Multi-Criteria Decision-Making in Software Production

Siamak Farshidi

SIKS Dissertation Serious No. 2020-35
The research reported in this thesis has been carried out under auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Multi-Criteria Decision-Making in Software Production
ISBN: 978-90-393-7350-7

Multi-Criteria Decision-Making in Software Production

Multi-criteria Besluitvorming in Softwareproductie

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. H.R.B.M. Kummeling, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op woensdag 2 december 2020 des morgens te 11.00 uur door

Siamak Farshidi

geboren op 8 augustus 1988,
te Teheran, Iran

Promotoren:	Prof. dr. S. Brinkkemper
	Prof. dr. ir. H.A. Reijers
Copromotor:	Dr. S.R.L. Jansen

This thesis was accomplished with financial support from NWO in the AMUSE project [project code 628.006.001].

Acknowledgments

Undertaking this Ph.D. has been a truly life-changing experience for me, and it would not have been possible to do without the support and guidance that I received from many people. The entire list of people who encouraged me during my Ph.D. is prolonged, whether within the research itself, forming a suitable environment, or even giving personal support. Probably, I will have forgotten to recall all names.

First and foremost, I would like to thank Slinger Jansen. To me, Slinger was more than a supervisor. He was an inspiring and enthusiastic co-promoter who showed me the right course of action in all situations. His encouragements and optimism were indispensable to keep me motivated. Slinger taught me how to do research and then report findings correctly to scientific communities.

I would like to express my sincere gratitude to Sjaak Brinkkemper. He actively supported my research and encouraged me to write down high-quality scientific papers. With a few questions, he always helped me to focus on contributions and structure my findings. Most of his remarks will be with me for my entire career as a researcher.

I would like to say a special thank you to Rolf de Jong. His support, guidance, and comprehensive insights into the software engineering field have made an inspiring experience for me. And special thanks to Michiel Overeem, whose support as an experienced software engineer and an eager researcher allowed my studies to go the extra mile (sorry for all the additional work, Michiel!).

Doing research and writing papers in collaboration with others was a pleasant experience as it allowed me to gain more insight and knowledge regarding what I was working on. I am profoundly grateful for my co-authors' hard work, who have not been mentioned above, and their substantial contribution to uplift the studies presented in this dissertation. Thanks, Sergio España, Jan Martijn van der Werf, Rolf de Jong, and Jacco Verkleij. I am also thankful to my motivated students who assisted me in the research projects, especially Elena Baninemeh, Andrey Krupskiy, Baharak Bakhtiari, and Mahdi Deldar. I am delighted to have worked with you, and I look forward to working with you again.

My research would have been impossible without the feedback from domain experts and case study participants from various software producing organizations that participated in the research projects' evaluation phases in this dissertation.

The work presented in this dissertation has been critically assessed and approved by an outstanding reading committee to whom I am more than grateful: Patricia Lago, Krzysztof Wnuk, Gabriele Keller, Bedir Tekinerdogan, and Diomidis Spinellis. I also

want to thank the numerous anonymous reviewers of the journals and conferences. Often I received constructive feedback that helped me to improve the quality of my publications.

To conclude, I cannot forget to thank my parents, beloved sister, and friends for all the unconditional love and support during my Ph.D. journey.

Siamak Farshidi,
November 2020

Contents

CHAPTER 1

Introduction

1.1 Decision-Making in Software Production

Decision-making is a continuous problem-solving process in our daily lives. Some people consider decision-making as an art, while others view it as practice. Decisions can be personal or professional, but, in each case, alternative solutions typically have permanent consequences. In other words, decision-making is one of the underlying cognitive processes of human behaviors by which a preferred alternative or a course of action is selected from among a set of options according to particular criteria (Wang et al., 2004). The decisions we make have the potential to affect ourselves and others in the short and long term.

Decision theories are widely applied in many disciplines. One example is software engineering (Rus et al., 2003), which has been defined as a continuous decision-making process (Fitzgerald & Stol, 2014). For instance, software producing organizations need to decide whether using their internal development resources (in-house), buying Commercial off-the-shelf (COTS) components, do subcontracting (outsourcing), or whether to use open-source software (Badampudi et al., 2018). In literature, decision-making is typically defined as a process or a set of ordered activities concerning stages of problem identifying, data collection, defining alternatives, selecting a shortlist of alternatives as feasible solutions with the ranked preferences. Fitzgerald et al. (2017) define decision-making as a process that consolidates critical assessment of evidence and a structured process that requires time and conscious effort. Kaufmann et al. (2012) states that the decision-making process encourages decision-makers to establish relevant decision criteria, recognize a comprehensive collection of alternatives, and assess the alternatives accurately.

Software engineers make a sequence of design decisions while developing a software product (Ruhe, 2002). Each design decision can be analyzed as an episode of complex problem-solving (Pressman, 2005) that relies on a substantial amount of knowledge and rationale. Design decisions in the software development lifecycle are significantly constrained by former decisions and lead to additional constraints on future decisions (Burge et al., 2008). Making informed design decisions in different phases of the software development lifecycle has critical impacts on the success of a software product.

Software producing organizations, private and public, are under pressure that forces them to react instantly to evolving conditions in business environments. They need to be innovative in how they operate and be agile in making strategic, tactical, and operational decisions. Such decisions require significant amounts of knowledge regarding decision domains. As software engineers who are working at such organizations are not experts in every field, they need to invest a considerable portion of their time in acquiring knowledge regarding each decision domain (Meyer et al., 2019). Additionally, they need to keep their knowledge up-to-date because of the provisional and volatile nature of knowledge in the software engineering field.

Software engineers as decision-makers, are biased to their prior knowledge, called familiarity bias (Tversky & Kahneman, 2000), to rapidly frame a solution to a problem they do not yet fully understand, called solution-first bias (Cross, 1999), or to inadequate or misleading knowledge, called confirmation bias (Phillips Brooks, 1995). Hence, such biases undermine decision-making outcomes (Burge et al., 2008). In the software production domain, unbiased decision support is required to mitigate the impacts of tacit expert knowledge, domain interpretation, over-learned professional practices, and evaluate decisions made by software engineers.

Software architecture is a subclass of software engineering that focuses on developing high-quality and successful software products based on fundamental design decisions (Medvidovic & Taylor, 2010). Accordingly, decision-making has been extensively studied in that domain. It is essential to highlight that well-known authors such as Jansen & Bosch (2005), Garlan & Shaw (1993), and Clements et al. (2003) have explained the decision-making process in software architecture and introduced a set of design decisions.

Nowadays, the development of software products, systems, and services typically results in complex decision models and decision-making processes (Badampudi et al., 2018). Decision support software has evolved in various disciplines, such as software engineering (Rus et al., 2003), to facilitate the decision-making process. Since the emergence of the concepts of decision support systems, in the mid-1960s, a considerable number of research articles regarding decision support has been published in the literature (Morton, 1971), moreover, a significant number of tools, including plugins in Eclipse, have been introduced to support software engineers with their decision-making problems. However, the complexity of the software engineering process and its socio-technical nature as the main barriers to the adoption of decision support systems (Donzelli, 2006). As decision support systems are traditionally designed for other equally complex application domains, such as clinical decision support systems (Bright et al., 2012; Musen et al., 2014), we need to identify an approach to overcome these challenges and support software engineers with their decision making processes.

1.2 Multi-Criteria Decision-Making Problems

A decision problem in software production is not addressed in the same way by all software engineers. Each software engineer has her priorities, tacit knowledge, and decision-making policy (Doumpos & Grigoroudis, 2013); consequently, one software engineer's judgment is expected to differ. Addressing such issues in building decision models in software production forms the focal point of interest in multiple-criteria decision making (MCDM).

MCDM is both an approach and a set of techniques, to provide an overall ranking of alternative solutions, from the most preferred to the least preferred solution (Dodgson et al., 2009). Alternative solutions may differ in the extent to which they achieve several objectives, and no one alternative solution will be best in achieving all objectives. Besides, some conflict or trade-off is usually evident amongst the objectives; alternative solutions that are more beneficial are usually more costly. Costs and benefits typically conflict, but so can short-term benefits compared to long-term ones, and risks may be higher for the otherwise more beneficial options.

MCDM problems consist of a finite set of alternative solutions, explicitly known at the beginning of the solution process Floudas & Pardalos, 2008. In multi-criteria design problems (multiple objective mathematical programming problems), alternative solutions are unknown. An alternative solution can be found by solving a mathematical decision model. Typically, the number of alternatives is either infinite or not countable, when variables are continuous, or typically very large if countable, when variables are discrete. However, both kinds of problems are considered as sub-classes of MCDM problems. The basic working principle of any MCDM technique is the same and known as six-step of the decision-making process (Majumder, 2015): (1) identifying the objective, (2) selection of the criteria, (3) selection of the alternatives, (4) selection of the weighing method, (5) applying the method of aggregation, and (6) decision-making based on the aggregation results.

Each decision-making problem in software production can be modeled as an MCDM problem that deals with evaluating a set of alternatives and considering a set of decision criteria (See Section 1.5). The challenge consists of evaluating and selecting the most suitable alternatives for software engineers (decision-makers) according to their preferences and requirements (Majumder, 2015).

In this dissertation, we focus on a set of decision-making problems pragmatically that software engineers face in software production (See Section 1.5). The following categories of decisions in software production are discussed: (1) decision-making regarding components for inclusion into software products (Chapters 2, 3, 4). (2) decision problems related to software development technologies that deal with finding the best fitting technologies for developing a software product (Chapters 5, 6). (3) architectural design decisions concerning pattern-driven software design (Chapters 7, 8, 9).

1.3 Multi-Criteria Decision-Making Techniques

The tools and techniques based on MCDM are mathematical decision models aggregating criteria, points of view, or features Floudas & Pardalos, 2008. Support is a fundamental concept in MCDM, indicating decision models are not developed following a process where the decision maker's role is passive (Dvořák et al., 2018). Alternatively, an iterative process is applied to analyze decision-makers' priorities and describe them as consistently as possible in a suitable decision model. This iterative and interactive modeling procedure forms the underlying principle of decision support tendency of MCDM, and it is one of the main distinguishing characteristics of the MCDM as opposed to statistical and optimization decision-making approaches (Gil-Aluja, 2013).

In literature, a wide range of MCDM techniques has been introduced to address MCDM problems in software production, such as the analytic hierarchy process (AHP) (Garg et al., 2011; Jusoh et al., 2014), Technique for Order Preference by Similarity to Ideal Solution(TOPSIS) (Oztaysi, 2014; Tang et al., 2019), analytic network process (ANP) (Lee & Kim, 2000; Yazgan et al., 2009), case-based reasoning (CBR) (Jadhav & Sonar, 2011; Li et al., 2009), fuzzy set theory (Rodriguez et al., 2017; Rouhani & Ravasan, 2015), Boolean Decision Tree (BDT) (Pahl et al., 2018; Staderini et al., 2018), weighted scoring method (WSM) (Davies & Reeves, 2010; Delgado et al., 2015), genetic algorithm (GA) (Doval et al., 1999; Guo et al., 2011), mathematical programming (Karsak & Özogul, 2009; Sahay & Gupta, 2003), and their hybrids.

The majority of the MCDM techniques in literature define domain-specific quality attributes to evaluate alternative solutions. Such techniques are mainly appropriate for specific case studies. Furthermore, MCDM approaches are valid for a specified period; therefore, the results of such studies, by technology advances, should be outdated. Additionally, a pairwise comparison is typically considered as the main method to assess the weight of criteria in MCDM techniques. For a problem with n number of criteria $\frac{n(n-1)}{2}$ number of comparisons are needed (Saaty, 1990). It means that the pairwise comparison is a time-consuming process, and gets exponentially more complicated as the number of criteria increases (Ribeiro et al., 2011). A subset of MCDM approaches, such as TOPSIS and AHP, are not scalable (Ibriwesh et al., 2018; Khari & Kumar, 2013), so in modifying the list of alternatives or criteria, the whole process of evaluation should be redone. Therefore, these methods are costly and applicable to only a small number of criteria and alternatives.

In this dissertation, we developed a theoretical framework to assist software engineers with a set of MCDM problems in software production. The framework provides a guideline for software engineers to capture knowledge systematically from different sources of knowledge to build decision models for MCDM problems in software production. Knowledge has to be collected and organized when it is needed to be employed. We designed, implemented, and evaluated a decision support system for software production, called SoProDSS, that utilizes such decision models to facilitate decision-making and support software engineers with their daily MCDM problems. A broad study has been carried out based on qualitative and quantitative research to evaluate the efficiency and effectiveness of the SoProDSS and the decision models inside its knowledge base to support software engineers with their decision-making process in software production.

1.4 Decision Support Systems

A DSS is an information system that comprises domain-specific knowledge and decision models to assist decision-makers by offering knowledge and the interpretation of several alternatives (Wang, 1997).

Researchers from different disciplines have been studying DSSs for more than five decades. The history of introducing such systems backed to the mid-1960s when Scott Morton published the concept of DSSs in February 1964. Scott Morton and his colleague Andrew McCosh carried out a study on decision models used to assist marketing and production managers with a recurring decision-making process in coordinating production planning for laundry equipment (Morton, 1971).

Since the 1970s, business publications have started to publish an extensive list of research articles on management decision systems, strategic planning systems, and DSSs (Sprague Jr & Watson, 1979). For instance, Little (1970) identified a set of numerical procedures for processing data and decision models to assist in managerial decision-making. Patrick Gerrity also presented a DSS for supporting investment managers in their daily administration of a client's stock portfolio (Gerrity, 1971). Afterward, countless research activities regarding designing, implementing, and investigating DSSs have occurred in academia and industry that resulted in expanding the scope of their applications. These research activities also expanded the field of DSSs beyond the initial business and management application domain. Nowadays, DSSs can be categorized into the following five broad categories (Power, 2008a):

(1) Model-Driven DSSs employ a single or a combination of multiple quantitative decision models with a fundamental level of functionality in their knowledge base. Typically, such DSSs are not data-intensive and receive a limited set of parameters from decision-makers to assist them in analyzing a complex decision problem (Power & Sharda, 2007). For instance, Interactive Financial Planning Systems can be categorized as Model-Driven DSSs (Sharda et al., 1988).

(2) Data-driven DSSs assist decision-makers by analyzing and mining big data existing in organizational systems. So that such systems ease the access to a large amount of accurate, well-organized multidimensional data (Power, 2008b). Management reporting systems, data warehousing, executive information systems, and business intelligence systems are a few examples of data-driven DSSs (Power & Sharda, 2007).

(3) Document-Driven DSSs manage, retrieve, and manipulate unstructured data in a variety of digital documents, such as textual policies and procedures, product specifications, catalogs, and corporate historical documents, including recorded meetings, corporate reports, and relevant correspondence (Power, 2000). For instance, a web-based search engine can be categorized as a Document-Driven DSS (Fedorowicz, 1993).

(4) Knowledge-Driven DSSs provide information, comprehension, recommendation, and suggestion to decision-makers based on the knowledge that has been captured using Artificial Intelligence or statistical tools like case-based reasoning, rules, frames, and Bayesian networks (Power & Sharda, 2007). Knowledge-Driven DSSs, sometimes called Intelligent Decision Support methods (Dhar & Stein, 1997), use problem-solving approaches to derive appropriate actions for particular problems (Baumeister & Striffler, 2015). Expert systems and recommender systems follow the knowledge-driven approach (Melle et al., 1984).

(5) Communication-Driven DSSs facilitate collaborative decision-making by using communication technologies (Power & Sharda, 2007). A Communication-Driven DSS shares its knowledge base among multiple decision-makers in geographically dispersed locations via interactive communication to support group decision-making on a specific decision making problem (Mohemad et al., 2010). For instance, voting mechanisms, and anonymous input of ideas and preferences are communications-driven DSSs (Desanctis & Gallupe, 1987).

Note, an information system may integrate multiple types of DSSs to develop thoroughly unique decision support (Power & Sharda, 2007); for example, an information system can be a model-driven DSS with a knowledge-driven DSS module for pre- or post-processing. Similarly, a DSS may incorporate both a data-driven and a model-driven subsystem.

1.4.1 Decision Support in Software Production

To make informed decisions, the decision-makers around a software product should either acquire knowledge themselves or hire external experts to support them with their decision-making process. The decision-making process becomes more complicated as the number of decision-makers, alternatives, and criteria increases. Therefore, software production is a suitable domain to deploy DSSs that intelligently support these decision-makers with the decision-making process.

In the history of software engineering, since the 1980s (Sommerville, 1985), academic researchers carried out a significant number of studies on decision support systems and their applications in the software engineers process. A subset of selected studies is presented as follows.

Holsapple et al. (1982) proposed a Document-Driven DSS for automated database design based on a set of managerial reports. They introduced a notion of report schemata, specified in terms of record types and binary relations, as a framework for analyzing report structures and interactions among reports. Next, four Knowledge-Driven DSSs were built using two approaches to knowledge acquisition, to facilitate early detection of potential problems that software engineers might face while coding and testing software development project(Ramsey & Basili, 1989). Carando (1989) presented a DSS that used hypertext systems and artificial intelligence techniques, as knowledge acquisition, to support software engineers in the software development process. She studied potential challenges in using the DSS in software engineering, including the difficulties of search versus browsing, user (dis)orientation, and the extra complexity that such tools add to the software development process. Afterward, a DSS was introduced to assist novice project managers in designing and tailoring the software development process to their specific projects for achieving required software reliability values (Rus & Collofello, 1999). The DSS had two subsystems: a knowledge-based component for reliability engineering strategy selection and a process simulator for strategy assessment. Ruhe (2002) described fundamental principles and expectations on the paradigm of software engineering decision support and then conducted two case studies of employing DSSs in the area of requirements negotiations. Then, a DSS was presented to model the software development process (Donzelli, 2006). The author conducted a study on the effects of requirements instability on software development projects. The DSS supported Project managers with simulating different possible unstable scenarios (e.g., different kinds of requirements behavior over time) to estimate the corresponding project trajectories. Therefore, managers could predict unpredictable environments and manage potential changes in conditions over time (e.g., to decide when and how to feed requirements to the project). Becker et al. (2013) introduced a multi-criteria decision support system (MCDSS) for software component selection. The MCDSS evaluates a total of 51 Commercial-Off-The-Shelf components against a total of 631 decision criteria. The authors specified metrics, such as the *key decision factors* and *efficient criteria sets*, for the quantitative evaluation of decision criteria and sets of criteria, and illustrated their application to a set of real-world decision cases. İmamoğlu & Çetinkaya (2017) designed and implemented a DSS to support software engineers with programming language selection in software production. A Knowledge-Driven DSS was introduced that supported software engineers with the risk estimation process in employing software components based on their prioritized requirements (Hettiarachchi & Do, 2019).

Perkusich et al. (2020) carried out an SLR to identify the application of intelligent techniques in software engineering. They defined an intelligent technique as a "technique that explores data (from digital artifacts or domain experts) for knowledge discovery, reasoning, learning, planning, natural language processing, perception, or supporting decision-making". They selected 104 unique primary studies and realized that less than half of them performed an empirical approach to evaluate their proposed solutions. The key findings of their SLR showed that (1) the number of studies employing intelligent techniques in software engineering is increasing, (2) reasoning under uncertainty, search-based solutions, and machine learning are the most used intelligent techniques in the field; (3) the main objectives of the selected studies were effort estimation, requirements prioritization, resource allocation, and requirements selection for a release or sprint; and requirements management; (4) the risks of applying such intelligent techniques in the software engineering field is considerably high, so (5) more empirical evaluation and validation for such methods are required.

1.4.2 A DSS for MCDM Problems in Software Production

As aforementioned, we develop a theoretical framework for building decision models for MCDM problems in software production. The decision-making process becomes more complicated as the number of decision-makers, alternatives, and criteria increase (Majumder, 2015). Therefore, software production is a suitable domain to deploy DSSs that intelligently support these decision-makers with the decision-making process.

In this dissertation, we designed and implemented the SoProDSS that integrates key aspects of Knowledge-Driven and Model-Driven DSSs to store and organize the extracted knowledge regarding decision models systematically, to facilitate the decision-making process, and to support software engineers with their design decisions in software production (see Figure 1.4).

Decision-makers define and prioritize their domain feature requirements based on *MoSCoW* prioritization technique (DSDM consortium and others, 2014), and then send them to the Inference Engine of the SoProDSS. The Inference Engine infers candidate solutions using the rules and facts of the decision models that it has in its knowledge base. In other words, the Inference Engine excludes infeasible solutions and assigns scores to the feasible ones, and then offers a ranked shortlist of feasible solutions to the decision-makers.

1.5 MCDM Problems in Software Production

Decision-making is the process in which a decision-maker explains the decision problems in their context, the results to be achieved, and the actions that can be taken. In this process, the decision-maker "must make sense of an uncertain situation that initially makes no sense" (Schon, 1984). So that we need first to identify the decision problems that we want to address and explain the uncertain situations that software engineers face during the software engineering process. This section identifies three diverse categories of decision problems in software production to evaluate the proposed theoretical framework in this dissertation from separate perspectives.

1.5.1 COTS-Based Software Engineering

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties (Szyperski et al., 2002). The primary goals of Component-based Software Engineering (CBSE) are the provision of support for the development of systems as assemblies of components, the development of components as reusable entities, and the maintenance and upgrading of systems by customizing and replacing their components (Heineman & Councill, 2001). CBSE can significantly reduce development cost and time-to-market, and improve maintainability, reliability, and overall quality of software systems (Cai et al., 2000; Pour et al., 1999).

Over the last few decades, a significant number of product software firms have realized that they can no longer develop an entire software product themselves and still meet all customers' requirements (Van Den Berk et al., 2010). Customers frequently demand new and more specific functionality, forcing product software firms to look for third parties, such as COTS software vendors, to add customers' needs as black boxes to their software products (Graaf et al., 2003). COTS software is developed for an entire market instead of a set of specific customers (Xu & Brinkkemper, 2007).

COTS software is either employed as is or slightly customized within the bounds of an application's ability to be easily modified without changing its primary functionality. A COTS product, such as an application or a component, is sold, leased, or licensed to the general public; offered by a vendor trying to profit from it; developed by the vendor, who retains the intellectual property rights; available in multiple, identical copies and utilized without source code modification (Brownsword et al., 2000).

Nowadays, almost all product software firms are involved in these networks of third-party vendors gathered around a single platform called software ecosystems (Jansen et al., 2013b). A software ecosystem is a collection of actors performing as a unit and interacting with a shared market for software products and services, together with the relationships among them (Jansen et al., 2019).

In CBSE, requirements statements should be more adaptable and less particular. Software engineers should define requirements intrinsically as desirable needs rather than as hard constraints (Cechich et al., 2003). Afterward, they need to select the best fitting component according to the requirements, and then they have to assemble the selected solution with a well-defined software architecture (Pour, 1998).

The selection process becomes more complicated as the number of potential alternative components, such as COTS software, and services that vendors offer (decision criteria) increases in their Software Ecosystems (Majumder, 2015). For instance, The DBMS selection problem is a subclass of the COTS selection problem, and both problems are a subclass of MCDM problems. Becker et al. (2013) presented a multi-criteria decision support system for software component selection. The SoProDSS evaluates a total of 51 COTS components against a total of 631 decision criteria.

1.5.2 Software Development Technologies

Software engineers have a broad knowledge of software development technologies, including programming languages, and they apply software engineering principles to develop software products. By employing such engineering principles in the software development lifecycle, from requirements elicitation to software implementation and then deployment, they can build customized software products for individual stakeholders.

The demand for highly skilled and qualified software engineers seems to have no end. This demand is growing by a changing economic landscape and fueled by the necessity of software development technologies. On the one hand, billions of dollars are spent annually on software products (Bhattacharya & Neamtiu, 2011) that are produced and maintained by software engineers. On the other hand, business processes are introduced and managed by stakeholders and top-level managers who principally understand businesses (Olariu et al., 2016).

Software development is not an independent activity: it typically requires interactions with stakeholders, which necessitate a level of agreement in the description of the technical phases of development. Moreover, software products are getting more complicated, so that they need to be discussed at different abstraction levels depending on the technical background of the involved domain experts, phase of the development process, and business objectives (Brambilla et al., 2017). Modeling is a handy tool for addressing such issues in software production as it simplifies the technical complexities and improves system understanding through visual analysis.

Over the last two decades, tools to support model-driven software development, such as low-code/no-code platforms and business process management systems, have gained more attention, and a significant number of them with a wide range of features and services have been introduced (Hutchinson et al., 2014). The primary aspiration of such tools is to boost productivity and decrease time-to-market by facilitating development at a higher level of abstraction and by employing concepts closer to the problem domain at hand, rather than the ones given by programming languages (Sendall & Kozaczynski, 2003).

Software production based on a model-driven software development platform is not initiated by programming but modeled using visual modeling or declarative development tools and pre-built templates and components that can be understood by the business. The business model transforms into an application, such as web-based or wearable apps, by generating code or model interpretation. The simplified interface leads many to believe that building applications using model-driven software development platforms require little or no coding knowledge. However, sometimes these predefined components should be customized, by using programming languages, to fulfill the required functionality.

Judging the suitability of a set of programming languages for a software product, as an application or a customized component, is a non-trivial task. For instance, a purely functional language like Haskell is the best-fit for writing parallel programs that can, in principle, efficiently exploit huge parallel machines working on large data sets (Peyton Jones et al., 2008). However, while developing a dynamic website, a software engineer might consider ASP.net as the best alternative, and others might prefer using PHP or a similar scripting language. It is interesting to highlight that successful projects have been built with both: StackOverflow is built in ASP.net, whereas Wikipedia is built in PHP. Furthermore, a software engineer might prefer particular criteria, such as scalability in enterprise applications, whereas other criteria, such as technology maturity level, might have lower priorities.

Acquiring and expanding knowledge about programming languages is a highly complex process, as significant numbers of criteria and alternatives exist in the market (Bhattacharya & Neamtiu, 2011). Various factors need to be taken into account, of which, not all are obvious. Simultaneously, the choice of programming languages can have repercussions on the implementation cost, quality of the result, and maintenance cost of the application (Holtz & Rasdorf, 1988).

For instance, Meyerovich & Rabkin (2013) conducted survey research to identify the factors that lead to language adoption. They evaluated 26 programming languages against 33 decision criteria (programming concepts). Moreover, Rymer et al. (2019) researched a list of low-code platforms, including 13 vendors, to consider for the evaluation. From that initial pool of vendors, they narrowed the final list based on several inclusion criteria, such as low-cost-of-entry commercial models, support of building many business use cases, and primarily targets large enterprises. Then, they collected data from products and strategies through a detailed questionnaire, demos and briefings, and a reference-customer survey. They used those inputs, along with the analyst's experience and expertise in the marketplace, to score the platforms, applying a relative rating system that compares each platform against the others in the evaluation.

Accordingly, it is essential to build decision models to support software engineers and citizen developers, such as non-professional developers with limited technical knowledge, with technology selection problems that they might face during the software development lifecycle.

1.5.3 Software Architecture Design

Software architecture is fundamental for developing a software product and plays an indispensable role in its success or failure as software architecture deals with the base structure, subsystems, and interactions among these subsystems (Clements et al., 2003). Software architecture comprises a set of architectural design decisions, concerns, variation points, features, and usage scenarios that address various system requirements, including functional and quality requirements (Bosch, 2004). The architecture on which a software product is built must adapt quickly to new requirements, not only to new user requirements but also to the always-changing environment, such as changes in the underlying database management system (Xu & Brinkkemper, 2007).

The software architecture field has evolved over the last four decades (Clements & Shaw, 2009; Shaw & Clements, 2006) from the early fundamental concepts from the mid-80s to the ubiquitous proliferation of roles of software architects in contemporary industrial practice (Capilla et al., 2016). In literature, different authors such as Clements et al. (2007), Farenhorst & Van Vliet (2009), De Boer & Farenhorst (2008), and Razavian et al., 2016 have highlighted the role of decision-making in the architecting process. In other words, software architecture design can be viewed as a decision-making process: software engineers consider a set of alternative solutions that could solve a system design problem and select the set that is evaluated as an optimal solution (Lago & Avgeriou, 2006; Razavian et al., 2016). Software engineers' duties in this decision-making process is a frequently recurring topic of discussion. Additionally, researchers and practitioners deliberated on the proper set of duties, skills, and knowledge of architects (Clements et al., 2007).

Design decisions are concerned with the system's application domain, architectural patterns employed in the system, COTS components, other infrastructure selections, and other aspects needed to satisfy all requirements (Bosch, 2004). Avgeriou et al. (2007) stated that lack of architectural design decisions in software production leads to well-known consequences, including expensive system evolution, poor stakeholder communication, limited reusability of architectural assets, and poor traceability between requirements and implementation.

Each architectural design decision is made with a design rationale (Dutoit et al., 2007), which represents the knowledge that provides the answers to questions about the design decision or the process followed to make that decision. In literature, various researchers such as Babar & Lago (2009) and Avgeriou et al. (2007) have highlighted the necessity of document design rationale to maintain and evolve software products and avoid violating rules of design decisions underpinning the original architecture. Design rationale is an essential part of an architecture description according to the IEEE 1471 recommended practices (IEEE-SA, 2000).

Architectural knowledge needs to be documented and codified in some way so that it can be searched and retrieved at different times (Tang et al., 2011a). An architectural pattern describes high-level structures and behaviors of software systems and addresses a particular recurring problem within a given context in software architecture design (Buschmann et al., 1996). Architectural patterns aim to satisfy several functional and quality attribute requirements and document the architectural design decisions (Avgeriou & Zdun, 2005). In this dissertation, we model the decision-making process and offer a solution to document software engineers' design decisions and design rationales in pattern-driven architecture design.

Software product audits should not be regarded as isolated projects (De Boer & Van Vliet, 2009). Instead, individual audits affect each other even if they target unrelated software products. For instance, lessons learned in one project might apply to another. Furthermore, the applicability of specific quality criteria is not limited to a single project alone. Similar projects might use similar quality criteria, and some general quality criteria might even apply to virtually all software products. For example, in high-security systems, some form of user authentication will always be needed. Accordingly, making informed design decisions is an acquired skill. No matter how relatively skilled they are, novice software engineers would not have enough collection of known and experienced situations, design problems, or practical solutions to pattern match current situations or problems (Razavian et al., 2016).

Additionally, architectural knowledge, specifically about architectural patterns, such as their application domains and their interactions with quality attributes, has been widely addressed in the literature. For instance, Avgeriou & Zdun (2005) proposed a pattern language, which was mainly focused on the relationships among

24 patterns and performed a categorization based on the concept of “architectural views”. Garlan & Shaw (1993) outlined six architectural patterns and showed how they could be applied and adapted to specific software systems. As the knowledge is fragmented over a wide range of heterogeneous studies (Buchgeher et al., 2016; Me et al., 2016; Tang et al., 2011b), a decision model is required to capture and aggregate this knowledge systematically and support software engineers with the architectural pattern selection problem (Babar et al., 2009; Falessi et al., 2011; Jansen & Bosch, 2005).

Kruchten (2006) stated that software architecting involves consensus decision-making in which software engineers balance between stakeholder concerns and requirements, including functional and quality requirements. Clements et al. (2007) explained that architecting is far more complicated than just making technical decisions. In a large-scale study in practice, they found that software engineers regularly interact with stakeholders and are involved in the organization and business-related issues but typically lead the architecting process.

Moreover, it is essential to note that lack of time or trust in a tool for storing or sharing knowledge is a well-known issue in architectural knowledge management literature (Babar et al., 2007a; Farenhorst & Van Vliet, 2009). To deal with such issues, incentives for sharing architectural knowledge should be created (Ghosh, 2004). One way to do this is by making the codification of architectural knowledge easier. Although some knowledge is inherently tacit in nature and therefore impossible to codify, i.e., detach from its owner (Nonaka & Takeuchi, 1995), researchers stress that a group’s or community’s performance increases significantly if everyone is informed of each other’s expertise or when more explicit knowledge is available to internalize (Cummings, 2003). Accordingly, a trustable knowledge management and decision support tool should be employed to support software engineers with their daily tasks in architectural design decisions.

1.6 Problem investigation

Research begins with inquiries or attempts to find solutions to a particular problem (Gregor, 2006). The developed theory should depend on the nature of the problem and the questions that are addressed. Whether the questions themselves are worth asking should be considered against the state of knowledge in the domain at the time. In other words, problem investigation is a knowledge question that asks for information and understanding of the given problem, without yet changing it. The problem investigation aims to formulate the problem, explain it, and possibly predict what would happen if nothing is done about it (Wieringa, 2009).

An essential step in the process of Design Science Research (DSR) is showing that existing theories, in the shared knowledge base of design scientists, are or are not adequate for addressing a particular problem (March & Storey, 2008). Before taking any further steps, an extensive literature study should be conducted to identify the potential solutions (theories) that might address the problem. The main objective of the problem investigation is to define the problem in its context, and possibly to indicate what would happen if nothing is done about it. Furthermore, it tries to consider the designers’ tacit knowledge regarding the problem as a part of the initial hypotheses and inputs of the theory development process. Note, research activities (see Section 1.7.2), such as expert interviews and case studies, define the primary sources of knowledge in the knowledge acquisition phase of the theory development process (Meredith et al., 1989). Figure 1.1 shows the problem investigation and the development process that we have followed to develop the MCDM framework.

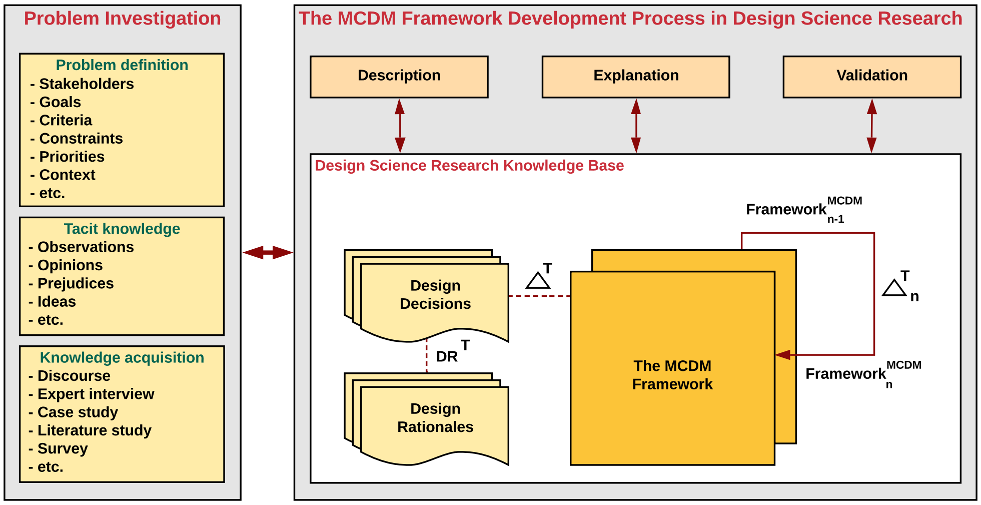


Figure 1.1: The design process of the MCDM framework.

In the problem investigation phase of the framework, the decision-making problem should be defined accurately, tacit knowledge should be converted to explicit knowledge properly that ca be used as an input to the development process, and different knowledge acquisition techniques can be employed to acquire knowledge from the decision-making problem domain.

Problem definition: Most real-world decision-making problems are ill-defined to some degree (Fortus et al., 2005), lacking required information and not having a well-defined ending state and, therefore, with neither a known correct nor best solution (Frederiksen, 1986; Nickerson, 1994). Accordingly, MCDM problems in each category of MCDM problems (see Section 1.5) should be defined precisely in their context. Furthermore, the evaluation criteria for solutions to the problem should be found by analyzing the problem, namely by identifying solution criteria, often called requirements, based on stakeholder goals, constraints, and priorities (Wieringa, 2009).

In this dissertation, we formulate each decision-making problem in software production as an MCDM problem: Let $Alternatives = \{a_1, a_2, \ldots a_{|Alternatives|}\}$ be a set of alternatives for an MCDM problem in software production. Moreover, $Features = \{f_1, f_2, \ldots t_{|Features|}\}$ be a set of domain features, including the most prominent technical and non-technical domain features of the alternatives. Each $a \in Alternatives$ supports a subset of the set $Features$. The goal is to find the best fitting solutions, where $Solutions \subset Alternatives$, that support a set of domain feature requirements, called $Requirements$, where $Requirements \subseteq Features$.

For example, in the database selection problem (See Chapter 2) we consider 73 database technologies (Oracle Enterprise Edition 12.1, MongoDB Enterprise Server 3.4.3, etc.) as alternative solutions (Set $Alternatives$). Moreover, we identify 307 decision criteria, such as database model (relational, graph, etc.), required functionality (transaction, backup, etc.), cost (license, support, etc.), as database features (Set $Features$). Next, the case study participants define their feature requirements based on the MoSCoW prioritization technique (DSDM consortium and others, 2014).

An MCDM technique receives $Alternatives$ and their $Features$ as its input, then applies a weighting method to prioritize the $Features$ based on the decision-makers' preferences to define the $Requirements$, and finally employs a method of aggregation to rank the $Alternatives$ and suggests $Solutions$. Accordingly, an MCDM technique can be formulated as follows:

$$MCDM_{technique} : Alternatives \times Features \times Requirements \rightarrow Solutions$$

Typically, a unique optimal solution for an MCDM problem does not exist, and it is necessary to employ decision-makers' preferences to differentiate between solutions (Majumder, 2015). Figure 1.2 visualises MCDM in a 3-dimensional space. It shows that the degree of satisfaction of the decision-makers with a suggested solution is fuzzy, which means that the satisfaction degree from a decision-maker (software engineer) perspective may range between completely true (best fit) and completely false (worst fit) (Dvořák et al., 2018), which is represented by a range of colors from red to dark green.

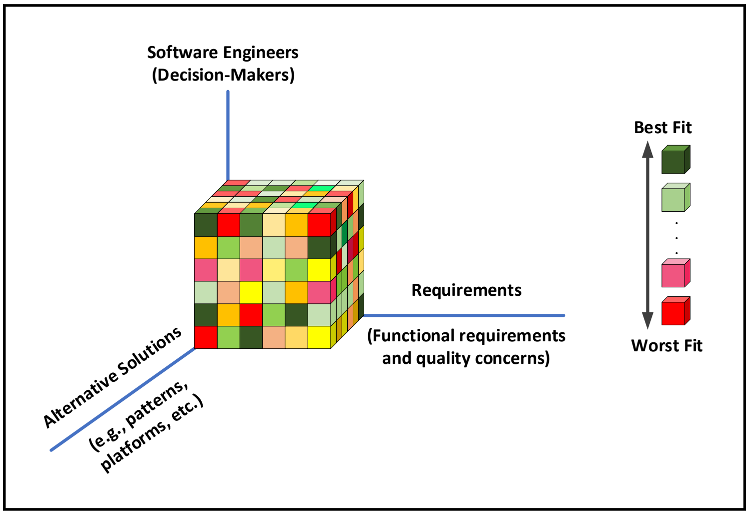


Figure 1.2: This figure shows MCDM in software production in a 3-dimensional space. Note, the degree of the decision-makers' satisfaction with a solution according to their priorities and preferences (requirements) ranges between the best and worst fit alternative solutions, which is represented by a range of colors from red to dark green.

Tacit knowledge: Polanyi (1966) defines tacit knowledge as "an individual's actions"

rather than "what that individual knows". Tacit knowledge has a significant effect on the outcome of the development process of a theory (Wong & Radcliffe, 2000). In other words, tacit knowledge is something typically derived from experience and difficult to express. In the theory development process, tacit knowledge of designers (such as observations, opinions, prejudices, and ideas) is highly valuable and has a significant influence on interpreting and making design decisions, which may result from a rich understanding and knowledge, but cannot be explained by explicit reasoning (design rationales).
Knowledge acquisition: Knowledge acquisition is the process of capturing, structuring, and organizing knowledge from multiple sources (Gruber, 1989). Human experts, discourse, internal meetings, case studies, literature studies, or other research activities are the primary sources of knowledge. The knowledge acquisition process can be divided into four phases (Chen, 2004): (1) *Planning:* In this phase, we gain knowledge regarding the problem domain, define domain experts, evaluate different knowledge acquisition techniques, and outline proper procedures. (2) *Knowledge extraction:* The main objective in this phase is to extract knowledge from sources of knowledge, including domain experts and literature study, by employing various knowledge acquisition techniques. (3) *Knowledge analysis:* The outcomes of the knowledge extraction phase, including concepts and heuristics, are interpreted in formal forms, such as facts, rules, ontologies, or fuzzy logic sets. (4) *Knowledge verification:* Typically, domain experts verify the formal forms of the heuristics and concepts. If the captured knowledge is insufficient to address the problem, alternative knowledge acquisition techniques, such as machine learning techniques, can be employed.

1.7 Research Approach

In this section, we elaborate on the research methods and relate them to individual research questions to which they apply (see Table 1.1). Apart from the methods discussed below, a literature study is carried out for each research question to reflect our work's objective and results.

Table 1.1: The mapping between the Research Questions (RQ) and Research Methods that have been employed in each chapter (Ch.). Note, ticks (✓) in black show the main research methods in the chapters.

Ch.	**RQ**	**Research Methods**					
		Design Science	*Conceptual Modeling*	*Case Study*	*Expert Interview*	*Document Analysis*	*Systematic Literature Review*
2	1,2,3,4,5,6	✔	✔	✔	✓	✓	
3	1,2,3,4,5,6	✓	✓	✔	✓	✓	
4	1,2,3,4,5,6	✓	✓	✔	✓	✓	
5	1,2,3,4,5,6	✓	✓	✔	✓	✓	
6	1,2,3,4,5,6	✓	✓	✔	✓	✓	
7	1,4	✓	✓		✔		✔
8	5	✔			✓		
9	1,2,3,4,6	✓	✓		✓		✓

1.7.1 Research Questions

Software engineering is a knowledge-intensive field that can be viewed as a decision-making process (Pressman, 2005). Software engineers spend a significant portion of their time collecting data regarding their daily tasks (Meyer et al., 2019). Moreover, they are, like most intelligent professionals, opinionated, moody, and convinced of their tacit knowledge. In order to mitigate the impacts of tacit expert knowledge, domain interpretation, and of overlearned professional practices, software engineering knowledge needs to be systematically captured and organized when it is required. Thus, the Main Research Question (MRQ) of this dissertation is stated as follows:

MRQ — ***How can software engineering knowledge be captured and organized systematically to support software engineers with software production decision-making?***

To answer the main question, we formulated the following Research Questions (RQs). First, we need to focus on the traditional decision-making process to understand how software engineers typically make decisions and tackle decision-making problems in software production. Second, we have to investigate potential decision-making problems that software engineers might face during software production. In order to address such decision-making problems, we need to identify sources of knowledge that can be used to capture knowledge regarding the domains of decision problems. Then, we need to determine the knowledge acquisition techniques that can be used to extract knowledge from the sources of knowledge systematically. Moreover, we should categorize and organize the extracted knowledge effectively when they are required. Lastly, we have to assess knowledge acquisition and organization techniques from software engineers' perspectives.

RQ_1 — ***How do software engineers make decisions in software production?***

We first need to know how software engineers typically make decisions and tackle decision-making problems in software production. Hence, we should identify potential challenges that software engineers face during the decision-making process. Then, we need to find out whether a framework can support them with systematically capturing knowledge regarding the decision-making problems, overcoming the challenges, and facilitating the decision-making process. RQ_1 is answered in Chapters 2, 3, 4, 5, 6, 7, and 9.

RQ_2 — ***How can a framework be developed that serves as a reference framework for decision problems in software production?***

The usability and suitability of the framework for addressing decision-making problems in software production should be investigated. To do so, we need to build several decision models based on the framework and evaluate them in their problem domain. The evaluation of each decision model leads to partial validation of the framework as a reference guideline that can be employed to address decision-making problems in software production. RQ_2 is addressed in Chapters 2, 3, 4, 5, 6, and 9.

RQ_3 — ***Which sources of knowledge should be used to build decision models in software production?***

After selecting the decision problems that we want to address, the fundamental sources of knowledge should be identified that can be used to build decision models for the problems. For instance, according to the problem definition (see Section 1.6), the right sets of decision alternatives, features, and their relationships should be identified. Tacit knowledge of the domain experts, scientific research papers, wikis, webinars, and white papers can be considered different sources of knowledge. Based on the nature of the decision-making problem, reliable and unbiased sources of knowledge should be identified. RQ_3 is answered in Chapters 2, 3, 4, 5, 6, and 9.

RQ_4 — ***How should domain knowledge for building a decision model be extracted and categorized?***

In order to build decision models, it is essential to follow a systematic approach to capture knowledge and then identify the concepts that should be used to build a decision model. In other words, we need to employ the right knowledge acquisition techniques according to the essence of a decision problem, then categorize and organize the extracted knowledge effectively. For instance, sometimes, domain features and quality attributes are conceptually intertwined. Similarly, domain features of a decision problem can be at different levels of abstraction. RQ_4 is answered in Chapters 2, 3, 4, 5, 6, and 9.

RQ_5 — ***How should the extracted knowledge for building a decision model be organized for facilitating the decision-making process?***

The decision-making process becomes more complicated as the number of decision-makers, alternatives, and criteria increases. Software production, therefore, is a suitable domain to deploy DSSs that intelligently support these decision-makers with the decision-making process. We need to design and implement a DSS to organize the extracted knowledge regarding decision models. We need to evaluate the DSS's efficiency and effectiveness in supporting software engineers with the decision problem. RQ_5 is addressed in Chapters 2, 3, 4, 5, 6, and 8.

RQ_6 — ***Will software engineers be willing to use the decision models within the decision support system to perform their tasks?***

Finally, we need to investigate the liabilities and strengths of employing the decision models within the knowledge base of the SoProDSS in addressing the decision problems. Additionally, we have to carry out a study concerning the willingness of software engineers to use the SoProDSS as tool support to perform their daily tasks and make design decisions. RQ_6 is answered in Chapters 2, 3, 4, 5, 6, 7, and 9.

1.7.2 Research methods

Research methods are classified based on their data collection techniques (interview, observation, literature, etc.), inference techniques (taxonomy, protocol analysis, statistics, etc.), research purpose (evaluation, exploration, description, etc.), units of analysis (individuals, groups, process, etc.), and so forth (Meredith et al., 1989). Multiple research methods can be combined to achieve a fuller picture and a more in-depth understanding of the studied phenomenon by connecting complementary findings that conclude from the use of methods from the different methodological traditions of qualitative and quantitative investigation (Johnson & Onwuegbuzie, 2004). In this dissertation, different research methods are combined and applied to answer the research questions. Table 1.1 gives an overview of the research questions and research methods that we have used in each chapter.

Design science (employed in Chapters 2, 3, 4, 5, 6, 8, and 9) is an iterative process (Simon, 1996), has its roots in engineering (Hevner et al., 2004), is broadly considered a problem-solving process (Fortus et al., 2005), and attempts to produce generalizable knowledge about design processes and design decisions. The design process, similar to a theory, is a set of hypotheses that eventually can be proven only by the creation of the artifact it describes (Walls et al., 1992). The feasibility of a design can, however, be supported by a scientific theory to the extent that the design comprises principles of the theory. Research investigations involve a continuous, repetitive cycle of *description*, *explanation*, and *testing* (Meredith et al., 1989). Accordingly, in most cases, theory development is a process of gradual change (Baxter, 2004). The research approach for creating decision models for MCDM problems is Design Science, which addresses research through the building and evaluation of artifacts to meet identified business needs (Hevner et al., 2008).

Conceptual modeling (used in Chapters 2, 3, 4, 5, 6, and 9) leads to a mental model of possible relationships and concepts. For example, taxonomies, ontologies, and categorizations are all conceptual models. A concept is a bundle of meanings or characteristics connected with particular events, objects, or conditions and used for representation, identification, communication, or understanding (Meredith, 1993). Concepts are linked to each other through relationships, by verbal or mathematical statements, called propositions. A conceptual model is a composition of concepts, with or without propositions, and used knowledge representation formalisms, such as mathematical formulation and a graphical representation, to describe (but not explain) an event, object, or process.

Case Study (conducted in Chapters 2, 3, 4, 5, and 6) is an empirical research method (Jansen, 2009) that investigates a phenomenon within a particular context in the domain of interest (Yin, 2017). Case studies can describe, explain, and evaluate a hypothesis. Researchers are free to carry out an empirical study in any way, as long as it takes place within a realistic context. A case study can be employed to collect data regarding a particular phenomenon, apply a tool, and evaluate its efficiency and effectiveness using interviews. Yin (2017) distinguishes four types of case study designs according to holistic versus embedded and single versus multiple. In this dissertation, we mainly employ holistic multiple case designs: examining multiple real-world companies' cases within their context to learn more about one specific unit of analysis, evaluating the decision models, and partially validating the MCDM framework. In deductive case study research, such as the studies that we have done within this dissertation, designers often employ qualitative research design (such as *Conceptual Modeling*) to build theories. The primary purpose of a qualitative research project is to carry out an in-depth analysis of the links among the concepts of the theory.

Expert Interview (applied in Chapters 2, 3, 4, 5, 6, 7, 8, and 9) is an essential knowledge acquisition technique (Chen, 2004) in qualitative research. We followed Myers and Newman guidelines (Myers & Newman, 2007) to conduct a series of qualitative semi-structured interviews with senior software engineers to explore expert knowledge regarding the decision-making problems and evaluate the outcomes of our study. We developed a role description for each decision-making problem before contacting potential experts to ensure the right target group. Then, we contacted the experts through email using the role description and information about our research topic. The experts were pragmatically and conveniently selected according to their expertise and experience that they mentioned on their *LinkedIn* profile. We considered a set of expert evaluation criteria (including "Years of experience", "Expertise", "Skills", "Education", and "Level of expertise") to select the experts. Each of the interviews followed a semi-structured interview protocol and lasted between 60 and 90 minutes. We used open questions to elicit as much information as possible from the experts minimizing prior bias. All interviews were done in person and recorded with the interviewees' permission, and then coded for further analysis.

Document analysis (employed in Chapters 2, 3, 4, 5, and 6) is a systematic procedure for reviewing or evaluating documents, including text and images that have been recorded without a researcher's intervention (Bowen et al., 2009). Document analysis is one of the analytical methods in qualitative research that requires data investigation and interpretation to elicit meaning, gain understanding, and develop empirical knowledge (Corbin & Strauss, 2014). In order to build a decision model for an MCDM problem, we reviewed webpages, whitepapers, scientific articles, fact sheets, technical reports, product wikis, product forums, product videos, and webinars to collect data and map domain features to alternatives of the MCDM problem. Afterward, a structured coding procedure was employed to extract knowledge from the selected sources of knowledge. Structured coding captures a conceptual area of the research interest (Saldaña, 2015). Next, the extracted knowledge has been classified into several categories such as *quality attributes*, *domain features*, and *alternatives*.

Systematic Literature Review (applied in Chapters 7 and 9) is one of the most broadly accepted research methods of evidence-based software engineering (Kitchenham et al., 2004). An SLR provides a prescribed process for identifying, evaluating, and interpreting all available evidence relevant to a particular research question or topic (Petersen et al., 2008). In this dissertation, the SLR functioned as a knowledge acquisition process to capture knowledge about patterns and ultimately making it available in forms of reusable knowledge. The SLR has been carried out following the steps and guidelines of Kitchenham (2004): reasoning the necessity of the SLR, defining research questions, searching relevant studies, applying inclusion/exclusion criteria, assessing the quality of studies, extracting knowledge, analyzing the results.

1.8 Theory Development in Design Science Research

Research investigations involve a continuous, repetitive cycle of *description*, *explanation*, and *testing* (Cooper & Emory, 1995; Meredith et al., 1989). Thus, proposing knowledge (explanation) and validating knowledge (testing) simply are two stages in the ongoing cycle of research. The design process is a Generate/Test Cycle (Simon, 1996) that begins with an investigation of a practical problem, then determines a set of possible solutions, validates them, and selects one of them, and implements the solution chosen. The outcome of each evaluation can be the start of a new cycle in the design process (Van Strien, 1997).

Theory development is a process of gradual change (Baxter, 2004). In other words, the development process of theories in DSR is an act of iterative interpretation. In a theory development process, designers make comparable design decisions in a particular domain. Such design decisions and their corresponding design rationales should be grouped and considered as repeatable design decisions for building similar theories in a particular domain. The captured knowledge from the development process of a theory provides an overview of the theory's design decisions and rationales. With such overviews, scientists can systematically develop and report their theories in DSR.

A theory is a coherent group of interrelated concepts and propositions accepted as principles of explanation and understanding (Meredith, 1993). In this dissertation, we develop a theoretical framework, called the MCDM framework (see Figure 1.4), in an iterative process. Figure 1.1 shows the design process that we have followed to develop the MCDM framework for building decision models for MCDM problems in software production. The applicability and validity of the framework have been tested by conducting multiple deductive case studies in each iteration of the theory development process.

1.8.1 Design Science Research Knowledge Base

The knowledge base is used to analyze, document, and track the MCDM framework revisions besides its design decisions and rationales. Moreover, the mapping among the design decisions, design rationales, and the framework revisions is recorded in the knowledge base for further reuse in the theory development process. In other words, the knowledge base captures the acquired knowledge during the theory development process and makes it available in forms of reusable knowledge (context, cause, and result).

Design decisions are made during the MCDM framework design process. Some design decisions are unique and involve situations that designers have not experienced before. However, some design decisions are repeatable during the development process of the framework. For instance, each decision-making problem in software production deals with evaluating a set of technology alternatives and taking into account a set of domain features. We considered $alternatives$ and $features$ as two primary constructs of the MCDM framework.

Design rationales indicate reasoning underlying the design process that explains, derive, and justify design decisions (Fischer et al., 1991). In other words, a design rationale is clear documentation of the reasons behind design decisions. The main goal of recording design rationale is to document the incremental changes and design decisions that designers make during the framework development process.

In the development process of the MCDM framework, a design decision is made according to a particular design rationale, and each design decision leads to a **revision of the theory**. In other words, we made multiple revisions to the framework during its development process. Suppose D_n^T is the set of all made design decisions of the framework until its n^{th} revision, then Δ_n^T denotes the made design decisions of the framework in its n^{th} revision, where $\Delta_n^T = D_n^T \setminus D_{n-1}^T$.

1.8.2 The MCDM Framework Development Process in DSR

Developing a theory is an incremental process (Simon, 1996) and requires making correct design decisions. Moreover, the development process involves a continuous and repetitive cycle consist of *description*, *explanation*, and *validation* of the theory (Cooper & Emory, 1995). The description must be preceded by explanation and validation. Therefore, the development process starts with the description stage. A theory may involve only one of the stages in the development process at a time (Meredith et al., 1989). In each cycle of the development process, (new) constructs and relationships are defined or revised accurately to keep consistency among components of the theory.

The **description** stage of the development process led to a level of understanding regarding the constructs and provided a well-documented characterization of the MCDM framework. This characterization is then used for building or testing the framework. Based on a description, an initial set of constructs and relationships of the framework is proposed. The description stage involved the initial explication of tacit knowledge in the textual description based on our understanding or prediction. Moreover, the improvement of the appearance of a knowledge representation formalism, such as intuitive naming and coloring, led to a higher level of understanding.

Hospers (1956) presents the **explanation** stage of the development process as the following three common interpretations: (1) Stating the scope of the framework, (2) Showing the framework is an instance of a familiar phenomenon, and (3) Bringing the framework under a law. In other words, the explanation stage translated interpreted observations, ideas, etc. into new constructs of the framework. This stage elaborated the understanding from the description stage into more detail by making conceptual design decisions. In this stage, constructs and relationships are (re)defined or (re)categorized.

The **validation** stage of the development process validated the identified constructs in the earlier stages. This stage involved a prediction based on the explanation constructed in the previous stage, and then observation to determine if the prediction was correct. Alternatively, a prediction could be proposed and then checked against observations already made or included in the description. In this stage, the conceptualization design decision was made for empirical evaluation and validation of the framework.

Figure 1.1 shows the development process of the MCDM framework. In each cycle of the MCDM framework development process, we instantiated the framework to build a decision model for a particular MCDM problem in software production. Next, we conducted a set of real-world case studies and expert interviews to evaluate the decision model and partially validate the framework. Each case study participant had several alternative solutions for the MCDM problem before participating in the research, and then, it defined a set of feature requirements based on the MoSCoW prioritization technique (DSDM consortium and others, 2014). Afterward, the SoProDSS generated a shortlist of ranked feasible solutions.

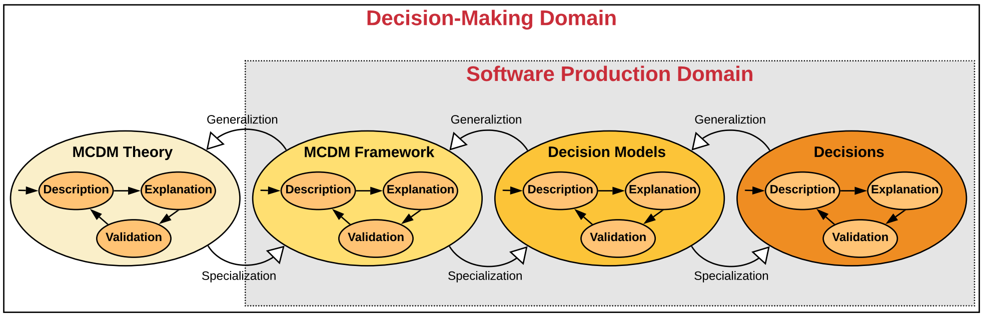


Figure 1.3: The MCDM framework, based on MCDM theory, is instantiated to build decision models to support software engineers with MCDM problems in software production.

We defined the results' success when they, in part, aligned with the case study participants' shortlist and when they provided new suggestions that were identified as being of interest to the case study participants. Note, using the case study participants' opinion as a measurement instrument was risky, as they may not have sufficient knowledge to make a valid judgment. We countered this risk by conducting more than one case study, assuming that the case study participants were handling in their interest and applying the SoProDSS to other problem domains in software production.

Software engineers make a sequence of design decisions while developing a software product (Ruhe, 2002). Each design decision can be analyzed as an episode of complex problem-solving (Pressman, 2005) that relies on a substantial amount of knowledge and rationale. Figure 1.3 shows that the MCDM framework, based on MCDM theory, is instantiated to build decision models to support software engineers with their decision-making process in software production. In this dissertation, we build six decision models for the following MCDM problems in software production: (1) Database Management System, (2) Cloud Service Provider, (3) Blockchain Platform, (4) Programming Language Ecosystem, (5) Model-Driven Software Development Platform, and (6) Architectural Pattern selection problems.

1.9 The MCDM Framework

The proposed theoretical framework, MCDM framework (see Figure 1.4), follows the six-step decision-making process (Majumder, 2015) to build decision models for MCDM problems in software production.

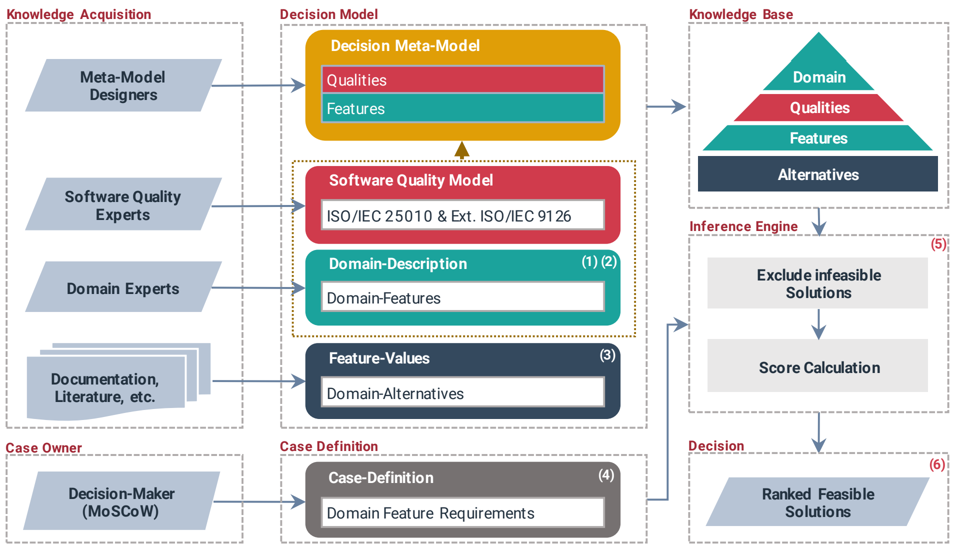


Figure 1.4: The MCDM framework that we follow to build decision models for MCDM problems in software production.

Gregor (2006) sets forth a taxonomy of five different types of theory in use within the field of information science: (1) A *theory for analyzing* does not extend beyond analysis and description. No causal relationships among phenomena are specified, and no predictions are made. (2) A *theory for explaining* provides explanations; however, it does not predict with any precision. Moreover, no testable propositions are offered by theory. (3) A *theory for predicting* provides predictions and has testable propositions; however, it does not have well-developed causal explanations. (4) A *theory for explaining and predicting* provides prediction and has both testable propositions and causal explanations. (5) A *theory for design and action* gives explicit prescriptions (such as models and techniques) for building an artifact in information science.

There is no complete agreement about the characteristics and components of theories in DSR (Baskerville & Pries-Heje, 2010). Gregor (2006) carefully delineated structures of theories and finally proposed seven components of a theory. We use Gregor's suggested components to formulate the MCDM framework in DSR as follows:

representations: is a subset of *means of representation* of theories in DSR. Different knowledge representation formalisms (Sloman, 1985), such as mathematical formulations and graphical representations, can be applied to represent a theory. Sometimes, multiple knowledge representation formalisms are utilized to represent a specific theory and improve the theory's depth of understanding. Note, we considered a graphical representation, mathematical formalization, and ontological modeling as three knowledge representation formalisms of the MCDM framework.

constructs: is a subset of *primary constructs* of theories in DSR. A construct is a collection of meanings or characteristics connected with particular phenomena of interest in a theory. All the other components of a theory depend on these primary constructs. The MCDM framework contains the following constructs: (1) *Domain* of the problem, (2) *Domain Features*, (3) *Alternatives*, (4) *Software Quality Model* to indicate the impacts of domain features on alternatives, (5) *Decision-Maker*, (6) *MoSCoW* prioritization technique as the weighing method, (7) *Domain Feature Requirements*, (8) the *Weighting Sum Model (WSM)* as the method of aggregation, (9) the *Inference Engine* of the decision support system to suggest feasible solutions, (10) *Ranked Feasible Solutions*.

relationships: is a subset of *statements of relationship* of theories in DSR. Constructs are linked to each other through relationships, by verbal or mathematical statements. A relationship can be associative, compositional, unidirectional, bidirectional, conditional, or causal. Table 1.2 shows verbal statements of the relationships in the MCDM framework. As we considered a mathematical formalization as one of the knowledge representation formalisms in this dissertation, so mathematical statements of relationships are defined to indicate the mapping among the sets of *Software Quality Model*, *Domain Features*, and *Alternatives*. For instance, the mapping between the sets of *Software Quality Model* and *Domain Features* is formulated as follows: $Qualities \times Features \rightarrow Boolean$.

For example, *consensus-mechanisms* as a blockchain feature influences the *Fault-tolerance* quality aspect. The framework does not enforce a blockchain feature to present in a single quality aspect; Blockchain features can be part of many quality aspects. For example, *Spam-attack resistant* as a blockchain feature might connect to multiple quality aspects such as *Recoverability* and *Availability*.

scope: is a member of all possible scopes of theories in DSR. The *scope* of a theory specifies the degree of generality of the statements of relationships. Moreover, the scope of a theory determines its boundaries and limitations.

causes: is a subset of *causal explanation* of theories in DSR. Causal explanations are revealing and explaining the causes of specific phenomena in theory. In other words, the theory gives statements of the relationship among phenomena that show causal reasoning. As aforementioned, the MCDM framework is based on the six-step of the decision-making process; therefore, all of the constructs and their relationships are implicitly inherited from this process.

propositions: is a subset of *testable propositions* of theories in DSR. Statements of relationships among constructs are asserted in such a form that they can be validated empirically.

prescriptions: is a subset of *prescriptive statements* of theories in DSR. Statements of relationships among constructs are expressed in a way to show how software engineers can perform something in practice (e.g., building an artifact).

A theory in DSR is formulated as a composition of primary constructs and statements of relationships, and used knowledge representation formalisms to describe (but not explain) particular phenomena. The causal explanation leads to go beyond the description and explain particular phenomena in a theory. Note, *means of representation*, *primary constructs*, *statements of relationship*, and *scope* are mandatory components to define a theory in DSR, however, *causal explanations*, *testable propositions*, *prescriptive statements* are optional and included in a theory based on the theory type. Therefore, the design space of the proposed theoretical Framework ($framework^{MCDM}$) in this dissertation is defined as follows:

$$\begin{aligned} Framework^{MCDM} : & \; representations \times constructs \times \\ & relationships \times scope \times causes \times \\ & propositions \times prescriptions \end{aligned}$$

The MCDM framework is *a theory for design and action* (Gregor, 2006) for building decision models in the context of MCDM problems in software production. In other words, this theoretical framework shows HOW decision-makers can efficiently make decisions to select the best fitting alternative solutions based on their requirements and priorities. Table 1.2 outlines the MCDM framework as a *theory for design and action* in DSR.

Table 1.2: This table outlines the components of the proposed theoretical framework in this dissertation, called the MCDM framework. Note, The framework is a *theory for design and action* (Gregor, 2006) for building decision models in the context of MCDM problems in software production.

Theory components	**Realization**
Means of representation	graphical representation, mathematical formalization, and ontological modeling
Primary constructs	(1) Domain; (2) Domain Features; (3) Alternatives; (4) Software Quality Model; (5) Decision-Maker; (6) MoSCoW; (7) Domain Feature Requirements; (8) Weighting Sum Method; (9) Inference Engine; (10) Ranked Feasible Solutions.
Statements of relationship	(1) Each decision model is built for the domain of an MCDM problem in software production. (2) Each alternative in the problem domain supports several domain features. (3) The mapping between domain features and alternatives is based on the documentation of the alternatives. (4) Software Quality Model indicates the impacts of domain features on alternatives. (5) The Software Quality Model is defined based on ISO/IEC 25010 and ISO/IEC 9126. (6) The relationship between quality aspects and domain features is based on domain experts' knowledge. (7) Each decision-maker has a set of prioritized domain feature requirements. (8) Decision-Makers assign priorities to their feature requirements base on the MoSCoW prioritization technique. (9) So that the set of domain feature requirements is a subset of features. (10) Each decision model should be uploaded to the knowledge-based of the decision support system. (11) The Inference Engine receives the prioritized domain feature requirements as its input. (12) The Inference Engine calculates the scores of the alternatives based on the Weighting Sum Method and the prioritized domain feature requirements. (13) The Inference Engine excludes infeasible solutions and ranks feasible solutions according to their scores. (14) A shortlist of ranked feasible solutions will be the outcome of the decision-making process. (15) A feasible solution is an alternative that supports the prioritized domain feature requirements.
Scope	MCDM problems in software production.
Causal explanations	The decision-making process, based on the MCDM theory, contains the following phases: (1) identifying the objective, (2) selection of the features, (3) selection of the alternatives, (4) selection of the weighing method, (5) applying the method of aggregation, and (6) decision making based on the aggregation results.
Testable propositions	(1) It is impossible to make a software production decision alone, so that decision support is required to make informed decisions. (2) Complex production processes require decision support, so software engineering processes need decision support. (3) Software engineers are, like most intelligent professionals, opinionated, moody, and convinced of their truths, so the framework can support them with making unbiased decisions. (4) The framework can be employed as a guideline to build decision models for MCDM problems in software production. (5) The decision support system facilitates the decision-making process and supports software engineers with their MCDM problems in software production. (6) Following the framework to build decision models leads to effective and efficient decisions.
Prescriptive statements	(1) the framework should be used to identify practical alternatives and the right set of domain features for each MCDM problem in software production. (2) the decision support system offers a short ranked list of feasible solutions; therefore, decision-makers should perform further investigations, such as performance testing, to find the best-fit alternatives for their software products. (3) the MoSCoW prioritization technique can be used in the feature requirements elicitation phase without employing the decision support system. (4) different Software Quality Models can be used to build decision models.

1.10 Dissertation Outline

This dissertation is structured into four separate parts to address the research questions. Chapters 2 to 9 are classified according to the three categories of decision problems in software production. The last chapter, Part IV, concludes the dissertation and provides an answer to the main research question.

Chapter 1 — The *Introduction* explains the motivation and highlights its relevance to software engineering. Additionally, it defines the multi-criteria decision-making problem in software production and positions the research in context through the research questions. Next, it describes the research methods that we have used as knowledge acquisition techniques to address the research questions. Moreover, it elaborates on the theory development process in this dissertation. Furthermore, it discusses the rationale behind the conducted research in each chapter of this dissertation. Finally, it outlines the SoProDSS that we have implemented as tool support for software engineers to facilitate their decision-making process.

COTS selection problems

Chapter 2 — *Database Management System Selection* is a crucial challenge for software producing organizations. Several decision factors come into play, such as database model (relational, graph, etc.), required functionality (transaction, backup, etc.), cost (license, support, etc.). Decision-makers are faced with an MCDM problem to find their suitable database technology because a large number of decisions of a similar kind have to make. Besides, the number of potential solutions and decision factors is significantly large. This chapter defines the selection problem as an MCDM problem and then build a decision model for it according to the MCDM framework. Finally, it reports how we have evaluated the decision model through three industry case studies.

Chapter 3 — *Cloud Service Provider Selection* is a significant challenge for businesses. Typically, cloud vendors' service portfolios are heterogeneous and combined with complicated service features and pricing models. The evaluation and selection process for an infrastructure-as-a-service provider requires collaboration, budgeting, and future-proofing of resources. Additionally, offered services are characterized using multiple criteria, such as their popularity, geographic location, and deployment model, so it is essential to have a reliable method to select desirable cloud vendors based on decision-makers' requirements. This chapter explains a decision model for the selection problem and explains the evaluation process through four industry case studies.

Chapter 4 — *Blockchain Platform Selection* is complicated because many criteria, such as security, interoperability, consensus mechanisms, and platform transaction speed, have to be considered. The selection process refers to the steps involved in choosing and evaluating the best fitting blockchain platforms for software-producing organizations according to their preferences and requirements. As the number of blockchain platforms in the market is increasing rapidly, the selection problem is becoming a significant challenge for software-producing organizations. Hence, knowledge regarding blockchain platforms has to be collected and organized when it needs to be applied. In this chapter, we build a decision model for the decision problem and then explain the evaluation results of conducting three real-world case studies.

Software development technology selection problems

Chapter 5 — *Programming Language Ecosystem Selection* is a highly complex process, as various factors need to be taken into account, of which, not all, are apparent. Third-party libraries play an essential role as many software applications are built by gluing together plenty of existing libraries in the market, so such libraries increase language growth. Additionally, communities generate wikis, forums, and tutorials to improve the learnability and understandability of languages. So that judging the suitability of programming languages for a software product is a non-trivial task. In this chapter, we build a decision model for this selection problem and evaluate it based on four industry case studies.

Chapter 6 — *Model-Driven Software Development Platform Selection* is challenging for enterprises to select the best fitting platforms that address their requirements and priorities. Model-driven software development platforms emphasize visual interfaces to enable citizen developers to build and deploy business applications with relative ease. Nowadays, a significant number of such platforms with a wide range of features and services are available on the market. Accordingly, a decision model for this selection problem is required to facilitate the decision-making process. In this chapter, we build a decision model for the decision problem and evaluate it based on four real-world case studies.

Decision-Making in Pattern-Driven Design

Chapter 7 — *Capturing Software Architecture Knowledge* is required for supporting software engineers with their design decisions in pattern-driven architecture. Selecting architectural patterns is a challenging task for software engineers, as knowledge about these patterns is scattered among a wide range of literature. Having this knowledge readily available supports software architects in making more efficient and effective design decisions that meet their quality concerns. In this chapter, we report on a systematic literature review, intending to build a decision model for the architectural pattern selection problem. Moreover, twelve experienced practitioners at software-producing organizations evaluated the usability and usefulness of the extracted knowledge.

Chapter 8 — *Decision Support for Pattern-Driven Architecture* is needed to support software engineers with their daily decision-making process with designing a pattern-driven architecture. In this chapter, we introduce a DSS that uses a decision model for supporting software architects with the pattern selection problem according to their requirements, including functional and quality requirements. Finally, a practical running example is presented to explain the usefulness and efficiency of the DSS to support software engineers with the pattern selection problem.

Chapter 9 — *Design Decisions in Pattern-Driven Architecture* are made continuously, based on design rationales and tacit knowledge of software engineers, while designing software architectures. A subclass of design decisions is selecting architectural patterns, which is a challenging process, as knowledge about them is fragmented over a wide range of heterogeneous studies. In this chapter, we introduce a decision model for the selection problem. Moreover, a study has been carried out with 24 software practitioners in the Netherlands to assess the user acceptance of the decision model based on qualitative analysis of the Technology Acceptance Model. They used the decision model in the knowledge base of the DSS that we introduce in Chapter 8.

Concluding the Research

Chapter 10 — The *Conclusion* chapter of this dissertation answers the research questions based on the results presented in the previous chapters. We provide an overview of our findings and explain our contributions to the software engineering field. Moreover, it discusses the limitations and threats to the validity of the research. Finally, we reflect on this dissertation research and describe future research perspectives.

Part I: Commercial Off-The-Shelf Components

CHAPTER 2

Database Management Systems

Software producing organizations face the challenge of including new technology in their products, such as cloud technologies and database management systems. As software architects and senior developers are not experts in this domain, they need to consult external experts or acquire the knowledge themselves. Therefore, software production is a suitable domain to deploy decision support systems that intelligently support these decision-makers in selecting the desirable technology for their product. We present a decision support system that supports decision-makers in choosing the most suitable database technology. The case studies and experts confirm that the approach increases insight into the selection process, provides a richer prioritized option list than if they had done their research independently, besides reduces the time and cost of the decision-making process.

keywords- multi-criteria decision-making; decision support system; technology selection; database management system

This chapter is based on the following publication:

Farshidi, S., S. Jansen, R. de Jong & S. Brinkkemper (2018c), "A decision support system for software technology selection", *Journal of Decision Systems*.

2.1 Introduction

Technology selection is the process of assessing the potential value of technologies and their contribution to the competitiveness and profitability of Software Producing Organizations (SPOs). Moreover, technology selection is one of the essential processes in evaluating innovation, popularity, and suitability of technologies for SPOs. Therefore, technology selection is an essential decision-making process for SPOs. The challenge consists of evaluating and selecting the most suitable technologies for SPOs according to their preferences and requirements. The selection process is complicated because too many factors, such as suitability and cost, should be considered. Therefore, the technology selection process can be modeled as a multi-criteria decision-making (MCDM) problem that deals with the evaluation of a set of alternatives, and taking into account a set of decision criteria (Triantaphyllou et al., 1998).

In recent years researchers introduced a significant variety of techniques, methods, and tools to solve different technology selection problems for SPOs. Many variations exist, but all share the vital phases of the decision-making process. The majority of MCDM approaches use pairwise comparison as the weighting method, which typically is not scalable. Thus, in the case of modifying the list of alternatives or criteria, the whole process of evaluation must be repeated. These methods are costly and only applicable to a small number of criteria and alternatives. Technology selection decisions are often made ad hoc, without reference to reliable models or sound methodologies. Furthermore, the results of technology selection solutions in the literature are valid for a specified period, so by technology advances, they should be performed again. Hence, a reusable, evolvable, and expandable decision-making approach is needed to make the right decision based on the characteristics of the environment.

This study introduces a Decision Support System (DSS) to help decision-makers with MCDM problems, such as DBMS selection. The DSS is a tool that can be used over the full life-cycle and can co-evolve its advice based on evolving requirements. The DSS applies the six-step decision-making process (Majumder, 2015) to build maintainable and evolvable decision models for MCDM problems, and makes the knowledge acquisition more reliable and trustful. The sets of criteria and alternatives, plus the relationship among them for an MCDM problem can be up-to-date and regularly manipulated without having impacts on the validity of its decision model. The novelty of the DSS lies in utilizing the MoSCoW prioritization technique (*MoSCoW*) (DSDM consortium and others, 2014) to assess criteria weights and reduce uncertainty, in introducing assessment models to measure the values of non-boolean criteria, and in using ISO/IEC quality aspects to indicate the relationship among criteria according to domain experts' knowledge.

This paper is structured as follows. Section 2.2 describes the design science method followed, and the exploratory theory-testing case studies that have been performed. Section 2.3 gives a window into the literature of software technology selection and the multiple approaches to solving decision-making problems, such as ours. Section 2.4 formulates the technology selection problem in SPOs and describes the proposed DSS. Then, section 2.5 illustrates an application of the DSS to address the Database Management System (DBMS) selection problem, using multiple case studies to evaluate and emphasize the significance of the approach. Afterward, section 2.6 interprets the results of the case studies according to expert interviews and opinions. Next, section 2.7 highlights and overcomes barriers to the knowledge acquisition and decision-making process. Finally, section 2.8 summarizes the proposed approach and offers directions for future studies.

2.2 Research Method

The problem we are trying to solve is that software-producing organizations typically are not knowledgeable in the domains in which they need to make technology selections for integration into their products. The technology selection process can be modeled as an MCDM problem that deals with structuring, planning, and solving the problem concerning a set of criteria: 1) Identifying the objective, 2) Selection of the features, 3) Selection of the alternatives, 4) Selection of the weighing method, 5) Applying the method of aggregation, 6) Decision making based on the aggregation results.

To support these organizations, we propose a DSS, created using design science, based on the six-step decision-making process. The DSS has the goal of finding suitable alternatives that support a set of domain feature requirements. The traditional design science cycle is followed, and the DSS is inspired by expert knowledge, which is gathered through three series of interviews. Fourteen experts (three DSS experts, two academics, five Software Developers, and four Software Architects) participated in this research to evaluate the DSS in interviews that lasted between 45 and 90 minutes. The domain experts were pragmatically selected according to their expertise and experience that they mentioned in their professional profile.

Secondly, the efficiency and usefulness of the DSS are evaluated through three exploratory theory-testing case studies. The unit of analysis is a unique technology selection decision in a software product. We performed three such case studies at two SPOs to evaluate the DSS. The case studies typically lasted one day and consisted of (1) defining the domain feature requirements, (2) prioritizing them, and (3) comparing the DSS feasible solutions with their solutions.

2.3 Related work

In recent years, researchers introduced a variety of MCMD methods to address technology selection problems for SPOs. The *Analytic Hierarchy Process(AHP)* is a structured method for organizing and analyzing MCDM problems. This method has been extensively applied and combined with other techniques to solve MCDM problems. The *Technique for Order Preference by Similarity to Ideal Solution(TOPSIS)* suggests that the selected alternative should have the shortest distance from an ideal solution and the farthest distance from the negative ideal solution. The *FAHP* and *FTOPSIS* are the combinations of Fuzzy logic with the *AHP* and *TOPSIS* methods. The *Fuzzy MCDM(FMCDM)* assesses the ratings of alternatives versus criteria and the importance weights of criteria based on semantic values represented by fuzzy numbers. The *Machine Learning(ML)* explores the study and construction of algorithms that can learn from and make predictions on data.

Table 2.1: This table compares selected MCDM methods from literature to address technology selection problems. The first column (Domain) points out the problem domain. The second column (MCDM) denotes the MCDM approach. The third column (PC) indicates whether the approach applies the pairwise comparison(PC) as a weight calculation method or not. The fourth column (QA) determines the type of quality attributes. The seventh and eighth columns (#F and #A) signify the number of criteria and alternatives that were considered in the problem domain.

Author(s)	**Domain**	**MCDM**	**PC**	**QA**	**#F**	**#A**
This paper	DBMS	DSS	No	ISO/IEC 25010 EX. ISO/IEC 9126	307	73
Jusoh et al. (2014)	DBMS	AHP	Yes	Domain specific	12	3
Brahimi et al. (2016)	DBMS	ML	No	Domain specific	20	3
Garg et al. (2017)	DBMS	FMCDM	Yes	Domain specific	14	5
Lin et al. (2007)	Data warehouse system	FAHP	Yes	Domain specific	16	6
Onut & Efendigil (2010)	ERP software	FAHP	Yes	ISO/IEC 9126	13	3
Kohli & Sehra (2014)	Software Quality Model	FMCDM	Yes	Domain specific	3	3
Rodriguez et al. (2017)	Risk management approach	FAHP	Yes	Domain specific	5	5
Fu et al. (2010)	Project management software	FAHP FMCDM	Yes	Domain specific	14	4
Büyüközkan & Güleryüz (2016)	Product development partner	FAHP FTOPSIS	Yes	Domain specific	16	6
Becker et al. (2013)	COTS	DSS	No	ISO/IEC 25010 Domain specific	631	51

Table 2.1 illustrates selected MCDM approaches from literature. The majority of the MCDM techniques use pairwise comparison to assess the weight of criteria. For a problem with n number of criteria, $\frac{n(n-1)}{2}$ comparisons are needed (Saaty, 1990). Pairwise comparison is a time-consuming process and gets more complicated as the number of criteria increases. Some of the methods, such as *AHP* and *FAHP*, are not scalable. For instance, when the list of alternatives or criteria is modified, the evaluation process should be conducted again. These methods are costly and applicable to a small number of criteria and alternatives. The MCMD techniques in literature mainly define domain-specific quality attributes to evaluate alternatives. Such studies are typically appropriate for specific case studies. Furthermore, the results of these MCDM approaches are valid for a specified period, so by technology advances, new updates and releases, they will be out-of-date.

The DBMS selection problem is a subclass of the COTS selection problem, and both problems are a subclass of MCDM problems. Becker et al. (2013) present a multi-criteria decision support system (MCDSS) for software component selection. The MCDSS evaluates a total of 51 COTS components against a total of 631 decision criteria. The authors specified metrics, such as the *key decision factors* and *efficient criteria sets*, for the quantitative evaluation of decision criteria and sets of criteria, and illustrated their application to a set of real-world decision cases. The proposed DSS and MCDSS provide a substantial number of criteria to support decision-makers in the technology selection problem. Furthermore, they use the ISO/IEC 25010 (ISO, 2011) as a standard set of quality attributes. The main difference between our and the MCDSS is their weighting methods. Our DSS utilizes the *MoSCoW* to assess the significance of criteria. Moreover, it introduces assessment models to measure the values of non-boolean criteria, such as the cost of alternatives.

2.4 Multi-Criteria Decision-Making

This study introduces a DSS that applies the six-step decision-making process (Majumder, 2015) to build maintainable and evolvable decision models for MCDM problems and makes the knowledge acquisition more reliable and trustful. Let $Alternatives = \{a_1, a_2, \ldots a_{|Alternatives|}\}$ be a set of alternatives (technologies) in the market. Moreover, $Features = \{f_1, f_2, \ldots t_{|Features|}\}$ be a set of domain features, which includes the most prominent technical and non-technical domain features of the alternatives, so each $a \in Alternatives$ supports a subset of the set $Features$. The goal is finding the suitable alternative a, which supports a set of essential domain features (set $Requirements$), where $Requirements \subseteq Features$. In other words, an alternative a is the suitable one that supports domain feature requirements and satisfies the decision-makers' preferences. Typically, a unique optimal solution for an MCDM problem does not exist, and it is necessary to use a decision-maker preference to differentiate between solutions (Majumder, 2015).

The fundamental components of a typical DSS (Sage, 1991) are the DataBase management system, the Model-Base management system, and the Dialog Generation management system. The *DataBase management system* is a set of domain features facts related to an MCDM problem. The *Model-Base management system* is a collection of rules, heuristics, and knowledge related to the MCDM problem. The *Dialog Generation management system* is a user interface to interact with decision-makers.

The Inference Engine of a standard DSS infers solutions and does not rely on knowledge base facts and rules, so it works independently from the other components. The Inference Engine receives domain feature requirements and their priorities according to *MoSCoW* from the *Dialog Generation management system* as its input. Next, it finds the most relevant rules from a collection of models in the *Model-Base management system*. Then, the Inference Engine, by using facts about the *DataBase management system*, deduces decisions. Eventually, it sends ranked feasible solutions to the *Dialog Generation management system*. The DSS[1] comprises of the standard DSS components and is illustrated in Figure 2.1.

2.4.1 Decision Model

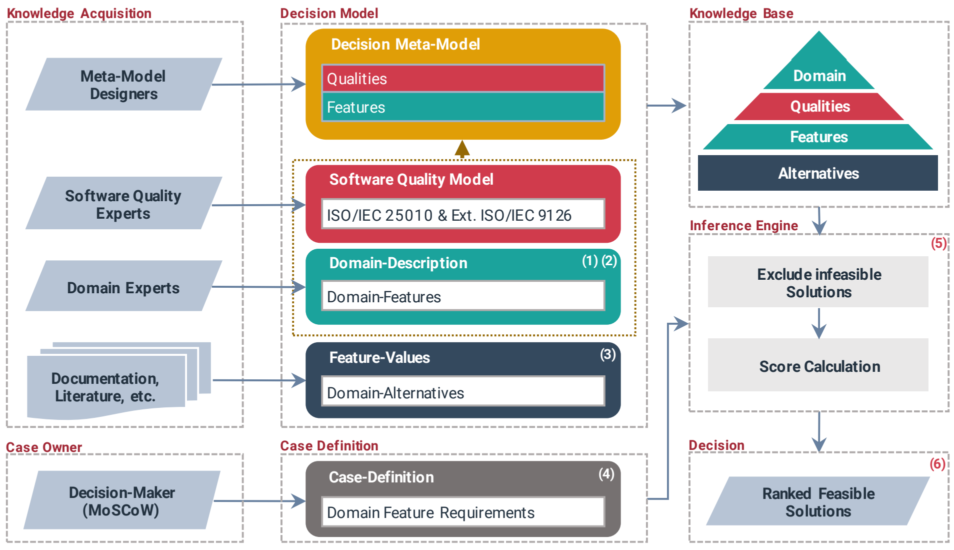
A decision model for an MCDM problem contains criteria, alternatives, and relationships among them (facts and rules). This section introduces the primary sources of knowledge and constituent parts of a decision model based on the six-step decision-making process.

Decision Meta-Model - The *Decision Meta-Model* defines the base structure of a decision model in the knowledge base. It includes two primary sets (*Qualities* and *Features*). The set *Qualities* is a set that contains software quality attributes, and the set *Features* is a set that consists of domain features of an MCDM problem.

Software Quality Model The *Software Quality Model* defines the software quality attributes and relationships among elements of the set *Qualities*. The DSS utilizes the *ISO/IEC 25010* standard (ISO, 2011) and *extended ISO/IEC 9126* stan-

[1]We implemented an online Decision Model Studio (https://dss-mcdm.com/) to build decision models for technology selection problems in SPOs.

Figure 2.1: A model-based decision support system for technology selection problems.



dard (Carvallo & Franch, 2006) in order to define the set *Qualities*. They are domain-independent software quality models and provide reference points by defining a top-down standard quality model for software systems. The elements of the *Software Quality Model* apply to classify domain features of an MCDM problem based on their impact on quality attributes of software technology alternatives.

Domain Description The *Domain Description* defines the first and second steps, denoted by *Identifying the objective* and *Selection of the features*, of the decision-making process. It specifies the domain features of an MCDM problem and maps the set *Qualities* to the set *Features*, where $Qualities \times Features \rightarrow Boolean$, based on domain experts' knowledge. Each domain feature has a data type, such as *Boolean* and *Numeric*. For example, the data types of domain features like the *popularity* and *Firewall* of a DBMS could be considered as *Numeric* and *Boolean* respectively.

Feature-Values The *Feature-Values* defines the third step, indicated by *Selection of the alternatives*, of the decision-making process. It determines a set of alternatives and maps them to the domain features set, where $Alternatives \times Features \rightarrow Boolean$. The primary source of knowledge in this phase could be documentation of alternatives, literature studies, social networks, alternative experts, etc.

2.4.2 Case Definition

The *Case Definition* defines the fourth step, denoted by *Selection of the weighing method*, of the decision-making process. The DSS employs *MoSCoW* to define decision-makers' domain feature requirements and assess the importance of required domain features. Domain feature requirements with *Must Have* or *Won't Have* priorities act as hard constraints and domain feature requirements with *Should Have* and *Could Have* priorities act as soft constraints. In other words, a case definition, based on the preferences of a decision-maker (*MoSCoW*), is a way to select domain feature requirements and assign priorities to them. Decision-makers specify desirable values for numeric domain feature requirements. For example, a decision-maker could be interested in prioritizing the DBMSs with TCOs lower than $5000 USD as more important than others. Therefore, the TCO lower than $5000 USD could be considered as a *should have* domain feature.

2.4.3 Inference Engine

The Knowledge Base is a collection of decision models, which are groups of rules and facts. The *Inference Engine* defines the fifth and Sixth steps, indicated by *Applying the method of aggregation* and *Decision making based on the aggregation results*, of the decision-making process. A feasible solution must support all domain feature requirements with *Must Have* priorities, and must not support all domain feature requirements with *Won't Have* priorities. The *Inference Engine* ranks the feasible alternatives based on their calculated scores. The score calculation process is based on the well-known Weighted Sum Model. Thus, by sorting the feasible solutions in descending order of their scores, the final ranked feasible solutions will be given as the result of the DSS.

2.5 DBMS Selection

The selection of efficient and cost-effective database technology is a crucial challenge for SPOs. A number of decision factors come into play, such as database model (relational, graph, etc.), required functionality (transaction, backup, etc.), cost (license, support, etc.). Decision-makers have to follow a trustworthy and iterative process to choose the DBMS, which best fulfills their requirements. Thus, SPOs are faced with an MCDM problem to find their suitable DBMS(s) because a large number of decisions of a similar kind have to make. Besides, the number of potential solutions and decision factors is significantly large.

As mentioned in section 2.4.1, Constituent parts of a decision model are *Decision Meta-Model*, *Software Quality Model*, *Domain Description*, and *Feature-Values*. The *Decision Meta-Model* defines the base structure of a decision model in the knowledge base, and it has two sets namely *Qualities* and *Features*. A decision model utilizes the *ISO/IEC 25010* standard and *extended ISO/IEC 9126* standard in order to define the set *Qualities*. The *Decision Meta-Model* and *Software Quality Model* are immutable for decision models based on the DSS approach. However, the *Domain Description* and *Feature-Values* should be define to structure a decision model for an MCDM problem.

This section presents a decision model according to the DSS approach to address the DBMS technology selection problem. Moreover, three case studies have been conducted to evaluate the efficiency and effectiveness of the DSS to solve the DBMS selection problem for SPOs.

2.5.1 Domain Description for DBMS Selection

As mention in section 2.4.1, a list of domain features of technology alternatives within the domain of interest should be specified. Domain experts are the main source of knowledge to identify the right set of domain features, although documentation and literature study regarding technology alternatives could be utilized to pinpoint an initial list of domain features. In order to define the domain of DBMS selection problem more than 250 features[2] (such as Auditing, Backup) have collected according to domain experts' suggestions. The *Software Quality Model* provides a general view of the software quality model. The decision model decomposes abstract concepts into more concrete ones, the domain features. Domain features have to define precisely to clarify the underlying quality concepts that they represent and to link them with the relevant quality aspects in the set $Qualities$. The *Domain Description* does not enforce a domain feature to present in a single quality aspect; Domain features can be part of many quality aspects. For example, *Immediate Consistency* as a DBMS feature might connect to multiple quality aspects such as *Recoverability* and *User error protection*.

In this study, domain features and the mapping between the sets $Qualities$ and *Features* for the DBMS selection problem are defined by nine domain experts, including two university professors, five Software Developers, and two Software Architects in the Netherlands. The domain features identified by six semi-structured interviews, then three experts participated in the research to map the considered domain features to the set *Qualities* based on a boolean adjacency matrix ($Qualities \times Features \rightarrow Boolean$).

2.5.2 Feature-Values for DBMS Selection

As mentioned in section 2.4.1, a list of technology alternatives of the domain of interest should be defined. Well-known technology solutions, websites, related forums, and domain experts are the primary source of knowledge to specify the list of technology alternatives. In this study, 73 DBMS technologies (Oracle Enterprise Edition 12.1, MongoDB Enterprise Server 3.4.3, etc.) from 10 data storage models (Relational, Document, etc.) have been considered. Next, the supportability of boolean domain features by the DBMS technologies investigated. The relationship between the sets *Features* and *Alternatives* defined based on the documentation and websites of the considered DBMS technologies. One of the principal challenges is the lack of standard terminology among documentation of DBMS technologies. Different vendors refer to the same concept by different names, or even worse, the same name might stand for different concepts in different DBMS technologies. Discovering conflicts in the *Feature-Values* is essential to prevent semantic mismatches throughout the DBMS selection process. Manufacturers tend to provide a partial view of their products. They emphasize their product's benefits, without mentioning weaknesses or provide only part of the truth. Some non-commercial articles compare DBMS technologies and features but are often based on the evaluators' limited knowledge of the technologies and their particular tastes (Franch & Carvallo, 2003). The next step in building a decision model for the DBMS selection problem is defining assessment models for each non-boolean domain features, such as *Popularity in the market* and *Total Cost of Ownership*.

[2]The entire list of the domain features and supportability of considered database technologies are available and accessible on the "DBMS Selection Model" website: https://dss-mcdm.com/

Popularity in the Market - In this study, the results of DB-Engines Ranking[3] is used to provide a metric on the popularity of DBMS technologies in the market. DB-Engines measures the popularity of a database system by using some parameters, such as the number of mentions of the system on websites and general interest in the system. *Popularity in the market* is a numeric domain feature of the DBMS selection problem that finds the most popular technologies in the market based on decision-makers' domain feature requirements.
Total Cost of Ownership - The cost of DBMS technologies varies widely, from entirely free to staggeringly expensive, and many factors and options should be considered. Database licensing can sometimes appear confusing, especially when it comes to well-known vendors, such as Oracle and Microsoft. A considerable variety of pricing methods and models, such as per core and server, for calculating the database licensing costs are available.

We defined four reference configurations[4], including the PC (1, 1, 4, 16, 25, 256, N, 5), Basic server (1, 1, 8, 64, 2×256, N, 25), Intermediate server (2, 2, 2×6, 2×256, 8×960, A/P, 15000), and Advanced server (2, 2, 2×24, 3000, 24×960, A/P, ∞), to get a rough estimate of the Total Cost of Ownership (TCO) of DBMS technologies.

The TCO of each alternative was asked directly from its vendor/maintainer or calculated via offered TCO calculators on websites of DBMS vendors. Many options, offers, and add-ons were not included in the TCO calculations because they were vendor-specific. The TCO is a domain feature of the DBMS selections problem that attempts to clear the fog somewhat regarding database licensing. However, the estimated values of the TCO cannot possibly provide a full and precise insight into the complex pricing and licensing schemes that DBMS providers use.

2.5.3 Empirical Evidence: The Case Studies

Three case studies have been conducted in the context of two SPOs to evaluate and signify the usefulness and efficiency of the DSS to address MCDM problems, precisely the DBMS selection problem. The case study companies considered a number of feasible DBMS technologies for their organizations through multiple internal expert meetings and extensive investigation into DBMS alternations before participating in this research.
AFAS Software - AFAS Software is an ERP vendor in the Netherlands with approximately 350 employees. One of AFAS' current challenges is validating whether they have chosen the right DBMS(s) for the new version of their main product. The new product requires two primary data storage, namely AFAS QS and AFAS SS.
ProcureComp - ProcureComp is an SPO that produces procurement software. ProcureComp's product is based on Microsoft technology. Presently, the ProcureComp product is being renewed and rebuilt using new Microsoft platforms, and this is a suitable time to rethink the data storage strategy for the new version of the ProcureComp product (NX1).

[3]The db-engines.com ranks database management systems according to their popularity. The ranking is updated monthly.

[4]Each reference configuration is indicated by a 7-tuple (CPU, Socket, Core, RAM, SSD, Failover, Max.DB), consisting of the number of CPUs, number of sockets, number of cores, amount of RAM (GB), SDD capacity (GB), failover type (None and Active/Passive), and maximum database file size (GB).

Table 2.2 demonstrates the number of domain feature requirements, which the case study participants indicated, of the AFAS QS, AFAS SS, and NX1 based on *MoSCoW*.

Table 2.2: The number of domain feature requirements of the case studies based on the MoSCoW priorities.

MoSCoW	AFAS QS	AFAS SS	NX1
Must Have	7	6	50
Should Have	8	4	5
Could Have	7	2	17

Table 2.3: The feasible solutions of the DSS for AFAS software and ProcureComp based on their domain feature requirements and MoSCOW priorities. The columns *Desirable* and *Undesirable* suggestions demonstrate which DSS feasible solutions already considered in the shortlist of case study participants based on their internal meetings and investigations. Moreover, the Columns *CP Rank* and *DSS score* of the table show the score calculation results of the DSS and the short ranked list of the feasible solutions based on the case study participants' opinions respectively.

Case Study	Feasible solutions	Desirable suggestion	Undesirable suggestion	DSS Score	CP Rank
AFAS QS	MySQL	✓		100 %	2
	DB2		✓	100 %	-
	Oracle Database		✓	100 %	-
	Postgres	✓		99.80 %	1
	SQL Server	✓		99.77 %	3
AFAS SS	Postgres	✓		100 %	1
	MySQL	✓		100 %	2
	MongoDB	✓		100 %	4
	DB2		✓	100 %	-
	Oracle Database		✓	100 %	-
	SQL Server	✓		99.45 %	3
NX1	SQL Server	✓		99.74 %	1

2.6 Results and Analysis

Table 2.3 illustrates the feasible solutions of the DSS for AFAS QS, AFAS SS, and NX1. The DSS deduced just one feasible solution for NX1 because ProcureComp experts restricted the search space by assigning 50 domain features as *Must Haves*, i.e., hard constraints. The reason is that the software architecture of NX1 depends heavily on the relational data storage model and Microsoft technology. Thus, ProcureComp experts were primarily interested in finding the edition of SQL Server (Enterprise edition 2016) that best covers their requirements and priorities. The AFAS software architecture is not dependent on a specific data storage model or vendor. Moreover, most of the domain feature requirements of AFAS QS and AFAS SS do not require specific DBMS technology.

The amount of annual TCO was a *Should Have* domain feature for AFAS. Hence, the DSS did not exclude any alternatives based on their TCO values. Table 2.3 shows that *Oracle* and *IBM DB2* database technologies are not desirable suggestions for AFAS. Because the case participants find that the annual TCO of these DBMS technologies, including extra options, end up being much higher than the other feasible solutions. In other words, they perceive that *MySQL*, *SQL Server*, and *Postgres* DBMS technologies are interesting suggestions because of their relatively low TCO for an intermediate server configuration, including extra options. Moreover, AFAS experts mentioned *IBM DB2* is undesirable, because they do not have enough experience with its performance, support, and licensing.

The case study participants at both companies confirm that the DSS provides practical solutions to help SPOs in their initial decisions for selecting DBMS technologies. In other words, the DSS recommended the same solutions as the case participants suggested to their companies after extensive analysis and discussions. However, the DSS offers a short ranked list of feasible solutions; therefore, SPOs should perform further investigations, such as performance testing and actual TCO calculation, to find the optimum DBMS technology for their software products. The case study participants state that their companies continuously improve and reevaluate their technologies, including the used DBMS technologies.

The case study participants entered a limited set of domain feature requirements. We were surprised to find that the experts have a limited view of what the domain feature requirements of the technology are. Furthermore, the case participants themselves were surprised to find what their primary concerns seem to be, especially when the opinions of different experts are combined. The DSS enables decision-makers to meet more complex requirements that they might have. More importantly, the case study participants confirm that the updated and validated version of the DSS is useful and valuable in finding the shortlist of feasible solutions. Finally, it reduces the time and cost of the decision-making process.

The consulted experts confirm that the DSS contains the main components of a standard DSS. Furthermore, they state that the DSS is a useful tool that provides more knowledge than they could have collected independently. The experts believe that experience in using a specific technology provides invaluable knowledge when selecting suitable technology. We, therefore, recommend that the DSS should use in combination with benchmarks where applicable.

2.7 Discussion

SPOs have different perspectives on their domain feature requirements in different phases of the Software Development Life-Cycle. Decision-makers typically consider generic domain features in the early phases of the life-cycle, whereas they are interested in more detailed and specific domain features as their development process matures. For instance, *Access Control* could be prioritized as a *Should Have* domain feature in the design phase, but in the implementation phase, one of its sub-features, e.g., *Label Based Access Control*, might be selected instead. Furthermore, domain features' priorities could be changed in different phases. Therefore, the DSS might come up with various solutions for an SPO in different phases of its software development life-cycle. The proposed DSS is a tool that can be used over the full life-cycle and can co-evolve its advice based on evolving requirements. As the choices of the participants are stored in the DSS, it does not cost a significant amount of time to rerun the decision-making process. Presently, we are designing solutions that enable "the crowd" to participate in contributing knowledge, without letting anyone commercial party influence the knowledge base. Furthermore, we are looking at methods to automatically extract domain features from manuals and documentation, using text mining techniques.

Decision-makers could bias the determination of domain feature requirements and their priorities. Biases, such as motivational and cognitive (Montibeller & Winterfeldt, 2015), arise because of shortcuts or heuristics that decision-makers use to solve problems and perform tasks. The Hawthorne effect, which is the tendency of decision-makers to change their behavior when observed, is a form of cognitive bias. The case study participants (AFAS and NX1 decision-makers) might have been more careful in the experimental setting than they would be in the real setting because they are being observed by scientists judging their selected domain feature requirements and priorities. Moreover, the Bandwagon effect, which is the tendency to do or believe things because many other decision-makers do or believe the same, is another form of cognitive bias. The Bandwagon effect typically shows up in group decisions. To mitigate the Hawthorne and Bandwagon effects, individual *and* group interviews conducted to collect the domain feature requirements for each case study.

2.8 Conclusion

Software producing organizations are faced with an MCDM problem when finding suitable COTS. The number of potential solutions (alternatives) and decision factors are significantly high. This paper is the first attempt at supporting architects in making complex decisions, where we ventured into the domain of DBMS technologies.

In recent years a variety of studies has been conducted to benchmark, compare, and evaluate database technologies. However, according to expert analysis, selecting a suitable DBMS technology for a software product is not utterly subjective. Finding a feasible solution for this problem based on decision-makers' priorities and requirements requires an in-depth investigation into the documentation of database technologies and extensive expert analysis. This study introduces a DSS to accelerate the process of finding the right DBMS technologies and suitable data storage models for SPOs. The novelty of the proposed DSS lies in utilizing *MoSCoW* to assess criteria weights and reduce uncertainty, in introducing assessment models to measure the values of non-boolean criteria, and in using ISO/IEC quality aspects to indicate the relationship among criteria according to domain experts' knowledge.

To keep the knowledge base of the DSS up-to-date and valid, a website[5] has created. We plan to create a community around the platform that will regularly update the curated knowledge base with new DBMS technologies and features. It could be imagined that the DSS implementation is used as a discussion platform that highlights conflicts and priorities to emphasize these and lead the decision process. Probing deeper, the decision model presented in this paper also provides a foundation for future work in technology selection problems. We intend to build trustworthy decision models to address*software architectural pattern*, *could service provider*, and *blockchain platform selection* as our (near) future work.

[5]https://dss-mcdm.com/

CHAPTER 3

Cloud Service Providers

Cloud computing enables software producing organizations to replace in-house IT infrastructure and provide them with scalable and flexible computing and flexible low cost. As cloud vendors and services on offer increase rapidly, cloud service provider selection is becoming a significant challenge for businesses. Cloud service providers and their offered services are characterized using multiple criteria, such as their popularity, geographic location, and deployment model, so it is essential to have a reliable method to select desirable cloud vendors based on decision-makers' requirements. In this study, we present a decision support system that supports decision-makers in choosing the most suitable Infrastructure-as-a-Service cloud providers. The case studies and experts confirm that the approach increases insight into the selection process, provides a richer prioritized option list than if they had done their research independently, and reduces the time and cost of the decision-making process.

keywords- multi-criteria decision-making, decision support system, knowledge management, cloud service provider selection, infrastructure-as-a-service

This chapter is based on the following publication:

Farshidi, S., S. Jansen, R. De Jong & S. Brinkkemper (2018a), "A decision support system for cloud service provider selection problems in software producing organizations", in: *2018 IEEE 20th Conference on Business Informatics (CBI)*, vol. 1, IEEE, pp. 139–148.

3.1 Introduction

Nowadays, cloud computing is influencing the IT landscape and becoming a significant economic factor for software producing organizations. Cloud computing is a fast-growing technology in a non-transparent market with diverse vendors, each of them having their specific services and deployment models. Typically, the service portfolios are heterogeneous and combined with complicated service features and pricing models. The challenge consists of evaluating and selecting the most suitable Infrastructure-as-a-Service Cloud Providers for software producing organizations according to their preferences and requirements.

The selection process is complicated because many factors, such as security and cost, have to be considered. In this study, the Infrastructure-as-a-Service Cloud Provider, in short, Cloud Service Provider (CSP) selection process is modeled as a multi-criteria decision-making (MCDM) problem that deals with the evaluation of a set of alternatives, and taking into account a set of decision criteria (Triantaphyllou et al., 1998).

In most cases, a unique optimal solution for an MCDM problem does not exist, and it is necessary to use the preferences of a decision-maker to differentiate between and prioritize solutions (Majumder, 2015). In recent years researchers introduced a considerable variety of techniques, methods, and tools to address MCDM problems. The majority of MCDM approaches in the literature use pairwise comparison techniques to calculate the weight of each decision criterion based on decision-makers' opinion. Pairwise comparison is a time-consuming process that gets more complex as the number of criteria increases(Saaty & Vargas, 2006). Moreover, most MCDM methods are not scalable, so in the case of modifying the list of alternatives or criteria, the whole process of evaluation has to be repeated. Traditional methods are costly and applicable to a small number of criteria and alternatives. A reusable, evolvable, and expandable decision-making approach is needed to make the right decision based on the decision-makers' requirements and preferences.

This study introduces a Decision Support System (DSS) to help decision-makers with MCDM problems, such as CSP selection. The DSS is a tool that can be used over the full life-cycle and can co-evolve its advice based on evolving requirements. The DSS applies the six-step decision-making process (Majumder, 2015) to build maintainable and evolvable decision models for MCDM problems, making the knowledge acquisition more reliable and trustful. In our previous work, we built a decision model for database technology selection problem (Farshidi et al., 2018c), then conducted three case studies to evaluate the DSS. The final results showed that the DSS performed well to address the database selection problem for the software-producing organizations. The novelty of the DSS lies in utilizing the MoSCoW prioritization technique (*MoSCoW*) (DSDM consortium and others, 2014) to assess criteria weights and reduce uncertainty, in introducing assessment models to measure the values of non-boolean criteria, and in using ISO/IEC quality aspects to indicate the relationship among criteria according to domain experts' knowledge.

This paper is structured as follows. Section 3.2 describes the design science method followed, and the exploratory theory-testing case studies that have been performed. Section 3.3 gives a window into the literature of software technology selection and the traditional approaches to solving decision-making problems such as ours. Section 3.4 outlines the details of the proposed decision support system and emphasizes the usage of novel techniques such as ISO qualities and the *MoSCoW*. Section 3.5 illustrates an application of the DSS to address the CSP selection problem, using four case studies to evaluate and emphasize the significance of the approach. Afterward, section 3.6 interprets the results of the case studies according to expert interviews and opinions. Section 3.7 highlights and overcomes barriers, such as motivational and cognitive biases, to the knowledge acquisition and decision-making process. Finally, section 3.8 summarizes the proposed approach, defends its novelty, and offers directions for future studies.

3.2 Research Method

Software-producing organizations typically are not knowledgeable in the problem domain, which is finding the most suitable CSPs for their businesses based on their requirements and priorities. The knowledge regarding the problem domain does not make any difference in the selection process, because the right selection requires regular studying and tracking available technologies and vendors in the market.

The selection process can be modeled as an MCDM problem that deals with structuring, planning, and solving the problem concerning a set of criteria: 1) Identifying the objective, 2) Selection of the features, 3) Selection of the alternatives, 4) Selection of the weighing method, 5) Applying the method of aggregation, 6) Decision making based on the aggregation results.

Knowledge acquisition and keeping the acquired knowledge up-to-date are time-consuming and costly processes for Software-producing organizations. To support these organizations, we propose a DSS, created using design science, based on the six-step decision-making process. The DSS has the goal of finding suitable alternatives that support a set of domain feature requirements.

The traditional design science cycle is followed, and the DSS is infused with expert knowledge gathered through three interviews. Twelve experts (three DSS experts, six cloud consultants, and three cloud architects) participated in this research to evaluate the DSS in interviews that lasted between 45 and 90 minutes. The domain experts were pragmatically selected according to their expertise and experience that they mentioned in their professional profile. Each of the interview series followed a semi-structured interview protocol. Data collected during one interview would typically be propagated to the next to build and validate the knowledge base incrementally. The knowledge base was sent to the interview participants afterward for final confirmation.

Secondly, the efficiency and usefulness of the DSS are evaluated through four exploratory theory-testing case studies. The unit of analysis is a unique CSP selection for a Software Producing Organization. We performed four such case studies at four software producing organizations to evaluate the DSS. The case studies typically lasted one day and consisted of (1) defining the domain feature requirements, (2) prioritizing them, and (3) comparing the DSS feasible solutions with their solutions.

3.3 Related Work

The proposed DSS applies the six-step decision-making process (Majumder, 2015) to build decision models for MCDM problems and distinguishes itself from the currently existing DSSs in the following ways: 1) the DSS utilizes the *MoSCoW* (DSDM consortium and others, 2014) to assess criteria weights and reduce uncertainty, 2) employs assessment models to measure the values of non-boolean criteria, and 3) uses the ISO/IEC quality aspects to indicate the relationship among criteria according to domain experts' knowledge.

Snowballing was employed as the principal method to investigate the existing literature related to the techniques which address MCDM problems for Software-producing organizations. Some MCDM methods can be listed as follows: The *Analytic Hierarchy Process (AHP)* is a structured method for organizing and analyzing MCDM problems. This method has been extensively applied and combined with other techniques to solve MCDM problems. The *Technique for Order Preference by Similarity to Ideal Solution (TOPSIS)* suggests that the selected alternative should have the shortest distance from an ideal solution and the farthest distance from the negative ideal solution.

The *Fuzzy Delphi Method (FDM)* is a more advanced version of the Delphi Method in that it utilizes triangulation statistics to determine the distance between the levels of consensus within the expert panel. The *FAHP* and *FTOPSIS* are the combinations of Fuzzy logic with the AHP and TOPSIS methods. The *FMCDM* assesses the ratings of alternatives versus criteria and the importance weights of criteria based on semantic values represented by fuzzy numbers.

Table 3.1 illustrates selected MCDM approaches from literature. The majority of the techniques in literature use pairwise comparison as the main method to assess the weight of criteria. For a problem with n number of criteria $\frac{n(n-1)}{2}$ number of comparison is needed (Saaty, 1990). It means that the pairwise comparison is a time-consuming process, and gets more complicated as the number of criteria increases. Some of the methods, such as AHP and FAHP, are not scalable, so in the case of modifying the list of alternatives or criteria, the whole process of evaluation should be conducted repeatedly. Therefore, these methods are costly and applicable to a small number of criteria and alternatives. The majority of the MCMD techniques in literature define domain-specific quality attributes to evaluate the alternatives. Such studies are mainly appropriate for specific case studies. Furthermore, the results of these MCDM approaches are valid for a specified period, so technology advances and new service offerings will be out-of-date.

Franch & Carvallo (2003) introduced a six-step method to solve the Commercial-Off-The-Shelf selection problem. The six-step method considers the ISO/IEC 9126-1 standard as for quality attributes and decomposes it into the domain features of the Commercial-Off-The-Shelf packages. Moreover, decision-makers should define specific metrics for each domain feature to assign a value to it. Finally, the results of considered Commercial-Off-The-Shelf packages will be compared. Becker et al. (2013) present a multi-criteria decision support system (MCDSS) for software component selection. The MCDSS evaluates a total of 51 Commercial-Off-The-Shelf components against a total of 631 decision criteria. The authors specified metrics, such as the

Table 3.1: This table compares selected MCDM methods from literature to address technology selection problems. The second column (Problem domain) points out the problem domain. The third column (MCDM) denotes the MCDM approach. The fourth column (Pairwise Comparison) indicates whether the approach applies pairwise comparison as a weight calculation method or not. The fourth column (Quality Attributes) determines the type of quality attributes. The seventh and eighth columns (Criteria and Alternatives) signify the number of criteria and alternatives that were considered in the problem domain.

Author(s)	**Problem domain**	**MCDM**	**Pairwise Comparison**	**Quality Attributes**	**Criteria**	**Alternatives**
This paper	Cloud service provider selection	DSS	NO	ISO/IEC 25010 EX. ISO/IEC 9126	300	40
Liu et al. (2016)	Cloud vendor selection	TOPSIS FDM	NO	Domain specific	4	4
Repschlaeger et al. (2014)	Software-as-a-Service product selection	AHP	YES	Domain specific	57	3
Garg et al. (2011)	Cloud service ranking	AHP	YES	ISO/IEC SMI	29	3
Halabi & Bellaiche (2017)	Cloud security service selection	AHP	YES	Domain specific	16	5
Lee & Seo (2016)	Cloud service selection	FAHP FDM	YES	Domain specific	18	5
Godse & Mulik (2009)	Software-as-a-Service product seletion	AHP	YES	Domain specific	21	4
Rouhani & Ravasan (2015)	ERP software	FMCDM	YES	Domain specific	23	4
Rodriguez et al. (2017)	Risk management approach	FAHP	YES	Domain specific	5	5
Büyüközkan & Güleryüz (2016)	Product development partner	FAHP FTOPSIS	YES	Domain specific	16	6
Oztaysi (2014)	Content Management System selection	AHP TOPSIS	YES	Domain specific	7	4
Becker et al. (2013)	Commercial Off-The-Shelf selection	DSS	NO	ISO/IEC 25010 Domain specific	631	51
Farshidi et al. (2018c)	Database technology selection	DSS	NO	ISO/IEC 25010 EX. ISO/IEC 9126	307	73

key decision factors and *efficient criteria sets*, for the quantitative evaluation of decision criteria and sets of criteria, and illustrated their application to a set of real-world decision cases.

One of the weaknesses of the six-step method (Franch & Carvallo, 2003) is that when the number of alternatives and domain features is high, measuring the qualities of domain features for each alternative is not possible a very time-consuming process. Furthermore, the assigned values for the domain features will be changed by technological advances. The proposed DSS is superior to the six-step method because it is an evolvable and expandable model-based approach that splits down the decision-making process into four maintainable phases (Section 3.4). The DSS and MCDSS both provide a substantial number set of criteria to support decision-makers. Furthermore, they use the ISO/IEC 25010 as a standard set of quality attributes. The main difference between the DSS and MCDSS is their weighting methods. We built a decision model for database technology selection problem (Farshidi et al., 2018c), then conducted three case studies to evaluate the DSS. The final results showed that the DSS performed well to address the database selection problem for the software-producing organizations. The DSS utilizes the *MoSCoW* to assess the importance of criteria and reduce the uncertainty. Moreover, it introduces assessment models to measure the values of non-boolean criteria, such as the cost and popularity of the alternatives.

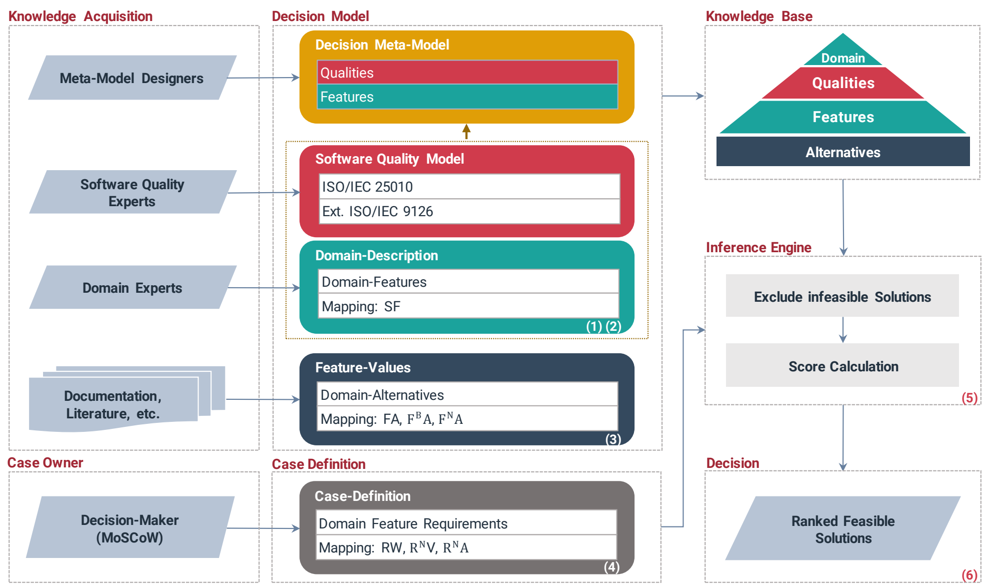
3.4 Multi-Criteria Decision-Making

The fundamental components of a typical DSS (Sage, 1991) are the Database management system, the Model-Base management system, and the Dialog Generation management system. The *Database management system* is a set of domain features related to an MCDM problem. The *Model-Base management system* is a collection of rules, heuristics, and knowledge related to the MCDM problem. The *Dialog Generation management system* is a user interface to interact with decision-makers.

The Inference Engine of a standard DSS infers solutions and does not rely on knowledge-based facts and rules, so it works independently from the other components. The Inference Engine receives domain feature requirements and their priorities according to *MoSCoW* from the *Dialog Generation management system* as its input. Next, it finds the most relevant rules from a collection of models in the *Model-Base management system*. Then, the Inference Engine, by using facts about the *DataBase management system*, deduces decisions. Eventually, it sends ranked feasible solutions to the *Dialog Generation management system*. The DSS[1] comprises the standard DSS components. The proposed DSS (Farshidi et al., 2018b) applies the six-step decision-making process (Majumder, 2015) to build decision models for MCDM problems. Furthermore, it makes knowledge acquisition more reliable and trustworthy. A decision model defines a decision structure to solve a specific MCDM problem. Figure 3.1 depicts the structure of the DSS.

[1]We implemented an online Decision Model Studio (https://dss-mcdm.com/) to build decision models for MCDM problems in Software-producing organizations.

Figure 3.1: A model-based decision support system for MCDM problems.



3.4.1 Decision Model

Knowledge acquisition is the process of extracting, structuring, and organizing knowledge from different sources of knowledge, including human experts, documentation, and literature. This process applies to define knowledge base facts and rules. This section elaborates on the knowledge acquisition process, the primary sources of knowledge, and constituent parts of a decision model based on the six-step decision-making process for building a decision model to address an MCDM problem.

Decision Meta-Model

The *Decision Meta-Model* defines the base structure (abstraction) of a decision model in the knowledge base. The *Decision Meta-Model* includes two primary sets (*Qualities* and *Features*). The set *Qualities*, denoted by Q, is a set that keeps software quality attributes, and the set *Features*, denoted by F, is a set that retains domain features of an MCDM problem.

Software Quality-Model

The *Software Quality-Model* determines the software quality attributes (Q), and defines relationships, based on a hierarchical structure, among elements of the set Q, thus, the Q is a nested set of quality attributes. The DSS utilizes the *ISO/IEC 25010* standard (ISO, 2011) and *extended ISO/IEC 9126* standard (Carvallo & Franch, 2006) in order to define the set Q. These quality standards are domain-independent software quality models and provide reference points by defining a top-down standard quality model for software systems. The ISO/IEC standard quality models have two hierarchical levels, being *Characteristics* and *Sub-characteristics*. The mappings between these levels are defined as follows. Suppose C and S are the sets of the *Characteristics* and *Sub-characteristics* of the ISO/IEC quality models. Then, the mapping between these two sets, $CS : C \times S \rightarrow \{0, 1\}$, are defined according to the ISO/IEC quality models. So that, $CS(c, s)$, where $c \in C$ and $s \in S$, is equal to one when c is connected to s, otherwise it is equal to zero. The elements of the *Software Quality-Model* apply to classify domain features (F) of an MCDM problem based on their impact on *Sub-characteristics* of the ISO/IEC quality models (SF); Moreover, they use to calculate the impact factor of domain feature requirements based on decision-makers' preferences and domain experts' knowledge (equation 3.3).

Domain Description

The *Domain Description* defines the first and second steps, denoted by *Identifying the objective* and *Selection of the features*, of the decision-making process. In other words, the *Domain Description* specifies the domain features (F) of an MCDM problem. Each domain feature has a data type, which could be *Boolean*, denoted by F^B, or *Numeric*, denoted by F^N, where $F^B \cap F^N = \varnothing$ and $F = F^B \cup F^N$. For example, the data type of a domain feature like the *popularity* of alternatives is *Numeric*. The mapping between sets Q and F is based on the domain experts' knowledge. As aforementioned, the DSS uses the ISO/IEC quality models to define the set Q, which is a nested set. The last level of the hierarchal structure in the set Q is the *Sub-characteristics* of the ISO/IEC quality models, denoted by S. Therefore, the mapping, $SF : S \times F \rightarrow \{0, 1\}$, defines the relationship between the *Sub-characteristics* (S) and domain features (F). So that, $SF(s, f)$, where $s \in S$ and $f \in F$, is equal to one when s is connected to f, otherwise it is equal to zero. Domain features could organize into conceptual hierarchical structures. So that, generic domain features split down into more specific domain features (sub-features). For instance, in the CSP selection problem, *Automation and orchestration* is a generic domain feature, moreover *Kubernetes*, *Docker Swarm*, and *Ansible* are considered as its sub-features. When an alternative supports a generic domain feature, it means that it supports at least one of the sub-features of the generic feature. The domain features were identified through interviews with the domain experts. The main aim of the interviews was to establish the prominent domain features and identify which domain features could be left out. Note that conceptual hierarchical structures of domain features help decision-makers prioritize the domain features based on their expertise and knowledge of the project requirements. Again, these hierarchies were established through domain expert interviews. Moreover, no difference exists between generic domain features and sub-features from the decision model perspective. Thus, the set F contains generic domain features and sub-features, and SF maps them to the set S.

Feature-Values

The *Feature-Values* defines the third step, indicated by *Selection of the alternatives*, of the decision-making process. The *Feature-Values* determines a set of alternatives, denoted by A, and maps them to the domain features set (F). The main source of knowledge in this phase is the documentation of alternatives, literature studies, social networks, domain experts, etc. The data type of domain features could be either *Boolean* or *Numeric*. The mapping, $F^B A : F^B \times A \rightarrow \{0, 1\}$, maps the boolean domain features (F^B), and the mapping, $F^N A : F^N \times A \rightarrow \mathbf{R}_{\geq 0}$, maps the numeric domain features (F^N) to alternatives (A). So that, $F^BA(f,a)$, where $f \in F^B$ and $a \in A$, is equal to one when f is connected to a (boolean domain feature f is supported by alternative a), otherwise it is equal to zero. Moreover, $F^NA(f,a)$, where $f \in F^N$ and $a \in A$, specifies the value of domain feature f regarding alternative a. In other words, the mapping F^NA assigns the values of assessment models to numeric domain features, such as *cost* and *popularity*.

3.4.2 Case Definition

The *Case Definition* defines the fourth step, denoted by *Selection of the weighing methods to indicate the importance of the features*, of the decision-making process. The DSS utilizes *MoSCoW* to define decision-makers' domain feature requirements and assess the importance of required domain features. Note that domain feature requirements set (R) is a subset of domain features, where $R \subseteq F$ and $R^B = R \cap F^B$ and $R^N = R \cap F^N$. Suppose $W_{MoSCoW} = \{w_{Must}, w_{Should}, w_{Could}, w_{Won't}\}$ is the set of priority weights according to the definition of the *MoSCoW* (DSDM consortium and others, 2014), where $\forall w \in W_{MoSCoW}; 1 \geq w \geq 0$. In other words, a case definition, based on a decision maker's preferences (*MoSCoW*), is a way to select domain feature requirements and assign priorities to them, $RW : R \to W_{MoSCoW}$.

The importance of a domain feature with *Must Have* priority must be greater than all domain features with *Should Have* priority, where $\sum_{\forall r \in R; RW(r) = w_{Should}} RW(r) < w_{Must}$. Furthermore, the importance of a domain feature with *Should Have* priority must be greater than all domain features with *Could Have* priority, where $\sum_{\forall r \in R; RW(r) = w_{Could}} RW(r) < w_{Should}$.

Decision-makers specify desirable values for numeric domain feature requirements, $R^NV : R^N \to \mathbf{R}_{\geq 0}$. For example, a decision-maker could be interested in prioritizing the CSPs with Total Cost of Ownership (TCO) less than 500 USD as more important than others. Therefore, the TCO less than 500 USD could be considered as a *should have* domain feature. Consequently, a mapping, $R^NA : R^N \times A \times \mathbf{R}_{\geq 0} \to \{0,1\}$, is considered to define these types of numerical criteria by decision-makers. The *Case Definition* receives mappings RW and R^NV as its input from the user interface of the DSS. Indeed, a decision-maker is the main source of knowledge in this phase. Domain feature requirements with *Must Have* or *Won't Have* priorities act as hard constraints (H) and domain feature requirements with *Should Have* and *Could Have* priorities act as soft constraints.

3.4.3 Inference Engine

Each decision model defines a decision structure for an MCDM problem systematically. Moreover, the mappings define rules and facts. Therefore, the Knowledge Base is a collection of decision-models, which are groups of rules and facts. The *Inference Engine* defines the fifth step, indicated by *Applying the method of aggregation*, of the decision-making process. The *Inference Engine* ranks the alternatives based on their calculated scores. The score calculation process begins with computing the weight of each aspect of a decision model. As mentioned prior the relationship between aspects are defined based on the mappings (CS, SF, F^BA, F^NA, RW, R^NV, and R^NA). The summary of the sets and mapping of a decision model and a case definition for an

Table 3.2: The summary of the sets and mapping of a decision model and a case definition for an MCDM problem.

Notation		Definition	Description	Source of Knowledge
C		-	The set of *Characteristics*.	ISO/IEC standards
S		-	The set of *Sub-characteristics*.	ISO/IEC standards
A		-	The set of alternatives.	Documentation
F	$=$	$F^B \cup F^N$	The set of domain features.	Domain Experts
F^B		-	The set of boolean domain features.	Domain Experts
F^N		-	The set of numeric domain features.	Domain Experts
R	$=$	$R \subseteq F$	The set of domain feature requirements.	Decision-Makers
R^B	$=$	$R \cap F^B$	The set of boolean domain feature requirements.	Decision-Makers
R^N	$=$	$R \cap F^N$	The set of numeric domain feature requirements.	Decision-Makers
W_{MoSCoW}	$=$	$\{w_{Must}, w_{Should}, w_{Could}, w_{Won't}\}$	The set of priority weights.	MoSCoW Priorities
CS	$:$	$C \times S \rightarrow \{0,1\}$	The mapping between the sets C and S.	ISO/IEC standards
SF	$:$	$S \times F \rightarrow \{0,1\}$	The mapping between the sets S and F.	Domain Experts
$F^B A$	$:$	$F^B \times A \rightarrow \{0,1\}$	The mapping between the sets F^B and A.	Documentation
$F^N A$	$:$	$F^N \times A \rightarrow \mathbf{R}_{\geq 0}$	The mapping between the sets F^N and A.	Documentation
RW	$:$	$R \rightarrow W_{MoSCoW}$	The mapping between the sets R and W_{MoSCoW}.	Decision-Makers
$R^N V$	$:$	$R^N \rightarrow \mathbf{R}_{\geq 0}$	Desirable values for the set R^N.	Decision-Makers
$R^N A$	$:$	$R^N \times A \times \mathbf{R}_{\geq 0} \rightarrow \{0,1\}$	Numerical criteria.	Decision-Makers
H	$=$	$\bigcup_{\substack{\forall r \in R \\ RW(r)=w_{Must} \vee RW(r)=w_{Won't}}} r$	Hard constraints	Decision-Makers

MCDM problem is shown in table 3.2. Note that the weight of domain feature requirements ($RW(r)$, where $r \in R$) assign by the decision-maker via the *MoSCoW*. Moreover, the weights *Sub-characteristics* (W_s) and *Characteristics* (W_c) of the ISO/IEC quality models in the set Q are the sum of the weights of their children.

$$W_{s \in S} = \sum_{\forall r \in R; r \notin H} SF(s,r).RW(r) \tag{3.1}$$

$$W_{c \in C} = \sum_{\forall s \in S} CS(c,s).\bar{W_s} \tag{3.2}$$

$\bar{W_s}$ is the normalized to unity weights of the *Sub-characteristics*. Next, the impact factor, denotes by I_r, of each domain feature requirement ($r \in R$) is equal to the sum of products of its parents' weights plus the weight of the domain feature requirement. The reason behind this impact factor calculation is finding the importance of domain feature requirements based on the decision-maker preferences and the relationship among domain feature requirements according to domain experts' knowledge. Moreover, it assures that the *MoSCoW* priorities of the domain feature requirements never change. In other words, if the decision-maker assigned the *Could Have* priority to a domain feature requirement, its importance would not become greater than a domain feature requirement with the *Should Have* priority.

$$I_{r \in R} = RW(r) + \sum_{\forall s \in S} SF(s,r).\bar{W_s} \sum_{\forall c \in C} CS(c,s).\bar{W_c} \tag{3.3}$$

$\bar{W_c}$ is the normalized to unity weights of the *Characteristics*. A feasible alternative a (feasible solution) must support all domain feature requirements with *Must Have* priorities, and must not support all domain feature requirements with *Won't Have* priorities. Equation 3.4 through mappings F^BA, R^NA, and R^NV indicate whether all boolean domain feature requirements ($H \cap R^B$) and numeric domain feature requirements ($H \cap R^N$) with *Must Have*and *Won't Have* priorities (hard constraints) are supported by the alternative a or not. Note, hard constraint numeric domain features contain numerical criteria which indicate by decision-makers. For example, a decision-maker could be interested in considering only CSPs which their TCO values are less than 500 USD, so $TCO < 500$ is a numeric domain feature with *Must Have* priority.

$$\begin{gathered} Sum = \\ \sum_{\forall r \in (H \cap R^B)} F^BA(r,a) + \sum_{\forall r \in (H \cap R^N)} R^NA(r,a,R^NV(r)) \\ Feasible_{a \in A} = \begin{cases} 1, & \text{if } Sum = |H| \\ 0, & \text{otherwise.} \end{cases} \end{gathered} \tag{3.4}$$

The score calculation process (equation 3.5) involves the sum of products of impact factors of domain feature requirements with *Should Have* and *Could Have* priorities.

$$Score_{a \in A} = Feasible_a . \left(1 + \sum_{\forall r \in (R^B \setminus H)} \bar{I}_r . F^B A(r,a) + \sum_{\forall r \in (R^N \setminus H)} \bar{I}_r . R^N A(r,a,R^N V(r))\right) \tag{3.5}$$

If in the score calculation process $RW(r)$ is equal to w_{Could} then $\bar{I}_r$ is normalized to $[w_{Could}, w_{Should})$, otherwise, $\bar{I}_r$ is normalized to $[w_{Should}, 1)$.

Equation 3.5 and equation 3.6 define the sixth step, denoted by *Decision making based on the aggregation results*, of the decision-making process. Note, the scores of feasible solutions are more than zero.

$$Solutions = A \setminus \bigcup_{\substack{\forall a \in A \\ Score_a = 0}} a \tag{3.6}$$

The final ranked feasible solutions will be given as the result of the DSS by sorting the feasible solutions in descending order of their scores.

3.5 Cloud Service Provider Selection

The main building blocks of a decision model are *Decision Meta-Model*, *Software Quality Model*, *Domain Description*, and *Feature-Values*. The *Decision Meta-Model* defines the base structure of a decision model in the knowledge base, and it has two sets, namely *Qualities* and *Features*. A decision model utilizes the *ISO/IEC 25010* standard and *extended ISO/IEC 9126* standard in order to define the set *Qualities*. The *Decision Meta-Model* and *Software Quality Model* are immutable for decision models based on the DSS approach. However, the *Domain Description* and *Feature-Values* should be define to structure a decision model for an MCDM problem.

This section presents a decision model based on the DSS approach to address the CSP selection problem. Moreover, four case studies have been conducted to evaluate the efficiency and effectiveness of the DSS to address CSP selection problem for software producing organizations.

3.5.1 Domain Description for CSP selection

A list of domain features (F) of the domain of interest should be specified in a decision model. Domain experts are the primary source of knowledge to identify domain features. In order to define the domain of CSP selection problem more than 250 features[2] (such as Automation and orchestration, Application Server, Certifications/Attestations, and Cost) have been collected according to domain experts' suggestions. The sub-characteristics of the *Software Quality-Model* provides a general

[2]The entire list of the domain features and supportability of considered cloud service providers are available and accessible on the "Cloud Service Provider Selection" website (https://dss-mcdm.com/)

view of the software quality model. The decision model decomposes abstract concepts into more concrete ones, the domain features. Domain features have to define precisely to clarify the underlying quality concepts that they represent and to link them with the appropriate sub-characteristics. Some domain features are related to more than one sub-characteristic. For example, *Automation and orchestration* as a CSP feature might include Availability, Reusability, and Installability. The DD does not enforce a domain feature to present in a single sub-characteristic; Domain features can be part of many quality aspects. The relationship between sets S and F is defined by the mapping SF according to domain experts' opinion. In this study, CSP features and the mapping SF defined by nine domain experts, including six cloud consultants and three cloud architects in the Netherlands.

Table 3.3: The reference configurations for calculating the Total Cost of Ownership of CSPs. Each reference configuration is indicated by its number of CPU cores, amount of RAM (GB), and SDD capacity (GB).

Server Configurations	CPU(Cores)	Memory	SSD
Basic Server	4	8 GB	100 GB
Intermediate Server	8	32 GB	100 GB
Memory-intensive Server	16	512 GB	200 GB
CPU-intensive Server	24	64 GB	500 GB

3.5.2 Feature-Values for CSP Selection

The decision model requires a list of CSP alternatives. Well-known CSPs, websites, related forums, and domain experts are the primary source of knowledge to specify the alternatives. In this study, 40 infrastructure-as-a-service CSPs (Leaseweb, Google Cloud, etc.) as the alternatives have been considered. The list of CSP alternatives collected from recent reports of the *Gartner*, *Glassdoor*, and *Forrester* websites.

Next, the supportability of boolean domain features (F^B) by the CSP alternatives (A) should be investigated. The relationship between sets F^B and A defined by the mapping F^BA based on the documentation and websites of the considered CSPs. One of the principal problems is the lack of standard terminology among documentation of CSPs. Different CSPs refer to the same concept (cloud service) by different names, or even worse, the same name might stand for different concepts in different CSPs. Discovering conflicts in the *Feature-Values* is essential to prevent semantic mismatches throughout the CSP selection process.

CSPs tend to provide a partial view of their cloud services. They emphasize their services' benefits without mentioning weaknesses, or they provide only part of the truth. Some non-commercial articles compare CSPs and features but are often based on the evaluators' limited knowledge and their particular tastes (Franch & Carvallo, 2003). The next step in building a decision model for the CSP selection problem is defining assessment models for each numeric domain features, such as cost and popularity. After defining suitable assessment models for numeric domain features (F^N), the mapping F^NA maps them to the corresponding CSP alternatives (A). For example, *Total Cost of Ownership*, *Popularity in the market*, *Company Maturity*, and and *Innovation* are non-boolean domain features in the decision model of the CSP selection problem.

Non-boolean domain features could be grouped into a number of categories (ranges) based on their values. Categories facilitate the usage of relational criteria. For example, a decision-maker could be interested in prioritizing the CSPs with Total Cost of Ownership values less than \$500 USD as more critical than others. Therefore, the Total Cost of Ownership values of less than \$500 USD could be considered as a *should have* domain feature.

Total Cost of Ownership

The cost of CSPs varies widely, and many factors and options should be considered. The Total Cost of Ownership (TCO) sometimes appear confusing, especially when it comes to well-known service providers (such as Oracle, Microsoft, Google), where a large variety of parameters (such as Operating System Licenses, Storage per GB/TB prices) for calculating the CSP costs are available. Thus, to get a rough estimate of the TCO of CSPs, four reference configurations for three cloud deployment models and server types (including Physical private cloud, virtual private cloud, and Virtual public cloud) are provided. Table 3.3 demonstrates the considered reference configurations.

The TCO value of each CSP alternative was asked directly from the CSP or calculated via the offered TCO calculator on the website of the CSP. Note that TCO values should be computed in the same currency (e.g., USD) and period (e.g., monthly) to provide a correct comparison. Many options, offers, and add-ons were not included in the TCO calculations because they were CSP specific. The TCO is a domain feature of CSP selection that attempts to clear the fog somewhat regarding CSP prices. However, estimation of TCO values cannot provide a full and precise insight into the complex pricing models that CSPs use.

Popularity in the Market

This non-boolean feature is one of the assessment models in the CSP selection problem. It ranks CSPs based on their popularity in the market by using the following parameters: a. The number of mentions of CSPs on websites, b. The frequency of technical discussions about CSPs on websites, c. The number of job offers on the leading job search engines, and d. Relevance in social networks.

Company Maturity

This assessment model measures the company maturity of CSPs based on three main factors, including company size (number of employees), company revenue, and date of establishment. In other words, a mature CSP company is well-established in the market, with well-known services and loyal customer following with average growth. Mature companies are categorized according to the business stage it is currently in. We considered a three-stage maturity level (high, middle, and low) for the CSPs.

Innovation

Innovation is often viewed as the application of better solutions that meet new requirements, unarticulated needs, or existing market needs (Maranville, 1992). This is accomplished through more effective products, processes, services, technologies, or business models that are readily available to the market. This assessment model measures the innovation of CSPs based on supportability of following factors: a. Internet of Things Cloud, b. Big Data Analytics, c. Business Intelligence, d. Enterprise reporting, e. Dynamically scale to meet capacity demands, and f. Multiple data centers.

Table 3.4: A part of AFAS, Negometrix, KPMG, and Health Diaries domain feature requirements based on the MoSCoW. Note that the numbers in the table indicate the number of domain feature requirements in a particular MoSCoW priority for each case study. For example, AFAS has nine domain feature requirements with *Could Have* priority.

MoSCoW	**AFAS**		**Negometrix**		**KPMG**		**Health Diaries**	
Must Have	Service Fabric, Disaster recovery, etc.	7	.Net, ISO 27001, etc.	21	Node.js, SOC2, etc.	22	Java, MySQL, etc.	13
Should Have	High Company Maturity, Memory-intensive server, etc.	9	Encryption, Packet Filtering, etc.	11	DevOps, GitLab, etc.	12	HL7, Auto Scaling, etc.	4
Could Have	Kubernetes, Windows Server Container, etc.	9	Free private transfer, Network IDS, etc.	17	Automation and orchestration	1	Big data analytics, Database Backup-as-a-service	2

3.5.3 Empirical Evidence: The Case Studies

Four case studies in the context of four software producing organizations have been conducted to evaluate and signify the usefulness and efficiency of the DSS. The case study companies considered a number of feasible CSPs for their organizations through multiple internal expert meetings and extensive investigation into CSP alternations before participating in this research.

AFAS Software - AFAS Software is an ERP vendor in the Netherlands with approximately 350 employees. One of AFAS' current challenges is validating whether they have chosen the right CSP for the new version of their product.

KPMG - KPMG is a professional service company with more than 189,000 employees and located in the Netherlands. KPMG has three lines of services: financial audit, tax, and advisory. KPMG participated in this research to select a well qualified CSP for one of its customers.

Health Diaries - Health Diaries is a small Software Producing Organization with ten employees and located in the Netherlands. Health Diaries is developing digital healthcare diaries based on expertise from healthcare professionals and medical science, which makes them useful for healthcare institutions. Health Diaries experts are interested in evaluating different CSPs in the market and selecting the suitable one that fulfills their requirements and priorities.

Negometrix - Negometrix produces procurement software. Its customers are one-third government, one-third non-profits, and one-third commercial organizations. Presently, the Negometrix product is being renewed and rebuilt using new Microsoft platforms, and this is a suitable time to rethink the CSP for the new version of the Negometrix product.

Table 3.4 demonstrates parts of the domain feature requirements of the case studies based on the MoSCoW priorities.

3.6 Results and Analysis

The feasible solutions of the DSS for the case studies are shown in Table 3.5. The KPMG domain feature requirements are mainly generic domain features, such as *Automation and orchestration* and *Auditing/Logging*, or standard features, which are supported by most of the CSP alternatives. Therefore, the DSS deduced eight feasible solutions for KPMG despite 22 domain feature requirements with *Must Have* priority (Hard constraints). Health Diaries domain feature requirements target specific CSPs, which support health care companies. Thus, the DSS suggested only four feasible solutions for Health Diaries. Negometrix domain feature requirements are mostly technical domain features. Moreover, Geo-locations of the data-centers are one of the feature requirements with *Must Have*. Consequently, the DSS inferred four feasible solutions for Negometrix. AFAS domain feature requirements are generic domain features. The number of hard constraints in the AFAS requirements is lower than the other case studies. As a result, the DSS recommended ten feasible solutions for AFAS.

The annual TCO was a *Should Have* domain feature requirement for AFAS and KPMG. Hence, the DSS did not exclude any alternatives based on their TCO values. Some of the feasible solutions proposed by the DSS were not on the shortlist of case

Table 3.5: The feasible solutions of the DSS for AFAS, Negometrix, KPMG, and Health Diaries based on their domain feature requirements and MoSCOW priorities. The column *CP (Case Participant) Shortlist* demonstrates which DSS feasible solutions already considered in the shortlist of case study participants based on their internal meetings and investigations. Moreover, the Columns *CP Rank* and *DSS score* of the table show the score calculation results of the DSS and the ranked shortlist of the feasible solutions based on the case study participants' opinions respectively.

Case Study	DSS Feasible solutions	CP Shortlist	DSS Score	CP Rank
AFAS	Microsoft Azure	✓	93.34	3
	IBM Cloud		92.41	-
	OVH		91.67	-
	DataPipe		86.12	-
	KPN (iS)	✓	84.81	4
	Google Cloud	✓	84.59	2
	Leaseweb	✓	83.05	1
	Interoute		77.22	-
	Amazon (AWS)		76.23	-
	1and1		75.15	-
Negometrix	Microsoft Azure	✓	99.86	1
	Leaseweb		99.69	-
	Google Cloud	✓	99.57	2
	KPN (iS)	✓	99.54	3
KPMG	Google Cloud	✓	100.00	3
	Rackspace		100.00	-
	Amazon (AWS)	✓	94.42	2
	Microsoft Azure	✓	94.42	1
	Fujitsu	✓	76.16	5
	Oracle Cloud		51.00	-
	IBM Cloud	✓	51.00	4
	Alibaba Cloud		32.75	-
Health Diaries	Microsoft Azure	✓	100.00	1
	Amazon (AWS)	✓	100.00	2
	Leaseweb	✓	82.24	3
	Fujitsu	✓	82.24	4

participants because they found that the annual TCO of these CSPs, including extra options, end up being much higher than the other feasible solutions. Also, the case participants stated that a lack of experience with the performance and Service-Level-Agreement of such CSPs is another reason for ignoring them. Columns *CP Rank* and *DSS score* of table 3.5 show the score calculation results of the DSS and the short ranked list of the feasible solutions based on the case participants' opinions respectively.

The case study participants confirm that the DSS provides practical solutions to help software producing organizations in their initial decisions for selecting CSPs. The DSS recommended the same solutions as the case participants suggested to their companies after extensive analysis and discussions. However, the DSS offers a short ranked list of feasible solutions; therefore software producing organizations should perform further investigations, such as performance testing and actual TCO calculation, to find the optimum CSPs for their software products. Twelve experts (three DSS experts, six cloud consultants, and three cloud architects) participated in this research to evaluate the DSS.

The consulted experts confirm that the DSS contains the main components of a standard DSS. Moreover, they asserted that the score calculation process in the *Inference Engine* of the DSS is not dependent on the knowledge-base facts and rules (i.e., the decision model). Therefore, if another replaces a decision model for an MCDM problem, the *Inference Engine* would not generate invalid solutions.

The experts believe that experience in using a specific technology provides invaluable knowledge when selecting suitable technology. Consequently, we recommend that our DSS is used in combination with benchmarks where applicable. Furthermore, the experts indicate that CSPs' supported domain features play a significant role in the CSP selection process. Specific CSPs support some domain features; for instance, *NEN 7510* is the standard for information security in health care. Also, the supported domain features are going to change due to technological advances. As such, the knowledge-base must be updated regularly. The experts state that their companies continuously improve and reevaluate their technologies, including the used CSPs.

The case study participants enter a limited set of domain feature requirements. We were surprised to find that the experts have a limited view of the technology's domain feature requirements. The case participants themselves were surprised to find what their primary concerns seem to be, especially when the opinions of different experts are combined. The fact that the DSS has led to discussions that determine decision-making for the technology illustrates that the DSS is a useful tool for software producing organizations and MCDM problems. More importantly, the case participants confirm that the updated and validated version of the DSS is useful and valuable in finding the shortlist of feasible solutions. Finally, it reduces the time and cost of the decision-making process.

3.7 Discussion

Software producing organizations have different perspectives on their domain feature requirements in different phases of the Software Development Life-Cycle. Decision-makers might want to consider generic domain features in the early phases of the life-cycle, whereas they are interested in more technical domain features as their development process matures. For instance, *Automation and orchestration* could be prioritized as a *Should Have* domain feature in the design phase, but in the implementation phase, one of its sub-features (more technical domain feature), e.g., *Service Fabric*, might be selected instead. Furthermore, domain features' priorities could be changed in different phases. Therefore, the DSS might come up with various solutions for a software producing organization in different phases of its software development life-cycle. As the participants' choices are stored in the DSS, it does not cost a significant amount of time to rerun the decision-making process.

Biases, such as motivational and cognitive (Montibeller & Winterfeldt, 2015), arise because of shortcuts or heuristics that decision-makers use to solve problems and perform tasks. The Hawthorne effect, which is the tendency for decision-makers to change their behavior when they are being observed, is a form of cognitive bias. The case study participants might have been more careful in the experimental setting than they would be in the real setting because they are being observed by scientists judging their selected domain feature requirements and priorities. Moreover, the Bandwagon effect, which is the tendency to do or believe things because many other decision-makers do or believe the same, is another form of cognitive bias. The Bandwagon effect typically shows up in group decisions. To mitigate the Hawthorne and Bandwagon effects, individual *and* group interviews have been conducted.

We define DSS success when it, in part, aligns with the CP's shortlist and when it provides new suggestions that are identified as being of interest to the CP. Using the CP experts' opinion as a measurement instrument is risky, as the CP may not have sufficient knowledge to make a valid judgment. We counter this risk by conducting more than one case study, by assuming that the CP expert is handling in its interest and applying the DSS to other problem domains, where we find similar results (Farshidi et al., 2018b; Farshidi et al., 2018c).

3.8 Conclusion

Finding a feasible solution for the Infrastructure-as-a-Service Cloud Provider selection problem based on decision-makers' priorities and requirements requires an in-depth investigation into the documentation of cloud vendors and extensive expert analysis. This study introduces a Decision Support System (DSS) to accelerate the process of finding the right Infrastructure-as-a-Service Cloud Provider for software producing organizations. The DSS comprises all of the fundamental components of a standard DSS. A decision model in the knowledge base of the DSS contains all facts and rules of an MCDM problem. In other words, a decision model defines a decision structure to solve a specific MCDM problem.

The novelty of the proposed DSS lies in utilizing the *MoSCoW* to assess criteria weights and reduce uncertainty, in introducing assessment models to measure the values of non-boolean criteria, and in using ISO/IEC quality aspects to indicate the relationship among criteria according to domain experts' knowledge. Our website[3] is up and running to keep the knowledge base of the DSS up-to-date and valid. We plan to create a community around the platform that will regularly update the curated knowledge base with new Infrastructure-as-a-Service Cloud Provider features.

Probing more in-depth, the decision model presented in this paper also provides a foundation for future work in MCDM problems. We intend to build trustworthy decision models to address *software architecture pattern* and *blockchain solution* selection problems as our (near) future work.

[3]https://dss-mcdm.com/

CHAPTER 4

Blockchain Platforms

Blockchain technology has received significant attention recently, as it offers a reliable decentralized infrastructure for all kinds of business transactions. Software-producing organizations are increasingly considering blockchain technology for inclusion in their software products. Selecting the best fitting blockchain platform requires the assessment of its functionality, adaptability, and compatibility with the existing software product. Novice software developers and architects are not experts in every domain, so they should either consult external experts or acquire knowledge themselves. The decision-making process gets more complicated as the number of decision-makers, alternatives, and criteria increases. Hence, a decision model is required to externalize and organize knowledge regarding the blockchain platform selection context.
Recently, we designed a decision support system to use such decision models to support decision-makers with their technology selection problems in software production. In this study, we introduce a decision model for the blockchain platform selection problem. The decision model has been evaluated through three real-world case studies at three software-producing organizations. The case-study participants asserted that the approach provides significantly more insight into the blockchain platform selection process, provides a richer prioritized option list than if they had done their research independently, and reduces the time and cost of the decision-making process.

keywords- blockchain platform selection; blockchain decision model; technology selection; multi-criteria decision making; decision support system;

This chapter is based on the following publication:

Farshidi, S., S. Jansen, S. España & J. Verkleij (2020c), "Decision support for blockchain platform selection: three industry case studies", *IEEE Transactions on Engineering Management*.

4.1 Introduction

Blockchain technology offers a reliable decentralized infrastructure, which means that it does not have to be controlled by one central authority for business applications. Blockchain technology can be employed as an application platform to build the underlying trust infrastructure of any distributed system. Since public blockchain platforms are open to the world, they can rapidly draw the attention of software development companies and communities to blockchain technology's strengths.

Software-producing organizations are increasingly considering distributed ledger and blockchain technology for inclusion into their software products. The selection process refers to the steps involved in choosing and evaluating the best fitting blockchain platforms for software-producing organizations according to their preferences and requirements. The selection process is complicated because many criteria, such as security, interoperability, consensus mechanisms, and platform transaction speed, have to be considered and fitted to the needs of the project at hand. Additionally, as a software product is typically a long-living system, such decisions determine the future of the product and the costs associated with its development.

Numerous public blockchain platforms have emerged recently and utilized in diverse business applications. For instance, Hyperledger[1] and Ethereum[2] offer public blockchain platforms. The fundamental difference between Hyperledger and Ethereum is the goal they are designed for. Hyperledger is an open-source development project that offers multiple blockchain platforms and supports the collaborative development of blockchain-based distributed ledgers. On the other hand, Ethereum is an open-source distributed public blockchain platform whose smart contracts enable decentralized applications to be implemented and deployed. As the number of blockchain platforms in the market is increasing rapidly, blockchain platform selection is becoming a significant challenge for software-producing organizations. Hence, knowledge regarding blockchain platforms has to be collected, organized, stored and quickly retrieved when it needs to be applied.

In literature, a variety of multi-criteria decision-making (MCDM) techniques have been introduced to address different technology selection problems for software-producing organizations. An MCDM problem deals with evaluating a set of alternatives and considers a set of decision criteria (Triantaphyllou et al., 1998). In our recent study (Farshidi et al., 2018a), we introduced a technology selection framework that is used to build decision models for MCDM problems and assist decision-makers at software-producing organizations with the decision-making process. Furthermore, we have instantiated the framework to build two decision models for the Database Management System (Farshidi et al., 2018c) and Cloud Service Provider (Farshidi et al., 2018a) selection problem. In this study, the blockchain platform selection process is modeled as an MCDM problem, and the technology selection framework is employed to build a decision model for this MCDM problem.

Recently, we designed and implemented a Decision Support System (DSS) (Farshidi et al., 2018b) for supporting decision-makers with their MCDM problems in software

[1]https://www.hyperledger.org
[2]https://www.ethereum.org

production. The DSS provides a decision model studio[3] for building decision models based on the technology selection framework. Moreover, such decision models can be uploaded to the knowledge base of the DSS to facilitate the decision-making process for software-producing organizations according to their requirements and preferences. The DSS provides a discussion and negotiation platform to enable decision-makers at software-producing organizations to make group decisions. Furthermore, the DSS can be used over the full life-cycle and can co-evolve its advice based on evolving requirements. During this research, we have built a decision model based on the technology selection framework for the blockchain platform selection problem, and then we have uploaded the decision model to the knowledge base of the DSS; finally, the outcomes of the DSS have employed in the case studies.

Please note that this study's proposed decision model contains reusable knowledge regarding potential blockchain platforms and features. Such knowledge can educate and support the decision-makers to understand: 1) which blockchain platforms are available at the moment, 2) the capabilities of the blockchain systems, and 3) which features are fulfilled by which blockchain platforms.

The rest of this study is structured as follows: Section 4.2 describes our research method, which is based on design science and exploratory theory-testing case studies. Section 4.3 positions the proposed approach in this study among the other blockchain platform selection techniques in the literature. Section 4.4 outlines a brief description of the DSS and the technology selection framework. This study has the following contributions:

- Section 4.4 introduces a decision model, in the form of reusable knowledge, for the blockchain platform selection problem based on the technology selection framework (Farshidi et al., 2018a).
- Section 4.5 describes the three conducted case studies that evaluate the effectiveness and usefulness of the approach to address the blockchain selection problem.
- Section 4.6 analyzes the final results of the DSS and compares the outcomes of the DSS with the case-study participants' shortlist of feasible blockchain platforms. The results show that the DSS recommended nearly the same solutions as the case-study participants suggested to their companies after extensive analysis and discussions, and does so more efficiently.

Section 4.7 highlights barriers to the knowledge acquisition and decision-making process, such as motivational and cognitive biases, and argues how we have minimized these threats to the validity of the results. Finally, section 4.8 summarizes the proposed approach, defends its novelty, and offers directions for future studies.

4.2 Research Approach

Rationality is the extent to which a decision-making process entails the compilation of information related to the domain of the problem and the degree of confidence upon analysis of this information in making the decision (Dean Jr & Sharfman, 1993). The decision-making process consolidates critical assessment of evidence and a struc-

[3]The decision model studio is available online on the DSS website: https://dss-mcdm.com

tured process that requires time and conscious effort (Fitzgerald et al., 2017). The decision-making process encourages decision-makers to establish relevant decision criteria, recognize a comprehensive collection of alternatives, and assess the alternatives accurately (Kaufmann et al., 2012). In other words, knowledge acquisition in the decision-making process is a time-consuming process in which the problem (decision context) should be interpreted accurately, and potential criteria and alternatives should be identified and compared precisely.

The decision-making process gets more complicated as the number of decision alternatives and criteria increases. Decision-makers at software-producing organizations are not experts in every domain, so they should either consult external experts or acquire knowledge themselves. In both cases, there is an investment of time (and eventually, money) that must be factored into the decision making process.

Recently, we introduced the technology selection framework (Farshidi et al., 2018a) to build decision models for technology selection problems in software production. Besides, we designed and implemented a decision support system to employ such decision models to facilitate the decision-making process for decision-makers at software-producing organizations. The framework provides a guideline for decision-makers to model their technology selection problems as MCDM problems. The framework incorporates the following six-step decision-making process (Majumder, 2015): 1) Identifying the objective, 2) Selection of the features, 3) Selection of the alternatives, 4) Selection of the weighing method, 5) Applying the method of aggregation, and 6) Decision making based on the aggregation results. In other words, the framework is employed to build decision models for MCDM problems and find suitable alternatives for software-producing organizations based on their requirements and priorities.

The research approach for creating decision models for MCDM problems is Design Science, which addresses research through the building and evaluation of artifacts to meet identified business needs (Hevner et al., 2008). Knowledge engineering theories have been employed to design and implement the DSS and the technology selection framework. Thirteen experts (three DSS experts, four blockchain researchers from Dutch research institutes, two blockchain-developers, and four blockchain consultants/public-speakers) participated in this research to evaluate the DSS and the decision model for the blockchain platform selection problem. The experts were pragmatically selected according to their expertise and experience that they mentioned on their *LinkedIn* profile. Each of the interview series followed a semi-structured interview protocol and lasted between 45 and 90 minutes.

The DSS experts confirm that the DSS contains the main components of a standard DSS. Moreover, they asserted that the DSS score calculation process is not dependent on the knowledge-based facts and rules (i.e., the decision model). Therefore, the DSS would not generate invalid solutions if the decision models in the knowledge-base change or evolve during the time.

In this study, the primary source of knowledge to build a valid decision model is blockchain experts. Acquired knowledge during each interview typically propagated to the next to build and validate the decision model incrementally. Finally, the decision model was sent to the interview participants afterward for final confirmation.

The efficacy and effectiveness of the decision model have been evaluated through three exploratory theory-testing case studies. The unit of analysis is a unique blockchain platform selection for software-producing organizations. We performed three industry case studies at three blockchain-based software development companies to evaluate the decision model. The case studies typically consisted of (1) defining the blockchain feature requirements, (2) prioritizing them, and (3) comparing the DSS feasible solutions with the solutions that the experts had suggested. None of the interviewed experts mentioned above were in any way involved in the subsequent case studies.

4.3 Related Work

In this research, Snowballing was employed as the primary method to investigate the existing literature related to the techniques that address the blockchain platform selection problem for software-producing organizations.

In literature, some studies point out that benchmarking and performance testing can be employed to evaluate and compare a collection of blockchain platforms against each other. A subset of such studies is presented as follows:

Dinh et al. (2017) present a benchmarking framework for evaluating private blockchain systems. The benchmark contains workloads for measuring the data processing performance and workloads for understanding the performance at different layers of the blockchain.

Hileman & Rauchs (2017) provide an empirical overview of both enterprise and public sector use of blockchain and distributed ledger technology. They report the emergence and evolution of the distributed ledger technology ecosystem, explore actors and their business models and examine the current state of the industry in terms of use cases, network/application deployments, and fundamental challenges to broadly distributed ledger technology adoption.

Maple & Jackson (2018) present the anatomy of blockchain platforms and analyze their essential technological features. Furthermore, the authors introduce a format for outlining generic blockchain building blocks. The anatomy ranges from permissions to consensus and can be referenced when evaluating blockchain platforms. Moreover, they represent a comparison among multiple blockchain platforms.

Kuo et al. (2019) conducted a systematic literature study to identify healthcare applications of blockchain technology, besides the blockchain platforms that have been proposed or implemented by the healthcare blockchain studies. The authors considered ten blockchain platforms and 21 blockchain features, then compared the blockchain platforms based on their features.

Yabo (2016) described the features of multiple blockchain platforms and compared them against each other. Macdonald et al. (2017) discussed how the blockchain is employed in Bitcoin cryptocurrency, besides some potential applications in other domains. Furthermore, the authors presented a comparison of five general-purpose blockchain platforms based on eight criteria related to usability, flexibility, and performance.

A variety of MCDM approaches have introduced by researchers recently. A subset of selected MCDM methods is presented as follows:

The *Weighted Sum Model (WSM)* is an aggregation function that transforms multiple criteria into a single value by multiplying each criterion by a weighting factor and summing up all weighted criteria. Frauenthaler et al. (Frauenthaler et al., 2019) introduce a WSM-based framework to monitor and evaluate several blockchain platforms according to user-defined settings.

The *Analytic Hierarchy Process (AHP)* is a structured and well-known method for organizing and analyzing MCDM problems based on mathematics and psychology. This MCDM approach considers a hierarchical structure of objectives, criteria, and alternatives to make complex decisions. Maček & Alagić (2017) present an AHP-based approach to evaluate the Bitcoin cryptocurrency system's security characteristics compared to other widely used online transaction systems.

The *Technique for Order Preference by Similarity to Ideal Solution (TOPSIS)* is an MCDM approach that employs information entropy to assess alternatives. The purpose of this approach is to come up with an ideal solution and a negative ideal solution and then identify a scenario nearest to the ideal solution and the furthest from the negative ideal solution. Tang et al. (2019) present a TOPSIS-based evaluation model to rank public blockchain platforms according to three dimensions, including technology, recognition, and activity.

The *Boolean Decision Tree (BDT)* is an MCDM method to chose one of the available and feasible decision alternatives. Staderini et al. (2018) propose a BDT-based requirements-driven methodology to support decision-makers to select suitable blockchain platform category (i.e., public or private, and permissionless or permissioned). Moreover, their proposed method assists the decision-makers with the configuration of selected blockchain platforms. Pahl et al. (2018) introduce a BDT-based framework to guide decision-makers to use blockchain technology. Furthermore, they categorize blockchain platforms into three categories (public permissionless, public permissioned, and private) and compare them against each other based on a set of decision criteria. Wüst & Gervais (2018) present a BDT approach to assist decision-makers on whether to select one of blockchain platforms or centralized databases. Additionally, they provide a comparison between permissionless, permissioned blockchain platforms, and centralized databases. Koens & Poll (2018) suggest three questions to find out if blockchain is the best fitting technology (Should you use a blockchain? If so, which blockchain variant is best? If not, which alternative is best?) and if so which type of blockchain platforms should be employed. Moreover, they introduce a BDT-based scheme for determining which type of database is appropriate such as public permissionless blockchain, distributed database, and central database.

Table 4.1 summarizes the selected studies that discuss the blockchain platform selection problem. Performance testing methods (Dinh et al., 2017) are time-consuming approaches and mainly applicable to a limited set of alternatives (blockchain platforms), as their implementation requires in-depth knowledge of blockchain platforms (such as APIs).

Blockchain is a relatively new and fast-evolving technology, so documentation is often out of date or not available; therefore, studies are based on documentation, and reports (Kuo et al., 2019; Macdonald et al., 2017; Maple & Jackson, 2018; Yabo, 2016) are likely to become outdated soon and should be kept up to date continuously.

The majority of the MCDM techniques in literature define domain-specific quality

Table 4.1: this table compares selected studies from the literature that address the blockchain platform selection problem. The first and second columns (Studies and Years) refer to the considered studies and their publication years. The third column (Decision-making technique) indicates the decision-making approach that the studies have employed to address the blockchain platform selection problem. The fourth column (MCDM) denotes whether the corresponding decision-making technique is an MCDM approach. The fifth column (Pairwise comparison) indicates whether the MCDM approach applied pairwise comparison as a weight calculation method or not. The sixth column (Quality Attributes) determines the type of quality attributes. The seventh and eighth columns (Criteria and Alternatives) signify the number of criteria and alternatives that were considered in the selected studies.

Studies	**Years**	**Decision-making technique**	**MCDM**	**Pairwise comparison**	**Quality Attributes**	**Criteria**	**Alternatives**
This study		DSS	Yes	No	ISO/IEC 25010 EX. ISO/IEC 9216	121	28
Dinh et al. (2017)	2017	Benchmarking	No	N/A	Domain specific	4	3
Maple & Jackson (2018)	2018	Benchmarking	No	N/A	Not Defined	3	6
Kuo et al. (2019)	2019	Benchmarking	No	N/A	Not Defined	21	10
Yabo (2016)	2016	Benchmarking	No	N/A	Not Defined	28	25
Macdonald et al. (2017)	2017	Benchmarking	No	N/A	Domain specific	8	5
Frauenthaler et al. (2019)	2019	WSM	Yes	No	Domain specific	8	4
Maček & Alagić (2017)	2017	AHP	Yes	Yes	Domain specific	6	4
Tang et al. (2019)	2019	TOPSIS	Yes	Yes	Domain specific	14	30
Staderini et al. (2018)	2018	BDT	Yes	No	Domain specific	8	4
Pahl et al. (2018)	2018	BDT	Yes	No	Domain specific	6	6
Wüst & Gervais (2018)	2018	BDT	Yes	No	Domain specific	6	4
Koens & Poll (2018)	2018	BDT	Yes	No	Domain specific	9	8

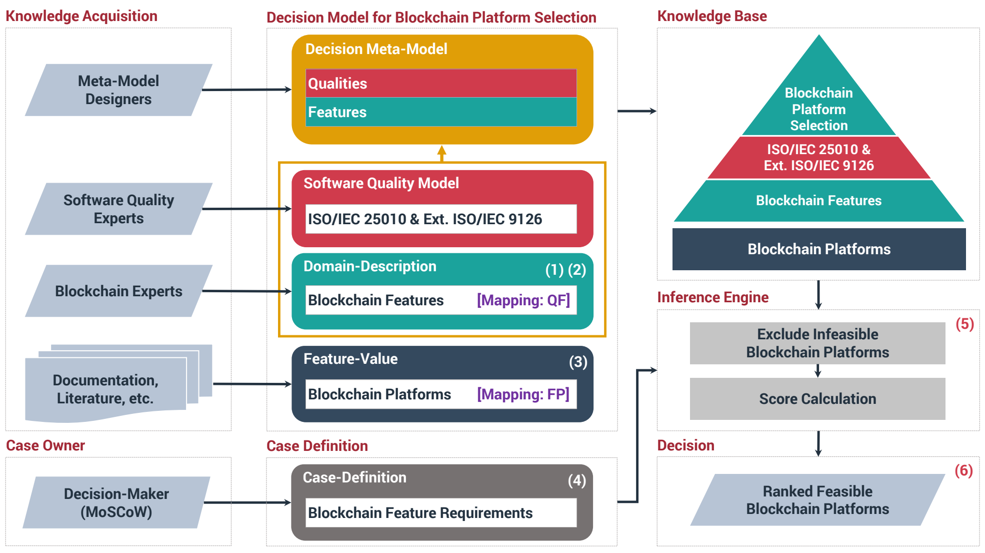
attributes to evaluate the alternatives. Such studies are mainly appropriate for specific case studies. Furthermore, the results of MCDM approaches are valid for a specified period; therefore, the results of such studies, by blockchain technology advances, will be outdated. Note that, in our proposal, this is also a challenge, and we propose a solution for keeping the knowledge base up to date, in section 4.7.

The number of criteria of BDT-based approaches (Koens & Poll, 2018; Pahl et al., 2018; Staderini et al., 2018; Wüst & Gervais, 2018) is limited, i.e., under 10, as processing the large decision-trees is time-consuming and complicated. BDT-based approaches suggest only one solution at the end of each evaluation. Furthermore, decision-makers cannot prioritize decision criteria based on their preferences. Some of the MCDM techniques in the literature use pairwise comparison as the main method to assess the weight of criteria. For a problem with n number of criteria $\frac{n(n-1)}{2}$ number of comparisons are needed (Saaty, 1990). It means that the pairwise comparison is a time-consuming process, and gets exponentially more complicated as the number of criteria increases. Some of the methods, such as AHP and TOPSIS, are not scalable, so in the case of modifying the list of alternatives or criteria, the whole process of evaluation should be redone. Therefore, these methods are costly and applicable to only a small number of criteria and alternatives. Please note that, in this study, we have considered 121 criteria and 28 alternatives to building a decision model for the blockchain platform selection problem.

In contrast to the mentioned MCDM approaches in the literature, the cost of creating, evaluating, and applying the proposed decision model is not penalized exponentially by the number of criteria and alternatives, because it is an evolvable and expandable approach that splits down the decision-making process into four maintainable phases (Farshidi et al., 2018c). Moreover, we introduce several parameters to measure the values of non-Boolean criteria, such as the cost and popularity of the blockchain platforms. The proposed decision model addresses main knowledge management issues, including capturing, sharing, and maintaining knowledge. Furthermore, it uses the ISO/IEC 25010 (ISO, 2011) as a standard set of quality attributes. This quality standard is a domain-independent software quality model and provides reference points by defining a top-down standard quality model for software systems.

Recently, we built two decision models based on the technology selection framework (Farshidi et al., 2018a) to address the Database Management System (Farshidi et al., 2018c) and Cloud-Service Provider (Farshidi et al., 2018a) selection problems. In both studies, several case studies were conducted to evaluate the effectiveness and usefulness of the DSS to address MCDM problems. The results showed that the DSS performed well to solve the Database Management System and Cloud-Service Provider selection problems for software-producing organizations. We believe that the technology selection framework can be considered a reference framework for building decision models for MCDM problems in software-producing organizations.

Figure 4.1: This figure is adapted from our previous study (Farshidi et al., 2018a) and shows the main building blocks of the decision support system beside the proposed decision model for the blockchain platform selection problem.



4.4 Multi-Criteria Decision-Making for Blockchain Platform Selection

We formulate the blockchain selection problem as an MCDM problem. Let $Platforms = \{p_1, p_2, \ldots p_{|Platforms|}\}$ be a set of blockchain platforms in the market (i.e., Hyperledger and Ethereum). Moreover, $Features = \{f_1, f_2, \ldots t_{|Features|}\}$ be a set of blockchain features (i.e., supporting JavaScript, Spam-attack resistant, and Sybil-resistant) of the blockchain platforms, and each $p \in Platforms$ supports a subset of the set $Features$. The goal is finding the suitable blockchain platform p, which supports a set of required blockchain features (set $Requirements$), where $Requirements \subseteq Features$. In other words, a blockchain platform p is the suitable one that supports blockchain feature requirements and satisfies the preferences of the decision-maker. Typically, a unique optimal solution for an MCDM problem does not exist, and it is necessary to employ decision-makers' preferences to differentiate between solutions (Majumder, 2015). The MCDM proposed in this article, therefore, provides a prioritized list of options for decision-makers.

4.4.1 Decision Model for Blockchain Platform Selection

In a previous study, we designed and implemented a DSS[4] (Farshidi et al., 2018b) that comprised of standard DSS components (Sage, 1991) and introduced the

[4]We implemented an online Decision Model Studio (https://dss-mcdm.com) to build decision models for MCDM problems in software-producing organizations.

technology selection framework (Farshidi et al., 2018a) that applies the six-step decision-making process (Majumder, 2015) to build maintainable and evolvable decision models for MCDM problems in software production. In this study, we follow the technology selection framework as modeled in Fig. 4.1 to build a decision model for the blockchain platform selection problem. Generally speaking, a decision model for an MCDM problem contains decision criteria, alternatives, and relationships among them. Fig. 4.1 illustrates the main building blocks of the decision support system besides the proposed decision model for the blockchain platform selection problem.

Decision Meta-Model:
As introduced in (Farshidi et al., 2018b), the *Decision Meta-Model* is a simplified view of decision models and highlights the fundamental structure of decision models. Furthermore, it provides ontological descriptions of MCDM problems. The Decision Meta-Model has two sets, namely, *Qualities* and *Features*. Software quality attributes such as interoperability, maturity, and performance of blockchain platforms are kept in the set *Qualities*. Additionally, Blockchain features such as *Smart-contracts* and *on-chain transactions* are listed in the set *Features*.

Software Quality Model:
The *Software Quality Model* is a set of characteristics, and relationships between them, which provides a structure for specifying quality requirements and assessing them (Farshidi et al., 2018b). The Software Quality Model supports the specification of quality requirements, assess blockchain features, or predict the quality of a blockchain platform. The decision model for the blockchain platform selection problem employs the *ISO/IEC 25010* standard (ISO, 2011) and *extended ISO/IEC 9126* standard (Carvallo & Franch, 2006) in order to define the set *Qualities*. Such domain-independent quality models suggest standard hierarchical quality models for software systems. The elements of the *Software Quality Model* are used to analyze blockchain features based on their impact on quality attributes of blockchain platforms.

Domain-Description:
As aforementioned in (Farshidi et al., 2018b), the *Domain-Description* determines the first and second steps, indicated by *Identifying the objective* and *Selection of the features*, of the decision-making process. As it is clear, the objective of the decision-making process in this study is *Blockchain Platform Selection*. Blockchain experts are the primary source of knowledge to identify the best fitting set of blockchain features, even though documentation and literature study of blockchain platforms can be employed to come up with an initial hypothesis about the blockchain feature set. Each blockchain feature has a data type, such as *Boolean* and *non-Boolean*. For example, the data types of blockchain features like the *popularity in the market* and supportability of *Smart-contracts* of a Blockchain Platform can be considered as *non-Boolean* and *Boolean*, respectively.
In this study, the initial set of blockchain features is extracted from online

Table 4.2: this table shows the considered blockchain features by the interviewees. Checkmarks (✓) denote the blockchain features suggested by the corresponding blockchain experts, and cross marks (✗) signify that the blockchain experts did not express a need for the blockchain features in the decision model. The *Agreement* columns denote the agreements among the blockchain experts regarding considering blockchain features. Note, this is not the final list of blockchain features. The full list of the blockchain features besides their definitions are available in the appendix section of this study. (https://dss-mcdm.com)

Blockchain features	Agreement	Interview 1	Interview 2	Interview 3	Interview 4	Interview 5	Interview 6	Interview 7	Interview 8	Interview 9	Blockchain features	Agreement	Interview 1	Interview 2	Interview 3	Interview 4	Interview 5	Interview 6	Interview 7	Interview 8	Interview 9
Proof-of-Work	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Solidity	88.89%	✓	✓	✓	✓	✗	✓	✓	✓	✓
Proof-of-Stake	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	C#	22.22%	✗	✓	✓	✗	✗	✗	✗	✗	✗
delegated Proof-of-Stake	55.56%	✓	✓	✓	✓	✗	✗	✗	✓	✗	Golang	55.56%	✗	✓	✗	✗	✗	✓	✓	✓	✓
practical Byzantine Fault Tolerance	77.78%	✓	✓	✓	✗	✗	✓	✓	✓	✓	JavaScript	55.56%	✗	✓	✗	✗	✗	✓	✓	✓	✓
federated Byzantine Agreement	77.78%	✓	✓	✓	✗	✗	✓	✓	✓	✓	Java	77.78%	✗	✓	✓	✓	✗	✓	✓	✓	✓
delegated Byzantine Fault Tolerance	77.78%	✓	✓	✓	✗	✗	✓	✓	✓	✓	C++	55.56%	✗	✓	✓	✓	✗	✗	✓	✗	✓
Proof-of-Authority	55.56%	✗	✗	✗	✓	✓	✓	✓	✓	✗	Python	55.56%	✗	✓	✓	✗	✗	✓	✓	✓	✗
Proof-of-Elapsed Time	55.56%	✗	✓	✓	✗	✗	✓	✓	✓	✗	Haskell	22.22%	✗	✓	✗	✗	✗	✗	✓	✗	✗
Proof-of-Burn	11.11%	✗	✗	✓	✗	✗	✗	✗	✗	✗	Tokenization	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓
Proof-of-Luck	11.11%	✗	✗	✓	✗	✗	✗	✗	✗	✗	Native	55.56%	✗	✓	✗	✓	✗	✗	✓	✓	✓
Directed Acyclic Graph (variants)	55.56%	✓	✗	✗	✗	✗	✓	✓	✓	✓	Non-native	55.56%	✗	✓	✗	✓	✗	✗	✓	✓	✓
SHA-256	44.44%	✗	✗	✓	✗	✗	✓	✗	✓	✓	Cryptocurrency	77.78%	✓	✓	✓	✓	✗	✓	✓	✗	✓
SHA-3	44.44%	✗	✗	✓	✗	✗	✓	✗	✓	✓	Utility	66.67%	✓	✓	✗	✓	✗	✗	✓	✓	✓
md-5	22.22%	✗	✗	✓	✗	✗	✗	✗	✓	✗	Security	55.56%	✓	✓	✓	✗	✗	✓	✗	✗	✓
SHA-512	33.33%	✗	✗	✗	✗	✗	✓	✗	✓	✓	Usage	55.56%	✗	✓	✓	✓	✗	✗	✓	✗	✓
ASIC-algorithm	33.33%	✗	✗	✓	✗	✗	✗	✗	✓	✓	Asset	66.67%	✓	✓	✗	✗	✗	✓	✓	✓	✓
Cryptonight	22.22%	✗	✗	✓	✗	✗	✗	✗	✓	✗	Work	33.33%	✗	✓	✗	✗	✗	✗	✓	✗	✓
Protocol Layer	77.78%	✓	✗	✗	✓	✓	✓	✓	✓	✓	Hybrid	33.33%	✗	✓	✗	✗	✗	✗	✓	✗	✓
Network Layer	77.78%	✓	✗	✗	✓	✓	✓	✓	✓	✓	On-chain transactions	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓
Application Layer	88.89%	✓	✗	✓	✓	✓	✓	✓	✓	✓	Off-chain transactions	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓
Public	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Side-chains	88.89%	✓	✗	✓	✓	✓	✓	✓	✓	✓
Private	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Sharding	88.89%	✓	✗	✓	✓	✓	✓	✓	✓	✓
Permissioned	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Plasma-chains	66.67%	✗	✗	✗	✓	✓	✓	✓	✓	✓
Permissionless	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Atomic-swaps	66.67%	✗	✗	✓	✓	✗	✓	✓	✓	✓
Smart-Contracts	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Cross-chain interoperable	66.67%	✓	✗	✗	✓	✓	✓	✗	✓	✓
Virtual Machine	66.67%	✗	✓	✓	✗	✗	✓	✓	✓	✓	Enterprise system integration	77.78%	✓	✓	✓	✗	✓	✓	✗	✓	✓
Docker	11.11%	✗	✓	✗	✗	✗	✗	✗	✗	✗	Zero-knowledge Proof	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓
Turing-Complete	66.67%	✗	✓	✓	✗	✗	✓	✓	✓	✓	Ring-signatures	55.56%	✗	✗	✓	✓	✗	✗	✓	✓	✓
Transaction Speed	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Hard-fork resistant	55.56%	✓	✓	✓	✗	✗	✓	✗	✓	✗
Block-Size	44.44%	✓	✓	✗	✗	✓	✓	✗	✗	✗	Spam-attack resistant	77.78%	✓	✓	✓	✗	✗	✓	✓	✓	✓
Maturity	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Sybil attack resistant	88.89%	✓	✓	✓	✓	✗	✓	✓	✓	✓
Platform Transaction Speed	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Quantum resistant	55.56%	✗	✗	✓	✓	✗	✓	✗	✓	✓
Popularity in the market	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Transaction Irreversibility	55.56%	✓	✗	✓	✓	✗	✗	✗	✓	✓
Innovation	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	Wallet	22.22%	✗	✗	✓	✓	✗	✗	✗	✗	✗

documentation of blockchain platforms. A list of prominent blockchain features was identified during nine blockchain expert interviews (three researchers from Dutch research institutes, two blockchain developers, and four blockchain consultants/public-speakers). Finally, 71 Boolean and four non-Boolean blockchain features[5] identified and validated by the blockchain experts. Table 4.2 shows the identified blockchain features based on blockchain experts' opinions. Checkmarks (✓) denote the blockchain features suggested by the corresponding blockchain experts, and cross marks (✗) signify that the blockchain experts did not express the blockchain features. Note, we excluded the blockchain features that were not considered at least by two blockchain experts.

The mapping (*QF*) between the sets *Qualities* and *Features* is identified based on blockchain experts' knowledge. Four blockchain experts participated in this phase of the research to map the considered blockchain features to the set *Qualities* based on a Boolean adjacency matrix ($Qualities \times Features \rightarrow Boolean$). For instance, *consensus-mechanisms* as a blockchain feature influences the *Fault-tolerance* quality

[5]The entire list of the blockchain features and supportability of considered blockchain platforms are available and accessible on the "Blockchain Platform Selection" website (https://dss-mcdm.com)

aspect. The *Domain Description* does not enforce a blockchain feature to present in a single quality aspect; Blockchain features can be part of many quality aspects. For example, *Spam-attack resistant* as a blockchain feature might connect to multiple quality aspects such as *Recoverability* and *Availability*.

Feature-Value:
As we discussed in our previous study (Farshidi et al., 2018b), the *Feature-Value* represents the third step, shown by *Selection of the alternatives*, of the decision-making process. Accordingly, a list of blockchain platforms (set *Platforms*) should be defined. Well-known blockchain platforms, websites, related forums, and blockchain experts are the primary source of knowledge to specify the list of blockchain platforms. In this study, 28 blockchain platforms (i.e., Hyperledger, Ethereum, and Chain) have been considered.

Blockchain features can be either Boolean or non-Boolean. A Boolean blockchain feature ($Feature^B$) is a feature that its supportability by the blockchain platforms is investigated. Moreover, a non-Boolean blockchain feature ($Feature^N$) assigns a non-Boolean value to a particular blockchain platform, for example, the maturity level of a blockchain platform. Therefore, the blockchain features in this study is a collection of Boolean and non-Boolean features, where $Features = Feature^B \cup Feature^N$.

The mapping $BFP : Feature^B \times Platforms \rightarrow \{0, 1\}$ defines the supportability of the Boolean blockchain features by the blockchain platforms. So that $BFP(f, p) = 0$ means that the blockchain platform p does not support the blockchain feature f and $BFP(f, a) = 1$ signifies that the platform supports the feature.

The mapping *BFP* is defined based on documentation of the blockchain platforms and expert interviews. One of the principal challenges is the lack of standard terminology among blockchain platforms. Different blockchain platforms refer to the same concept by different names, or even worse, the same name might stand for different concepts in different blockchain platforms. Discovering conflicts in the *Feature-Value* is essential to prevent semantic mismatches throughout the blockchain platform selection process. Table 4.3 shows a sample of the *BFP* mapping between the **B**oolean blockchain **F**eatures and **P**latforms in the knowledge base of the DSS.

In this study, the non-Boolean blockchain features are *Blockchain Platform Maturity, Popularity in the Market, Transaction Speed,* and *Innovation*. The assigned values to these non-Boolean blockchain features for a specific blockchain platform is a 3-point Likert scale, where $NFP : Features^N \times Platforms \rightarrow \{High, Medium, Low\}$, based on several predefined parameters.

Table 4.4 illustrates the non-Boolean blockchain features besides their parameters. The blockchain experts assigned the 3-point Likert scale values to these non-Boolean blockchain features according to the corresponding values of the parameters.

4.4.2 Knowledge Base

Each decision model defines a decision structure for an MCDM problem systematically based on the technology selection framework (Farshidi et al., 2018a). The *Knowledge Base* is a collection of decision models, which are groups of rules and facts. The blockchain decision model has been uploaded to the DSS knowledge base to facilitate

Table 4.3: this table lists a sample of the **BFP** mapping between the **B**oolean blockchain **F**eatures and **P**latforms. The first row and column of the table designate the blockchain features and platforms, respectively. Furthermore, 1s on each row indicates that the corresponding platforms support the blockchain feature of that row. Conversely, 0s mean that the corresponding platforms do not support that blockchain feature. Note, the *Coverage* column denotes the percentage of blockchain platforms that support each feature. The complete list of the blockchain features and platforms are available in the appendix. We plan to keep the list up-to-date, so the list is available on the following website as well: https://dss-mcdm.com

BFP Boolean blockchain Features and Platforms	Coverage	Ethereum	R3 Corda	JPMorgan Quorum	Hyperledger	BigChainDB	MultiChain	HydraChain	Chain	Symbiont	Stratis (Azure BaaS)	OpenChain	NEO	Cardano	Stellar	Ripple	Bitshares	QTUM	ICON	VeChain	IOTA	Factom	Cosmos Network	Lisk	Waves Platform	Wanchain	Neblio	Zilliqa	Komodo
Consensus																													
Consensus Mechanism	96.43%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
Proof-of-Work	25.00%	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	1
Proof-of-Stake	28.57%	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	1	0	1	1	1	0	0
Delegated Proof-of-Stake	14.29%	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0
practical Byzantine Fault Tolerance	28.57%	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0
delegated Byzantine Fault Tolerance	14.29%	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
Proof-of-Authority	14.29%	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
Federated Byzantine Agreement	17.86%	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0
Proof-of-Elapsed Time	7.14%	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
SIEVE	3.57%	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Cross-Fault Tolerance	3.57%	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Directed Acyclic graph	3.57%	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
Layers																													
Protocol Layer	96.43%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
Network Layer	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Application Layer	82.14%	1	1	1	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1
Authorization and Authentication																													
Public	64.29%	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Private	35.71%	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Permissioned	53.57%	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	1	0	0
Permissionless	57.14%	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1
Contracts																													
Smart-contracts	71.43%	1	1	1	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	1
Virtual Machine	39.29%	1	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1	0	1	0	1	0
Turing Complete	35.71%	1	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1	0	1	0	0	0
Programming																													
Programming Language Support	89.29%	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1
Solidity	28.57%	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0
Python	46.43%	1	0	0	1	1	1	1	0	1	0	1	1	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	1
Golang	32.14%	1	0	1	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	0	0	1	0	1
Java	42.86%	1	1	0	1	0	1	0	1	0	0	0	1	0	1	1	0	1	0	0	1	0	0	0	0	0	1	0	1
JavaScript	35.71%	1	0	0	1	0	0	0	0	0	0	1	1	0	1	0	0	1	0	0	0	0	0	1	1	0	1	0	1

the decision-making process for software-producing organizations for any blockchain platform selection.

4.4.3 Case-Definition

As discussed in (Farshidi et al., 2018b), the *Case-Definition* defines the fourth step, denoted by *Selection of the weighing method*, of the decision-making process. Decision-makers prioritize the blockchain feature requirements using the *MoSCoW* technique.

Suppose $W_{MoSCoW} = \{w_{Must}, w_{Should}, w_{Could}, w_{Won't}\}$ is the set of priority weights according to the definition of the *MoSCoW* (DSDM consortium and others, 2014). Blockchain feature requirements with *Must Have* or *Won't Have* priorities act as hard constraints and blockchain feature requirements with *Should Have* and *Could Have* priorities act as soft constraints. In other words, a case definition, based on the decision-maker preferences, is a way to define blockchain feature requirements and

Table 4.4: this table shows the **NFP** mapping between the **N**on-Boolean blockchain **F**eatures and **P**latforms. Note, the *Platform Transaction Speed*, *Popularity in the market*, *Innovation*, and *Blockchain Platform Maturity* are the Non-Boolean blockchain features that were considered in this study. The parameters of these features are listed below each features, for example, *Founded*, *Revenue*, *Size*, and *Consensus-mechanism* are the parameters of the *Blockchain Platform Maturity*.

NFP Non-Boolean blockchain Features and Platforms	Ethereum	R3 Corda	JPMorgan Quorum	Hyperledger	BigChainDB	MultiChain	HydraChain	Chain	Symbiont	Stratis (Azure BaaS)	OpenChain	NEO	Cardano	Stellar	Ripple	Bitshares	QTUM	ICON	VeChain	IOTA	Factom	Cosmos Network	Lisk	Waves Platform	Wanchain	Neblio	Zilliqa	Komodo	Source of Knowledge
Transaction Speed	Low	Highigh	Highigh	Highigh	Highigh	Medium	Medium	Highigh	Highigh	Highigh	Highigh	Medium	Medium	Medium	Highigh	Medium	Low	Highigh	Medium	Low	Low	Highigh	Medium	Medium	Low	Low	Highigh	Low	*Domain Experts*
Confirmation Time (sec)	15	1	1	1	1	1	1	1	1	1	1	15	10	4	3.5	3	120	2	5	120	300	1	10	3	15	120	120	60	*The platform website*
Speed of consensus mechanism (transactions/second)	15	170	140	3500	1E+06	1000	0	15000	80000	20000	2500	1000	257	1000	1500	3300	70	3000	50	500	45	3000	28	1000	15	27	2000	80	*The platform website*
# Scalability technologies implemented	2	2	2	2	0	0	1	0	0	2	0	2	0	2	2	0	0	3	2	0	0	3	2	0	0	0	2	0	*The platform website*
Popularity in the market	Highigh	Highigh	Medium	Highigh	Low	Medium	Low	Low	Low	Medium	Low	Medium	Medium	Medium	Highigh	Medium	Medium	Medium	Low	Medium	Low	Low	Medium	Medium	Low	Low	Low	Low	*Domain Experts*
Market Capitalization (x 1.000.000 dollar)	58676	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3458	5327	5214	23562	508	1200	1049	1895	4049	N/A	N/A	932	455	470	108	870	275	*https://coinmarketcap.com*
Google monthly searches	2240000	405000	201000	405000	3600	4400	300	1000	9900	5000	1900	368000	201000	201000	1830000	60500	14800	40500	9900	673000	27100	480	90500	5400	4400	2900	1600	6600	*https://keywordseverywhere.com*
Twitter Followers	396491	20954	432	45454	7560	3581	0	26527	2359	138774	336	318939	135938	236565	873583	85177	162920	101272	92828	112199	76913	10912	189261	125094	10208	74635	39925	88178	*https://twitter.com*
Reddit subscribers	362000	N/A	N/A	N/A	N/A	53	N/A	N/A	N/A	10800	N/A	92900	64500	79200	187000	7000	14300	22600	47800	110000	8400	2200	29800	12600	17100	5900	7400	6600	*https://www.reddit.com*
Transactions+Operations per day	833377	1224000	10000	2.4E+07	N/A	N/A	N/A	N/A	N/A	1200000	N/A	76334	2420	150000	955000	155000	3000	1000	N/A	146880	40000	N/A	3470	22000	250	800	800	10000	*https://bitinfocharts.com*
Linkedin followers	60931	10308	1026784	15809	1220	16	4	2835	1446	783	46	2761	3001	2141	2043	481	170	127	810	2961	1962	1724	1865	882	756	0	641	291	*https://www.linkedin.com*
Innovation	Highigh	Highigh	Medium	Highigh	Medium	Medium	Low	Medium	Low	Highigh	Low	Medium	Medium	Medium	Medium	Low	Low	Medium	Medium	Medium	Low	Medium	Low	Low	Medium	Low	Low	Low	*Domain Experts*
Internet of Things	1	1	1	1	0	1	0	1	0	1	0	1	0	0	0	1	0	0	1	1	1	1	1	1	0	1	0	0	*The platform website*
Artificial Intelligence	1	1	1	1	0	1	0	1	0	1	0	1	0	0	1	0	0	1	1	1	0	1	0	1	0	0	1	0	*The platform website*
Plasma Technology	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	*The platform website*
Sharding	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	*The platform website*
Supply-Chain Management	1	1	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	*The platform website*
Financial Sector	1	1	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	1	1	0	0	1	1	1	1	0	1	1	*The platform website*
Zero-knowledge proofs technology	1	1	1	1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	*The platform website*
Consortium-Research Support	1	1	1	1	1	1	0	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	0	0	1	1	0	0	*The platform website*
Cross-chain interoperability	1	1	0	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1	0	1	1	0	0	1	*The platform website*
Blockchain Platform Maturity	High	High	High	High	High	Medium	Low	Medium	High	High	Low	Low	Low	Medium	High	Low	Low	Low	High	Low	Medium	Low	Low	Medium	Low	Low	Low	Low	*Domain Experts*
Founded	2013	2014	2017	2014	2014	2014	2015	2014	2015	2017	2015	2017	2017	2014	2012	2014	2017	2015	2015	2015	2015	2016	2016	2016	2017	2017	2017	2014	*The platform website*
Revenue	Between 1 and 5 million	Between 1 and 5 million	More than 5 million	More than 5 million	More than 5 million	Less than 1 million	Less than 1 million	Between 1 and 5 million	Between 1 and 5 million	More than 5 million	Less than 1 million	Less than 1 million	Less than 1 million	Less than 1 million	More than 5 million	Less than 1 million	Less than 1 million	Less than 1 million	More than 5 million	Less than 1 million	Between 1 and 5 million	Less than 1 million	Less than 1 million	Between 1 and 5 million	Less than 1 million	Less than 1 million	Less than 1 million	Less than 1 million	*https://www.owler.com*
Size (employees+github devs)	35000	2000	157	7351	214	573	10	25	504	337	66	31	13	250	274	44	33	33	129	55	44	31	46	35	47	9	25	16	*https://www.owler.com* *https://github.com*
Consensus-mechanism	Proof-of-Work	Byzantine Fault Tolerance	Byzantine Fault Tolerance	Byzantine Fault Tolerance	Byzantine Fault Tolerance	Proof-of-Work	Byzantine Fault Tolerance	Byzantine Fault Tolerance	Byzantine Fault Tolerance	Proof-of-Stake	Proof-of-Authority	Byzantine Fault Tolerance	Proof-of-Stake	Byzantine Fault Tolerance	Byzantine Fault Tolerance	Delegated Proof-of-Stake	Proof-of-Stake	Byzantine Fault Tolerance	Proof-of-Authority	Directed Acyclic Graph	Proof-of-Work	Byzantine Fault Tolerance	Delegated Proof-of-Stake	Proof-of-Stake	Proof-of-Stake	Proof-of-Stake	Byzantine Fault Tolerance	Proof-of-Work	*The platform website*

assign priorities to them. Note, we could have used other prioritization techniques, but we purposely wanted to keep it simple.

Decision-makers specify desirable values for numeric blockchain feature requirements. For example, a decision-maker could be interested in prioritizing blockchain platforms with the *Blockchain Platform Maturity* above average. Therefore, the *Blockchain Platform Maturity* above average is considered as a *should have* blockchain feature.

Fig. 4.2 shows a decision structure from a case definition. The DSS generated the decision structure and inferred the solutions based on the proposed decision model for the blockchain platform selection problem. At the top of the figure (Domain), the domain of the decision structure is shown (*ShareCompany BIQH Blockchain Platform (BP) Selection*). The next level of decision structure (Qualities) illustrates the characteristics and sub-characteristics of the ISO/IEC 25010 (ISO, 2011) plus ISO/IEC 9126 (Carvallo & Franch, 2006) standards, respectively. The relationship among the quality attributes is based on the definitions of these two quality models. The third part of the decision structure (Features) presents the blockchain feature requirements based on decision-makers' priorities and preferences. As we mentioned earlier, the decision model utilizes the *MoSCoW* prioritization technique to assign blockchain features weights. In this figure, the colors identify the blockchain features priorities. Finally, the lowest segment of the decision structure (Platforms) shows the final results for the specific case definition according to the decision-makers' blockchain features requirements and priorities. The mapping QF, which is based on the knowledge of blockchain experts (tacit knowledge), demonstrates the relationship between sub-characteristics of the ISO/IEC standards and blockchain Features. Moreover, the mapping *FP* partially illustrates the mapping between the blockchain features and platforms. The mapping between the sets of Boolean blockchain features and platforms is denoted by *BFP*, and the mapping between the sets of non-Boolean blockchain features and platforms is signified by *NFP*. The mappings *NFP* and *BFP* are two subsets of the mapping *FP*, where $FP = NFP \cup BFP$.

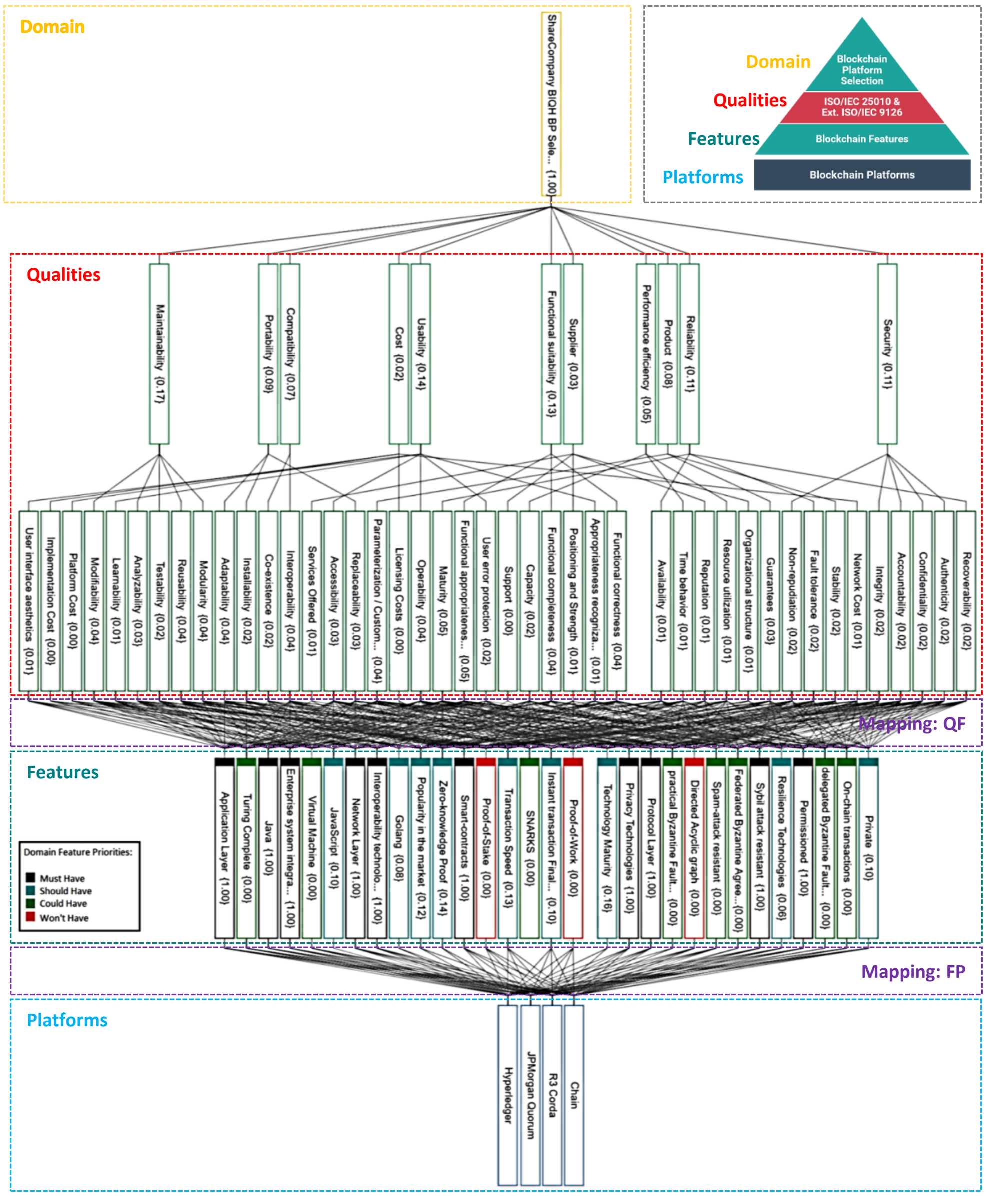
4.4.4 Inference Engine

The *Inference Engine* comprises two steps: the fifth step, i.e., *Applying the method of aggregation* and the sixth step, i.e., *Decision making based on the aggregation results*, of the decision-making process (Farshidi et al., 2018b). The *Inference Engine* makes logical deductions about knowledge assets, intending to find the best fitting blockchain platforms.

A feasible blockchain platform must support all blockchain feature requirements with *Must-Have* priorities, and must not support all blockchain feature requirements with *Won't-Have* priorities.

The *Inference Engine* excludes infeasible blockchain platforms, calculates the scores of the feasible blockchain platforms, and finally suggests a sorted shortlist of them. The score calculation process is based on the *Weighted Sum Model*, as described in our previous work (Farshidi et al., 2018a). The scores of feasible blockchain platforms are non-negative integers, so by sorting the feasible blockchain platforms in descending order of their scores, the final ranked list of feasible blockchain platforms will be provided as the result of the DSS.

Figure 4.2: this figure illustrates a decision structure based on a case definition (ShareCompany BIQH Blockchain Platform Selection). The DSS has automatically generated the decision structure. The first level of the decision structure (Domain) indicates the goal of the decision structure. The second level (Qualities) denotes the relevant quality attributes that impact the prioritized blockchain platform feature requirements, signified in the third level (Features). The last level (Platforms) shows the list of feasible blockchain platforms. Note, the mapping FP and QF define the relationship between the qualities, features, and platforms.



4.4.5 Group Decision-Making

The DSS provides a discussion and negotiation platform to enable decision-makers to make group decisions. The DSS asks decision-makers to define individual blockchain feature requirements based on the *MoSCoW*. Next, it collects the individual prioritized blockchain feature requirements of decision-makers and considers the maximum the *MoSCoW* priority for each blockchain feature requirement (Farshidi et al., 2018b). It detects and highlights the conflicts in the assigned priorities to the blockchain feature requirements by decision-makers, and asks them to resolve disagreements.

4.5 Empirical Evidence: The Case Studies

Three industry case studies at three software development companies are conducted to evaluate and signify the decision model's usefulness and effectiveness. The case-study participants have identified a number of potentially feasible blockchain platforms for their organizations through multiple internal expert meetings and investigation into blockchain platforms before participating in this research. Moreover, the case-study participants have employed the DSS to analyze, document, track, and prioritize their blockchain feature requirements. The remaining sections describe the case studies and discuss the outcome of the DSS.

4.5.1 Case Study 1: ShareCompany BIQH

ShareCompany BIQH, a FinTech company in the Netherlands, supports two well-known Dutch banks with accommodating the requirements put forth by the European Union regarding packaged retail investment and insurance-based products (PRIIP/KID regulation). ShareCompany BIQH is now interested in investigating deploying its current centralized financial system on a blockchain platform. Some of the envisioned system requirements and corresponding blockchain feature requirements asserted by the case study participants are listed as follows.

- The envisioned system requires extensive integration with the current system (e.g., APIs), so *Enterprise System Integration* is considered as a *Must-Have* feature. Since only a limited number of end-users are authorized to make changes in the system, *Permissioned* is a *Must-Have* feature.
- The current system and its data are already publicly accessible, so a *private platform* is not a necessary blockchain feature and considered as a *Should-Have* feature. The *Protocol Layer*, *Network Layer*, and the *Application Layer* are all prioritized as *Must-Have* features. The protocol layer supports reaching consensus on the accuracy of the data; the network layer defines the communication among system end-users, and the application layer employs to build the required infrastructure to connect with current enterprise systems.
- The clients of the system should remain anonymous. So, the envisioned system should follow the General Data Protection Regulation and force its employed technologies to support data privacy regulation. Therefore, supporting *Smart Contracts* in the Java programming language have been prioritized as a *Must-Have* feature.

- The selected blockchain platform must be *Sybil-attack resistant* to prevent such attacks in peer-to-peer networks.
- ShareCompany BIQH is operating in a financial data environment with major internationally-operating banks, so the selected blockchain platform *Should-Have* both a *High Maturity* and a *High Popularity* in the market to reduce potential risks.
- The envisioned system should provide anonymous and secure transactions. Thus, *Zero-knowledge Proof* feature is prioritized as a *Should-Have* feature.
- The speed of transactions in the envisioned system is not a crucial issue; therefore, *High Transaction speed* can be considered as a *Should-Have* blockchain feature.
- ShareCompany BIQH has professional Golang and JavaScript developers, however, coding in other programming languages is not a major obstacle. Thus, supporting *Golang* and *JavaScript* programming language are two *Should-Have* blockchain features.
- ShareCompany BIQH does not decide on a consensus mechanism. As they are not working with cryptocurrency and not interested in the consensus mechanisms designed for a cryptocurrency, they prioritized *Proof-of-Work* or *Proof-of-Stake* as two *Won't-Have* blockchain features. Three remaining considered consensus-mechanisms in the decision model are *Practical Byzantine Fault Tolerance*, *Federated Byzantine Fault Tolerance*, and *Delegated Byzantine Fault Tolerance* prioritized as *Could-Have* blockchain features.
- The case-study participants mentioned that *Directed-Acyclic-Graph* is too experimental and immature yet, so they considered it as another *Won't-Have* blockchain feature.

Finally, the case-study participants themselves concluded that *Hyperledger* and *JPMorgan Quorum* are two potential blockchain platforms that meet all their requirements.

4.5.2 Case Study 2: DUO

DUO is the administrative and executive agency of the Dutch government for managing the educational system. DUO operates in the name of the Ministry of Education, Culture, and Science and the Ministry of Social Affairs and Employment. DUO has eight different main functions, with several activities as their core focus. This case study will merely focus on the process of student financing in the form of granting loans. DUO is interested in building a decentralized application based on blockchain technology to address the requirements of the student financing activities. Some of the envisioned system requirements and corresponding blockchain feature requirements that were asserted by the case-study participants are listed as follows.

- The DUO financing system requires the three layers of a blockchain platform (including the protocol, network, and application layers). Therefore, these three layers are considered as three *Must-Have* blockchain features.
- The system has no strict requirement for a specific consensus-mechanism. Therefore, all considered consensus-mechanisms in the decision model are prioritized as *Could-Have* blockchain features.
- Supporting *Smart Contracts* is a *Must-Have* blockchain feature, as it mainly influences the functionality of the system. For example, the *Smart Contracts* handle

paying out the loans each month if a specific date has passed, grant conventional loans to students if they meet the specified conditions, or deny additional loans to students who try deceiving the system.

- The payout of credits can be either done by the system in fiat currency (Euro) or in the form of *Cryptocurrency*, which acts as Native token to the system. Thus, the *Cryptocurrency* is a *Should-Have* blockchain feature.
- The DUO financial system executes transactions as *On-chain transactions*, moreover, it utilizes *Cryptographic Tokens*. Therefore, both of them are prioritized as *Must-Have* blockchain features.
- The DUO financial system has to be deployed on a public, private, permission, or permissionless blockchain platform. The Dutch government prefers not to utilize public blockchain platforms. However, selecting a public blockchain platform that follows privacy regulations dramatically increases transparency and possibly credibility, as Cyber Capital indicates. Therefore, *Permission* and *Permissionless* are considered as two *Could-Have* blockchain features.
- Currently, Solidity is the most common programming language to create smart contracts and specifically designed for it. Thus, *Solidity* prioritized as a *Must-Have* blockchain feature.
- *Spam-attack resistant* and *Sybil attack resistant* are *Must-Have* blockchain features from the case-study participants to guarantee a base level of security and resilience.
- Supporting *JavaScript*, being *Turing-complete* are two *Should-Have* blockchain features.
- Selecting a blockchain platform with *High Maturity* decreases potential unnecessary risks as much as possible, so it as a *Should-Have* blockchain feature in this case study.
- The case-study participants asserted that they would not utilize a *Directed Acyclic Graph* for now since it is still too immature; therefore, they prioritized it as a *Won't-Have* blockchain feature.

Finally, the case-study participants selected three blockchain platforms as their main potential solutions, namely *Ethereum*, *NEO*, and *Hyperledger*.

4.5.3 Case Study 3: Veris Foundation

The Veris Foundation is an organization focusing on the American healthcare system. The Veris Foundation addresses the problem of bringing healthcare service providers, insurers, and banks together to authorize the provisioning and payment for healthcare services. The Foundation is a nonprofit whose core objective is to establish a platform to reduce the cost of healthcare and make it more affordable to patients. Traditional, centralized healthcare systems are slow, redundant, and expensive, because, service providers and payers employ their staff and separate software stacks to facilitate their medical claims processes. These isolated systems make the sharing of necessary information complicated, costly, and prone to errors and fraud. The Veris Foundation is interested in finding the best fitting blockchain platform, as they believe that creating decentralized databases enables all parties to securely access and share data within and across organizations, eliminating the need to hire and maintain expensive third-party information systems. Some of the requirements and corresponding blockchain feature requirements that were stated by the case-study participants are listed as follows.

- case-study participants prioritized supportability of *smart contracts* as a *Must-Have* blockchain feature, because smart contracts define the rules and penalties related to agreements among parties and automatically enforce those obligations.
- the Veris platform is currently a forked version of the NEO blockchain platform, so the *Protocol Layer*, *Network Layer*, and the *Application Layer* are all prioritized as *Must-Have* features. Note, the NEO blockchain platform already supports most of the Veris blockchain feature requirements.
- case-study participants stated that the *delegated Byzantine Fault Tolerance* and *Proof-of-Stake* are two consensus mechanisms that can be employed interchangeably, so that they are two *Should-Have* blockchain features.
- The Veris platform provides different Graphical User Interfaces for its stakeholders; moreover, their authority is required to provide the validation of blocks of transactions. Therefore, case-study participants prioritized *Permissioned* as a *Must-Have* blockchain feature.
- The platform interacts with other parties, such as banks, so it requires a specific type of interoperability, and in particular *Enterprise system integration*. Thus, the case-study participants have considered it as a *Must-Have* blockchain feature.
- The dual-currency structure of the Veris platform gives rise to discuss the *Cryptographic tokens* as a *Must-Have* blockchain feature.
- The Veris platform has not decided on a special type of token. Therefore, *Share-like token*, *Security token*, *Network token*, *Network value token*, *Work token*, and *Usage token* are considered as *Should-Have* blockchain features.

In this study, the case-study participants selected *Ethereum* and *NEO* as two potential blockchain platforms for their system.

4.6 Results and Analysis

The case-study participants specified their blockchain features requirements according to the MoSCoW priorities (table 4.5), so three industry cases are defined and stored in the knowledge base of the DSS. Next, the Inference Engine of the DSS generated feasible solutions for each case definition.

4.6.1 The DSS Results

Table 4.6 shows the deduced feasible blockchain platforms along with their calculated scores by the DSS. Moreover, it compares the case-study participants' shortlists and their ranks, which results from internal meetings and investigations, with the outcomes of the DSS.

ShareCompany BIQH

The case-study participants at ShareCompany BIQH considered *Hyperledger* and *JP-Morgan Quorum* as the first and second potential solutions in their shortlist. *Hyperledger* supports all the *Must-have* and most of the *Should-have* and *Could-have* blockchain feature requirements (such as JavaScript programming language, Zero-

Table 4.5: this table presents a subset of blockchain feature requirements of the considered three industry case studies. The blockchain feature requirements are defined based on the MoSCoW prioritization technique.

MoSCoW	**ShareCompany BIQH**	**DUO**	**Veris Foundation**
Must-Have	Permissioned Platform Interoperability technologies Smart Contract Java Sybil-attack resistant Privacy Technologies Enterprise System Integration Network Layer Application Layer Protocol Layer	Protocol Layer Network Layer Application Layer Smart contracts On-chain transactions Cryptographic Tokens Sybil attack resistant Spam-attack resistant	Permissioned Blockchain Smart Contracts Cryptographic Token Protocol Layer Network Layer Application Layer Interoperability technologies Enterprise system integration On-chain transactions
Should-Have	Golang Private Platform JavaScript Resilience technologies Instant Transaction Finality High Transaction Speed Zero-knowledge Proof High Maturity High Popularity	Turing-complete JavaScript High Maturity Native token Cryptocurrency (purpose) Solidity	Private Blockchain delegated Byzantine Fault Tolerance Delegated Proof-of-Stake Share-like token Security token Network token Network value token Work token Usage token
Could-Have	zk-SNARKS Spam-attack resistant Virtual Machine Turing-complete On-chain transactions Practical Byzantine Fault Tolerance Federated Byzantine Fault Tolerance Delegated Byzantine Fault Tolerance	Proof-of-Work Proof-of-Stake delegated Proof-of-Stake practical Byzantine Fault Tolerance federated Byzantine Agreement delegated Byzantine Fault Tolerance Proof-of-Authority Proof-of-Elapsed Time Public Platform Private Platform Permissioned Platform Permissionless Platform Zero-knowledge Proof zk-SNARKS Hard-fork and Quantum resistant Instant transaction finality Medium Popularity and Innovation High Transaction speed	Privacy Technologies, Virtual Machine, Turing Complete
Won't-Have	Proof-of-Work Proof-of-Stake Directed Acyclic Graph	Directed Acyclic Graph	(None)

knowledge Proofs, and Golang), therefore, the DSS assigned the highest score to this blockchain platform. However, the second DSS feasible blockchain platform is *R3 Corda*, which has higher values for the non-Boolean blockchain feature requirements, such as *Popularity in the market* and *technology maturity*, compared to *JPMorgan Quorum*.

DUO

The case-study participants at DUO ranked *Ethereum*, *NEO*, *Hyperledger* as their three potential blockchain platform solutions. *Ethereum* has gained the highest score among the top-10 DSS feasible blockchain platforms for DUO according to their blockchain feature requirements. *Wanchain* was not on the case-study participant shortlist, but since it is an *Ethereum-based* blockchain platform, its high score is not surprising. Despite *Hyperledger* reaching the second place in the DSS feasible blockchain platforms, it forces DUO to make intensive use of *cryptographic tokens*, so it is not as suitable as the previous two platforms. Moreover, *Hyperledger* does not support *native-tokens*, so it is not a suitable blockchain platform for token-based applications. Also, several blockchain feature requirements with *Should-have* priority are *token-based*. The case-study participants at DUO considered 23 blockchain feature requirements with *Could-have* priority, and *Hyperledger* supports all of them, so *Hyperledger* received the second-highest score among the DSS feasible blockchain platforms.*NEO* does not support all of the blockchain feature requirements with *Should-have* and *Could-have* priories. However, the gap between the calculated scores of *NEO* and *Hyperledger* is not too much.

Veris Foundation

The case-study participants at the Veris Foundation considered *NEO* and *Ethereum* as the first and second potential blockchain platform in their shortlist. *Cosmos Network* has gained the highest score among the DSS feasible blockchain platforms for the Veris Foundation, because *Cosmos Network* is flexible regarding different *pluggable consensus mechanisms* and supports any combinations of *permission/permission-less* and *public/private* blockchain platforms compared to both NEO and Ethereum. *Hyperledger* is a feasible solution once again. However, the same possible difficulties, as in the DUO case study, could arise with a heavy reliance on different *token-types*, which are harder to implement in practice.

4.6.2 Analysis of the results

Table 4.6 states that *Chain* is a feasible blockchain platform for all three case studies, which means that this blockchain platform at least supports all of the blockchain feature requirements with *Must-have* priority and does not support the blockchain feature requirements with *Won't-Have* priority. None of the case-study participants considered *Chain* as a potentially feasible blockchain platform for their company, demonstrating that the DSS can potentially come up with more feasible blockchain platforms than human experts.

Another interesting observation is that the main decision that has to be made is the choice between *permission* or *permission-less* blockchain platforms and whether *cryptographic tokens* are required or not.

Table 4.6: this table presents the outcomes of the DSS for ShareCompany BIQH, DUO, and Veris Foundation based on their blockchain feature requirements' priorities. The columns *DSS Feasible Solutions* and *CP Shortlist* demonstrate the deduced feasible solutions by the DSS and the shortlist of potential solutions by the case-study participants. Moreover, the Columns *CP Rank* and *DSS score* show the order of potential feasible solutions based on the case-study participants' opinions and the calculated scores of the feasible solutions by the DSS.

Case Study	DSS Feasible Solutions	CP Shortlist	DSS Score	CP Rank
ShareCompany BIQH	Hyperledger	✓	99.39	1
	R3 Corda		68.13	-
	JPMorgan Quorum	✓	61.92	2
	Chain		40.05	-
DUO	Ethereum	✓	98.25	1
	Hyperledger	✓	73.22	3
	Wanchain		64.68	-
	NEO	✓	62.10	2
	Cosmos Network		51.10	-
	Stellar		37.91	-
	Komodo		37.65	-
	Waves Platform		37.25	-
	Chain		34.30	-
	VeChain		31.31	-
Veris Foundation	Cosmos Network		99.64	-
	NEO	✓	69.42	1
	Ethereum	✓	54.52	2
	Stellar		53.33	-
	Hyperledger		44.48	-
	Chain		44.48	-
	VeChain		30.27	-
	ICON		28.63	-
	Symbiont		28.16	-
	Neblio		21.37	-

Concerning effectiveness, the case-study participants asserted that the updated and validated version of the decision model is useful and valuable in finding the shortlist of feasible blockchain platforms. Moreover, the DSS reduces the time and cost of the decision-making process. The case-study participants expressed that the DSS enabled them to meet more detailed blockchain feature requirements. Furthermore, they were surprised to find what their primary concerns seem to be, especially when the opinions of different experts are combined.

The validity metric defined as the degree to which an artifact works correctly. There are two ways to measure validity: 1) the results of the DSS compared to the predefined case-study participant shortlist of potentially feasible blockchain platforms, and 2) according to the blockchain experts' opinion.

The case-study participants confirm that the DSS provides effective blockchain platforms to help software-producing organizations in their initial decisions for selecting blockchain platforms. In other words, the DSS recommended nearly the same blockchain platforms as the case-study participants suggested to their companies after extensive analysis and discussions. However, the DSS offers a short ranked list of feasible blockchain platforms; therefore, software-producing organizations should perform further investigations, such as performance testing, to find the best fitting blockchain platform for their software products.

4.7 Discussion

The DSS assists requirements engineers in the requirements elicitation activity by offering a list of prominent blockchain features. Software-producing organizations have different perspectives on their blockchain feature requirements in different phases of the software development life-cycle. Requirement engineers (decision-makers) might want to consider generic blockchain features in the early phases of the life-cycle, whereas they are interested in more technical blockchain features as their development process matures. For instance, *Consensus Mechanism* could be prioritized as a *Should Have* blockchain feature in the design phase, but in the implementation phase, one of its sub-features (more technical blockchain feature), e.g., *Proof-of-Work*, might be selected instead. Furthermore, blockchain features' priorities could be changed in different phases. Therefore, the DSS might come up with various blockchain platforms for a software-producing organization in phases of its software development life-cycle. As the blockchain feature requirements for each *Case Definition* are stored in the knowledge base of the DSS, it does not cost a significant amount of time to rerun the decision-making process. Therefore, the DSS, as a requirements management tool, provides a platform to enable decision-makers to analyze, document, track collaboratively, and prioritize their blockchain features requirements.

Biases, such as motivational and cognitive (Montibeller & Winterfeldt, 2015), arise because of shortcuts or heuristics that decision-makers use to solve problems and perform tasks. The Hawthorne effect, which is the tendency for decision-makers to change their behavior when they are being observed, is a form of cognitive bias. The case-study participants might have been more careful in the observational setting than in the real setting because they are being observed by scientists judging their selected blockchain feature requirements and priorities. Moreover, the Bandwagon effect, which is the tendency to do or believe things because many other decision-makers do or believe the same, is another form of cognitive bias. The Bandwagon effect typically shows up in group decisions. To mitigate the Hawthorne and Bandwagon effects, individual *and* group interviews have been conducted. The DSS provides a discussion and negotiation platform to enable requirement engineers to make group decisions. It detects and highlights the conflicts in the assigned priorities to the blockchain feature requirements by decision-makers and asks them to resolve disagreements. Thus, the DSS supports requirements engineers in the Requirements verification and validation activity by avoiding conflict between blockchain feature requirements and generating feasible solutions according to the blockchain feature requirements. Moreover, the DSS is a communication tool among the decision-makers to facilitate the requirements specification activity.

We define DSS success when it, in part, aligns with the case-study participants shortlist and when it provides new suggestions that are identified as being of interest to the case-study participants. Using the case-study participants' opinion as a measurement instrument is risky, as the case-study participants may not have sufficient knowledge to make a valid judgment. We counter this risk by conducting more than one case study, by assuming that the case-study participants are handling in their interest, and by applying the DSS to other problem domains, where we find similar results (Farshidi et al., 2018a; Farshidi et al., 2018b; Farshidi et al., 2018c).

4.8 Conclusion

Blockchain technology is evolving rapidly, and the number of blockchain platforms in the market is proliferating. Furthermore, software-producing organizations are increasingly considering blockchain technology for inclusion in their products. Therefore, a decision model is required to externalize and organize knowledge regarding the current state of blockchain technology, and assist decision-makers at software-producing organizations with selecting right blockchain platforms based on their preferences and requirements.

In this study, the blockchain platform selection process is modeled as a multi-criteria decision-making problem that deals with the evaluation of a set of alternatives, and taking into account a set of decision criteria (Triantaphyllou et al., 1998). Moreover, we introduced a decision model for the blockchain selection problem based on the technology selection framework (Farshidi et al., 2018a). We have designed and implemented a decision support system for supporting decision-makers with their technology selection problems in software production. The decision support system provides a modeling studio to build such decision models for technology selection problems. The decision models can be uploaded to the knowledge base of the decision support system to facilitate the decision-making process for software-producing organizations. The proposed decision support system addresses common knowledge management issues, including capturing, sharing, and maintaining knowledge.

The novelty of the approach provides knowledge about blockchain platforms to support uninformed decision-makers while contributing a sound decision model to knowledgeable decision-makers. Furthermore, it incorporates deeply embedded requirements engineering concepts, such as the ISO software quality standards and the MoSCoW prioritization technique, besides knowledge engineering theories, to develop the decision support system. We conducted three case studies to evaluate the decision support system's usefulness and effectiveness to address multi-criteria decision-making problems. Our website[6] is up and running to keep the knowledge base of the decision support system up-to-date and valid. We aim to create a community around the platform that will regularly update the curated knowledge base with new blockchain platform features.

Probing more in-depth, the decision model presented in this paper also provides a foundation for future work in multi-criteria decision-making problems. We intend to build trustworthy decision models to address *Software Architecture Pattern* and *Model-Driven Development Platform* selection problems as our (near) future work.

[6]https://dss-mcdm.com

Part II: Software development technology selection problems

CHAPTER 5

Programming Language Ecosystems

Abstract - Software development is a continuous decision-making process that mainly relies on the software engineer's experience and intuition. One of the essential decisions in the early stages of the process is selecting the best fitting programming language based on the project requirements. A significant number of criteria, such as developer availability and consistent documentation, besides potential programming languages in the market, lead to a challenging decision-making process. A decision model is required to analyze the selection problem using systematic identification and evaluation of potential alternatives for a development project. *Method:* Recently, we introduced a framework to build decision models for technology selection problems in software production. Furthermore, we designed and implemented a decision support system that uses such decision models to support software engineers with their decision-making problems. This study presents a decision model based on the framework for the programming language selection problem. *Results:* The decision model has been evaluated through seven real-world case studies at seven software development companies. The case study participants declared that the approach provides significantly more insight into the programming language selection process and decreases the decision-making process's time and cost. *Conclusion:* With the knowledge available through the decision model, software engineers can more rapidly evaluate programming languages. Having this knowledge readily available supports software engineers in making more efficient and effective decisions that meet their requirements and priorities.

keywords- programming language selection; decision model; industry case study; software production; multi-criteria decision-making; decision support system;

This chapter is based on the following publication:

Farshidi, S., S. Jansen & M. Deldar (2020a), “A decision model for programming language ecosystem selection”, *(Submitted)*.

5.1 Introduction

Software engineers make a sequence of design decisions while developing a software product (Ruhe, 2002). Each design decision can be analyzed as an episode of complex problem solving (Pressman, 2005) that relies on a substantial amount of knowledge and rationale. Design decisions in the software development lifecycle are significantly constrained by former decisions and lead to additional constraints on future decisions (Burge et al., 2008). Making informed design decisions in different phases of the software development lifecycle has critical impacts on the success of a software product.

Over the last decades, thousands of programming languages belonging to several programming paradigms have been introduced. Despite the significant number of programming languages, only a few fundamental programming concepts and languages have survived for more than ten years (Vujošević-Janičić & Tošić, 2008). Some languages have grown into extensive software ecosystems (Jansen et al., 2013a), while others have failed to grow beyond their niche or disappeared altogether (Meyerovich & Rabkin, 2013).

No unique programming language is the *best option* for all potential scenarios. Judging the suitability of a programming language for a software product, as an application or a customized component, is a non-trivial task. For instance, a purely functional language like *Haskell* is the best-fit for writing parallel programs that can, in principle, efficiently exploit huge parallel machines working on large data sets (Peyton Jones et al., 2008). However, while developing a dynamic website, a software engineer might consider *ASP.net* as the best alternative, and others might prefer using *PHP* or a similar scripting language. It is interesting to highlight that successful projects have been built with both: StackOverflow is built-in *ASP.net*, whereas Wikipedia is built-in *PHP*. Furthermore, a software engineer might prefer particular criteria, such as scalability in enterprise applications, whereas other criteria, such as technology maturity level, might have lower priorities.

Selecting and employing multiple programming languages in one development project is quite common (Kochhar et al., 2016). For example, a software engineer might code the back-end of a website using *PHP* or *C#* and then use the combination of *HTML*, *CSS*, and *JavaScript* to design layouts and styles of the front-end. Furthermore, leading popular software available in the market are developed in multiple languages. For example, some essential components of the Linux operating system[1] are designed and implemented in *C*; however, the majority of its utilities and applications are built-in *C++*, *Perl*, and *Python*. Comparably, OpenCV4[2], an open-source computer vision and machine learning software library, is developed utilizing an assemblage of programming languages such as *C++*, *C*, *Python*, *Java*, and *JavaScript*.

Acquiring and expanding knowledge about programming languages is a highly complex process, as significant numbers of criteria and alternatives exist in the market (Bhattacharya & Neamtiu, 2011). Various factors need to be taken into account, of which not all are obvious. Simultaneously, the choice of programming languages

[1]https://www.linux.com/

[2]https://opencv.org/opencv-4-0/

can have repercussions on the implementation cost, quality of the result, and maintenance cost of the application (Holtz & Rasdorf, 1988). Some of these consequences may not be felt for years after the initial programming language choice decision has been made.

Each programming language has different characteristics, communities, and ecosystems that should be considered. The selection process is mainly based on surrounding ecosystems and communities. Third-party libraries play an essential role as many software applications are built by gluing together plenty of existing libraries in the market, so such libraries increase language growth. Additionally, communities generate wikis, forums, and tutorials to improve the learnability and understandability of languages.

Nowadays, the development of software products, systems, and services typically results in complex decision models and decision-making processes (Badampudi et al., 2018). Selecting the best fitting programming language(s) for a software project can be modeled as a multi-criteria decision-making (MCDM) problem that deals with the evaluation of a set of alternatives and takes into account a set of decision criteria (Triantaphyllou et al., 1998) (e.g., features of the programming languages).

Knowledge about programming languages is scattered among a wide range of literature, documentation, and software engineers' experience. This study's main motive is to build a decision model to capture knowledge about programming languages and concepts systematically and make it available in a reusable and extendable format. Accordingly, we have followed our framework (Farshidi et al., 2018a) to build such a decision model for the programming language selection problem. The framework and a Decision Support System (DSS) (Farshidi & Jansen, 2020a; Farshidi et al., 2018b) were introduced in our previous studies for building MCDM decision models in software production. The DSS is a platform[3] for capturing MCDM decision models based on the framework. Decision models can be uploaded to the DSS's knowledge base to facilitate software-producing organizations' decision-making process according to their requirements and preferences. The DSS provides a discussion and negotiation platform to enable decision-makers at software-producing organizations to make group decisions. Furthermore, the DSS can be used over the full lifecycle and co-evolve its advice based on evolving requirements.

The rest of this study is outlined as follows: Section 5.2 describes our research method, which is based on design science and exploratory theory-testing case studies. This study has the following contributions:

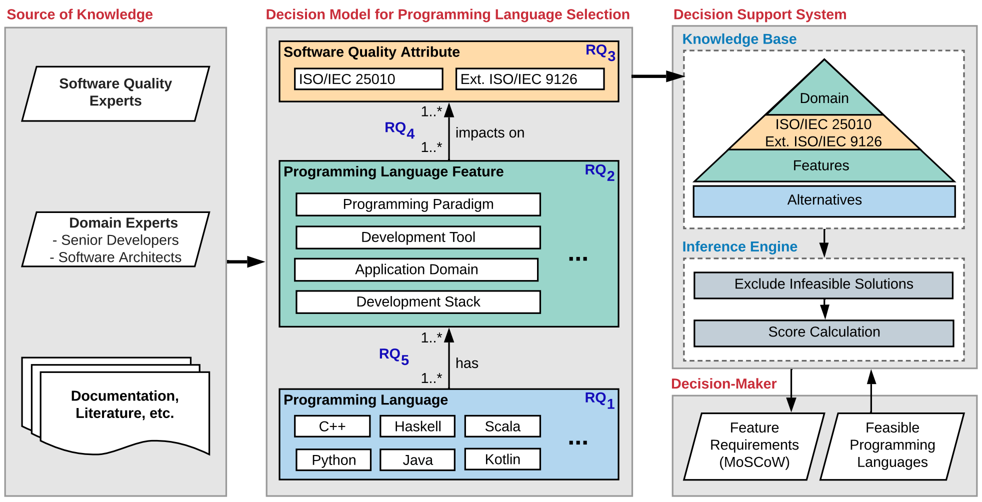
- Section 5.3 explains the integration of the captured tacit knowledge of software engineers through interviews and the explicit knowledge that is scattered in an extensive list of websites, articles, and reports. Acquired knowledge is presented in forms of reusable knowledge that can be used by software engineers in their decision-making process.
- Section 5.4 describes seven conducted case studies, performed in the Netherlands and Iran, to evaluate the effectiveness and usefulness of the decision model.
- Section 5.5 analyzes the results of the DSS and compares them with the case

[3]The decision studio is available online on the DSS website: https://dss-mcdm.com

study participants' ranked shortlists of feasible programming languages. The results show that the DSS recommended nearly the same solutions as the case study participants suggested to their companies after extensive analysis and discussions and do so more efficiently.

Section 5.6 highlights barriers to the knowledge acquisition and decision-making process, such as motivational and cognitive biases, and argues how we have minimized these threats to the validity of the results. Section 5.7 positions the proposed approach in this study among the other programming language selection techniques in the literature. Finally, Section 5.8 summarizes the proposed approach, defends its novelty, and offers directions for future studies.

Figure 5.1: The main building blocks of the framework, adapted from our previous study (Farshidi et al., 2018a), is shown in this figure. On the left, the sources of knowledge, in the middle, the proposed decision model for the programming language selection problem, and on the right, the decision support system is modeled.



5.2 Research Method

Research methods are classified based on their data collection techniques (interview, observation, literature, etc.), inference techniques (taxonomy, protocol analysis, statistics, etc.), research purpose (evaluation, exploration, description, etc.), units of analysis (individuals, groups, process, etc.), and so forth (Meredith et al., 1989). Multiple research methods can be combined to achieve a fuller picture and a more in-depth understanding of the studied phenomenon by connecting complementary findings that conclude from the methods from the different methodological traditions of qualitative and quantitative investigation (Johnson & Onwuegbuzie, 2004).

Knowledge acquisition is the process of capturing, structuring, and organizing knowledge from multiple sources (Gruber, 1989). Human experts, discourse, internal meetings, case studies, literature studies, or other research methods are the primary sources of knowledge. The rest of this section outlines the research questions and elaborates on a mixed research method based on design science research, expert interviews, documentation analysis, and case study research to capture knowledge regarding the programming languages, to answer the research questions, and to build a decision model for the programming language selection problem.

5.2.1 Research Questions

We formulated the following research questions to capture the required knowledge based on the framework (Farshidi et al., 2018a):

RQ_1: Which programming languages should be considered in the decision model?

RQ_2: Which programming concepts should be considered as the programming language features in the decision model?

RQ_3: Which software quality attributes can be utilized to evaluate the programming languages?

RQ_4: What are the impacts of the programming language features on the quality attributes of the programming languages?

RQ_5: Which programming languages currently support the programming language features?

5.2.2 Design science

Design Science is an iterative process (Simon, 1996), has its roots in engineering (Hevner et al., 2004), is broadly considered a problem-solving process (Fortus et al., 2005), and attempts to produce generalizable knowledge about design processes and design decisions. Similar to a theory, the design process is a set of hypotheses that can eventually be proven only by creating the artifact it describes (Walls et al., 1992). However, a design's feasibility can be supported by a scientific theory to the extent that the design comprises principles of the theory.

Recently, we designed a framework (Farshidi et al., 2018a) and implemented a DSS (Farshidi & Jansen, 2020a; Farshidi et al., 2018b) for supporting software engineers (decision-makers) with their MCDM problems in software production. Knowledge engineering theories have been employed to design and implement the DSS and the framework. The framework provides a guideline for decision-makers to build decision models for MCDM problems in software production following the six-step of the decision-making process (Majumder, 2015): (1) identifying the objective, (2) selection of the features, (3) selection of the alternatives, (4) selection of the weighing method, (5) applying the method of aggregation, and (6) decision making based on the aggregation results.

In this study, we applied the framework to build a decision model for the programming language selection problem. The research approach for creating the decision model is Design Science, which addresses research through the building and evaluation of artifacts to meet identified business needs (Hevner et al., 2004). We carried out seven industry case studies in the context of seven software development companies to evaluate the decision model.

5.2.3 Expert Interviews

The primary source of knowledge to build a valid decision model for this work is domain experts. We followed Myers & Newman (2007) to conduct a series of qualitative semi-structured interviews with senior software engineers to explore expert knowledge regarding the programming language selection problem. We developed a role description before contacting potential experts to ensure the right target group. We contacted the experts through email using the role description and information about our research topic. The experts were pragmatically and conveniently selected according to their expertise and experience mentioned on their *LinkedIn* profile. We considered a set of expert evaluation criteria (including *Years of experience*, *Expertise*, *Skills*, *Education*, and *Level of expertise*) to select the experts.

Each of the interviews followed a semi-structured interview protocol and lasted between 60 and 90 minutes. We used open questions to elicit as much information as possible from the experts minimizing prior bias. All interviews were done in person, recorded with the interviewees' permission, and then coded for further analysis.

Fifteen experts (fourteen senior software engineers and one business consultant) participated in this research to answer the research questions and build a decision model for the programming language selection problem. Acquired knowledge during each interview is typically propagated to the next to validate the captured knowledge incrementally. Finally, the findings were sent to the interview participants afterward for final confirmation. Note, for the validity of the results, the research's data collection phases were not affected by the case study participants; moreover, none of the researchers were involved in the case studies.

5.2.4 Documentation analysis

Document analysis is one of the analytical methods in qualitative research that requires data investigation and interpretation to elicit meaning, gain understanding, and develop empirical knowledge (Corbin & Strauss, 2014). To build a decision model for the programming language selection problem, we reviewed webpages, whitepapers, scientific articles, fact sheets, technical reports, product wikis, product forums, product videos, and webinars to collect data. Accordingly, we reviewed 489 unique resources (including webpages, whitepapers, and scientific articles) to map the programming language features to the programming language.

A structured coding procedure is employed to extract knowledge from the selected sources of knowledge. Structured coding captures a conceptual area of the research interest (Saldaña, 2015). The extracted knowledge has been classified into five categories: *quality attributes*, *programming languages*, *programming language features*, *impacts of the programming language features on the quality attributes*, and *supportability of the programming language features by the programming languages*. Afterward, the extracted knowledge was employed to build a decision model for the programming language selection problem. Then, the decision model was uploaded to the knowledge base of the DSS.

5.2.5 Case Study

Case study research is an empirical research method (Jansen, 2009) that investigates a phenomenon within a particular context in the domain of interest (Yin, 2017). Case studies can describe, explain, and evaluate a hypothesis. A case study can be employed to collect data regarding a particular phenomenon, apply a tool, and evaluate its efficiency and effectiveness using interviews. Note, we followed the guidelines outlined by Yin (1981) to conduct and plan the case studies.

Objective: Building a valid decision model for the programming language selection problem, was the main goal of this research.

The cases: The analysis units were seven industry case studies, performed in the Netherlands and Iran, in the context of seven software development.

Methods: We conducted multiple expert interviews with the case study participants to collect data and identify their requirements and preferences regarding the programming language selection problem.

Selection strategy: In this study, we selected *multiple case study* (Yin, 1981) to analyze the data both within each situation and across situations, to more extensive exploring the research questions and theoretical evolution, and to create a more convincing theory.

Theory: The proposed decision model is a valid reference model to support software engineers with the programming language selection problem.

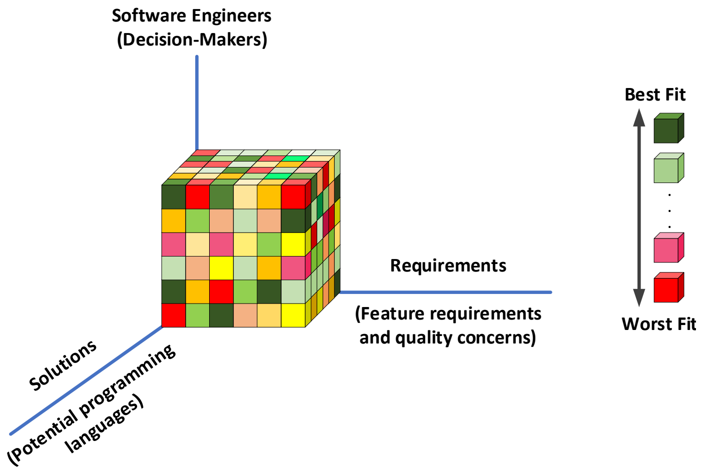
Protocol: To conduct the case studies and evaluate the proposed decision model, we followed the following protocol:

> **Step 1. Requirements elicitation:** The participants defined their programming language feature requirements and prioritized them based on the *MoSCoW* prioritization technique (DSDM consortium and others, 2014). Furthermore, they identified a set of programming languages as potential solutions for their software projects.
>
> **Step 2. Results and recommendations:** We defined seven separate cases on the DSS portal according to the case studies' requirements and priorities. Next, the DSS suggested a set of feasible solutions per case individually. Then, the outcomes were discussed with the case study participants.
>
> **Step 3. Analysis:** We compared the DSS feasible solutions with the experts' solutions at the case study companies had suggested. Moreover, we analyzed the outcomes and our observations and then reported them to the case study participants.

Note, based on the framework, we built decision models for database management systems (Farshidi et al., 2018c), cloud service providers (Farshidi et al., 2018a), software architecture patterns (Farshidi & Jansen, 2020a; Farshidi et al., 2020e), and blockchain platforms (Farshidi et al., 2020c)[4]. Several case studies were conducted to evaluate the DSS's effectiveness and usefulness to address these MCDM problems. The results showed that the decision models could reduce the decision-making time and support the decision-makers with the decision-making process.

[4]The decision models and modeling studio are available on the DSS website: https://dss-mcdm.com.

Figure 5.2: This figure an MCDM approach for the programming language selection problem in a 3-dimensional space. The degree of the decision-makers' satisfaction with solutions (potential programming languages) according to their priorities and preferences (requirements) ranges between the best and worst fit solutions, which is represented by a range of colors from red to dark green.



5.3 Multi-Criteria Decision-Making for Programming Language Selection

Decision theories are widely applied in many disciplines. One example is software engineering (Rus et al., 2003), which has been defined as a continuous decision-making process (Fitzgerald & Stol, 2014). Software producing organizations need to decide whether using their internal development resources (in-house), buying commercial off-the-shelf components, do subcontracting (outsourcing), or whether to use open-source software (Badampudi et al., 2018). A decision problem in software production is not addressed in the same way by all software engineers. Each software engineer has her priorities, tacit knowledge, and decision-making policy (Doumpos & Grigoroudis, 2013); consequently, one software engineer's judgment is expected to differ. Addressing such issues in building decision models in software production forms the focal point of interest in multiple-criteria decision making (MCDM). In this study, we formulate the programming language selection problem as an MCDM problem in software production:

Let $Languages = \{l_1, l_2, \ldots l_{|Languages|}\}$ be a set of programming languages in the market (i.e., *C++*, *Ruby*, and *Python*), and $Features = \{f_1, f_2, \ldots t_{|Features|}\}$ be a set of programming language features (i.e., Supporting threading and Multi-platform) of the programming languages. Each language l, where $l \in Languages$, supports a subset of the set $Features$. The goal is finding the best fitting programming languages as solutions, where $Solutions \subset Languages$, that support a set of programming language feature requirements, called $Requirements$, where $Requirements \subseteq Features$.

MCDM is both an approach and a set of techniques to provide an overall ranking of alternative solutions, from the most preferred to the least preferred solution (Dodgson et al., 2009). Alternative solutions may differ in how they achieve several objectives, and no one alternative solution will be best in achieving all objectives. Besides, some conflict or trade-off is usually evident amongst the objectives; alternative solutions that are more beneficial are usually more costly. Costs and benefits typically conflict, but so can short-term benefits compared to long-term ones, and risks may be higher for the otherwise more beneficial options.

An MCDM approach for the selection problem receives *Languages* and their *Features* as its input, then applies a weighting method to prioritize the *Features* based on the decision-makers' preferences to define the *Requirements*, and finally employs a method of aggregation to rank the *Languages* and suggests *Solutions*. Accordingly, an MCDM approach can be formulated as follows:

$$MCDM : Languages \times Features \times Requirements \rightarrow Solutions$$

Typically, a unique optimal solution for an MCDM problem does not exist, and it is necessary to employ decision-makers' preferences to differentiate between solutions (Majumder, 2015). Figure 5.2 visualises MCDM approach for the programming language selection problem in a 3D space. It shows that the degree of satisfaction of the decision-makers with a suggested solution is fuzzy, which means that the satisfaction degree from a decision-maker perspective may range between completely true (best fit) and completely false (worst fit) (Dvořák et al., 2018), which is represented by a range of colors from red to dark green.

As aforementioned, we follow the framework (Farshidi et al., 2018a) as modeled in Figure 5.1 to build a decision model for the programming language selection problem. Generally speaking, a decision model for an MCDM problem contains decision criteria, alternatives, and relationships among them. Figure 5.1 represents the main building blocks of the framework, including the source of knowledge, the proposed decision model, and the decision support system.

5.3.1 Programming Language Alternatives (RQ_1)

This study only focuses on programming languages used in computer programming to develop software-intensive applications or implement algorithms. Accordingly, we are not interested in domain-specific programming languages such as *Arduino*[5], DOT[6], or *CFML*[7].

To answer the first research question, we identified a set of alternatives, including 594 programming languages, based on a variety of programming language websites and related forums as our initial hypothesis. Next, we reviewed the published surveys and reports from well-known knowledge bases, including *Small place to discover languages in GitHub* (2014), *Developer Survey Results* (2019), *Visualizing Language Migration Over Time* (2017), *Dear Developers: Coding Languages That Will Set You Apart*

[5]Arduino is mainly used to program micro-controllers.

[6]DOT is a development tool that is optimized for the processing of graph-structured data.

[7]The ColdFusion Markup Language (CFML) is a set of tags used in ColdFusion pages to interact with data sources, manipulate data, and display output.

Table 5.1: This table shows the programming languages that were mentioned on at least three sources of knowledge, including experts and well-known websites. This list has been considered as the programming language alternatives in the decision model.

	Agreement	Initial Hypothesis	GitHut Ranks	stackoverflow Ran	Language Migratic	hired.com	infoq.com	codingame.com	jetbrains.com	TIOBE Index	PYPL	hackernoon	Coding infinite	statista	stackify	dzone	Redmonk	IEEE	Domain Expert 1	Domain Expert 2	Domain Expert 3	Domain Expert 4	Domain Expert 5	Domain Expert 6	Domain Expert 7	Domain Expert 8	Domain Expert 9	Domain Expert 10	Domain Expert 11	Domain Expert 12
Python	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C#	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Java	100.00%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C++	96.55%	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
JavaScript	96.55%	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PHP	96.55%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Visual Basic .NET	72.41%	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ruby	68.97%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓
R	65.52%	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓
Swift	65.52%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓
Go	62.07%	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓
HTML / CSS	58.62%	✗	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Objective-C	58.62%	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓
SQL	55.17%	✓	✗	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓
Matlab	48.28%	✓	✓	✗	✓	✗	✗	✗	✓	✓	✓	✗	✗	✗	✓	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓
Kotlin	44.83%	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
Scala	44.83%	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
TypeScript	41.38%	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Rust	37.93%	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Assembly language	37.93%	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓
Perl	34.48%	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Clojure	31.03%	✓	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Visual-Basic	31.03%	✓	✗	✗	✓	✓	✗	✗	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Haskell	31.03%	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
shell	27.59%	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Dart	27.59%	✓	✓	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
F#	27.59%	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
PowerShell	24.14%	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Lua	24.14%	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Julia	20.69%	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Groovy	20.69%	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Elixir	17.24%	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Bash	17.24%	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ASP.net	17.24%	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Delphi	13.79%	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Fortran	13.79%	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Erlang	13.79%	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
D	10.34%	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CoffeeScript	10.34%	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ActionScript	10.34%	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
WebAssembly	10.34%	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Common Lisp	10.34%	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
COBOL	10.34%	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Logo	10.34%	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Scheme	10.34%	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
OCaml	10.34%	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Object Pascal	10.34%	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

(2019), *Programming Languages InfoQ Trends Report* (2019), *Top Programming Languages Rankings* (2019), *Do You Speak Code?* (2019), *The State of Developer Ecosystem* (2019), *TIOBE Index* (2020), *PYPL PopularitY of Programming Language* (2019), *Eight Top Programming Languages and Frameworks of* (2019), *What Stats & Surveys Are Saying About Top Programming Languages* (2019), *Most used programming languages among developers worldwide* (2019), *Look At 5 of the Most Popular Programming Languages* (2019), *The RedMonk Programming Language Rankings* (2019), and *The Top Programming Languages* (2019). Afterward, we conducted a set of expert interviews with twelve experts to gain more insight into the popular and applicable programming languages and to evaluate our findings. It is interesting to highlight that most of the domain experts were familiar with a limited number of the list's programming languages (See the twelve Domain Experts' columns on Table 5.1). To prevent potential biases, we only considered the programming languages mentioned on at least three resources. Finally, we analyzed the data and ended with 47 alternative programming languages mentioned on at least three resources. Table 5.1 shows the complete list of the programming languages that we have selected in the decision model.

Table 5.2: This table shows a subset of the Boolean Features ($Feature^B$), the programming languages ($Languages$), and the BFL mapping, where $BFL : Feature^B \times Languages \rightarrow \{0, 1\}$. Note, we reviewed 489 unique resources (including webpages, whitepapers, and scientific articles) to map the programming language features to the programming language. The entire list of the features and mappings is available on Mendeley Data (Farshidi et al., 2020b).

Boolean Features **Programming Languages**	**Coverage**	Java	Python	C++	C#	C	Scala	Swift	Perl	D	Object Pascal	Ruby	Common Lisp	F#	PHP	OCaml	Lua	JavaScript	Erlang	Go	Fortran	Visual Basic .NET	Clojure	HTML / CSS	Objective-C	TypeScript	Elixir	Visual-Basic	Kotlin	ASP.net	Delphi	Rust	ActionScript	Groovy	Dart	R	Haskell	Julia	Assembly language	Scheme	Matlab	Bash / Shell	CoffeeScript	SQL	COBOL	PowerShell	Logo	WebAssembly
Application Domain																																																
Interactive System	80.85%	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	1	1	1	0
Compiler Design	31.91%	1	1	1	1	1	0	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0
Commercial-Off-The-Shelf (COTS)	8.51%	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Data Base Systems	27.66%	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Distributed Systems	29.79%	1	0	1	1	1	1	0	1	1	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Cloud Computing Applications	36.17%	1	1	1	1	1	0	0	0	1	1	0	0	1	1	1	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
Mobile Applications	65.96%	1	1	1	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0
Web-Based Systems	70.21%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0
Web Services	48.94%	1	1	1	1	1	0	1	1	0	0	1	0	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	0	1	0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
Service-Based Systems	70.21%	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0	0	0	0	0
Plug-and-Play Environment	17.02%	1	1	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Real-Time Systems	23.40%	1	1	1	0	1	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1
File-Sharing Applications	31.91%	1	1	1	1	1	0	1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
Exchange Data And Information	27.66%	0	1	1	0	1	1	0	0	1	1	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	1	0	0	0	0
Banking System	44.68%	1	1	1	1	0	1	1	1	0	0	1	0	1	1	1	0	0	1	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	1	0	0	0
Platforms																																																
Cross platform	89.36%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0
Windows	89.36%	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0
Linux	82.98%	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0
macOS	68.09%	1	0	1	1	1	1	1	1	1	1	0	0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	0	1	1	0	1	0	0	0	0
Docker	72.34%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	0	1	1	0	1	0	0	1	0	0	0	1	0	0	0	0
AWS	61.70%	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Microsoft Azure	55.32%	1	1	0	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	0	0	1	1	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0
IBM Cloud or Watson	34.04%	1	0	0	1	0	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
Kubernetes	38.30%	0	0	0	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1	0	1	0	1	1	0	1	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Android	57.45%	1	1	1	1	1	1	0	0	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
iOS	57.45%	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Web-based	72.34%	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0	0	0	0	0

5.3.2 programming language features (RQ_2)

Domain experts were the primary source of knowledge to identify the right set of programming language features, even though documentation and literature study of programming languages can be employed to develop an initial hypothesis about the programming language feature set. Each programming language feature has a data type, such as *Boolean* and *non-Boolean*. For example, the data types of programming language features, such as the *popularity in the market* and supportability of *Object-oriented programming*, can be considered as *non-Boolean* and *Boolean*, respectively.

The initial set of programming language features was extracted from online documentation of programming languages. Then, a list of essential programming language features was identified during twelve domain expert interviews. Finally, 94 Boolean and 13 non-Boolean programming language features[8] were identified and confirmed by the domain experts.

[8]The entire lists of the programming language features and their mapping with the considered programming languages are available and accessible on the *Programming Language Selection* website (https://dss-mcdm.com)

5.3.3 Software Quality Attributes (RQ_3)

Quality attributes are characteristics of a software product that are intrinsically non-functional. One of the primary concerns of software engineers in the implementation phase of a software product is to satisfy the quality requirements. In other words, the quality of a system is the degree to which the system meets its requirements (functionality, performance, security, maintainability, etc.). It is necessary to find quality attributes widely recommended by other researchers to measure the system's characteristics.

The literature study results confirmed that researchers do not agree upon a set of standard criteria, including quality attributes and features, to evaluate the programming languages (See Table 5.6). Additionally, we realized that the suggested criteria were mainly applied to specific domains to address different research questions. Thus, a set of generic and domain-independent criteria is required to assess programming languages.

The ISO/IEC 25010 (ISO, 2011) provides best practice recommendations on the foundation of a quality evaluation model. The quality model determines which quality characteristics should be taken into account when evaluating a software product's properties. A set of quality attributes should be defined in the decision model (Farshidi et al., 2018a). In this study, we employed the *ISO/IEC 25010* standard (ISO, 2011) and *extended ISO/IEC 9126* standard (Carvallo & Franch, 2006) as two domain-independent quality models to analyze programming language features based on their impact on quality attributes of programming languages. The key rationale behind using these software quality models is that they are a standardized way of measuring a software product. Moreover, they describe how easily and reliably a software product can be used.

The last four columns of Table 5.6 show the results of our analysis regarding the common criteria and alternatives of this study with the selected publications. Let us define the coverage of the i-th selected study as follows:

$$Coverage_i = \frac{CQ_i + CF_i}{C_i} \times 100$$

Where, CQ_i and CF_i denote the numbers of common quality attributes (column #CQ) and features (column #CF) of the i-th selected study with this study, respectively. Furthermore, C_i signifies its number of suggested criteria. The last column (*Cov.*) of Table 5.6 indicates the percentage of the coverage of the considered criteria within the selected studies. On average, 80% of those criteria are already considered in this study.

5.3.4 Impacts of the programming language features on the Software Quality Attributes (RQ_4)

The mapping between the sets *software quality attributes* and *programming language features* has been determined based on domain experts' knowledge. Three domain experts participated in this phase of the research to map the programming language features (*Features*) to the quality attributes (*Qualities*) based on a Boolean

Table 5.3: This table shows the Non-Boolean Features ($Feature^N$), the programming languages ($Languages$), and the NFL mapping, where $NFL : Features^N \times Languages \rightarrow \{H, M, L\}$. Note, this table is mainly the result of the expert interviews and reviewing the sources of knowledge indicated in the *source of knowledge* column.

Non Boolean Features Programming Languages	Java	Python	C++	C#	C	Scala	Swift	Perl	D	Object Pascal	Ruby	Common Lisp	F#	PHP	OCaml	Lua	JavaScript	Erlang	Go	Fortran	Visual Basic .NET	Clojure	HTML / CSS	Objective-C	TypeScript	Elixir	Visual-Basic	Kotlin	ASP.net	Delphi	Rust	ActionScript	Groovy	Dart	R	Haskell	Julia	Assembly language	Scheme	Matlab	Bash / Shell	CoffeeScript	SQL	COBOL	PowerShell	Logo	WebAssembly	Source of Knowledge
Popularity in the market	H	H	H	H	H	H	H	L	L	L	M	L	L	H	L	L	H	L	H	L	M	L	H	M	M	L	L	L	H	L	H	L	M	L	H	L	L	M	M	H	L	L	H	L	M	L	L	Domain Experts
TIOBE Index (January 2020)	16.90%	9.70%	5.57%	5.34%	15.77%	0.37%	1.80%	0.74%	0.82%	0.99%	1.06%	0.00%	0.23%	2.40%	0.00%	0.23%	2.45%	0.00%	0.90%	0.31%	5.28%	0.00%	0.00%	0.92%	0.18%	0.00%	0.83%	0.29%	0.00%	0.99%	0.40%	0.00%	0,61%	0.63%	0.80%	0.19%	0.18%	0.87%	0.20%	0.73%	0.00%	0.00%	1.50%	0.38%	0.00%	0.00%	0.00%	https://www.tiobe.com
LinkedIn (Jobs)	12K+	9K+	4K+	11K+	3K+	8K+	781	543	-	-	1K+	-	6	10K+	2	45	17K+	50	11K+	97	46	65	7K+	459	2K+	123	299	625	5K+	316	6K+	18	473	100	5K+	90	529	1K+	2K+	1K+	103	75	20K+	56	2K+	-	4	https://www.linkedin.com/
GitHub (Repositories)	808K+	833K+	171K+	121K+	114K+	59K+	126K+	27K+	838	14	240K+	4K+	4K+	348K+	7K+	19K+	362K+	11K+	224K+	3K+	454	25K+	218K+	16K+	72K+	26K+	156	66K+	3K+	6K+	62K+	1K+	6K+	14K+	112K+	39K+	9K+	2K+	3K+	42K+	49K+	3K+	5K+	777	25K+	365	189	https://github.com/
Google Trends (mean of the last 5 years)	58.15	66.02	79.48	83.73	78.58	76.24	76.44	62.57	30.67	54.44	75.85	47.97	31.25	75.06	56.64	74.03	84.18	66.69	64.89	70.30	63.16	69.93	73.47	67.19	55.53	60.46	60.76	39.57	65.47	34.27	43.82	32.31	82.16	31.53	81.56	65.84	31.16	74.87	67.66	71.44	16.76	40.11	80.08	68.34	81.48	14.35	11.42	https://trends.google.com/
Reusability	H	H	H	H	H	H	H	H	H	H	H	H	M	H	M	M	H	M	M	M	M	M	H	M	M	M	M	M	M	M	M	M	M	L	M	M	L	M	M	L	L	M	L	L	M	L	L	Domain Experts
Application domains	34	33	34	33	30	19	19	18	20	23	19	20	16	22	16	15	18	16	16	11	12	11	21	17	14	13	15	13	8	10	10	13	7	7	13	9	6	11	12	4	9	10	9	4	9	2	3	BFL mapping
Modular paradigms	5	5	4	4	3	3	3	5	3	3	5	5	3	4	3	4	4	2	3	2	2	2	0	3	2	2	1	2	3	2	3	4	5	2	4	2	4	0	1	4	1	1	0	3	2	1	0	BFL mapping
Maturity level	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	M	M	M	H	M	H	M	H	M	M	M	H	M	M	M	M	M	H	M	M	L	M	H	L	L	L	L	L	L	L	Domain Experts
Boolean Features' coverage	96%	92%	88%	91%	80%	70%	69%	69%	71%	70%	67%	67%	67%	78%	64%	64%	69%	59%	59%	58%	65%	59%	60%	58%	64%	56%	59%	56%	59%	56%	55%	54%	52%	51%	51%	51%	45%	40%	42%	40%	40%	39%	33%	35%	28%	25%	11%	BFL mapping
Comprehensive documentation	H	H	H	H	H	M	H	M	L	L	L	M	M	H	L	L	H	M	H	H	H	M	H	H	L	L	H	H	H	M	H	M	M	H	H	M	M	M	M	H	M	L	H	M	L	M	L	Domain Experts
Google hits	3M+	7M+	1M+	500K+	4M	100K+	400K+	255K	272K	12K+	313K	21K	16K+	371K	8K+	79K+	283K	35K+	527K	76K+	1M+	23K+	458K	208K	10K+	18K+	323K	108K	104K	63K+	179K	59K+	24K+	180K	750K	76K+	45K+	69K+	54K+	141K	20K+	1K+	205K+	61K+	5K+	61K+	199	https://www.google.com/
GitHub (Topics)	3K	2K	25	25	25	411	715	181	1	3	611	6	3	3K	47	402	1K	141	5K	60	1	103	1K	2	393	160	16	501	2	80	392	8	31	122	2	165	75	20	192	156	212	12	989	13	155	2	12	https://github.com/
GitHub (Wikis)	252K	241K	511K	511K	511K	23K	13K	24K	13K	2K	80K	2K	210K	189K	3K	23K	238K	7K	510K	6K	4K	13K	408K	15K	13K	3K	21K	4K	12K	1K	15K	2K	12K	3K	240K	9K	3K	6K	46K	15K	65K	9K	91K	976	12K	1K	651	https://github.com/
Active community	H	H	H	H	H	H	H	H	M	M	H	L	H	H	H	H	H	M	H	M	L	M	M	H	H	M	L	M	M	L	H	L	H	H	H	H	H	M	H	M	H	L	H	L	L	L	L	Domain Experts
stackoverflow	500	500	500	500	500	500	500	500	450	471	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	https://stackoverflow.com/
GitHub (Commits)	22M	19M	336M+	362M+	362M+	1M	1M	6M	47K	40K	4M	53K	76M+	18M	727K	3M	3M	252K	150M+	955K	4K	189K	48M+	982K+	709K	202K	34K	380K	242K	67K	964K	33K	723K	1M	244M+	1M	23M	165K+	17M	373K	6M+	130K	6M	15K	278K	476	327K	https://github.com/
GitHub (Issues)	2M	2M	272K	377K	1M	221K	146K	227K	3K	10K	761K	2K	12K	1M	33K	84K	1M	61K	1M	6K	59	33K	613K	30K	374K	46K	238	48K	98	14K	210K	6K	24K	102K	138K	51K	96K	305	3K	17K	76K	11K	913+	155	39K	1K	2K	https://github.com/
Github (Marketplace)	28	49	603	603	603	5	21	1	-	-	26	-	270	53	3	5	31	1	58	1	-	4	13	2	7	7	-	3	-	-	15	-	1	3	382	1	2	-	1	-	7	1	3	-	-	-	-	https://github.com/
GitHub (Packages)	219	117	5K	5K	5K	21	10	8	5K	-	103	-	1K	156	1	12	884	3	488	-	11	2	751	6	13K	3	15	50	47	1	23	-	3	7	3K	4	-	6	16	-	492	4	94	1	16	3	15	https://github.com/
Availability of developers	H	H	H	H	H	M	M	M	M	L	H	L	L	H	L	L	H	L	H	M	M	L	H	M	M	L	H	L	H	M	L	M	L	M	M	L	L	M	M	H	M	L	H	M	M	L	L	Domain Experts
GitHub (Users)	27K	22K	0	7K	3K	866	2K	3K	7	46	5K	42	0	12K	57	375	47K	804	3K	39	0	368	4K	8	486	310	0	382	25	221	689	13	75	209	2K	395	209	10	12	248	353	9	13K	12	652	2	0	
LinkedIn (People)	9M+	5M+	5M+	3M+	11M+	241K+	574K+	761K+	430K+	24K+	682K+	9K+	13K+	3M+	18K+	85K+	6M+	24K+	4M+	205K+	365K+	20K+	4M+	517K+	258K+	39K+	1M+	99K+	1M+	334K+	95K+	134K+	90K+	235K+	763K+	69K+	24K+	320K+	120K+	3M+	124K+	30K+	10M+	341K+	295K+	94K+	2K+	https://www.linkedin.com/
Scalability	H	H	M	H	M	M	L	L	L	M	H	L	L	H	L	L	H	L	H	L	L	M	H	L	H	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	H	L	L	L	L	L	L	Domain Experts
Distributed Systems	5K	1K+	891	324	805	214	-	-	-	-	173	-	-	-	-	-	813	-	1K+	-	-	-	304	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	https://github.com/
Cloud native systems	652	123	-	51	-	-	-	-	-	49	-	-	-	-	-	-	2K+	-	312	-	-	-	250	-	58	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	176	-	-	-	-	-	-	https://github.com/
Service based systems	202	69	34	34	-	-	-	-	-	-	18	-	-	77	-	-	71	-	-	-	-	-	45	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	25	-	-	-	-	-	-	https://github.com/
Microservices	15K	3K	-	2K	-	-	-	-	-	-	1K+	-	-	1K+	-	-	11K	-	3K	-	-	-	2K	-	1K+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1K	-	-	-	-	-	-	https://github.com/
Component based systems	67	33	71	41	23	-	-	-	-	-	-	-	-	36	-	-	79	-	-	-	-	12	20	-	19	-	-	-	-	-	-	-	-	-	-	-	-	-		-	-	-	-	-	-	-	-	https://github.com/

adjacency matrix[9] ($Qualities \times Features \rightarrow Boolean$). For instance, *support threading* as a programming language feature influences the *Time behavior* quality attribute. The experts believed that about 68% percent of the programming language features have impacts on the following quality aspects of the programming languages:

- **Usability** defines the degree to which a programming language can be used to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use. Moreover, it embraces quality attributes such as *Learnability*, *Operability*, *User error protection*.
- **Cost** denotes the amount of money that a company spends on implementing a

[9]The final Boolean adjacency matrix is available on Mendeley Data (Farshidi et al., 2020b). It is made publicly available to enable other researchers to use it for their research purposes.

software product using a programming language. It includes quality attributes such as *Implementation Cost*, *Platform Cost*, and *Licensing Costs*.

- **Product** defines a set of quality attributes regarding the state or fact of exclusive rights and control over the property. For instance, *Stability*, *Ownership*, and *Guarantees* are part of this characteristic.
- **Supplier** includes a set of quality attributes such as *Reputation* and *Support* of the programming languages.
- **Maintainability** is the degree to which a programming language can be effectively and efficiently modified without introducing defects or degrading existing product quality. It can be supported by a programming paradigm, such as *object-oriented programming*, that encourages *Modularity*, *Reusability*, *Analyzability*, *Modifiability*, and *Testability*.

The acquired knowledge regarding the impacts of the programming language features on the quality attributes was used to calculate the Impact Factors (Farshidi et al., 2018a) that apply in the score calculation of the DSS. The framework does not enforce a programming language feature to present in a single quality attribute; programming language features can be part of many quality attributes. For example, *object-oriented programming* as a feature might connect to multiple quality attributes such as *Functional completeness* and *Interoperability*.

In this study's knowledge extraction phase, we realized some inconsistencies regarding the programming language features' impacts on the quality aspects. For example, one of the experts asserted that when a programming language supports object-oriented programming as one of its paradigms, the programming language's operability and User error protection will be increased. However, the other asserted that object-oriented programming does not have any impact on these two quality aspects. Thus, we used the fuzzy Delphi technique to reach a consensus among experts regarding the impacts of the programming language features on the quality aspects.

5.3.5 Supportability of the programming language features by the Programming Languages (RQ_5)

A Programming language has a set of programming language features that can be either Boolean ($Feature^B$) or non-Boolean ($Feature^N$). A Boolean programming language feature is a feature that is supported by the programming language; for example, supporting the *socket programming*. Additionally, a non-Boolean programming language feature assigns a non-Boolean value to a particular programming language; for example, the *maturity level* of a programming language can be *high*, *medium*, or *low*. Therefore, the programming language features in this study are a collection of Boolean and non-Boolean features, where $Features = Feature^B \cup Feature^N$.

The mapping $BFL : Feature^B \times Languages \rightarrow \{0,1\}$ defines the supportability of the Boolean programming language features by the programming languages. So that $BFL(f,l) = 0$ means that the programming language l does not support the programming language feature f and $BFL(f,l) = 1$ signifies that the language supports the feature. The mapping *BFL* is defined based on documentation of the programming languages and expert interviews. One of the principal challenges is the lack of standard terminology among programming languages. Sometimes different programming languages refer to the same concept by different names, or even worse, the same name might stand for different concepts in different programming languages. Discovering conflicts is essential to prevent semantic mismatches throughout the programming language selection process.

A structured coding procedure is employed to extract knowledge from the selected sources of knowledge.

Table 5.2 shows a subset of the Boolean Features that we have considered in the decision model.

We defined thirteen non-Boolean programming language features, such as Scalability and Popularity in the market, and developer availability. The assigned values to these non-Boolean programming language features for a specific programming language is a 3-point Likert scale (**H**igh, **M**edium, and **L**ow), where $NFL : Features^N \times Languages \rightarrow \{H, M, L\}$, based on several predefined parameters. For instance, the *popularity in the market* of programming languages was defined based on the following four parameters: *TIOBE Index* (2020), *LinkedIn (Jobs)*, *Github (Repositories)*, and the mean of the last five years of *Google Trends*. Table 5.3 shows a subset of the non-Boolean programming language features, their parameters, and sources of knowledge.

5.3.6 Programming language feature Requirements:

The DSS (Farshidi & Jansen, 2020a; Farshidi et al., 2018b) receives the programming language feature requirements based on the *MoSCoW* prioritization technique (DSDM consortium and others, 2014). Decision-makers should prioritize their feature requirements using a set of weights ($W_{MoSCoW} = \{w_{Must}, w_{Should}, w_{Could}, w_{Won't}\}$) according to the definition of the *MoSCoW* prioritization technique. programming language feature requirements with *Must-Have* or *Won't-Have* priorities act as hard constraints and programming language feature requirements with *Should-Have* and *Could-Have* priorities act as soft constraints. The DSS excludes all infeasible programming languages which do not support programming language features with *Must-Have* and support programming language features with *Won't-Have* priorities. Then, it assigns non-negative scores to feasible programming languages according to the number of programming language features with *Should-Have* and *Could-Have* prioritizes (Farshidi et al., 2018a).

Decision-makers specify desirable values, from their perspectives, for non-Boolean programming language feature requirements. For example, a decision-maker could be interested in prioritizing programming languages with the *Maturity level* above average. Therefore, the *Maturity level* above average is considered as a *Should-Have* feature.

5.4 Empirical Evidence: The Case Studies

Seven industry case studies at seven software development companies have been conducted to evaluate and signify the decision model's usefulness and effectiveness to address the programming language selection problem. We selected the case study companies from seven different application domains for increasing diversity in our

Table 5.4: This table shows the feature requirements, based on the MoSCoW prioritization technique (Must-Have (M), Should-Have (S), and Could-Have (C)). Note, The entire list of the features requirements is available on Mendeley Data (Farshidi et al., 2020b).

	Oceaneering	Dooman ltd.	Saanaa DP	ASC	FinanceComp	SecureSECO	ENVRI-FAIR	
Debugger	M	M	M	M	M	M	M	R01
Object-oriented programming (OOP)	S	M	M	M	M	M	M	R02
Testing tools	M	C	M	M	M	M	M	R03
Socket programming	M	M	C	M	M	M	M	R04
Support threading	M	S	M	M	S	M	M	R05
Scalability	M	M	M	M	S	M	S	R06
Popularity in the market	M	S	M	M		M	M	R07
Maturity level	M	M	M	S	M		M	R08
Open source compiler or Interpreter	M	M	C	C	M	M	M	R09
Web-based		M	M	M	M	C	M	R10
General-Purpose PL	M	M	S	M	M	S	S	R11
Web-Based Systems	S	M	M	M	M	S	S	R12
Web Services	S	M	C	M	M	S	M	R13
Windows	S	M	M	M	M			R14
Free implementation of the core libraries	M	C	C	C	M	M	M	R15
Cross platform / Multiplatform	M	M	C			M	M	R16
Open source	M	M	C			M	M	R17
Back-end		M	C	M	M	M		R18
Full-stack		M	C	M	M		M	R19
Event-driven programming	M	M	M	S	S	S	S	R20
Comprehensive consistent documentation	M	M		M	S	S	S	R21
Software Architecture Patterns	M	M	M	S	C	S	C	R22
Package Manager	S	S		M	C	M	M	R23
Database Systems	S	M	M	M		S	C	R24
Toolchain	M	M	C	S		M	C	R25
Reusability	M	S		M			M	R26
Compiler	M	M	M	C		C	C	R27
Code coverage	M	C		M		M		R28
Front-end		M	C	M	M			R29
Imperative programming	M			M		M	S	R30
Maintainability	M	S	S	S	M		S	R31
Software Architecture Design Patterns	M	M	S	S	C	S	C	R32
Static code analysis	M		C	M	S	S	C	R33
Generic programming	S	M		M	S			R34
Human Resource Availability (Developers)		M			M	S	S	R35
Profiler	M	C	C	M		S	C	R36
Compiler Design	M	M				S		R37
Linux	M	M	C	C				R38
Interactive System	M	M	C	C			S	R39
Easy to write new code	S	S	M	S		S	S	R40
Easy to read existing code	S	M	S	S		S	S	R41
Easy to reuse existing code	S	M	S	S		S	S	R42
Service-Based Systems	C	C	S	M		S	S	R43
Functional programming	M		C	S	S	S		R44
Accessible and friendly community	M	C		S		S	S	R45
Docker	M	C	C	S		C		R46
Code refactoring	C	C		M		S		R47
Reflective programming	C	M		S			S	R48
Network and Communication Systems		M				S		R49
Banking System			S		M			R50
Amazon Web Services (AWS)	C	C		M		C	C	R51
Declarative programming	C			M				R52
Licensed				M				R53
Operating Systems		M						R54
ORM	C	S	C	S	S	S	S	R55
Self-documenting (or self-describing) syntax	S	S	C	S		S	S	R56
Simple and concise syntax	S	S		S		S	S	R57
Procedural programming	C			S	S	S	S	R58
Distributed Systems	S	C				S	S	R59
Mobile Applications	C	S	S				S	R60
Cloud Computing Applications	C	S	C	C		S	C	R61
Aspect-Oriented Programming (AOP)	C					S	S	R62
Data-Dominant Software						S	S	R63
Data-driven programming	C		C			S		R64
Commercial-Off-The-Shelf (Cots)		C		C			S	R65
GUI builder	C	C	C	C		C	C	R66
Interpreter	C	C	C	C		C	C	R67
Dynamic programming	C			S				R68
macOS	S	C						R69
Information Management and DSSs				C			S	R70
Pattern Recognition	C					S		R71
Bytecode	C		C	C		C	C	R72
Array programming							S	R73
Embedded Systems	S							R74
Real-Time Systems	S							R75
Microsoft Azure	C	C				C	C	R76
IBM Cloud or Watson	C	C				C	C	R77
Kubernetes	C	C		C		C		R78
Constraint programming	C			C		C		R79
Google Cloud Platform	C					C	C	R80
Android	C	C	C					R81
iOS	C	C	C					R82
Flow-Based Programming (FBP)	C			C				R83
Heroku	C					C		R84
Plug-and-Play Environment		C	C					R85
File-Sharing Applications	C			C				R86
Exchange Data And Information	C			C			C	R87
Metaprogramming	C							R88
Non-structured programming	C							R89
Arduino	C							R90
Multi-Processors Environment		C						R91
Expert System				C				R92
Management Information Systems				C				R93
Internet Of Things (Iots)	C							R94

evaluation, including control systems, content management systems (CMS), helpdesk systems, booking systems, financial systems, distributed ledgers, and search engines. Moreover, the selected case study companies were located in two different countries, namely Iran and the Netherlands.

The case study participants have identified a shortlist of ranked feasible programming languages (Table 5.5), as their potential solutions, for the **back-end** of their projects through multiple internal expert meetings and investigation into programming languages before participating in this research. The experts at the case study companies specified their programming language feature requirements based on the MoSCoW prioritization technique (Table 5.4), so seven industry cases were defined and stored in the knowledge base of the DSS [10]. Next, the Inference Engine of the DSS generated feasible solutions for each case. The rest of the section describes the case study companies' ranked shortlists and analyzes the DSS outcomes.

5.4.1 Case Study 1: Oceaneering

Oceaneering AGV Systems work on logistics and Automated Guided Vehicle (AGV) technology, widely used for transporting materials in industry and commerce. An AGV system is a portable robot that follows along marked long lines or wires on the floor or uses radio waves, vision cameras, magnets, or lasers for navigation.

One of the Oceaneering's branches is located in the Netherlands and is mainly active in developing, implementing, and marketing AGV Systems. They specialize in providing mission-critical mobile robotics solutions for material handling applications involving mixed fleets deployed globally in the automotive and manufacturing sectors.

The Oceaneering experts designed a centralized control system that collects data from several field devices and transmits control instructions. The system is responsible for fleet and traffic management and various logistics functions, including order fulfillment. The system is customized to meet specific requirements related to defining vehicle traffic rules, presenting performance data, and optimizing battery consumption.

Requirements

The case study participants defined the following subset of requirements of the control system (for more detail, see Table 5.4):

- The system is expected to coordinate multiple AGVs to guarantee that no collisions occur while tasks are performed. The system must dispatch AGVs so that the quickest cycle time is achieved. Accordingly, Socket programming (R04) and support threading (R05) are two Must-Have features from their perspective.
- The system's main architecture design is based on a centralized management system to coordinate AGVs from a single point. Thus, the potential programming languages must have support predefined Software Architecture Patterns (R22) and Design Patterns (R32). Moreover, supporting functional programming (R44), free implementations of the core libraries (R15), and a wide range of Package Managers (R23) facilitate the development phase of a centralized management system.

[10]The industry cases are available on the DSS website: `https://dss-mcdm.com`

- A real-time or semi-real-time data processing unit is required for data streams, such as collection, classification, storage, and analysis of various event messages output from multiple sources (R20, R74, R94, and R75).
- A mobile app. can be used for real-time handling of on-site tasks related to composition, opening, malfunction, and tests for home site response (R39, R60, R81, and R82).
- The system architecture supports security monitoring and behavioral analysis. Additionally, it supports the development of data-driven systems based on collecting and processing security-related data to assess risks, identify and visualize threats, and produce alerts, among other security services (R13, R64, R43, and R61).
- As maintainability (R31), reusability (R26), and scalability (R06) were part of the quality concerns of the experts, they were looking for highly popular (R07) and mature programming languages (R08).

Results

The case study participants at Oceaneering have considered four potential programming languages (including *C++*, *C*, *Java*, and *Python*) to implement the centralized control system based on 77 programming language feature requirements (see Table 5.5). Although more than 60% of the feature requirements for this case study were Could-Have and Should-Have features (soft constraints), a significant portion of their features prioritized as Must-Have features (almost 39%). Accordingly, the DSS excluded 41 infeasible solutions and ranked the rest of the programming languages (including *Java*, *C#*, *C++*, *Python*, *C*, and *PHP*) based on the soft constraints.

The experts were looking for programming languages that can be used in *Web Services* and *Web-Based Systems*, so they prioritized these features as Should-Have. The DSS suggested *Java* and *C#* as two better alternative solutions for these application domains. The experts at Oceaneering were looking for programming languages that can be employed in *Embedded Systems*, so they considered *C* and *C++* as two alternatives. As it was a Should-Have feature for them, the DSS did not exclude *PHP* as an infeasible solution but scored it lower than the other potential solutions (64%).

5.4.2 Case Study 2: Author-it Software Corporation (ASC)

Author-it Software Corporation (ASC) is enterprise software for authoring, content management, publishing, and localization. ASC centralizes the content creation process, writing in components, and storing the pure content information in a database. Author-it supports the assembly and generation of this information into various documents to be published to an array of outputs. Author-it can be used for documenting Pharma & Biotech, Medtech, Technical Publications and Training & eLearning. One of the branches of ASC is located in the Netherlands.

The experts at this case study design and implement cloud-based component authoring solutions for collaborative content development and multi-channel publishing. The platform enables organizations to author, share, and reuse information across multiple forms of content for critical business needs. Author-it Honeycomb is the latest version of ASC's responsive *HTML5* output for delivering eLearning, mobile learning, and assessments on desktops, tablets, and smartphones.

Requirements

The experts at ASC specified the following subset of requirements of their system (for more detail, see Table 5.4):

- The system architecture is designed based on the service-oriented and architecture (R43) and object-oriented design (R02), so potential programming languages should support software architecture design patterns (R32) and predefined software architecture patterns (R22).
- The system is a Software-as-a-Service solution and enables content creation in a web browser (R61, R10, and R12).
- Author-it enables users to publish single-source content to other content formats, such as PDF, Word, and PPT (R86, and R87).
- Author-it supports translations with its Localization Manager (R67 and R70).
- The content manager has various editors, such as image and text editors (R65).
- The system supports permission management and version control (R93 and R24).
- Declarative programming (R52), Dynamic programming (R68), Imperative programming (R30), and Functional programming (R44) are essential programming paradigms for implementing the system.
- The case study participants were looking for languages that can be used to code, build, run, test, and debug software for cloud platforms, such as Amazon Web Services (R51).
- The system requires a highly integrated and consumerized user experience with consistency across all devices that can be provided by a web-based user interface (R12, R39, and R66).
- The popularity in the market (R07), Reusability (R26), Maturity level (R08), and Scalability (R06) are the main quality concerns of the experts at ASC when they want to select potential programming languages.

Results

The case study participants at ASC came up with *C#*, *VB.Net*, *TypeScript*, and *JavaScript* as four main programming language alternatives to building the content management system. The experts defined 65 programming language feature requirements and assigned Must-Have priority to almost 45% percent of them. The DSS results were similar to the case study participants and suggested *Java* and *Python* as two equal alternatives to *C#*.

The "popularity in the market", "Reusability", "Maturity level", and "Scalability" of the suggested solutions had differed from each other, so the DSS scored them differently. The experts were looking for programming languages that were employed mainly in *Cloud computing applications*. Accordingly, the DSS ranked *C#*, *Java*, and *Python* higher than *JavaScript*, *Visual Basic.net*, and *TypeScript*.

5.4.3 Case Study 3: Dooman ltd.

Dooman ltd. is an Iranian software development company implementing and maintaining a help desk system called *Gamma*. Gamma is a suite of tools that enables organizations to provide information and support customers with concerns, complaints, or inquiries about their products or services. Gamma unifies queries from various customer-facing support channels, such as live chat, email integration, web contact forms, phone, mobile, and social media.

The experts at this case study company designed and implemented the system based on the Multitier and Model–View–Controller (MVC) software architecture patterns. Additionally, the system is deployed and maintained on a private cloud. The experts are mainly interested in the Microsoft ecosystem, so currently, Gamma is developed based on Microsoft technologies, such as *.NET Framework* and *SQL Server*.

Requirements

The experts at this company indicated the following subset of requirements of their system (for more detail, see Table 5.4):

- It should be possible for a user to log an incident by sending an email which will generate a new incident reference and be added to the queue. This will ensure that all questions sent by email to the Help Desk inbox, no matter how small, will be logged and tracked, and no-one will have to monitor the Service Desk inbox manually (R13, R39, and R49).
- Gamma converts all emails from customers to tickets and facilitates ticket management (R37).
- Gamma automates the ticket assignment process and guarantees all customer requests will be replied to within one business day (R20).
- The system needs a reporting system to track team performance, customer satisfaction, and identify potential bottlenecks (R65).
- Gamma must quickly generate ad hoc reports, reporting on any fields, including text strings (R24).
- The system should be able to differentiate between (1) a new incident and (2) an update to an existing incident (e.g., by auto-detecting the incident reference number in the subject line/body of the email). If the latter, it should be added as a note to the relevant incident (R48).
- It must be possible for end-users to submit suggestions via the portal, which can then be reviewed and added to the frequently asked questions' list if appropriate (R12).
- The experts modeled their system based on object-oriented design, so Object-oriented programming is an essential paradigm for them (R02).
- Gamma needs to link an article in the knowledgebase to an incident (e.g., to indicate that all actions in that article were attempted, possibly with a method to indicate with a single-click (e.g., tick/cross) which actions were successful/unsuccessful (R67 and R39).
- The system must be accessible through a dedicated web client (i.e., with no client software installed on the PC, even behind the scenes) (R10).
- Gamma must be integrated with Change Management software (for generating Change Requests, etc.) (R04).
- In order to implement their web-based solution, the experts preferred to employ popular and mature enough programming languages (R07 and R08).
- The experts believed that such programming languages have consistent, comprehensive documentation, and typically their communities are more friendly and accessible (R21 and R45).

- The experts mentioned that simplicity in writing, reading, and reusing codes are vital factors (R40, R41, R42).

Results

The case study participants selected *C#*, *PHP*, and *Python* as their potential solutions before participating in this research. Next, they identified 65 feature requirements based on the MoSCoW prioritization technique. More than half of those features have been prioritized as Must-Have features, so the DSS excluded 43 alternatives and suggested top-4 alternative solutions.

Table 5.5 shows that the DSS offered *Java* as an alternative that was scored equal to *C#* and *Python*. According to our research documenting analysis phase, we realized that *PHP* was not used as a tool to implement Commercial-Off-The-Shelf (COTS) components, and it cannot be used to develop MacOS-based applications. As both of these features prioritized as Could-Have, the DSS did not exclude *PHP* from the set of feasible solutions; however, *PHP* gained the lowest score among the other feasible solutions.

5.4.4 Case Study 4: Saanaa DP

Saanaa DP is specialized in designing and building web-based systems for a variety of customers in Iran. The experts at Saanaa DP mainly determine their clients' requirements and goals and then provide an estimate of the cost to create web applications. They are also active in hosting their websites and debugging any problems. They redesign websites for pre-existing clients, as well as new clients. One of their customers requested an online ticket reservation and hotel booking system, so they looked for the best fitting programming language to implement a web-based solution.

The backbone of the booking system architecture is designed based on the MVC and Client-Server software architecture patterns. Typically, the experts at Saanaa DP deploy their web applications on public cloud providers. The majority of their websites are currently developed based on Microsoft technology, so their initial choice for selecting a programming language ecosystem is *C#.net*.

Requirements

The experts at Saanaa DP defined the following subset of requirements for the booking system (for more detail, see Table 5.4):

- The booking system needs to have a user-friendly interface (R66 and R39).
- The system should show up-to-date availability and immediate price quotations, request any information on the booking form, handle cancellations, modifications, and set up automatic confirmations (R24 and R20).
- The system should be able to handle the creation and delivery of invoices to clients. Additionally, the system should accept deposits or full payments, whether optional or obligatory, processed securely by one of the partner payment gateways (R43 and R50).
- The system must send a booking confirmation email after successful payment (R13).
- The case study participants stated that object-oriented programming (OOP) and data-driven programming are mainly considered as programming paradigms at their company to model ticket reservation systems (R02 and R64).

- The potential programming languages for implementing the system have to support multithreading to simultaneously handle multiple tasks (R05).
- The system should support almost all popular web browsers, such as Internet Explorer, Safari, Chrome, and Firefox (R39, R19, and R29).
- The case study participants preferred to select programming languages that have been employed to implement web-based and transaction processing systems (R12 and R50).
- The potential programming languages must be mature enough and trendy in the market because they have comprehensive documentation and friendly communities (R07, R08, R21, and R45).

Results

As the software engineers of Saanaa DP depend heavily on Microsoft technology, they have selected *C#* as their primary choice for their customers. The case study participants defined 49 feature requirements and prioritized almost 70% of them as soft constraints (Should-Have and Could-Have features). Thus, the DSS had to suggest more feasible solutions; however, they assigned the Must-Have priority to several particular features, such as web-based systems and supporting predefined software architecture patterns supported by a limited list of programming languages (see Table 5.4).

The DSS offered *C#*, *Python*, and *Java* as three almost equal alternatives, besides *Object Pascal* and *Go* as two potential solutions with lower scores. During the data collection phase to build the decision model, we did not find any evidence that shows *Go* and *Object Pascal* can be employed in *transaction-based systems (banking Systems)*.

5.4.5 Case Study 5: FinanceComp

FinanceComp is an Iranian financial institution that provides personal loans, commercial loans, and mortgage loans; moreover, it allows financial transactions at its branches. Additionally, it provides an internet banking system to help customers view and operate their respective accounts through the internet.

The software architecture of the system is based on the Multitier and Client-Server architecture patterns. Security is one of the main quality concerns of the software engineers in the case study; accordingly, they employed multi-tenant databases to store financial transactions. Furthermore, the system is deployed and maintained on a private cloud.

Requirements

The experts at FinanceComp defined the following subset of requirements for the system (for more detail, see Table 5.4):

- Standard data classifications (definition and formats) should be established and used for recording financial events (R50).
- Internal controls over data entry, transaction processing, and reporting should be applied consistently (R20 and R50).
- The system must provide timely and useful financial reports to support managers (R13).
- The system should define, maintain, and execute the posting and editing rules for processed transactions (R50 and R44).

- The system should provide and maintain online queries and reports on balances separately for the current and prior months (R50).
- The case study participants highlighted that *Object-oriented programming* and *Event-driven programming* are the essential programming paradigms that they have used to model the system to incorporate the advantages of modularity and reusability (R02 and R20).
- The system should generate an audit log that identifies all document additions, changes, approvals, and deletions by users (R18).
- The experts mentioned that the potential programming languages have to support *socket programming*, enabling them to exchange information between processes across the network (R04).
- The programming languages should facilitate their software architecture implementation and assist them with employing software architecture design patterns (R22 and R32).
- *Maintainability*, *scalability*, and *security* were part of their quality concerns, so that they preferred to hire highly mature programming languages (R08, R06, and R31).
- The developers' availability besides their current programming knowledge and experience is a vital factor that has profoundly impacted their decision-making process (R35).

Results

The case study participants at FinanceComp stated that their system's former implementation was based on *Java*, and now they want to consider *C#* as another alternative solution for implementing the system. They defined 30 programming language feature requirements and assigned Must-Have priority to 60% percent of them. Thus, they were looking for a limited set of programming languages with unique capabilities.

The complexity of programming languages was not an issue for them; however, they wanted to select a language that can hire enough senior developers to work with. Finally, the DSS concluded that *C#*, *Java*, *Python*, and *PHP* could be used as the most suitable alternatives to their case.

5.4.6 Case Study 6: SecureSECO

SecureSECO is a research organization in the Netherlands. The researchers at SecureSECO work in close collaboration with Utrecht University and the Delft University of Technology. The goal of the research group is to secure and increase trust in the software ecosystem by using distributed ledger technology and empirical software engineering research.

The Software Heritage Graph (SHG) is a database that contains 8 billion source code files that have been collected from the worldwide software ecosystem. This archive is a treasure trove, but it is a big challenge to extract value from the SHG. The researchers at SecureSECO propose the SearchSECO, a hash-based index for code fragments that enables searching source code at the method level in the worldwide software ecosystem. They want to create a set of parsers that extract fragments (methods) from the code files and make them findable.

Requirements

The experts at SecureSECO defined the following subset of requirements for developing the SearchSECO (for more detail, see Table 5.4):

- An extensible data structure, a meta-model representing the relevant entities for SecureSECO, is needed (R55).
- The system needs a parsing and extraction architecture that enables the rapid extraction of code file methods, including the call graph in a project (R67).
- The system will initially use parsers for *Java*, *C*, *JavaScript*, and *Python* (R44).
- SearchSECO requires a project data extractor to collect data about the code fragments, such as author data and version data (R24).
- The system needs several repository spiders that collect source code and project data from different repositories (R43).
- SearchSECO has multiple generic extraction techniques to extract code from languages for which it does not have parsers available individually (R63).
- The system needs smart hashing techniques for hashing the abstract syntax tree of code clones and for hashing code fragment meta-data (R64).
- SearchSECO requires a search API that enables one to search through the SecureSECO ledger rapidly (R59).
- a Job distribution architecture, Worker nodes that can perform the SecureSECO maintenance jobs, is required (R13 and R59).
- SearchSECO needs a data collection dashboard that shows the current state of the SearchSECO platform's growth (R71 and R20).

Results

Recently, the SecureSECO organization experts designed the SearchSECO architecture, so it has not been implemented yet. The case study participants identified 59 programming language feature requirements and prioritized more than 70% of them as soft constraints (Could-Have and Should-Have) features based on their assumptions at the current stage of the software development life cycle. Accordingly, they have not limited themselves to a specific technology or third party vendor. The SecureSECO experts indicated *Python*, *Rust*, *Java*, *C#*, and *C++* as their top-5 potential programming languages.

We did not find any evidence showing that *Rust* supports a toolchain and a code coverage tool during this study's data collection phase. Accordingly, the DSS excluded *Rust* from the ranked shortlist of solutions because the case study participants prioritized these two features as two Must-Have feature requirements. Based on the feature requirements, the DSS suggested to employ *Java*, *C#*, *Python*, *PHP*, *C++*, and *JavaScript* as potential solutions. As aforementioned, each software-intensive project can be implemented by employing a set of programming languages; for instance, the case study participants, after getting feedback from us, asserted that SearchSECO could be developed using a combination of *Python* and *JavaScript* programming languages.

5.4.7 Case Study 7: ENVRI-FAIR

The Environmental Research Infrastructure (ENVRI) community is a community of Environmental Research Infrastructures, projects, networks, and other diverse stakeholders interested in environmental Research Infrastructure matters. The overarching goal of ENVRI-FAIR is for all participating Research Infrastructures to improve their FAIRness and prepare the connection of their data repositories and services to the European Open Science Cloud. With the development of FAIR implementations from the participating Research Infrastructures and integrated services among the environmental subdomains, these data and services will be brought together at a higher level (for the entire cluster), providing more efficient researchers' services and policymakers.

One of the deliverable projects in ENVRI FAIR context is Open Semantic Search Engine (OSSE), which is a platform for building own Search Engine, Explorer for Discovery of extensive document collections based on Apache Solr or Elasticsearch open-source enterprise-search and Open Standards for Linked Data, Semantic Web, and Linked Open Data integration. In this case study, the experts were interested in various OSSE functionality, such as importing and indexing linked data from semantic knowledge graphs for full-text search and faceted search.

Requirements

The experts of the ENVRI-FAIR project defined the following subset of requirements for developing the OSSE (for more detail, see Table 5.4):

- As the most common type for knowledge storage, representation, reasoning, RDF's support is the core requirement in the design and development of the knowledge base of the search engine. This requirement can include the following specific options, such as RDF import/export, RDF storage, owl import, and SPARQL support (R10, R24, R30, and R70).
- An interface for search and discovery of knowledge base content should be provided. This could be the conventional keyword-based search or faceted search. Rather than strict adherence to a single controlled vocabulary or keyword set, a semantic search function is further expected to permit search based on 'similar' or 'related' terms (R39 and R61).
- Due to the variance of source types in the ENVRi community, various methods should be supported for knowledge acquisition, like form-based manual RDF ingestion, Questionnaire-based RDF triple generation, existing RDF integration, structured and unstructured information transformation, etc. Specific measures should be considered to facilitate non-technical users adding knowledge in a straight-forward way (R48, R59, and R87).
- A graph/network analysis view can provide a visualization of the direct and indirect relations, connections and networks between named entities like persons, organizations or main concepts which occur together in your content, data sources, and documents or are connected in your Linked Data Knowledge Graph (R65, R66, and R19).
- Considering the typical case where multiple users contribute to the knowledge base, provenance is of fundamental importance. This primarily refers to tracking individual additions, deletions, and updates and their administration, i.e., approval, rejection, reversion (R05 and R20).

Table 5.5: this table presents the context of the case study companies (Context), the feature requirements (Requirements), the case study participants' ranked shortlists (CP ranked shortlists), and the outcomes of the DSS for the case studies based on their requirements and priorities (DSS Solutions). Moreover, the numbers of features requirements (#Feature Req) and the percentages of the MoSCoW priorities are shown in the table. Note, the numbers in percentages beside the solutions signify the calculated scores by the DSS. For instance, the score of the *C* programming language for *Oceaneering* is 78% (see (Farshidi et al., 2018a) for the details).

		Oceaneering	ASC	Dooman ltd.	Saanaa DP	FinanceComp	SecureSECO	ENVRI-FAIR
Context	App. Domain	Control systems	CMS	Helpdesk systems	Booking systems	Financial systems	Distributed ledgers	Search engines
	#Employees	800-1000	101-250	50-80	20-50	4500-5000	20-50	101-250
	Country	Netherlands	Netherlands	Iran	Iran	Iran	Netherlands	Netherlands
Requirements	Must-Have	38.96%	44.62%	50.77%	30.61%	60.00%	27.12%	27.59%
	Should-Have	20.78%	27.69%	16.92%	16.33%	30.00%	50.85%	44.83%
	Could-Have	40.26%	27.69%	32.31%	53.06%	10.00%	22.03%	27.59%
	#Feature Req.	77	65	65	49	30	59	58
CP ranked Shortlists	1.	C++	C#	C#	C#	C#	Python	PHP
	2.	C	VB.net	PHP	-	Java	Rust	Python
	3.	Java	TypeScript	Python	-	-	Java	Java
	4.	Python	JavaScript	-	-	-	C#	C#
	5.	-	-	-	-	-	C++	-
DSS Solutions	1.	Java 99%	C# 99%	C# 99%	C# 99%	C# 100%	Java 90%	Java 94%
	2.	C# 92%	Java 99%	Python 99%	Python 99%	Java 100%	C# 89%	C# 93%
	3.	C++ 84%	Python 99%	Java 99%	Java 99%	Python 100%	Python 88%	PHP 92%
	4.	Python 84%	JavaScript 78%	PHP 85%	Object Pascal 87%	PHP 100%	PHP 84%	Python 91%
	5.	C 78%	VB.Net 72%	- -	Go 87%	- -	C++ 71%	JavaScript 75%
	6.	PHP 64%	TypeScript 65%	- -	- -	- -	JavaScript 65%	Ruby 72%

Results

The case study participants stated that after analyzing the requirements of the OSSE, they investigated the potential open-source tools that can be used and customized to meet the requirements. After performing an extensive evaluation, they selected an open-source tool called open semantic search. The backend of the tool was implemented in *PHP* and *Python*. Accordingly, the first two solutions for the case study participants were these programming languages. However, they assumed that *Java* and *C#* could be employed in developing additional components of the OSSE.

More than 70% of the feature requirements were prioritized as soft constraint features so that the DSS could offer a broader list of alternative solutions. The DSS results showed that besides the case study participants' solutions, *JavaScript* and *Ruby* could be considered as two more options. Scalability, popularity, and maturity of the programming languages were prioritized as three Must-Have features, so the DSS prioritized *Java* and *C#* higher than the offered solutions.

5.5 Analysis of the Results

The validity metric is defined as the degree to which an artifact works correctly. There are two ways to measure validity: (1) the results of the DSS compared to the predefined case-study participant shortlist of potentially feasible programming languages, and (2) according to the domain experts' opinion.

Concerning effectiveness, the case study participants asserted that the updated and validated version of the decision model is useful and valuable in finding the shortlist of feasible programming languages. Moreover, the DSS reduces the time and cost of the decision-making process. The case study participants expressed that the DSS enabled them to meet more detailed programming language feature requirements. Furthermore, they were surprised to find their primary concerns, especially when different experts' opinions are combined.

The DSS suggests that *C#*, *Java*, and *Python* can be feasible solutions for all seven case studies (see Table 5.5), which means that these programming languages support all of the features with *Must-have* priority. It makes sense as these programming languages are in the top-5 list of popular solutions in the market (see Table 5.3); moreover, their maturity levels are relatively high, as they support most of the programming language features that we have considered in this study (see Tables 5.2).

Scalability and maturity of the programming languages were two key quality concerns of the case study participants (see Table 5.4) so that they considered at least one of the top-5 programming languages as their potential solutions. Table 5.5 represents that the DSS can come up with more feasible programming languages than human experts.

Table 5.4 shows that supporting *Debugger*, *Object-Oriented Programming*, *Testing tools*, *Socket programming*, *threading*, *Scalability*, *Open source compiler or Interpreter*, *General-Purpose PL*, *Web-Based Systems*, *Web Services*, *Free implementation of the core libraries*, *Event-driven programming*, *Software Architecture Patterns*, *Software Architecture Design Patterns*, and *Object–Relational Mapping (ORM)* were programming language features that all of the case studies assigned priorities to them and defined them as their programming language feature requirements. All of the case study participants somehow declared that the Object-Oriented Programming paradigm leads to cheaper and faster development. They mainly preferred to employ a programming language, or a set of languages, that supports debugging, tracking, and testing tools.

It is not surprising that *socket programming* and *threading* were prioritized as two essential features, as all of the case studies were mainly involved with network programming and web-based applications. In other words, Client-Server was one of the software architecture patterns of the backbone of their systems.

Easy to reuse and read existing code and write new code have considered three programming language features in the decision model. These features were prioritized as Must-Have and Should-Have features at four case study companies so that the case study participants were looking for programming languages that facilitate the development process and increase the reusability and readability of their source code.

Besides the free implementation of their core libraries, open-source programming languages, as two programming language features, arouse almost all of the case study participants' attention. We realized that the development teams' budget at the case study companies was impactful on their decisions to select either licensed or open-source programming languages. In other words, the budget constraint can be led to selecting the cheapest feasible solution as the best fitting solution.

5.6 Discussion

5.6.1 Case Study Participants

Almost all of the case study participants asserted that for evaluating a programming language from its usability point of view, they should be familiar with the language first, essentially when it is a cutting-edge programming language. It is a time-consuming process depending on the language's complexity, making it more challenging to evaluate. Thus, they require technical knowledge to comprehend the programming language and its features to create suitable test setups and compare it with other potential solutions. Typically, they need to focus on particular parts of the programming language - assessing the entire language at once is nearly impossible because it would take much time. Additionally, to evaluate the programming language, the case study company's experts need to answer the following question: which programming language features are the most important ones? Can a language's complexity be evaluated in a questionnaire or a usability test, or do we need test users evaluating the language over a more extended period? What are the right set of criteria to declare a language is usable? How can we assume a language is *better* than another one? What should be measured to find this out? Several measurements could be more practical, such as learnability, understandability, or consistency – What is most substantial? The case study participants confirmed that the decision model, including its programming language features and languages, can address such questions and reduce the decision-making process's time and cost.

One of the participants mentioned that *in contrast to licensed programming languages, in which programmers are limited to use them for a few days only, open-source programming languages are available to evaluate before actual implementation. However, after selecting an open-source programming language as the primary language, sometimes multiple modifications are made in the source code of an open-source programming language by unknown developers. Eventually, this leaves programmers questioning about the current version of the code they are using. In the case of licensed programming languages, the changes are made systematically by authorized developers of the source code, and they notify the version of the code.* Accordingly, a development team should be able to organize the source code and increase its reusability.

One of the case participants stated that *code reuse is the practice of using existing code for a new function or component. The existing code may be reused to perform the same function or to do a similar but slightly modified function providing for efficiencies, cost savings, and improved overall quality. In order to reuse code, that code should be high-quality, so it should be secure and reliable. Code reusable in practice means that developers have to build libraries that other projects requiring that same functionality can utilize. So the developers should identify the core competence of each module.* .

Biases, such as motivational and cognitive (Montibeller & Winterfeldt, 2015), arise because of shortcuts or heuristics that decision-makers use to solve problems and perform tasks. The Hawthorne effect (Jones, 1992), which is the tendency for decision-makers to change their behavior when observed, is a form of cognitive bias. The case study participants might have been more careful in the observational setting than in the real setting because they are being observed by scientists judging their selected programming language feature requirements and priorities. Moreover, the Bandwagon effect (Nadeau et al., 1993), which is the tendency to do or believe things because many other decision-makers do or believe the same, is another form of cognitive bias. The Bandwagon effect typically shows up in group decisions. To mitigate the Hawthorne and Bandwagon effects, individual and group interviews have been conducted.

5.6.2 Experts

One of the experts asserted that *some programming languages are certainly better options for solving different problems, making the selection process more straightforward. For instance, if you are integrating heavily with Microsoft products, then you have to consider .NET on your list of potential alternatives. If you perform functions, such as scientific modeling, Python can be a better choice because of its well-defined math libraries and the ability to scale well with Hadoop. If enterprise-level security for a business-to-business application is critical, then Java can be the best fitting language. Thus, some factors, including the problem domain, business case, and the types of customers are impactful in the evaluation process.*

The experts expressed that programming languages' supported programming language features play a significant role in the programming language selection process. For instance, one of the experts expressed that *we are a developer-centric company and need a comprehensive set of software development kits for various well-known programming languages, including Java, Python, Ruby, and PHP. Each programming language has different features, communities, support, and ecosystems to consider when making your choice.*

Almost all of the experts mentioned that their companies continuously improve and reevaluate their technologies, including the used programming languages. They mainly consider a limited set of programming language features and languages in their selection process; thus, a decision model, such as the one in this study, can ease their evaluation process.

5.6.3 Lessons learned

General-purpose programming languages, such as *C#* and *Java*, are nearly at the same level of maturity and support almost the same feature set, so programming language selection is majorly concerned with the ecosystem, the community, and the availability of programmers. The ecosystem can be considered as libraries, tools, and frameworks that support programming languages. The community is typically the one who maintains the ecosystem. Even when backed by a company, the community is the main drive for improvements, often implementing them. The availability of programmers is self-explanatory. Some programming languages, such as *Haskell*, are appreciated in theory, but fewer programmers are available than *Java*.

When a software development company selects a popular programming language, it can count on sourcing numerous well-qualified developers in the market who are available to work at the company. Such programming languages are surrounded by huge communities that provide developers with samples and solutions to solve critical tasks and problems much quicker.

Experience in using technology provides invaluable knowledge when selecting suitable technology. In other words, software engineers typically prefer to select the programming languages they have employed before and have some experience. The main factor is the cost of adding a new programming language. Hiring new developers, changing the infrastructure, and learning the best practices are costly for many companies. Therefore, the answer to the best programming language question usually is what the company was already using. There are many risks associated with this, as an organization could end up being stuck in a legacy technology for which there is no longer a demand (Khadka et al., 2014). It is thereby advisable for these companies to primarily use the DSS to avoid innovation stasis.

For small projects, selecting a programming language is much faster and more straightforward, as the brevity of the lines of code is essential. In other words, software engineers prefer to select a programming language requiring fewer code lines to develop the project in such projects. The intention is to get the solution out first, next to be worried about the performance. However, for large organizational projects, programming language selection is a different story. Various development teams will develop different modules and services expected to interact and interconnect with another to address a particular problem. In this case, the programming language selection might be involved with the high level of portability of the language to run on different platforms or exchangeability of data and information.

5.6.4 The Decision Model

The case study participants confirm that the DSS provides programming languages to help software development companies in their initial decisions for selecting programming languages. In other words, the DSS recommended nearly the same programming languages as the case study participants suggested to their companies after extensive analysis and discussions. However, the DSS offers a shortlist of feasible programming languages; therefore, software development companies should perform further investigations, such as performance testing, to find the best fitting programming language for their software products.

The case study participants confirm that the updated and validated version of the DSS is useful and valuable in finding the shortlist of feasible solutions. Finally, it reduces the time and cost of the decision-making process. Our website[11] is up and running to keep the decision support system's knowledge base up-to-date and valid. The supported programming language features are going to change due to technological advances. As such, the decision model must be updated regularly. We envision a community of users of the DSS, who maintain and curate the system's knowledge and consider building such a community as future work.

Decision support systems can be employed to make decisions quicker and more efficiently; however, they suffer from adoption problems (Donzelli, 2006). A DSS supports rational decision making by recommending alternative solutions basis the objectivity. Although limited rationality plays a crucial role in a decision-making process, subjectivity should not be discarded. A DSS promotes objectivity and dismisses subjectivity, which can have a drastic consequence on the decisions' reliability.

[11]https://dss-mcdm.com

The DSS provides a discussion and negotiation platform to enable requirement engineers to make group decisions. It detects and highlights the conflicts in the assigned priorities to decision-makers' programming language feature requirements and asks them to resolve disagreements. Thus, the DSS supports requirements engineers in the requirements verification and validation activity by avoiding conflict between programming language feature requirements and generating feasible solutions according to the programming language feature requirements. Moreover, the DSS can be considered as a communication tool among the decision-makers to facilitate the requirements specification activity (Farshidi et al., 2020c).

It is essential to highlight that sometimes unpopular programming languages have unique features that the popular ones do not support them. For instance, *OptiML* is an embedded domain-specific language for machine learning. *OptiML* enables software developers to run statistical inference algorithms expressible by the statistical query model to be easy to express and execute quickly (Sujeeth et al., 2011). Alternatively, there are many command-line tools on UNIX-like operating systems (such as Linux, Mac, and BSD), each one accepting instructions in their format. This format can be considered a domain-specific language that allows defining the tasks to be executed. For example, *SED* executes the text transformations indicated using its domain-specific language (Dougherty & Robbins, 1997). Consequently, we need a decision model for selecting such programming languages. We believe that the decision model can be extended in the future to consider such programming languages as its alternative solutions.

5.6.5 Limitations and Threats to Validity

The validity assessment is an essential part of any empirical study. Validity discussions typically involve Construct Validity, Internal Validity, External Validity, and Conclusion Validity.

Construct validity refers to whether an accurate operational measure or test has been used for the concepts being studied. In literature, decision-making is typically defined as a process or a set of ordered activities concerning stages of problem identifying, data collection, defining alternatives, selecting a shortlist of alternatives as feasible solutions with the ranked preferences (Fitzgerald et al., 2017; Kaufmann et al., 2012). To mitigate the threats to the construct validity, we followed the MCDM theory and the six-step of a decision-making process (Majumder, 2015) to build the decision model for the programming language selection problem. Moreover, we employed document analysis and expert interviews to capture knowledge regarding programming languages as two different knowledge acquisition techniques. Additionally, the DSS and the decision model have been evaluated through seven real-world case studies at seven different real-world software development companies in the Netherlands and Iran.

A challenge for this study is that the qualities and features that we have identified with the support of twelve experts can vary wildly with the expert's perception. Some of the experts, for instance, indicated that most features could quickly be built using the language, although it is not necessarily included in the language's standard libraries. While we are convinced that the twelve experts have added a significant amount of extra knowledge to the model, one might argue we need a large number of experts per programming language to reach consensus on each feature. We should also be aware of the strong opinions surrounding programming languages, making it somewhat more complicated to find consensus in the data. A potential solution to this validity threat is the introduction of the community, as mentioned earlier.

Internal validity attempts to verify claims about the cause-effect relationships within the context of a study. In other words, it determines whether the study is sound or not. To mitigate the threats to the decision model's internal validity, we define DSS success when it, in part, aligns with the case study participants' shortlist and when it provides new suggestions that are identified as being of interest to the case study participants. Emphasis on the case study participants' opinion as a measurement instrument is risky, as they may not have sufficient knowledge to make a valid judgment. We counter this risk by conducting more than one case study, assuming that the case study participants are handling their interest and applying the DSS to other problem domains, where we find similar results (Farshidi & Jansen, 2020a; Farshidi et al., 2018a; Farshidi et al., 2018b; Farshidi et al., 2020c; Farshidi et al., 2018c; Farshidi et al., 2020e).

External validity concerns the domain to which the research findings can be generalized. External validity is sometimes used interchangeably with generalizability (feasibility of applying the results to other research settings). We evaluated the decision model in the context of Dutch enterprises. To mitigate threats to the research's external validity, we captured knowledge from different knowledge sources without any regional limitations to define the constructs and build the decision model. Accordingly, we hypothesize that the decision model can be generalized to all software development companies worldwide who face uncertainty in the programming language selection problem. Another question is whether the framework and the DSS can be applied to other problem domains as well. The problem domains (Farshidi et al., 2018a; Farshidi et al., 2020c; Farshidi et al., 2018c; Farshidi et al., 2020e) were selected opportunistically and pragmatically, but we are convinced that there are still many decision problems to which the framework and the DSS can be applied. The categories of problems to which the framework and the DSS can be applied successfully can be summed up as follows: (1) the problem regards a technology decision in system design with long-lasting consequences, (2) there is copious scientific, industry, and informal knowledge publicly available to software engineers, and (3) the (team of) software engineer(s) is not knowledgeable in the field but very knowledgeable about the system requirements. We believe that the framework can be employed as a guideline to build decision models for MCDM problems in software production.

Conclusion validity verifies whether the methods of a study such as the data collection method can be reproduced, with similar results. We captured knowledge systematically from the sources of knowledge following the MCDM framework (Farshidi et al., 2018a). The accuracy of the extracted knowledge was guaranteed through the protocols that were developed to define the knowledge extraction strategy and format. A review protocol was proposed and applied by multiple research assistants, including bachelor and master students, to mitigate the research's conclusion validity threats. By following the framework and the protocols, we keep consistency in the knowledge extraction process and check whether the acquired knowledge addresses

Table 5.6: this table compares a subset of selected studies from the literature that addresses the programming language selection problem. The first and second columns (Studies and Years) refer to the considered studies and their publication years. The third column (DMA) indicates the decision-making approach that the studies have employed to address the problem. The fourth column (MCDM) denotes whether the corresponding decision-making technique is an MCDM approach. The fifth column (PC) indicates whether the MCDM approach applied pairwise comparison as a weight calculation method or not. The sixth column (QA) determines the type of quality attributes. The seventh and eighth columns (#C and #A) signify the number of criteria and alternatives that were considered in the selected studies. The next three columns indicate the numbers of common quality attributes (#CQ), features (#CF), and alternatives (#CA) of this study (the first row) with the selected studies. The last column (Cov.) shows the percentage of the coverage of the considered criteria (quality attributes and features).

Studies	**Years**	**DMA**	MCDM	**PC**	**QA**	**#C**	**#A**	**#CQ**	**#CF**	**#CA**	**Cov.**
This study		DSS	Yes	No	ISO/IEC 25010 EX. ISO/IEC 9216	164	47	57	107	47	100%
Mishra et al. (2020)	2020	Fuzzy Logic	Yes	No	Domain Specific	7	8	5	2	7	100%
Costanza et al. (2019)	2019	Benchmarking	No	N/A	Domain Specific	2	3	2	0	3	100%
Yıldızbaşı & Daneshvar (2018)	2018	TOPSIS	Yes	Yes	Domain Specific	18	4	8	8	4	89%
Feraud & Galland (2017)	2017	Benchmarking	No	N/A	Domain Specific	14	5	5	8	0	92%
Yoon et al. (2016)	2016	FDM	Yes	No	Domain Specific	29	6	3	12	3	52%
Ray et al. (2014)	2014	Statistical Analysis	No	N/A	Domain Specific	26	17	2	12	17	82%
Lesani et al. (2014)	2014	FAHP	Yes	Yes	Domain Specific	8	5	3	3	5	75%
Meyerovich & Rabkin (2013)	2013	Statistical Analysis	No	N/A	Domain Specific	14	33	3	7	26	71%
Bissyandé et al. (2013)	2013	Statistical Analysis	No	N/A	Domain Specific	3	30	2	1	17	100%
Bhattacharya & Neamtiu (2011)	2011	Statistical Analysis	No	N/A	Domain Specific	4	2	2	0	2	50%
Mannila & Raadt (2006)	2006	Benchmarking	No	N/A	Domain Specific	17	11	9	7	9	94%
Parker et al. (2006)	2006	AHP	Yes	Yes	Domain Specific	23	7	9	12	7	91%
Cochran & Chen (2005)	2005	Fuzzy Logic	Yes	No	Domain Specific	36	3	9	2	1	31%
Holtz & Rasdorf (1988)	1988	Benchmarking	No	N/A	Domain Specific	23	4	18	3	3	91%

the research questions. Moreover, we crosschecked the captured knowledge to assess the results' quality, and we had at least two assistants extracting data independently.

5.7 Related Work

Decision analysis, which is the study of decision making for problems with multiple objectives, has been developed and widely employed in solving complex decision-making problems. In literature, decision-making is typically defined as a process or a set of ordered activities concerning stages of problem identifying, data collection, defining alternatives, selecting a shortlist of alternatives as feasible solutions with the ranked preferences. Fitzgerald et al. (2017) define decision-making as a process that consolidates critical assessment of evidence and a structured process that requires time and conscious effort. Kaufmann et al. (2012) state that the decision-making process encourages decision-makers to establish relevant decision criteria, recognize a comprehensive collection of alternatives, and assess the alternatives accurately. Over the past few years, various methods and underlying theories have been introduced for solving decision-making problems in software production, such as programming language selection.

In this research, Snowballing was the primary method to investigate the existing literature regarding techniques that address the programming language selection problem. Table 5.6 summarizes a subset of selected studies that discuss the problem.

5.7.1 Benchmarking and Statistical Analysis

In literature, some studies employed Benchmarking and Statistical Analysis to evaluate and compare a collection of programming languages against each other. For instance, Meyerovich & Rabkin (2013) conducted survey research to identify the factors that lead to language adoption. They concluded that only a limited number of programming languages were used for most applications, but the programming market supports many programming languages with niche user bases. Furthermore, essential programming language features have only secondary importance in adoption. Open-source libraries, existing code, and experience strongly influence developers when selecting a project's programming language.

Holtz & Rasdorf (1988) introduced and discussed various attributes of programming languages that can positively or negatively affect the computer-aided design and computer-aided engineering software. Four programming languages, *Fortran*, *C*, *Pascal*, and *Modula-2*, were compared using the attributes.

Bhattacharya & Neamtiu (2011) introduced a methodology for quantifying the impact of programming language on software quality and developer productivity. They formulated four hypotheses that investigated whether using *C*++ leads to better software than using *C*. Then, they tested their hypotheses on large data sets to ensure statistically significant results.

Mannila & Raadt (2006) suggested a set of criteria, including learnability, suitability, and availability, of programming languages. Next, they compared eleven programming languages (including *Eiffel*, *Haskell*, *Java*, *JavaScript*, *Logo*, *Pascal*, *Python*, and *Scheme*) according to the criteria. Finally, they assigned scores to the languages based on the number of criteria that they support.

Ray et al. (2014) collected an extensive data set from GitHub to study the effect of programming language features such as static versus dynamic typing, strong versus weak typing on software quality. By triangulating findings from different methods and controlling for confounding effects such as team size, project size, and project history, they reported that programming language design does have a significant but modest effect on software quality.

Bissyandé et al. (2013) investigated a large number of open-source projects from GitHub to measure the *popularity*, *interoperability* and *impact* of various programming languages in terms of lines of code, development teams, issues, etc.

Feraud & Galland (2017) compared five agent-based programming languages according to a number of criteria, such as Code extensibility and Debugging tools.

Costanza et al. (2019) performed performance testing to analyze and compare the performance of three programming languages (*Go*, *Java*, and *C++*). Based on their benchmark results, the authors selected *Go* as their implementation tool and recommended considering *Go* as a valid candidate for developing other bioinformatics applications.

Studies based on benchmarking and statistical analysis are typically time-consuming approaches and mainly applicable to a limited set of alternatives and criteria, as they require a thorough knowledge of programming languages and concepts. Decision-making based on such analysis can be challenging as decision-makers cannot assess all their requirements and preferences at the same time, especially when the number of requirements and alternatives is significantly high. Furthermore, benchmarking and statistical analysis are likely to become outdated soon and should be kept up to date continuously, which involves a high-cost process.

5.7.2 MCDM approaches

As aforementioned, finding the best fitting programming language(s) for a software project is a decision-making process that evaluates several alternatives and criteria. The selected programming language(s) should address the concerns and priorities of the decision-makers.

Conversely to MCDM approaches, studies based on *Benchmarking* and *Statistical Analysis* principally offer generic results and comparisons and do not consider individual decision-maker needs and preferences.

The tools and techniques based on MCDM are mathematical decision models aggregating criteria, points of view, or features (Floudas & Pardalos, 2008). Support is a fundamental concept in MCDM, indicating decision models are not developed following a process where the decision maker's role is passive (Dvořák et al., 2018). Alternatively, an iterative process is applied to analyze decision-makers' priorities and describe them as consistently as possible in a suitable decision model. This iterative and interactive modeling procedure forms the underlying principle of decision support tendency of MCDM, and it is one of the main distinguishing characteristics of the MCDM as opposed to statistical, and optimization decision-making approaches (Gil-Aluja, 2013).

A variety of MCDM approaches have been introduced by researchers recently. A subset of selected MCDM methods is presented as follows: The *Analytic Hierarchy Process (AHP)* is a structured and well-known method for organizing and analyzing MCDM problems based on mathematics and psychology. Parker et al. (2006) presented a set of criteria for selecting a programming language for use in an introductory programming course. Next, they applied the AHP to evaluate the seven programming languages they considered potential alternatives. This MCDM approach considers a hierarchical structure of objectives, criteria, and alternatives to make complex decisions.

The *Technique for Order Preference by Similarity to Ideal Solution (TOPSIS)* is an MCDM approach that employs information entropy to assess alternatives.

Yıldızbaşı & Daneshvar (2018) employed the TOPSIS method to evaluate a set of programming languages based on seven criteria such as ease of use and ability of programming languages. This approach aims to come up with an ideal solution and a negative ideal solution and then identify a scenario that is nearest to the ideal solution and farthest from the negative ideal solution.

Fuzzy logic is an approach to computing based on *degrees of truth* rather than the usual Boolean logic. Cochran & Chen (2005) and Mishra et al. (2020) employed fuzzy set theory and fuzzy operations to address the programming language selection problem using based on weighted scores. Sometimes combinations of fuzzy logic with other MCDM approaches, such as AHP, are employed to solve MCDM problems. For instance, Lesani et al. (2014) introduced a FAHP-based approach to evaluate five object-oriented programming languages against eight decision criteria.

The *Fuzzy Delphi Method (FDM)* is a more advanced version of the Delphi Method in that it utilizes triangulation statistics to determine the distance between the levels of consensus within the expert panel. Yoon et al. (2016) identified a set of key factors for educational programming language selection and then applied the Delphi method based on a 20-expert panel to evaluate six programming languages.

The majority of the MCDM techniques in literature define domain-specific quality attributes to evaluate the alternatives. Such studies are mainly appropriate for specific case studies. Furthermore, the results of MCDM approaches are valid for a specified period; therefore, the results of such studies, by programming language advances, will be outdated. Note that, in our proposal, this is also a challenge, and we propose a solution for keeping the knowledge base up to date in section 5.6.

Additionally, pairwise comparison is typically considered as the main method to assess the weight of criteria in MCDM techniques. For a problem with n number of criteria $\frac{n(n-1)}{2}$ number of comparisons are needed (Saaty, 1990). It means that the pairwise comparison is a time-consuming process and gets exponentially more complicated as the number of criteria increases (Ribeiro et al., 2011). A subset of MCDM approaches, such as TOPSIS and AHP, are not scalable (Ibriwesh et al., 2018; Khari & Kumar, 2013), so in modifying the list of alternatives or criteria, the whole process of evaluation should be redone. Therefore, these methods are costly and applicable to only a small number of criteria and alternatives. In this study, we have considered 164 criteria and 47 alternatives to building a decision model for the programming language selection problem.

In contrast to the named approaches, the cost of creating, evaluating, and applying the proposed decision model is not penalized exponentially by the number of criteria and alternatives because it is an evolvable and expandable approach that splits down the decision-making process into four maintainable phases (Farshidi et al., 2018c). Moreover, we introduce several parameters to measure non-Boolean criteria' values, e.g., the maturity level and popularity of programming languages. The proposed decision model addresses the main knowledge management issues, such as capturing, sharing, and maintaining knowledge. Moreover, it uses the ISO/IEC 25010 (ISO, 2011) as a standard set of quality attributes. This quality standard is a domain-independent software quality model and provides reference points by defining a top-down standard quality model for software systems.

5.8 Conclusion

The development of software systems is well recognized as an engineering activity, hence the term *software engineering*. As with all engineering activities, supervisors and practitioners must have a firm understanding of the fundamentals. The complexity of software engineering has increased dramatically in the past decade. With the continuing increase in the variety, functionality, and complexity of software engineering, more attention must be paid to programming language suitability to make rational decisions regarding language selection.

In this study, the programming language selection process is modeled as a multi-criteria decision-making problem that deals with evaluating a set of alternatives and considering a set of decisions criteria (Triantaphyllou et al., 1998). Moreover, we presented a decision model for the programming language selection problem based on the technology selection framework (Farshidi et al., 2018a). The novelty of the approach provides knowledge about programming languages to support uninformed decision-makers while contributing a sound decision model to knowledgeable decision-makers. Furthermore, it incorporates deeply embedded requirements engineering concepts, such as the ISO software quality standards and the MoSCoW prioritization technique, besides knowledge engineering theories, to develop the decision model. We conducted seven industry case studies to evaluate the decision model's usefulness and effectiveness to address the decision problem. We find that while organizations are typically tied to particular ecosystems by extraneous factors, they can benefit significantly from our DSS use.

The case studies show that this article's decision model also provides a foundation for future work on MCDM problems. We intend to build trustworthy decision models to address *Programming Language Framework* and *Model- Driven Development Platform* selection problems as our (near) future work.

CHAPTER 6

Model-Driven Development Platforms

Context: Model-Driven Development platforms shift the focus of software development activity from coding to modeling for enterprises. A significant number of such platforms are available in the market. Selecting the best fitting platform is challenging, as domain experts are not typically model-driven deployment platform experts and have limited time for acquiring the needed knowledge. *Method:* We model the problem as a multi-criteria decision-making problem and capture knowledge systematically about the features and qualities of 30 alternative platforms. *Results:* Through four industry case studies, we confirm that the decision model can support decision-makers with the selection problem by reducing the time and cost of the decision-making process and by providing a richer list of options than the enterprises considered initially. *Conclusion:* Our industry contribution is the decision model. We show that having decision knowledge readily available supports decision-makers in making more rational, efficient, and effective decisions. The study's theoretical contribution is the observation that the decision framework provides a reliable approach for creating decision models in software production.

keywords- Model-Driven Development platform; decision model; multi-criteria decision-making; decision support system; industry case study

This chapter is based on the following publication:

Farshidi, S., S. Jansen & S. Fortuin (2021), "Model-driven development platform selection: four industry case studies", *Software and Systems Modeling*.

6.1 Introduction

Software applications are produced and maintained by software engineers, and business processes are introduced and managed by domain experts (non-developers) who mainly understand business (Olariu et al., 2016). Software development requires interactions with domain experts, necessitating a level of agreement in describing the technical phases of development. Moreover, software products are getting more complicated, so they need to be discussed at different abstraction levels depending on the technical background of the involved domain experts, the development process's phase, and the business objectives (Brambilla et al., 2017).

Model-Driven Development (MDD) is a vision of software development where models play a core role as primary development artifacts (Staron, 2006). Modeling tools in software production are widespread and have reached a degree of maturity where their performance and availability are increasingly accepted, also by non-technical users. Over the last two decades, an extensive list of modeling tools (García-Borgoñon et al., 2014) has been introduced (Richardson & Rymer, 2016) to support MDD, such as low-code/no-code platforms and business process management systems. Such modeling tools and platforms' primary aspiration is to boost productivity and decrease time-to-market by facilitating development at a higher level of abstraction and employing concepts closer to the problem domain at hand, rather than the ones given by programming languages (Sendall & Kozaczynski, 2003).

According to Gartner, by 2024, MDD platforms will be responsible for over 65% of the application development activity, and three-quarters of large enterprises will be using at least four MDD platforms, as such platforms enable enterprises to develop applications quicker using more capabilities and fewer conventional developers (Vincent et al., 2019).

A significant number of MDD platforms with a broad list of features and services are available in the market (Richardson & Rymer, 2016), so it is challenging for enterprises to select the best fitting platforms that address their requirements and priorities (Hutchinson et al., 2014). The selection problem can be modeled as a multi-criteria decision-making (MCDM) problem that deals with evaluating a set of alternatives and considers a set of decisions criteria (Triantaphyllou et al., 1998). MCDM poses a cost-effective solution based on its mathematical modeling method for finding the best fitting feasible alternative according to decision-makers' preferences (Dhiman & Deb, 2020).

Knowledge about MDD platforms is fragmented in a wide range of literature, documentation, and software engineers' experience. To systematically capture such knowledge and make it available in a reusable and extendable format, we have followed the framework (Farshidi et al., 2018a) to build a decision model for the MDD platform selection problem. The framework and a Decision Support System (DSS) (Farshidi et al., 2018b) were introduced in our previous studies for building decision models for MCDM problems in software production.

The DSS is a platform[1] for building MCDM decision models based on the framework. Decision models can be uploaded to the DSS's knowledge base to facili-

[1]The decision studio is available online on the DSS website: https://dss-mcdm.com

tate software-producing organizations' decision-making process according to their requirements and preferences. Furthermore, the DSS can be used over the full lifecycle and co-evolve its advice based on evolving requirements.

The rest of this study is structured as follows: Section 6.2 presents a brief description of the MDD platforms and determines the scope of the study. Section 6.3 formulates the MDD platform selection problem as an MCDM problem, defines the research questions of the study, and describes our research method, which is based on the design science, expert interviews, document analysis, and exploratory theory-testing case studies. This study has the following contributions:

- Section 6.4 explains the integration of the captured tacit knowledge of software engineers through interviews and the explicit knowledge scattered in an extensive list of websites, articles, and reports. This study's findings provide knowledge that can educate and support the decision-makers to understand: 1) which MDD platforms are available at the moment, 2) the capabilities of the MDD platforms, and 3) which features are fulfilled by which platforms.
- Section 6.5 describes four industry case studies in the Netherlands to evaluate the effectiveness and usefulness of the decision model to address the decision problem. Moreover, This section analyzes the DSS results and compares them with the case study participants' shortlist of feasible MDD platforms. The results show that the DSS recommended nearly the same solutions as the case study participants suggested to their companies after extensive analysis and discussions and do so more efficiently.

Section 6.6 highlights barriers to the knowledge acquisition and decision-making process, such as motivational and cognitive biases, and argues how we have minimized these threats to the validity of the results. Section 6.7 positions the proposed approach in this study among the other MDD platform selection techniques in the literature. Finally, Section 6.8 summarizes the proposed approach, defends its novelty, and offers directions for future studies.

6.2 Background

Hailpern & Tarr (2006) give a general definition of MDD: "A software engineering approach consisting of the application of models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life cycle".

Software production with an MDD platform is not initiated by programming (high-coding) but modeled using visual modeling or declarative development tools and pre-built templates and components understood by citizen developers. The conceptual model transforms into an application (Ceri et al., 2009), such as web-based or wearable apps, by generating code or model interpretation (Brambilla et al., 2017). The process of transforming a conceptual model into an application is called *conceptual modeling compilation* (Pastor & Molina, 2007). The development of conceptual models comprises the real world's representation using an abstraction level higher than that of source code. Likewise, source code represents a higher abstraction level than that for machine code obtained through a conventional compilation process.

Therefore, it seems logical to refer to the process of transforming a conceptual model into a software product using the term *compilation*.

The simplified interface leads many to believe that building applications using MDD platforms requires little or no coding knowledge. However, sometimes these predefined components need to be customized using programming languages. For example, maybe a developer wants to place a particular widget on a web application home page, which is not a part of the MDD platform's default widget library. In this case, she needs to extend the platform capabilities by developing a unique widget and making it a new component for future projects. Additionally, some MDD platforms offer the flexibility to deploy and maintain applications on public or private clouds or even on-premises. Automated deployment, together with a cloud-native and stateless architecture, leads to high availability and fail-over to support large-scale deployments, especially in an enterprise context.

Four principles underlie the architecture of an MDD platform (Brown, 2004): (1) Models expressed in a well-defined notation are a cornerstone to system understanding for enterprise-scale solutions. (2) Building systems can be organized around a set of models by imposing a series of transformations between models, organized into an architectural framework of layers and transformations. (3) A formal underpinning for describing models in a set of metamodels facilitates meaningful integration and transformation among models and is the basis for automation through tools. (4) Acceptance and broad adoption of the model-based approach require industry standards to provide consumers with openness and foster competition among vendors. Accordingly, in literature, there are a significant number of tools and platforms that are based on the MDD paradigm, for instance, modeling notations and software process modeling languages, such as UML and Petri-nets (García-Borgoñon et al., 2014).

Gartner (Dunie et al., 2019) and Forrester (Rymer et al., 2019) categorized MDD platforms, including low-code/no-code platforms and business process management systems into the following two sets:

Business Process Management Suite (BPMS) is a set of platforms to support business process management initiatives. BPMS is an MDD approach that aids a process improvement lifecycle from start to end – from process discovery, definition, design, implementation, monitoring, and analysis. BPMS platforms are used for automating, measuring, and optimizing business processes. Note, BPMS is an extension of classical workflow management systems and approaches (Van Der Aalst, 2003).

intelligent Business Process Management Suite (iBPMS) is a subset of low-code/no-code application development platforms. It provides the functionality needed to support more intelligent business operations, such as real-time analytics and collaborative capabilities (Dunie et al., 2019). According to Gartner's report, iBMPS is the next generation of BPMS that leverages recent technological advances to attain a degree of operational responsiveness not possible with BPMS platforms.

This study focuses on BMPS and iBPMS as two essential sets of MDD platforms to build a decision model for the decision-making process. Note that we use *MDD platforms* to refer to BMPS and iBPMS platforms for the sake of brevity.

6.3 Research Approach

This section defines the problem, indicates the study objective, and formulates the research questions. Moreover, it elaborates on the research methods and relates them to individual research questions to which they apply. Additionally, the knowledge acquisition techniques, analysis procedures, and the tactics used to mitigate the threats to this study's validity are presented in this section.

6.3.1 Problem Definition

In this study, we formulate the MDD Platform selection problem as an MCDM problem:

Let $Platforms = \{p_1, p_2, \dots p_{|Platforms|}\}$ be a set of MDD platforms in the market (i.e., Mendix, Outsystems, and ServiceNow). Furthermore, $Features = \{f_1, f_2, \dots t_{|Features|}\}$ be a set of MDD features (i.e., Supporting Native modeling tool and Decision table) of the MDD platforms, and each platform p, where $p \in Platforms$, supports a subset of the set $Features$. The goal is finding the best fitting MDD platforms as solutions, where $Solutions \subset Platforms$, that support a set of MDD feature requirements, called $Requirements$, where $Requirements \subseteq Features$. An MCDM approach for the selection problem receives $Platforms$ and their $Features$ as its input, then applies a weighting method to prioritize the $Features$ based on the decision-makers' preferences to define the $Requirements$, and finally employs a method of aggregation to rank the $Platforms$ and suggests $Solutions$. Accordingly, an MCDM approach can be formulated as follows:

$$MCDM : Platforms \times Features \times Requirements \rightarrow Solutions$$

Typically, a unique optimal solution for an MCDM problem does not exist, and it is necessary to employ decision-makers' preferences to differentiate between solutions (Majumder, 2015). Particular platforms might fit into an enterprise; however, some might be better than others. It is tough to state which platform is the best one, partially because we can not predict the future or know how the enterprise would have evolved if a different platform was selected. Therefore, it is inevitable to note that such a technology selection process can never be completely objective because humans have to make decisions.

Figure 1.2 visualises MCDM approach for the MDD platform selection problem in a 3D space. It shows that the degree of satisfaction of the decision-makers with a suggested solution is fuzzy, which means that the satisfaction degree from a decision-maker perspective may range between completely true (best fit) and completely false (worst fit) (Dvořák et al., 2018), which is represented by a range of colors from red to dark green.

6.3.2 Research Question

The Main Research Question (MRQ) of this study is as follows:

MRQ: How can knowledge regarding MDD platforms be captured and organized systematically to support enterprises with the decision-making process?

We formulated the following research questions to capture knowledge regarding the MDD platform systematically and to build a decision model for the decision problem based on the framework (Farshidi et al., 2018a):

> RQ_1: Which MDD concepts should be considered as the MDD features in the decision model?
> RQ_2: Which MDD platforms should be considered in the decision model?
> RQ_3: Which software quality attributes can be used to evaluate the MDD platforms?
> RQ_4: What are the impacts of the MDD features on the quality attributes of the MDD platforms?
> RQ_5: Which MDD platforms currently support the MDD features?

6.3.3 Research Methods

Research methods are classified based on their data collection techniques (interview, observation, literature, etc.), inference techniques (taxonomy, protocol analysis, statistics, etc.), research purpose (evaluation, exploration, description, etc.), units of analysis (individuals, groups, process, etc.), and so forth (Meredith et al., 1989). Multiple research methods can be combined to achieve a fuller picture and a more in-depth understanding of the studied phenomenon by connecting complementary findings that conclude from the methods from the different methodological traditions of qualitative and quantitative investigation (Johnson & Onwuegbuzie, 2004).

The data collection is an empirical study that can be quantitative or qualitative (Runeson & Höst, 2009). Quantitative data comprises numbers and classes, while qualitative data involves descriptions and explanations of phenomena. Quantitative data is analyzed using statistics, while qualitative data is analyzed using expert interviews and case study research to provide a more detailed and more in-depth explanation. However, a combination of qualitative and quantitative data often provides a better understanding of the studied phenomenon (Seaman, 1999) (Mixed research). Table 6.6 shows the research methods and data collection types of a subset of studies in the literature that address the MDD platform selection problem.

We designed a framework (Farshidi et al., 2018a) and implemented a DSS (Farshidi et al., 2018b) for supporting software engineers (decision-makers) with their MCDM problems in software production. Knowledge engineering theories have been employed to design and implement the DSS and the framework. The framework provides a guideline for decision-makers to build decision models for MCDM problems in software production following the six-step of the decision-making process (Majumder, 2015): (1) identifying the objective, (2) selection of the features, (3) selection of the alternatives, (4) selection of the weighing method, (5) applying the method of aggregation, and (6) decision making based on the aggregation results.

In this study, we applied the framework to build a decision model for the MDD platform selection problem. Moreover, we used design science, expert interviews, and document analysis as a mixed data collection method to capture knowledge regarding MDD platforms and to answer the research questions. Then, we identified

30 MDD platforms and 94 MDD features by conducting semi-structured interviews with 26 domain experts. We also indicated the mapping between the MDD features and platforms by analyzing MDD platforms' documents and the experts' tacit knowledge. Moreover, we mapped the MDD features to the quality attributes suggested by ISO/IEC 25010 standard (ISO, 2011) and extended ISO/IEC 9126 standard (Carvallo & Franch, 2006) and calculate the impacts of the MDD platforms on the quality attributes based on three expert interviews. According to the acquired knowledge and guidelines of the framework (Farshidi et al., 2018a), we modeled the decision problem as an MCDM problem and built a decision model for the MDD platform selection problem. Finally, we have evaluated the decision model by conducting four real-world case studies.

Design Science

Design science is an iterative process (Simon, 1996), has its roots in engineering (Hevner et al., 2004), is broadly considered a problem-solving process (Fortus et al., 2005), and attempts to produce generalizable knowledge about design processes and design decisions. The design process is a set of hypotheses that can eventually be proven by creating the artifact it describes (Walls et al., 1992). However, a design's feasibility can be supported by a scientific theory to the extent that the design comprises the theory's principles. Research investigations involve a continuous, repetitive cycle of *description*, *explanation*, and *testing* (Meredith et al., 1989). Accordingly, in most cases, theory development is a process of gradual change (Baxter, 2004). The research approach for creating decision models for MCDM problems is Design Science, which addresses research by building and evaluating artifacts to meet identified business needs (Hevner et al., 2008).

Recently, we designed a theoretical framework (Farshidi et al., 2018a) and implemented a DSS (Farshidi et al., 2018b) for supporting software engineers (decision-makers) with their MCDM problems in software production. Knowledge engineering theories have been employed to design and implement the DSS and the framework. In this study, we applied the framework to build a decision model for the MDD platform selection problem. Additionally, we employed the DSS to facilitate the decision-making process. The research approach for creating the decision model is Design Science, which addresses research through the building and evaluation of artifacts to meet identified business needs (Hevner et al., 2004), accordingly, we carried out four industry case studies in the context of four real-world enterprises to evaluate the decision model.

Expert Interviews

Twenty-six domain experts from different software producing organizations have participated in this research to answer the research questions and build a decision model for the MDD platform selection problem. *Expert Interview* is an essential knowledge acquisition technique (Chen, 2004) in qualitative research. The primary source of knowledge to build a decision model is domain expertise. Please note that these interviews are different from the interviews we conducted during the case study interviews with the case participants.

We followed Myers' and Newman's guidelines (Myers & Newman, 2007) to conduct a series of qualitative semi-structured interviews with senior software engineers to explore expert knowledge regarding MDD Platforms and evaluate the outcomes of our study.

We developed a role description before contacting potential experts to ensure the right target group. Then, we contacted the experts through email using the role description and information about our research topic. The experts were pragmatically and conveniently selected according to their expertise and experience mentioned on their *LinkedIn* profile. We considered a set of expert evaluation criteria (including "Years of experience", "Expertise", "Skills", "Education", and "Level of expertise") to select the experts.

Each interview followed a semi-structured interview protocol and lasted between 45 and 60 minutes. We used open questions to elicit as much information as possible from the experts minimizing prior bias. All interviews were done in person and recorded with the interviewees' permission, then transcribed for further analysis.

Acquired knowledge during each interview was typically propagated to the next to validate the captured knowledge incrementally. Finally, the findings were sent to the interview participants afterward for final confirmation. Note, for the validity of the results, the research's data collection phases were not affected by the case study participants; moreover, none of the interviewees or researchers were involved in the case studies.

Please note that we did not use formal coding for the analysis of the interviews and the literature. What we did do, however, could be termed incremental concept development. During the literature study and interviews, concepts were identified that were relevant. Candidate qualities and features were identified, defined, and fine-tuned with the interviewees and later confirmed by asking the interviewees for post-analysis of the interview and literature results. While this did not constitute formal coding, we did mark concepts related to the domain, came up with the literature study, and came up with the interviews. Secondly, these concepts were incrementally fine-tuned until an agreement was reached with the interviewees.

Document analysis

Document analysis is a systematic procedure for reviewing or evaluating documents, including text and images that have been recorded without a researcher's intervention (Bowen et al., 2009). Document analysis is one of the analytical methods in qualitative research that requires data investigation and interpretation to elicit meaning, gain understanding, and develop empirical knowledge (Corbin & Strauss, 2014). We reviewed webpages, whitepapers, scientific articles, fact sheets, technical reports, product wikis, product forums, product videos, and webinars to map the platforms' MDD features.

One of the principal challenges while document analysis is the lack of standard terminology among MDD platforms. Sometimes different MDD platforms refer to the same concept by different names, or even worse, the same name might stand for different concepts in different MDD platforms. Discovering conflicts is essential to prevent semantic mismatches throughout the knowledge extraction phase.

We followed the framework to capture a conceptual phrase representing a segment of data related to a particular research question. In other words, we used a conceptual mapping of MDD concepts to identify potential conflicts in knowledge sources.

For instance, conceptually similar phrases and definitions regarding MDD platforms' features were collected together for more detailed analysis. Based on the framework, document analysis and conceptual mapping were employed to extract knowledge from the selected sources of knowledge and prevent semantic mismatches.

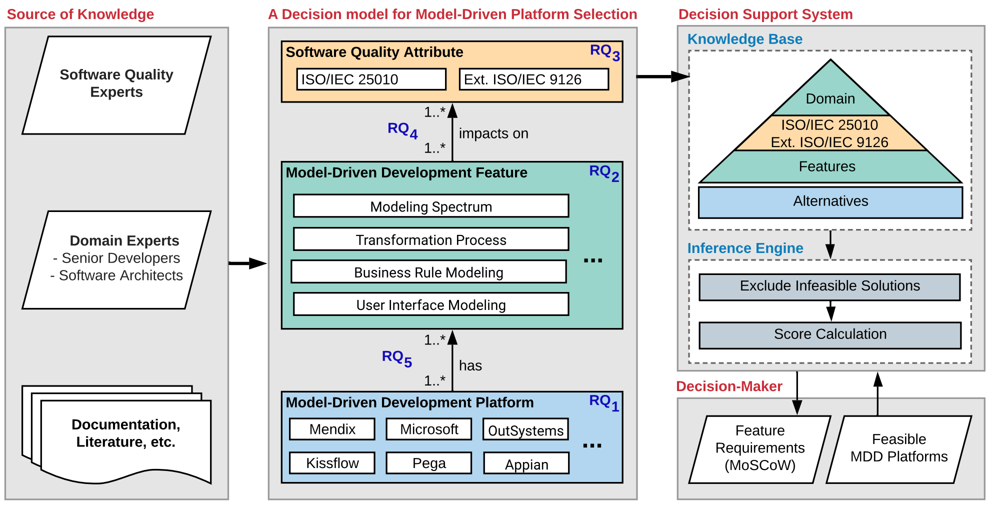
We defined an extract extraction form to obtain consistent extraction of relevant knowledge and checked whether the acquired knowledge would address the research questions. The extracted knowledge, which correspond to the research questions, has been classified into five categories: *quality attributes*, *MDD platforms*, *MDD features*, *impacts of the MDD features on the quality attributes*, and *supportability of the MDD features by the MDD platforms*. Next, the extracted knowledge was employed to build a decision model for the MDD platform selection problem. Finally, the decision model was uploaded to the knowledge base of the DSS.

Case Study

Case Study is an empirical methodology that investigates a phenomenon within a particular context in the domain of interest (Yin, 2017). A case study can be employed to collect data regarding a particular phenomenon or to apply a tool and evaluate its efficiency and effectiveness using interviews. Yin (2017) distinguishes four types of case study designs according to holistic versus embedded and single versus multiple. In this study, we employ holistic multiple case designs: examining multiple real-world companies' cases within their context to learn more about one specific unit of analysis and evaluating the decision model for the MDD platform selection problem. To conduct the case studies and evaluate the proposed decision model, we followed the following *case study protocol*:

1. **- Step 1: Requirements elicitation.** At least two managers or team leaders of the case study companies' IT departments should participate in the research, as such participants are thoroughly informed on the design decisions and the requirements of their decision context. During the interview session with each company's case study participants, we first ask them to explain the decision context and requirements. Next, we show the MDD feature list to the participants and explain the features completely. Afterward, we ask the participants to identify their MDD feature requirements and prioritize them based on the MoSCoW prioritization technique (DSDM consortium and others, 2014). Additionally, they have to express the rationales behind the requirements elicitation. Finally, they should identify a set of MDD platforms as potential solutions for their software projects.
2. **- Step 2: Results and recommendations.** We need to define four separate cases in the DSS knowledge base according to the case studies' requirements and priorities. Next, the DSS can suggest a set of feasible solutions per case individually. Then, the outcomes should be discussed with the case study participants.
3. **- Step 3: Analysis.** We need to compare the DSS feasible solutions with the case study participants solutions suggested by the experts at the case study companies based on their internal meetings. Furthermore, we should analyze the outcomes and observations and then report them to the case study participants and receive their feedback on the results.

Figure 6.1: This figure is adapted from our previous study (Farshidi et al., 2018a) and shows the main building blocks of the decision support system beside the proposed decision model for the MDD Platform selection problem.



We ensured validity in the conversations with the case participants in the following ways. First, we made sure that all terms we used in the discussions were known by providing the list of features and qualities and discussing these terms with the participants during the interviews. Secondly, the discussions with the case participants were noted down by the researchers during the interviews, and these were processed within 24 hours to ensure that none of the results would be forgotten. We did not record the meetings to avoid tension during these discussions. Finally, we confirmed the results from the decision support model with the case participants and discussed whether our inputs into the decision support tool were correct.

We have considered other study designs, such as action research studies, to support the engineers during their selection. However, by performing these case studies post hoc, we could ensure that the case participants had used their selected platforms for a more extended period of time. Another possibility would have been to survey end-users of MDD Platforms. However, as we were particularly interested in how our model and tool were used, performing multiple case studies at companies provided us with the highest level of detail for the empirical results.

6.4 MCDM for MDD Platform Selection

We follow the framework (Farshidi et al., 2018a) as modeled in Figure 6.1 to build a decision model for the MDD platform selection problem. Generally speaking, a decision model for an MCDM problem contains decision criteria, alternatives, and mappings. Figure 6.1 represents the main building blocks of the decision support system besides the proposed decision model.

6.4.1 RQ_1: MDD Features

Domain experts were the primary source of knowledge to identify the right set of MDD features, even though documentation and literature studies of MDD platforms can be employed to develop an initial hypothesis about the MDD feature set. Each MDD feature has a data type, such as *Boolean* and *non-Boolean*. For example, the data types of MDD features, such as the *popularity in the market* and supportability of *Real-time Analytics*, can be considered as *non-Boolean* and *Boolean*, respectively.

The initial set of MDD features was extracted from the following sources: White papers, Fact sheets, Technical reports, Instruction manuals, Product wikis, Product forums, Product videos, Webinars. Additionally, twenty-three domain experts have participated in this phase of the research to identify a potential list of MDD features. Accordingly, 90 Boolean and four non-Boolean MDD features were identified and extracted from the expert interviews' results. Eventually, the validity and reliability of the final list of the features[2] was evaluated and confirmed by the domain experts.

6.4.2 RQ_2: MDD Platforms

To answer the second research question, we identified 104 MDD platforms as our initial hypothesis based on the following three methods:

- Exploring literature with the keywords “Low-Code”, “No-Code”, “BPMS”, “iBPMS”, and “Model-Driven Development” platforms.
- Exploiting our network of domain experts, including software engineers and academics. Note, we conducted 26 expert interviews.
- Asking interviewed domain experts at the end of each interview whether they know of products that should be researched.

Next, we reviewed the published surveys and reports from well-known knowledge bases, such as Gartner (Dunie et al., 2019; Vincent et al., 2019) and Forrester (Rymer et al., 2019); eventually, we selected 30 MDD platforms that at least three sources of knowledge confirmed the necessity of their existence in the decision model. The first row of Table 6.1 shows the list of the selected MDD platforms.

6.4.3 RQ_3: Software Quality Attributes

Based on the IEEE Standard Glossary of Software Engineering Terminology (Committee et al., 1998; Samadhiya et al., 2010), the quality of software products is the degree to which a system, component, or process meets specified requirements (such as functionality, performance, security, and maintainability) and the extent to which a system, component or process meets the needs or expectations of a user. It is necessary to find quality attributes widely recommended by other researchers to measure the system's characteristics.

The literature study results approved that researchers do not agree upon a set of conventional criteria, including quality attributes and features, to evaluate the MDD

[2]The entire list of the MDD features and supportability of considered MDD platforms are available and accessible on the data repository (Farshidi et al., 2020d).

platforms (See Table 6.6). Additionally, we realized that their suggested criteria were mainly applied to specific domains to address different research questions. Consequently, a set of nonexclusive and domain-independent criteria is needed to evaluate MDD platforms.

The ISO/IEC 25010 (ISO, 2011) presents best practice recommendations on the base of a quality assessment model. The quality model defines which quality characteristics should be considered when assessing the qualities of a software product. A set of quality attributes should be defined in the decision model (Farshidi et al., 2018a). In this study, we used the *ISO/IEC 25010* standard (ISO, 2011) and *extended ISO/IEC 9126* standard (Carvallo & Franch, 2006) as two domain-independent quality models to analyze MDD features based on their impact on quality attributes of MDD platforms. The key rationale behind using these software quality models is that they are a standardized way of measuring a software product. Moreover, they describe how easily and reliably a software product can be used.

The last four columns of Table 6.6 show the results of our analysis regarding the common criteria and alternatives of this study with the selected publications. Let us define the coverage of the i-th selected study as follows:

$$Coverage_i = \frac{CQ_i + CF_i}{C_i} \times 100$$

Where CQ_i and CF_i denote the numbers of common quality attributes (column #CQ) and features (column #CF) of the i-th selected study; furthermore, C_i signifies its number of suggested criteria. The last column (*Cov.*) of Table 6.6 indicates the percentage of the coverage of the considered criteria within the selected studies. On average, 75% of those criteria are already considered in this study.

6.4.4 RQ_4: The impacts of MDD features

The mapping between the sets *software quality attributes* and *MDD platforms* was identified based on domain experts' knowledge. Three domain experts participated in this phase of the research to map the MDD platforms (*Features*) to the quality attributes (*Qualities*) based on a Boolean adjacency matrix[3] ($Qualities \times Features \rightarrow Boolean$). For instance, *Entity-Attribute-Relationship (EAR)* as an MDD feature influences the *functional correctness* quality attribute. The framework does not enforce an MDD feature to present in a single quality attribute; MDD features can be part of many quality attributes. For example, *Native modeling tool* as an MDD feature might be linked to multiple quality attributes such as *resource utilization* and *functional appropriateness*. The experts believed that about 74% percent of the MDD features impact the following key characteristics of the MDD platforms:

- **Functional suitability** is defined in ISO/IEC 25010 as the degree to which an MDD platform functions that meet the stated or implicit requirements when used under specific conditions.
- **Usability** defines the degree to which an MDD platform can be used to achieve

[3]The acquired knowledge regarding the impacts of the MDD platforms on the quality attributes was used to calculate the Impact Factors (Farshidi et al., 2018a) that apply in the score calculation of the DSS. The final Boolean adjacency matrix is available online on the data repository (Farshidi et al., 2020d).

Table 6.1: This table shows the first part of the Boolean Features ($Feature^B$), MDD Platforms ($Platforms$), and the "BFP" mapping. Note, 1s on each row indicates that the corresponding platforms support the MDD feature of that row, and 0s signify the corresponding platforms do not support that feature, or we did not find any strong evidence of their supports based on the documentation analysis. Moreover, the rows in black indicate the categories of the features, and the rows in blue show the features, and the rows below them are their subfeatures. The definitions of the features are available on the data repository (Farshidi et al., 2020d).

Boolean Features MDD Platforms (BFP)	**Coverage**	Outsystems	Mendix	Appian	Salesforce (Lightning)	Microsoft PowerApps	Kissflow	Zoho (Creator)	ServiceNow (Now Platform)	Oracle APEX	Pega (Infinity)	Google App Maker	Quick Base	TrackVia	WaveMaker	Kony (Quantum)	GeneXus	Codeless Platforms	IBM (Digital App Builder)	Software AG (AgileApps Live)	Betty Blocks	Servoy	42windmills	Thinkwise	Usoft	Triggre	Aptean	OpenText (AppWorks 16)	WEM Modeler	GorillaIT (Karooda Platform)	AgilePoint (NX)
Platform Types																															
General-purpose platform	68.97%	1	1	0	1	1	0	1	0	1	1	1	1	0	1	1	1	1	1	0	1	1	0	1	0	1	0	0	1	1	1
Process app platform	20.69%	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1	0	0	0
Mobile app platform	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Request-handling platform	13.79%	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
Modeling spectrum																															
Model-centric (Low-code)	93.10%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	1	1	1	1
Model only (No-code)	51.72%	0	1	1	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	1	0	1	0	0	1	0	0	0	1	1
Developer Citizen																															
Domain Experts	68.97%	0	1	1	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	1	0	1	1	1	1	1	1	1	1	1
Business Analysts	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Professional Developers	65.52%	1	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	0	0	0	0	0	0	1	1
Development																															
Visual IDE	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Multi-channel/Cross-platform Application	65.52%	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	0	1	0	1	0	0	1	0	0	0	0
Programming mandatory	31.03%	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	1	1	0	0
Programming optional	65.52%	1	1	1	1	1	1	0	1	0	1	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	1	1
Integration	89.66%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0
Cross-Platform Integration	79.31%	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1	1
Integrate with an ERP system	55.17%	0	1	1	0	0	1	1	0	1	1	0	0	1	0	0	1	1	1	0	1	1	0	1	0	0	0	1	1	0	1
Data mapping	72.41%	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0
REST	82.76%	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	0	0
SOAP	68.97%	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	1	1	0	1	1	0	1	0	1	0	0
OData services	41.38%	1	1	0	1	1	0	1	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0
Importing data	68.97%	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0
Deployment	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Public Cloud platform	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Private Cloud platform	37.93%	1	1	1	0	1	0	0	0	1	0	0	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
On-premise	72.41%	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0	0	1
Access control	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Identity and permissions management	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Multi-factor authentication (MFA)	51.72%	1	1	1	1	1	1	1	0	1	1	0	1	0	0	1	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0
OAuth	75.86%	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	0	0	0	0	1	1	1	0	1
Single Sign-On (SSO)	75.86%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	1	0	0	1	1	0	1
Data Management	93.10%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1
SQL or NoSQL databases	65.52%	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	0	1	0	1	1	0	1	1	0	1	0	0	0	0
Web API	86.21%	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1
Service Calls	72.41%	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	0
Local application-specific databases	44.83%	1	1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0
Connectors to various back-ends or services	55.17%	1	1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	1	0	1	0	0	0	0	1	0	0
Real-time Analytics	68.97%	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	0	0	1
Report and Analytics	68.97%	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	0	0	1	1	0	0	1
Application lifecycle manager	82.76%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0
user stories	41.38%	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	1	0	0	0	1	1	0	0	1	0	0	0	0	0	0
Built-in team collaboration	82.76%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0
sandbox-to-production phases	34.48%	1	1	0	0	1	0	0	1	1	1	0	1	0	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
component catalogue	37.93%	1	1	0	0	1	0	0	1	1	0	0	1	0	1	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0
Application and portfolio management	51.72%	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0
Version control	48.28%	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0
Administrative controls	44.83%	1	1	1	1	1	0	1	1	1	1	0	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0
User Interface																															
Tool set	62.07%	1	1	1	1	1	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0
Form & View	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Predefined components	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Multilingual Apps	62.07%	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	0	1	1	1	1	0	0	1	0	0	0	1	0	0
Company-Branded Templates & Styling	72.41%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0
Create extensions and widget libraries	58.62%	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0
Data Modeling																															
Entity-Attribute-Relationship (EAR)	86.21%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	0	1
Object-Role Modeling (ORM)	20.69%	0	1	0	1	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
Ontology modeling	13.79%	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Business rule modeling																															
Data rules	96.55%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
Process rules	93.10%	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
Decision table	41.38%	1	1	1	1	1	1	1	1	1	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
Decision tree	44.83%	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
Expression modeling																															
Expression editor	79.31%	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	1	1	0	1
Natural language rules	27.59%	0	0	0	1	0	0	1	1	0	1	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Close to programming language	37.93%	1	1	0	0	1	0	0	1	1	0	0	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
Programmed	27.59%	1	0	0	0	1	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	0	0

Table 6.2: This table shows the second part of the Boolean Features ($Feature^B$), MDD Platforms ($Platforms$), and the "BFP" mapping. Note, 1s on each row indicates that the corresponding platforms support the MDD feature of that row, and 0s signify the corresponding platforms do not support that feature, or we did not find any strong evidence of their supports based on the documentation analysis. Moreover, the rows in black indicate the categories of the features, and the rows in blue show the features, and the rows below them are their subfeatures. The definitions of the features are available on the data repository (Farshidi et al., 2020d).

Boolean Features MDD Platforms (BFP)	**Coverage**	Outsystems	Mendix	Appian	Salesforce (Lightning)	Microsoft PowerApps	Kissflow	Zoho (Creator)	ServiceNow (Now Platform)	Oracle APEX	Pega (Infinity)	Google App Maker	Quick Base	TrackVia	WaveMaker	Kony (Quantum)	GeneXus	Codeless Platforms	IBM (Digital App Builder)	Software AG (AgileApps Live)	Betty Blocks	Servoy	42windmills	Thinkwise	Usoft	Triggre	Aptean	OpenText (AppWorks 16)	WEM Modeler	GorillaIT (Karooda Platform)	AgilePoint (NX)
Model Transformation																															
Modeling Tool	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Native modeling tool	48.28%	1	1	1	1	1	0	0	1	1	0	0	0	0	0	0	1	1	0	1	1	1	0	0	1	0	0	1	0	0	1
Web modeling tool	79.31%	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	0	1	1	0	1	1	1
Engine	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Code Generation	55.17%	0	1	0	1	1	0	1	1	0	0	0	1	0	1	1	1	1	0	0	0	1	1	0	0	1	1	0	0	1	1
Model interpretation	44.83%	1	0	1	0	0	1	0	0	1	1	1	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	1	1	0	0
Storage	82.76%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	0	0	1	0	1
XML/JSON as data storage	68.97%	1	1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	1	0	1	1	0	0	0	1	0	1
Store model locally	51.72%	1	1	1	1	1	0	1	1	0	0	0	0	1	1	0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	1
Portability	65.52%	1	1	1	1	1	0	1	1	1	1	1	0	0	1	0	1	1	1	0	1	1	1	0	0	1	0	1	1	0	0
Convert model to text	31.03%	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	1	0	0
Support different stacks	34.48%	1	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0
Plug and play architecture	37.93%	1	1	1	1	0	0	1	1	1	1	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0
Business Process Automation																															
Workflow	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Process flow	93.10%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	1
Case flow	44.83%	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1	0	0	0
Application Types																															
Mobile Apps	96.55%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
Web portals	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Web applications	100.00%	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Smartwatch (wearable) Apps	13.79%	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Security and Protection (Compliance)																															
HIPAA compliant	51.72%	0	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0
ISO 27001-2013 certification	65.52%	1	1	1	1	1	1	1	1	0	1	1	0	1	0	1	0	0	1	1	1	0	0	0	1	0	1	1	1	0	1
SOC 1	26.67%	0	1	1	0	1	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
SOC 2	56.67%	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	1	1	0	0	1
SOC 3	33.33%	0	1	1	0	1	1	0	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0
PCI DSS	34.48%	1	1	1	1	1	0	1	0	0	1	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0

specified goals with effectiveness, efficiency, and satisfaction in a specified context of use. Moreover, it embraces quality attributes such as "Learnability", "Operability", "User error protection".

- **Maintainability** is the degree to which an MDD platform can be effectively and efficiently modified without introducing defects or degrading existing product quality. For instance, the experts believe that "Modularity", "Reusability", "Analyzability", "Modifiability", and "Testability" can be considered as key strengths of the MDD platforms that support "Model only" or "Model-centric" features.
- **Supplier** includes a set of quality attributes such as "Reputation" and "Support" of the MDD platforms.
- **Cost** denotes the amount of money that a company spends on implementing a software product using an MDD platform. It includes quality attributes such as "Implementation Cost", "Platform Cost", and "Licensing Costs".
- **Product** defines a set of quality attributes regarding the state or fact of exclusive rights and control over the property. For instance, "Stability", "Ownership", and "Guarantees" are part of this characteristic.

Table 6.3: This table shows the **NFP** mapping between the **N**on-Boolean MDD **F**eatures and **P**latforms. Note, the *Popularity in the market*, *Active Community*, *Maturity level of the company*, and *Future Roadmap* are the Non-Boolean MDD features that were considered in this study. The parameters of these features are listed below each feature, for example, *Founded*, *Number of Employees*, *Type*, and *Revenue per year* are the parameters of the *Maturity level of the company*.

Non-Boolean Features-MDD Platforms (NFP)	Outsystems	Mendix	Appian	Salesforce (Lightning)	Microsoft PowerApps	Kissflow	Zoho (Creator)	ServiceNow (Now Platform)	Oracle APEX	Pega (Infinity)	Google App Maker	Quick Base	TrackVia	WaveMaker	Kony (Quantum)	GeneXus	Codeless Platforms	IBM (Digital App Builder)	Software AG (AgileApps Live)	Betty Blocks	Servoy	42windmills	Thinkwise	Usoft	Triggre	Aptean	OpenText (AppWorks 16)	WEM Modeler	GorillaIT (Karooda Platform)	AgilePoint (NX)	Source of Knowledge
Popularity in the market	H	H	H	H	H	H	H	H	H	H	M	M	M	M	M	M	M	M	L	L	L	L	L	L	L	L	L	L	L	L	Domain experts
Google hits	13.500	12.800	8.960	39.700	2.780	18.300	45.700	9.030	36.300	8.840	2.240	3.910	8.010	2.420	3.700	710	8.520	2.600	5	3.470	802	3	301	6	11	50	15	33	2	1.620	Query string: "Product Name"+ "Low code"+ "No code"
Google Trends (Means of the past 12 months)	80,84	82,78	67,67	82,61	65,14	66,96	85,49	76,14	89,20	67,55	72,78	16,63	58,69	47,51	53,35	77,69	66,61	81,00	65,41	36,71	53,88	0,00	0,00	34,76	0,00	55,53	29,53	31,92	0,00	0,00	www.trends.google.com
Twitter (follower)	23.9K	10.8K	15,9K	483.8K	23.3K	3,5K	55,3K	32,2K	6,2K	47.7K	23	5,4K	3,1K	18,3K	11,6K	6,7K	4,5K	4,5K	17	3,9K	1K	849	290	220	86	1,3K	2K	138	9	15K	www.twitter.com
Gartner	1	1	1	1	1	0	1	1	1	1	0	1	1	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	www.gartner.com
The Forrester Wave	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	www.forrester.com
softwaretestinghelp (2020)	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	www.softwaretestinghelp.com
pcmag	1	1	1	1	1	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	www.pcmag.com
TrustRadius	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	www.trustradius.com
G2	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	www.g2.com
predictiveanalyticstoday	1	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	www.predictiveanalyticstoday.com
featuredcustomers	1	1	1	0	1	1	1	0	0	0	0	1	1	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	www.featuredcustomers.com
apriorit	1	1	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	www.apriorit.com
objectivity	1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	www.objectivity.co.uk
altitudemarketing	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	www.altitudemarketing.com
Active Community	H	H	H	H	H	L	M	H	H	H	H	M	L	L	M	L	L	H	L	L	L	L	L	L	L	H	L	L	L	L	Domain experts
LinkedIn (job openings)	149	132	170	3K	8,7K	1	6	1,4K	614	913	4K	10	14	3	17	N/A	N/A	8,9K	N/A	3	20	N/A	64	74	N/A	80	N/A	N/A	N/A	N/A	www.linkedin.com
LinkedIn (Followers)	56K	20,4K	29,1K	2M	10,15M	5,3K	255,7K	277K	5M	191,65K	16,63M	7,9K	2,5K	2,1K	40,2K	9,2K	2,9K	7,97M	N/A	5,4K	1K	48	2,1K	497	211	20,20K	3,1K	N/A	157	1,2K	www.linkedin.com
Glassdoor (jobs)	132	157	280	1.9K	10K	N/A	15	4,2K	11K	646	6,8K	44	15	N/A	9	N/A	7	17K	N/A	23	2	N/A	27	N/A	N/A	116	N/A	N/A	N/A	6	www.glassdoor.com
Maturity level of the company	H	M	H	H	H	L	H	H	H	H	H	M	M	M	H	M	M	H	L	M	M	M	M	M	L	H	M	L	M	M	Domain experts
Founded	2001	2005	1999	1999	1975	2012	1996	2004	1977	1983	1998	1999	2006	2014	2007	1988	1988	1911	2013	2012	2001	2009	2002	1987	2013	2012	2001	N/A	2001	2003	www.glassdoor.com
Number of Employees	1001 to 5000	501 to 1000	1001 to 5000	10000+	10000+	101 to 250	201 to 500	5001 to 10000	10000+	1001 to 5000	10000+	201 to 500	51 to 200	51 to 200	1001 to 5000	201 to 500	51 to 200	10000+	N/A	51 to 200	51 to 200	11 to 50	51 to 200	11 to 50	11 to 50	1001 to 5000	501 to 1000	N/A	11 to 50	51 to 200	www.glassdoor.com www.crunchbase.com www.linkedin.com
Type	Private	Private	Public	Public	Public	Private	Private	Public	Public	Public	Public	Private	Private	Subsidiary	Private	Private	Private	Public	Private	Private	Private	Private	Private	Private	Private	Private	Private	Private	Private	Private	www.glassdoor.com www.crunchbase.com www.linkedin.com
Revenue per year	$100 to $500 M	$100 to $500 M	$100 to $500 M	$10+ B	$10+ B	N/A	$100 to $500 M	$2 to $5 B	$10+ B	$500 M to $1 B	$10+ B	N/A	18,6 M	15,9 M	$100 to $500 M	$5 to $10 M	N/A	$10+ B	N/A	$5 to $10 M	N/A	N/A	N/A	N/A	N/A	$100 to $500 M	N/A	N/A	N/A	$10 to $25 M	www.glassdoor.com www.crunchbase.com www.linkedin.com
Future Roadmap	H	H	M	H	H	L	L	H	H	L	H	L	M	H	L	M	L	H	L	M	L	L	L	M	L	M	H	L	L	L	Domain experts
Research and Development department	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	Product website
science-industry collaboration	1	1	1	1	1	0	0	1	1	0	1	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	0	0	Product website
Co-publications in peer reviewed journals	1	1	0	1	1	0	0	1	1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	Web of Science

6.4.5 RQ_5: Supportability of the MDD Features

An MDD platform contains a set of MDD features that can be either Boolean or non-Boolean. A Boolean MDD feature ($Feature^B$) is a feature that is supported by the MDD platform, for example, supporting the *Native modeling tool*. A non-Boolean MDD feature ($Feature^N$) assigns a non-Boolean value to a particular MDD platform; for example, the popularity in the market of an MDD platform can be "high", "medium", or "low". Accordingly, this study's MDD features are a collection of Boolean and non-Boolean features, where $Features = Feature^B \cup Feature^N$.

The mapping $BFP : Feature^B \times Platforms \rightarrow \{0,1\}$ defines the supportability of the Boolean MDD features by the platforms. So that $BFP(f, p) = 0$ means that the platform p does not support the MDD feature f or we did not find any evidence for proof of this feature's supportability by the MDD platform. Moreover, $BFP(f, l) = 1$ signifies that the platform supports the feature. The mapping *BFP* is defined based on documentation of the MDD platforms and expert interviews. Tables 6.1 and 6.2 present the Boolean Features that we have considered in the decision model.

The experts defined four non-Boolean MDD features, including "Popularity in the market", "active community", "Maturity level of the company", and "future roadmap". The assigned values to the non-Boolean MDD features for a specific MDD platform is based on a 3-point Likert scale (**H**igh, **M**edium, and **L**ow), where $NFP : Features^N \times Platforms \rightarrow \{H, M, L\}$, based on several predefined parameters. For instance, the "popularity in the market" of an MDD platform was defined based on the following parameters: the number of the "Google hits", "Google Trends (Means of the past 12 months)", "Twitter (follower)", and the popular forums and reports that considered the platform in their evaluation. Table 6.3 shows the non-Boolean programming features, their parameters, and sources of knowledge.

6.4.6 MDD Feature Requirements:

The DSS (Farshidi et al., 2018b) receives the MDD feature requirements based on the *MoSCoW* prioritization technique (DSDM consortium and others, 2014).

Decision-makers should prioritize their MDD feature requirements using a set of weights ($W_{MoSCoW} = \{w_{Must}, w_{Should}, w_{Could}, w_{Won't}\}$) according to the definition of the *MoSCoW* prioritization technique. MDD feature requirements with *Must-Have* or *Won't-Have* priorities act as hard constraints and MDD feature requirements with *Should-Have* and *Could-Have* priorities act as soft constraints. So that, the DSS excludes all infeasible MDD platforms which do not support MDD features with *Must-Have* and support MDD features with *Won't-Have* priorities. Then, it assigns non-negative scores to feasible MDD platforms according to the number of MDD features with *Should-Have* and *Could-Have* prioritizes (Farshidi et al., 2018a).

Decision-makers specify desirable values, according to their preferences, for non-Boolean MDD feature requirements. For example, a decision-maker could be interested in prioritizing MDD platforms with the *Maturity level* above average. Therefore, the *Maturity level of the company* above average is a *Should-Have* feature.

6.5 Empirical Evidence: the Case Studies

Four industry case studies in the context of four real-world enterprises have conducted to evaluate and signify the decision model's usefulness and effectiveness to address the MDD platform selection problem. To increase diversity in our evaluation, we selected the case studies from different domains, such as workflow management, project management, and Enterprise Resource Planning (ERP) systems. Furthermore, one of the case studies was a software consultancy company working in close collaboration with Mendix and was interested in evaluating other potential alternatives. We had a session with the decision-makers at each case study company to capture their functional and quality requirements, constraints and assumptions, and technology acceptance criteria. They explained their concerns and barriers, such as their budget or lack of time and technical knowledge in the development team to build the desired software product. Then, we showed the feature list[4] besides their explanations (Farshidi et al., 2020d) and asked them to identify their feature requirements based on the MoSCoW prioritization technique (Table 6.4).

Note, the case study participants have identified several potentially feasible MDD platforms for their projects through multiple internal expert meetings and investigation into MDD platforms before participating in this research (see the CSP row in Table 6.5). Four industry cases were defined based on the MoSCoW prioritization technique and stored in the DSS knowledge base. Next, the Inference Engine of the DSS generated feasible solutions for each case. The rest of the section describes the case study companies' shortlists and analyzes the DSS outcomes.

6.5.1 Case Study 1: Nederlandse Spoorwegen (NS)

Nederlandse Spoorwegen (Dutch: NS; English: Dutch Railways) is a Dutch state-owned company, the principal passenger railway operator in the Netherlands. Founded in 1938 and with more than 4500 employees, NS provides rail services on the Dutch leading rail network. The Information and Communications Technology (ICT) department of NS has more than 400 employees and is responsible for monitoring the IT projects' short-term and long-term progress.

The experts at the ICT department implemented a workflow management system called NOVA, based on the Mendix platform, to enable users to define different workflows for different types of jobs or processes in the context of IT projects. The case study participants stated that NOVA mappings out the IT projects' workflows in an ideal state; it finds redundant tasks, automates the processes, and identifies bottlenecks or areas for improvement.

The case study participants stated that they want to evaluate a shortlist of feasible MDD platforms that they came up with through an extensive investigation into potential alternatives. They considered Mendix and Microsoft PowerApps as two MDD platforms before participating in this research. The case study participants mentioned that the third alternative for them could be "High-Coding" or implementing a workflow management system by hiring a team of senior software engineers and architects.

NS Requirements

The case study participants at NS defined a subset of the key functionality of NOVA as follows:

- It gets a clear overview of work in progress in a particular workflow (R17).
- It allows creating, prioritizing, assigning tasks, divides tasks into workflow stages, decides who works on what part, and allows monitoring work moving through multiple stages (R1, R16, R17).
- It enables centralized governance, so creating task lists, adding tasks/subtasks, subscribing to the entire tasklist, assigning tasks/subtasks to one or multiple people, logging activities, and scheduling events can be managed and monitored in one place (R5, R6).
- It creates custom roles and grants access to people based on their responsibilities (R1).

[4]The Boolean and non-Boolean features are presented in Tables 6.1, 6.2, and 6.3

- It provides a dynamic structure for executing non-routine unpredictable business processes that require coordination of multiple tasks and complex decision-making (R8, R16).
- It has a policy access-control mechanism that restricts access to authorized users and defines users' roles and privileges (R1).

The participants expressed that the MDD platform should be popular enough in the market (R24), as popularity indicates that an MDD platform is widely purchased by other businesses and has an active community (R58). Furthermore, they believed that the maturity level of the MDD platform is an essential criterion that assesses the efficiency and effectiveness of an MDD platform (R9). Additionally, the potential MDD platform should deploy the workflow management system on a private cloud due to higher security, flexibility, and availability (R6).

The case study participants identified their feature requirements based on the MoSCoW prioritization technique (See Table 6.4). Then, we defined a case according to their requirements and priorities in the DSS knowledge base.

Results and Analysis

The case study participants at NS identified 40 MDD feature requirements that more than half of them prioritized as "Must-Have" features (see Table 6.5). The DSS excluded 26 MDD platforms from the 30 platforms in its knowledge base and offered four potential MDD platforms to NS. Table 6.5 shows that Mendix was the first feasible platform for NS. Additionally, Oracle APEX, Microsoft PowerApps, and OutSystems were scored as the second to fourth potential solutions.

The case study participants were looking for a Low-code or No-code platform to prioritize "Model-only" (R20) and "Model-centric" (R10) as two Should-Have features. Based on our assessment, Mendix and Microsoft PowerApps support both of these features. "Ontology modeling" (R27) as a Should-Have feature is only supported by Oracle APEX. Moreover, "Plug and play architecture" as another Should-Have feature does not support by Microsoft PowerApps.

The experts who participated in this case study expressed that NOVA is currently based on Mendix as they had some legal limitations to select a suitable platform that meets their requirements. NS is a semi-government organization that should follow some government bureaucracy; for instance, they have to deploy NOVA on a national platform that uses a local data center inside the Netherlands borders. They mentioned that the DSS results showed that they made the right decision in their selection process; additionally, they can consider more potential solutions in their future evaluation.

6.5.2 Case Study 2: Innovation-Kite customer

Innovation-Kite is a software development company in the Netherlands and Germany with more than 500 employees. They have a "Solution Center" with an international network of experienced ICT-specialists and developers. The experts at Innovation-Kite stated that the agile development methodology requires closer interaction between end-users and developers. The local Agile / Scrum Business Engineering should make general use of specialized developers who can make specific integrations and adjust-

Table 6.4: This table represents the entire list of the feature requirements that were defined by the case study participants. Note, the Boolean and non-Boolean features that are presented in Tables 6.1, 6.2, and 6.3. The first column signifies the requirement id (RID). We used this column to determine the link between the feature requirements and the actual needs of the case study participants.

RID	MDSD Feature Requirements	Nederlandse Spoorwegen (NS)	Innoviation-Kite Customer	Bizzomate	Royal IHC
R1	Identity and permissions management	Must-Have	Must-Have	Must-Have	Must-Have
R2	SQL or NoSQL databases	Must-Have	Must-Have	Must-Have	Must-Have
R3	Tool set	Must-Have	Must-Have	Must-Have	Must-Have
R4	Predefined components	Must-Have	Must-Have	Must-Have	Must-Have
R5	Web portals	Must-Have	Must-Have	Must-Have	Must-Have
R6	Private Cloud platform	Must-Have	Should-Have	Must-Have	Must-Have
R7	Entity-Attribute-Relationship (EAR)	Must-Have	Must-Have	Should-Have	Must-Have
R8	Data rules	Must-Have	Must-Have	Should-Have	Must-Have
R9	Maturity level of the company	Must-Have	Must-Have	Should-Have	Must-Have
R10	Model-centric (Low-code)	Should-Have	Must-Have	Must-Have	Should-Have
R11	Business Analysts	Should-Have	Should-Have	Must-Have	Must-Have
R12	Form & View	Must-Have	Should-Have	Should-Have	Must-Have
R13	Plug and play architecture	Should-Have	Must-Have	Should-Have	Must-Have
R14	Programming optional		Must-Have	Must-Have	Must-Have
R15	Workflow		Must-Have	Must-Have	Must-Have
R16	Process rules	Must-Have	Must-Have	Should-Have	Could-Have
R17	Case flow	Must-Have	Must-Have	Could-Have	Should-Have
R18	Decision table	Should-Have	Must-Have	Could-Have	Must-Have
R19	Process flow		Should-Have	Must-Have	Must-Have
R20	Model only (No-code)	Should-Have	Should-Have	Must-Have	Could-Have
R21	Decision tree	Should-Have	Must-Have	Could-Have	Should-Have
R22	Visual IDE	Should-Have	Should-Have	Could-Have	Must-Have
R23	Professional Developers		Must-Have	Could-Have	Must-Have
R24	Popularity in the market	Should-Have	Should-Have	Should-Have	Should-Have
R25	Expression editor		Should-Have	Should-Have	Must-Have
R26	Web modeling tool		Should-Have	Could-Have	Must-Have
R27	Ontology modelling	Should-Have	Could-Have	Could-Have	Should-Have
R28	Native modeling tool		Could-Have	Could-Have	Must-Have
R29	Object-Role Modeling (ORM)		Should-Have	Could-Have	Should-Have
R30	Natural language rules		Should-Have	Could-Have	Should-Have
R31	Support different stacks		Could-Have	Should-Have	Should-Have
R32	XML/JSON as data storage	Must-Have		Could-Have	
R33	Integrate with an ERP system			Must-Have	Could-Have
R34	Code Generation			Could-Have	Must-Have
R35	Two step generation			Could-Have	Must-Have
R36	Store model locally			Could-Have	Must-Have
R37	General-purpose platform	Must-Have			
R38	REST	Must-Have			
R39	SOAP	Must-Have			
R40	OData services	Must-Have			
R41	Web API	Must-Have			
R42	Service Calls	Must-Have			
R43	Connectors to various back-ends or services	Must-Have			
R44	Company-Branded Templates & Styling	Must-Have			
R45	Domain Experts		Could-Have	Should-Have	Could-Have
R46	Convert model to text		Should-Have	Could-Have	Could-Have
R47	Real-time Analytics	Could-Have	Could-Have	Could-Have	Could-Have
R48	Report and Analytics	Could-Have	Could-Have	Could-Have	Could-Have
R49	Version control	Could-Have	Could-Have	Could-Have	Could-Have
R50	Close to programming language			Could-Have	Should-Have
R51	Model interpretation			Could-Have	Should-Have
R52	Public Cloud platform		Should-Have		
R53	Create extensions and widget libraries	Should-Have			
R54	Multi-channel/Cross-platform Application	Could-Have			
R55	data mapping	Could-Have			
R56	importing data	Could-Have			
R57	Mobile Apps	Could-Have			
R58	Active Community	Could-Have			
R59	Future Roadmap	Could-Have			
R60	Programmed				Could-Have

ments (customizations) within an existing customer infrastructure and environment, so that test work and specific optimizations can also be performed cost-effectively.

One of their customers was a small startup company with around ten employees. The startup company requested a software application to help them estimate activities, scheduling, cost control, and budget management. The experts at Innovation-Kite wanted to design and implement a customized project management system for this customer by employing one of their strategic technology partners, Betty Blocks, and Mendix. Additionally, the case study participants stated that without the time and budget limitation of their customers, they have enough in-House expertise and knowledge to build an entirely new software product so that High-Coding can be considered the third potential solution.

The case study participants joined this research to evaluate the shortlist of potential solutions (Betty Blocks, Mendix, and High-Coding) for this startup company. Moreover, they desired to know about other possible MDD platforms that they have to take into account.

Innovation-Kite Requirements

The case study participants at Innovation-Kite defined a subset of the customized project management system's key functionality:

- Viewing progress across all ongoing projects, identifying projects at risk, monitoring timelines, and sharing project status in real-time (R15, R47, R48)
- Keeping workflow tools in one place, having a centralized management unit for details and updates, storing projects' files in a secured data storage, and keeping templates always consistent (R2, R5, R6, R52)
- Reporting a clear picture of how the resources are being used (R48)
- Offering account management and provisioning system to define new end-users, roles, and privileges (R1)
- Managing all activities and tasks required to maintain a desired level of excellence (R19)

The Innovation-Kite experts expressed that they want to be independent of particular programming languages and development processes. However, they required a level of flexibility to add new functionality or customize an existing component. Therefore, they prioritized the "Model-centric (Low-code)" feature as Must-Have and the "Model only (No-code)" as a Should-Have feature (R10, R20). Moreover, they wanted to employ their technical knowledge so that a potential MDD platform has to support professional developers (R23). The case study participants mentioned that their customers always want to deploy their software products on the cloud; however, according to their budgets and infrastructures, it can be on private or public clouds; thus, they considered both of these options as Should-Have features (R6, R52).

Results and Analysis

The case study participants identified 37 MDD feature requirements, including 43.24% hard-constraint features (Must-Have) and 56.76% soft-constraint features (Should-Have and Could-Have). The DSS suggested five possible solutions, namely Mendix, Salesforce (Lightning), Betty Blocks, OutSystems, and ServiceNow (Now Platform). Table 6.5 shows that Mendix was the first, and Betty Blocks was the third feasible platform for this case study. They did not consider Salesforce (Lightning) as a solution, as they had no experience with employing this platform.

The DSS scored Betty Blocks as the third solution as it does not support "Decision

Table" (R18), "Natural language rules" (R30), and "Object-Role Modeling" (R29). According to our assessment, Mendix does not support "Natural language rules" and "Ontology modeling". Please note that the Should-Have features have higher priorities than Could-Have features, so MDD platforms that support more Should-Have features scores higher.

The experts who participated in this case study stated that "OutSystems" could be a potential solution as it supports all the feature requirements that they required. However, they needed to perform a cost-benefit analysis to evaluate its usefulness. The experts mentioned that the DSS could support them for their future evaluation, reducing decision-making time. However, we need to keep the knowledge base of the DSS besides the decision model regularly up-to-date.

6.5.3 Case Study 3: Bizzomate

Bizzomate is a software consultancy company in the Netherlands to support organizations with technical and technological problems, such as MDD platform selection. Organizations hire external consultancy companies, such as Bizzomate, when their internal resources and expertise are insufficient. External consultants analyze an organization's existing setup and make suggestions. Additionally, the experts at Bizzomate advises its customers on how to configure their software applications, write code, fix bugs, or customize their software systems for specific tasks or businesses.

The experts at Bizzomate stated that they use MDD platforms to increase agility in software development. Such platforms assist them with working closely together in one environment, and various stakeholders can collaborate, create, iterate, and release solutions in a fraction of the time compared to traditional development methods.

Mendix is the leading partner of Bizzomate in the software development process. The experts at Bizzomate typically employ the Mendix platform to implement customized software solutions for their customers. They have recently investigated a little bit regarding other potential platforms and considered Betty Blocks an alternative solution for their customers. The case study participants joined this research to evaluate the selected shortlist of MDD platforms based on requirements that typically take into account to build a customized software solution.

Bizzomate Requirements

The case study participants at Bizzomate defined a subset of their essential MDD feature requirements that they generally consider to select an MDD platform for their customers:

- The MDD platform has to enable developers to use models to develop software (R10) and generate code automatically (R35).
- Developers have to use models to build software and solely communicate with each other about the system in terms of models (R20). Note, coding terminology is absent.
- A full application could be created without any programming, but developers can use one programming language (R14).
- The platform has to be able to govern users by enforcing both authentication and authorization. Authentication verifies a user's identity. Once authenticated,

the verified user may use any of the resources their account is authorized to access (R1).

- Toolset gives modelers access to a what-you-see-is-what-you-get editor, in which access is provided to different user interface components. Within such an editor, the user is free to edit those elements' height and width (R3).
- The platform has to allow users to model user interface predefined components without altering the user interface components' location, width, and height (R4).
- The platform has to support developers with designing and implementing web portals (R5). A web portal is a specially designed website that brings information from diverse sources, like emails, online forums, and search engines, together in a uniform way. Usually, each information source gets its dedicated area on the page for displaying information; often, the user can configure which ones to display.
- The platform should support EAR[5] (R7), which is used to represent attributes as well as entities and relationships.

The participants expressed that non-Boolean features such as popularity in the market (R24) and the company's maturity level (R9) play an essential role in the MDD platform selection process.

Results and Analysis

The case study participants identified 42 MDD feature requirements, including 30.95% hard-constraint features (Must-Have) and 69,05% soft-constraint features (Should-Have and Could-Have). They defined a limited number of Must-Have features as they did not focus on a particular IT project. The case study participants put their emphasis on the platform-specific features that they were already familiar with. In other words, their feature requirements were biased to the features that their shortlist of MDD platforms were supported them. Thus, the DSS results did not surprise them. Table 6.5 shows that Mendix was the first, and Betty Blocks was the third feasible platform for this case study.

The DSS scored Betty Blocks as the third solution as it does not support a set of the soft constraint requirements, such as "Code Generation" (R34), "Decision Table" (R18), "Natural language rules" (R30), and "Object-Role Modeling" (R29).

The case study participants stated that they have never considered Appian as an alternative solution for their project because they do not have any experience with this platform; however, they will investigate its functionality and possibilities for future projects.

6.5.4 Case Study 4: Royal IHC

Royal IHC is an international supplier of innovative equipment, ships, and services for offshore, dredging, and wet mining. Royal IHC enables customers to execute complex projects from the water surface to the ocean floor in the most challenging maritime environments. The head office is located in the Netherlands, but more than 3,000 employees work from offices worldwide. Thus, customer support is provided

[5]Entity-Attribute-Relationship

Table 6.5: DSS Results

		Nederlandse Spoorwegen (NS)	**Innovation-Kite Customer**	**Bizzomate**	**Royal IHC**
	Context	Workflow management system	Project management system	Software consulting company	ERP system
	#Employees	5-10	5-10	20-50	50-100
Requirements	Must-Have	52.50%	43.24%	30.95%	50.00%
	Should-Have	25.00%	37.84%	23.81%	28.57%
	Could-Have	22.50%	18.92%	45.24%	21.43%
	Won't-Have	0.00%	0.00%	0.00%	0.00%
	#Feature Req.	40	37	42	42
CP Shortlist	1	Mendix	Betty Blocks	Mendix	Mendix
	2	Microsoft PowerApps	Mendix	Betty Blocks	Betty Blocks
	3	High-Code	High-Code		
DSS Solutions	1	Mendix 91%	Mendix 93%	Mendix 99%	Mendix 77%
	2	Oracle APEX 86%	Salesforce (Lightning) 92%	Appian 90%	OutSystems 67%
	3	Microsoft PowerApps 84%	Betty Blocks 72%	Betty Blocks 87%	Betty Blocks 48%
	4	OutSystems 77%	OutSystems 71%		
	5		ServiceNow (Now Platform) 70%		

on every continent. The company faces ever-changing customer needs and healthy global competition.

Digitization is an inevitable factor for every organization, including Royal IHC, to provide customers with innovative solutions. The case participants asserted that Royal IHC would gain increased business agility, transparency, and uniformity of information by consolidating the IT environment, a vital part of its national and international value chain. Recently, the ICT department at Royal IHC introduced an ERP application called “One IHC”, based on the low-code platform from Mendix, to support their key strategic goals of collaboration, globalization, and growth.

Royal IHC participated in this research to assess their current platform (Mendix) and an alternative to it (Betty Blocks). Moreover, they wanted to know about any potential MDD platforms that could be considered in the near future.

Royal IHC Requirements

The case study participants defined the following subset of requirements of One IHC to select the best fitting MDD platform.

- The platform has to support Royal IHC’s demand for applications with a native mobile experience on multiple devices. The platform needs to be available both online and offline (R22, R26, R29).
- The platform has to enhance the collaboration between business stakeholders and IT (R11). It leads to an increase in the organization’s development capacity to meet the growing demand for applications, dashboards, and portals (R23, R47, R48).
- The platform has to support component-based architectures (R13).

- The platform has to enable users to save models as byte code on their local machine (R36).
- The platform could be integrated with an ERP system (R33).
- A set of data rules must be defined to ensure that only values compliant with the data rules are allowed within a data object. Data rules will form the foundation for correcting or removing data (R8).

Results and Analysis

The case study participants identified 42 MDD feature requirements that half of them were Must-Have features. Their hard-constrained feature requirements were mainly biased toward the features the shortlist of MDD platforms already supported, and they were completely aware of them. They wanted to know about alternative solutions, so the other half of their feature requirements were mainly about nice-to-have MDD features (soft-constrains). Table 6.5 shows that Mendix was the first, and Betty Blocks was the third feasible platform for this case study.

The DSS scored Betty Blocks as the third solution as it does not support a set of the soft constraint requirements, such as "Code Generation" (R34), "Decision Table" (R18), "Ontology modeling (R27)" (R30), and "Object-Role Modeling" (R29).

The case study participants asserted the OutSystems platform could be an interesting alternative to them, and they have not considered it up to now because of a lack of knowledge and expertise regarding this platform; however, they will consider it an option in their future evaluations. Note, Royal IHC hired some experts from Bizzomate to support them with their decision-making process, so it was not surprising that their shortlists were the same.

6.6 Discussion

6.6.1 Case Study Participants

Software products may be more successful in some regions. Not every MDD platform is equally represented in different regions of the world. We observed specific MDD platforms that are primarily active in the Netherlands and focus their support efforts on the Netherlands because there is the most business for them. As aforementioned, the case study companies were located in the Netherlands. Almost all of them preferred to select one of the locally produced MDD platforms, specifically Mendix and Betty Blocks, because of their concerns regarding legal issues and safety. While we did not collect each platform's sales data, we have to remain cognizant that some MDD platforms may be over- or underrepresented in particular geographic regions.

The total cost of ownership of MDD platforms plays an inevitable factor in the decision-making process. However, none of the case study participants considered it a "Must-Have" feature, as they believed that functional suitability, maturity, and popularity of potential solutions should be prioritized higher. One of the case study participants expressed that "we have to examine total costs besides total benefits to make a rational decision about the potential solution that will provide the highest positive impact on the future of our business". The DSS assigns higher scores to the general-purpose platforms, such as Mendix and Appian, as they offer a vast set of services and functions.

Biases, such as motivational and cognitive (Montibeller & Winterfeldt, 2015), arise because of shortcuts or heuristics that decision-makers use to solve problems and perform tasks. The Hawthorne effect (Jones, 1992), which is the tendency for decision-makers to change their behavior when they are being observed, is a form of cognitive bias. The case study participants might have been more careful in the observational setting than in the real setting because they are being observed by scientists judging their selected MDD feature requirements and priorities. Moreover, the Bandwagon effect (Nadeau et al., 1993), which is the tendency to do or believe things because many other decision-makers do or believe the same, is another form of cognitive bias. The Bandwagon effect typically shows up in group decisions. To mitigate the Hawthorne and Bandwagon effects, individual *and* group interviews have been conducted.

Please note that sometimes the case study participants are biased toward a specific alternative solution. For instance, in this study, all case study participants asserted that Mendix was one of their alternative solutions. Regional limitation, popularity in the market, financial plus political issues, and tacit knowledge of the case study participants can be considered potential factors were limiting alternative solutions. Accordingly, conducting case studies in different regions could lead to different MDD feature requirements; consequently, the DSS can suggest different rankings or even entirely different alternative solutions.

6.6.2 Domain Experts

The experts expressed that the decision-making process is a lot different for small organizations than large ones. Looking at the IT landscape, we notice a difference in the selection process because the requirements of small and large organizations are remarkably different. Small enterprises typically start purchasing a unique MDD platform to solve multiple problems; they cannot invest in multiple platforms to perform different tasks because of financial constraints. Larger enterprises can invest more money to employ multiple MDD platforms for different tasks. However, using multiple platforms requires more training costs and knowledge sharing possibilities. The best-fitting MDD platform for a company should add values instead of just solving quick issues.

The experts asserted that MDD platforms should not be employed in three use cases: (1) Complex applications with rich functionality, such as software products, require continuous development and maintenance to integrate a significant number of services and components from third parties. Thus, an MDSM platform is not the best way to develop complex applications, and it is better to hire a development team that can address the functional requirements and quality concerns. (2) The MDD platforms should not be used to build applications for enterprises that employ the generated applications to perform their core businesses. The key rationale to avoid using MDD platforms for such scenarios is that MDD platforms are mainly designed for simple reoccurring problems, and they limit creativity by simplifying the complexity of the real-world. Thus, such enterprises will be limited to a set of predefined functions and cannot make new wild decisions. (3) The MDD platforms would not be a cost-effective solution for businesses that rely principally on freemium end-users, as MDD platforms may charge their customers based on the number of their end-users.

Software development is an iterative and incremental process, based on a collection of invaluable concepts and principles (Beck et al., 2001; Pressman, 2005). As extensively discussed in (Asadi & Ramsin, 2008; Embley et al., 2011), it should be kept in mind that MDD is not a concrete methodology, but a generic approach that can be applied to software development processes to take advantage of its promises. The main issue with using models to drive software engineering directly is that they are far from flexible. First, end-users are limited by the type of the MDD platforms they use. Second, they are only flexible in the parts of the solution covered by the used Domain-Specific Languages. The higher level of abstraction, the more commonalities will be 'hard-coded' in the MDD platforms. Third, sometimes models are made flexible at only a limited set of predefined components by lower-level languages.

6.6.3 The Decision Model

The case study participants confirm that the updated and validated version of the DSS is useful in finding the shortlist of feasible solutions. Finally, it reduces the time and cost of the decision-making process. Our website[6] is up and running to keep the knowledge base of the decision support system up-to-date and valid. We aim to create a community around the platform that regularly updates the curated knowledge base with new MDD platform features. We consider it as future work to enable third parties to add new features and products to the database in a wiki-style manner. These additions need to be approved by us, as it may be tempting for product marketers to overstate the features present in the platform.

The study of heuristics-and-biases has investigated various decision-making shortcuts and has documented their inferior performance (Kahneman et al., 1982; Tversky & Kahneman, 1974). However, these uncomplicated heuristics can be viewed as smart approaches to save time so that a decision-maker can respond immediately (Gigerenzer & Selten, 2002). Applying simple rules is sometimes an answer to complexity (Simon, 1955). When faced with a problem that is highly complex to solve optimally, the decision-maker falls back on a simple rule that makes sense based on what is understood. Fast-and-frugal heuristics can perform well in certain domains (Gigerenzer & Todd, 1999), such as MDD selection, to find the best fitting alternatives based on a limited set of criteria, for instance, background knowledge and experience of the decision-maker. Thus, the decision model can be considered a method to evaluate the shortlist of decision-makers' alternative solutions and assist them with decision-making under uncertainty.

We believe that the theoretical contribution and the answer of the main research question (see section 6.3.2) of this study is a decision model that can be used to make informed decisions in software production, and models from software engineering, such as the ISO standard quality model and the MoSCoW prioritization technique, are fundamental building blocks in such decisions. Researchers can replace the ISO standard quality model with more specific quality attributes to customize the decision model. Although we employ the MoSCoW prioritization technique to simplify the

[6]`https://dss-mcdm.com`

understanding and manage priorities, other researchers can employ other types of prioritization techniques to define the feature requirements.

With the knowledge available through the decision model, researchers can more rapidly evaluate MDD platforms in the market, add more platforms or features to the decision model systematically according to the presented guideline, employ the reusable knowledge (presented in Tables 6.1, 6.2, 6.3, and 6.4) to develop new concepts and solutions for future challenges.

6.6.4 Limitations and Threats to Validity

The validity assessment is an essential part of any empirical study. Validity discussions typically involve Construct Validity, Internal Validity, External Validity, and Conclusion Validity.

Construct validity refers to whether an accurate operational measure or test has been used for the concepts being studied. In literature, decision-making is typically defined as a process or a set of ordered activities concerning stages of problem identifying, data collection, defining alternatives, selecting a shortlist of alternatives as feasible solutions with the ranked preferences (Fitzgerald et al., 2017; Kaufmann et al., 2012). To mitigate the threats to the construct validity, we followed the MCDM theory and the six-step of a decision-making process (Majumder, 2015) to build the decision model for the MDD platform selection problem. Moreover, we employed document analysis and expert interviews as two different knowledge acquisition techniques to capture knowledge regarding MDD platforms. Additionally, the DSS and the decision model have been evaluated through four real-world case studies at four different real-world enterprises in the Netherlands.

Internal validity attempts to verify claims about the cause-effect relationships within the context of a study. In other words, it determines whether the study is sound or not. To mitigate the threats to the internal validity of the decision model, we define DSS success when it, in part, aligns with the case-study participants shortlist and when it provides new suggestions that are identified as being of interest to the case study participants. Emphasis on the case study participants' opinion as a measurement instrument is risky, as the case study participants may not have sufficient knowledge to make a valid judgment. We counter this risk by conducting more than one case study, assuming that the case study participants are handling their interest and applying the DSS to other problem domains, where we find similar results (Farshidi et al., 2018a; Farshidi et al., 2020c; Farshidi et al., 2018c; Farshidi et al., 2020e).

External validity concerns the domain to which the research findings can be generalized. External validity is sometimes used interchangeably with generalizability (feasibility of applying the results to other research settings). We evaluated the decision model in the context of Dutch enterprises. To mitigate threats to the research's external validity, we captured knowledge from different sources of knowledge without any regional limitations to define the constructs and build the decision model. Accordingly, we hypothesize that the decision model can be generalized to all enterprises worldwide who face uncertainty in the MDD platform selection problem. Another question is whether the framework and the DSS can be applied to other problem domains as well. The problem domains (Farshidi et al., 2018a; Farshidi et al., 2020c; Farshidi et al., 2018c; Farshidi et al., 2020e) were selected opportunistically and prag-

matically, but we are convinced that there are still many decision problems to which the framework and the DSS can be applied. The categories of problems to which the framework and the DSS can be applied successfully can be summed up as follows: (1) the problem regards a technology decision in system design with long-lasting consequences, (2) there is copious scientific, industry, and informal knowledge publicly available to software engineers, and (3) the (team of) software engineer(s) is not knowledgeable in the field but very knowledgeable about the system requirements.
Conclusion validity verifies whether the methods of a study such as the data collection method can be reproduced, with similar results. We captured knowledge systematically from the sources of knowledge following the MCDM framework (Farshidi et al., 2018a). The accuracy of the extracted knowledge was guaranteed through the protocols that were developed to define the knowledge extraction strategy and format. A review protocol was proposed and applied by multiple research assistants, including bachelor and master students, to mitigate the threats to the research's conclusion validity. By following the framework and the protocols, we keep consistency in the knowledge extraction process and check whether the acquired knowledge addresses the research questions. Moreover, we crosschecked the captured knowledge to assess the quality of the results, and we had at least two assistants extracting data independently.

6.7 Related Work

In this study, Snowballing was applied as the primary method to investigate the existing literature regarding techniques that address the MDD platform selection problem. Table 6.6 summarizes a subset of selected studies that discuss the problem. As aforementioned, the last column (*Cov.*) of Table 6.6 indicates the percentage of the coverage of the considered criteria within the selected studies. On average, 75% of those criteria are already considered in this study. In other words, the decision model contains a significant number of criteria, including features and quality attributes, that have been mentioned in literature.

6.7.1 Intelligent Business Process Management Suite Selection

In literature, a wide range of publications has assessed different iBPMS platforms and compared them against a set of criteria. Dunie et al. (2019) reported considered 26 criteria, such as sales execution/pricing and marketing strategy, to evaluate nineteen iBPMS platforms as leaders, challengers, niche players, and visionaries.

Sanchis et al. (2020) introduced a framework to manage the overall network of a collaborative manufacturing and logistics environment that enables humans, applications, and Internet of Things devices to seamlessly communicate and interoperate in the interconnected environment, promoting resilient digital transformation. Then, the authors conducted a literature study regarding MDD platforms to identify sixteen features that they support. Finally, the authors compared their framework with six Low-Code platforms against the features.

Rymer et al. (2019) researched a list of low-code platforms, including 13 vendors, to consider for the evaluation. From that initial pool of vendors, they narrowed the

Table 6.6: compares a subset of selected studies from the literature that addresses the MDD platform selection problem. The first six columns indicate the selected study (Study), the publication type (Type) (including Research Paper (RP), Master Thesis (MT), and Report (R)), the research methods (R. Method) (including Expert Interview (Experts), Document Analysis (Doc Analysis), Design Science (Design Sc.), Systematic Literature Review (SLR), Survey, and Case Study), the data collection type (Data Col.), and MDD platforms (Platforms), and the publication year (Year) of the corresponding selected studies, respectively. The seventh column (Approach) indicates the decision-making approach that the studies have employed to address the MDD platform selection problem. The eighth column (MCDM) denotes whether the corresponding decision-making technique is an MCDM approach. The ninth column indicates whether the MCDM approach applied pairwise comparison (PC) as a weight calculation method or not. The tenth column (QA) determines the type of quality attributes. The eleventh and twelfth columns (#C and #A) signify the number of criteria and alternatives considered in the selected studies. The next three columns indicate the numbers of common quality attributes (#CQ), features (#CF), and alternatives (#CA) of this study (the first row) with the selected studies. The last column (Cov.) shows the percentage of the coverage of the considered criteria (quality attributes and features).

Study	Type	R. Method	Data Col.	Platforms	Year	Approach	MCDM	PC	QA	#C	#A	#CQ	#CF	#CA	Cov.
This study	RP	Design Sc. Experts Doc Analysis Case Study	Mixed	iBPMS BPMS	2020	DSS	Yes	No	ISO/IEC 25010 EX. ISO/IEC 9216	151	30	57	94	30	100%
Sanchis et al. (2020)	RP	SLR	Quantitative	iBPMS	2020	Benchmarking	No	N/A	Domain Specific	16	7	0	11	2	68%
Vincent et al. (2019)	Report	Survey Doc Analysis	Quantitative	iBPMS	2019	SA	No	N/A	Domain Specific	15	18	7	5	12	80%
Rymer et al. (2019)	Report	Survey Doc Analysis	Quantitative	iBPMS	2019	SA	No	N/A	Domain Specific	20	13	2	14	9	80%
Dunie et al. (2019)	Report	Survey Doc Analysis	Quantitative	iBPMS	2019	SA	No	N/A	Domain Specific	26	19	9	4	6	50%
Vugec et al. (2019)	RP	Doc Analysis	Quantitative	iBPMS	2019	Benchmarking	No	N/A	Domain Specific	4	9	1	3	5	100%
Sattar (2018)	MT	Case Study Experts Doc Analysis	Mixed	iBPMS	2018	Benchmarking	No	N/A	Domain Specific	10	13	2	8	3	100%
Zolotas et al. (2018)	RP	Doc Analysis	Quantitative	iBPMS	2018	Benchmarking	No	N/A	N/A	4	11	0	2	5	50%
Şen et al. (2018)	RP	Experts	Qualitative	BPMS	2018	AHP FTOPSIS	Yes	Yes	Domain Specific	4	5	2	2	0	100%
Hendriks et al. (2017)	MT	Experts	Qualitative	iBPMS	2017	WSM	Yes	No	Domain Specific	16	5	5	7	3	75%
Melo et al. (2017)	RP	Doc Analysis	Quantitative	iBPMS	2017	Benchmarking	No	N/A	ISO/IEC 25010 Domain Specific	38	2	20	12	2	84%
Meidan et al. (2017)	RP	SLR	Quantitative	BPMS	2017	SA	No	N/A	Domain Specific	41	7	10	21	0	76%
Richardson & Rymer (2016)	Report	Survey Doc Analysis	Quantitative	iBPMS	2016	SA	No	N/A	Domain Specific	5	42	0	4	12	80%
Wasilewski (2016)	RP	Doc Analysis	Quantitative	BPMS	2016	SA	No	N/A	N/A	35	27	0	23	6	66%
Vukšić et al. (2016)	RP	SLR	Quantitative	BPMS	2016	Benchmarking	No	N/A	Domain Specific	11	3	1	8	2	82%
Delgado et al. (2015)	RP	Case Study Doc Analysis	Qualitative	BPMS	2015	WSM	Yes	No	Domain Specific	11	13	3	8	1	100%
Mejri et al. (2015)	RP	Survey	Quantitative	BPMS	2015	WSM	Yes	No	Domain Specific	4	8	0	3	0	75%
Marín et al. (2014)	RP	Survey	Qualitative	MDA	2014	Benchmarking	No	N/A	Domain Specific	9	8	4	3	0	78%
Ravasan et al. (2014)	RP	Experts Doc Analysis	Qualitative	BPMS	2014	FTOPSIS	Yes	Yes	Domain Specific	48	5	19	16	0	73%
Davies & Reeves (2010)	RP	Experts Doc Analysis	Qualitative	BPMS	2010	WSM	Yes	No	Domain Specific	53	10	10	40	0	94%
Kapteijns et al. (2009)	RP	Case Study Doc Analysis	Qualitative	BPMS	2009	Benchmarking	No	N/A	Domain Specific	12	4	1	6	1	58%
Štemberger et al. (2009)	RP	Case Study Doc Analysis	Qualitative	BPMS	2009	AHP	Yes	Yes	Domain Specific	10	5	5	3	0	80%

final list based on several inclusion criteria, such as low-cost-of-entry commercial models, building many business use cases, and primarily targeting large enterprises. Then, they collected data from products and strategies through a detailed questionnaire, demos and briefings, and a reference-customer survey. They used those inputs, along with the analyst's experience and expertise in the marketplace, to score the platforms, applying a relative rating system that compares each platform against the others in the evaluation.

Vugec et al. (2019) identified the following four social business process management dimensions based on literature study: Egalitarianism, Collective intelligence, Self-organization, and Social production. Next, the authors selected nine iBPMSs reported by Dunie et al. (2019) to compare their functionality against the social dimensions.

Sattar (2018) assessed 13 low-code platforms based on supporting ten criteria, such as cloud platform attributes, and then introduced a decision tree for the low-code platform selection problem.

Zolotas et al. (2018) presented a low-code platform based on the REST architecture that enables developers to model attribute-based access control policies without requiring any code writing. Then, the authors compared their approach with eleven low-code platforms against four security features. Melo et al. (2017) considered the ISO/IEC 25010 standard besides a set of quality aspects, such as vendor and cost, to evaluate Oracle Apex and OutSystems.

Hendriks et al. (2017) researched for essential characteristics of low-code platforms and how they should be matched to each other. They performed interviews with industry experts to determine how the industry looks upon matching situations and platforms. Finally, they introduced a framework based on the characteristics to support organizations with the iBPMS platform selection.

Richardson & Rymer (2016) carried out an online vendor survey to assess 42 low-code platforms. The authors of the report then divided low-code platforms into the following five categories using their background and functionality: General purpose platforms, Process application platforms, Database application platforms, Request handling platforms, and Mobile application platforms.

Wasilewski (2016) compared the Gartner's reports about BPMS and iBMPS markets from 2009 to 2015 to analyze the behavior of the leaders in the Magic Quadrant.

6.7.2 Business Process Management Suite Selection

A vast range of BPMS platforms is currently available on the market to cater to a wide variety of modeling objectives. A subset of publications that reported on the evaluation of BPMS platforms is presented as follows.

Şen et al. (2018) applied the Analytic Hierarchy Process (AHP)[7] and the Fuzzy Technique for Order Preference by Similarity to Ideal Solution (FTOPSIS)[8] to address

[7]AHP is an MCDM technique for making decisions between alternatives. AHP allows decision-makers to capture their strategic goals as a set of weighted criteria that they then use to rank alternatives.

[8]The *TOPSIS* is an MCDM approach that employs information entropy to assess alternatives. Fuzzy logic is an approach to computing based on "degrees of truth" rather than the usual Boolean logic. Sometimes combinations of fuzzy logic with other MCDM approaches, such as FTOPSIS, are employed to solve MCDM problems.

the problem of choosing BPMS for a retailer operating in the textile sector. In the first step, the AHP implementation and the pairwise comparisons were taken from the seven decision-makers determined the decision criteria' weights. In the second step, the FTOPSIS method was performed to select the best fitting BPMS with the decision-makers' quantitative and qualitative evaluations.

Meidan et al. (2017) performed a formal survey based on a systematic literature review method and quality models to classify and compare BPMSs according to a set of characteristics of open source BPMS. Additionally, they observed that every BPMS provider used its terminology to explain business process concepts (e.g., join and fork elements, exception handling, events). Unifying the terminology could help to improve interoperability and portability among BPMSs. Furthermore, the evaluation showed that BPMSs could be classified into two families: the first one is oriented to normal users (e.g., Bonita and ProcessMaker), and the second one is platform-oriented to developers and expert users (e.g., Activiti, Camunda, and jBPM).

Vukšić et al. (2016) presented a guideline and a set of selection criteria such as maturity, reporting and analytics, business rules, user interface and user experience, and modeling notation to evaluate three BPMS platforms (IBM, K2, and Software AG).

Kapteijns et al. (2009) performed research regarding BMPS platforms in small-scale development projects to investigate the case study participants' level of satisfaction. Moreover, the authors collected a set of BMPS features from the literature to compare four BMPS platforms against each other.

Delgado et al. (2015) presented a systematic approach based on the Weighted Sum Model (WSM)[9] for assessing BPMS tools, both open-source and proprietary. The authors suggested a list of relevant vital characteristics for BPMS tools and a way of evaluating the provided support using test cases and a case study to provide an overall view of the tool support.

Mejri et al. (2015) used a questionnaire to capture a set of BPMSs' strengths and weaknesses in terms of flexibility from their researchers and developers. Then, they used the Weighted Sum Method to calculate the scores of the BPMSs and rank them accordingly. Ravasan et al. (2014) introduced a set of functional and non-functional criteria for selecting the right BPMS for an organization. The authors then applied the FTOPSIS approach to calculate the weight of the criteria based on decision-makers' requirements and priorities.

Davies & Reeves (2010) reported on the Australian government department's experiences in selecting a BPM tool to support its process modeling, analysis, and design activities. Candidate solutions were identified for evaluation by researching case studies and market overviews.

Štemberger et al. (2009) presented a method for BPMS selection to support decision-makers (managers and IT experts) with the BPMS selection process. Their approach was based on the AHP method and developed BPM tools features from project goals and critical success factors. Their research reported two points: (1) to choose the best fitting software tool for a particular application and business, an organization requires a method for the evaluation of a BPMS, and (2) the selection of a

[9]*Weighted Sum Model* is an aggregation function that transforms multiple criteria into a single value by multiplying each criterion by a weighting factor and summing up all weighted criteria.

BPMS is a multi-criteria decision process. Accordingly, a suitable method for making multi-objective decisions should be employed.

6.7.3 Strengths and Liabilities

Studies based on Benchmarking (Kapteijns et al., 2009; Marín et al., 2014; Melo et al., 2017; Sanchis et al., 2020; Sattar, 2018; Vugec et al., 2019; Vukšić et al., 2016; Zolotas et al., 2018) and Statistical Analysis (SA) (Dunie et al., 2019; Meidan et al., 2017; Richardson & Rymer, 2016; Rymer et al., 2019; Vincent et al., 2019; Wasilewski, 2016) are typically time-consuming approaches and mainly applicable to a limited set of alternatives and criteria, as they require in-depth knowledge of programming languages and concepts. Such analysis is subject to increased error, particularly when a relational analysis is used to attain a higher interpretation level. One of the critical issues regarding statistical analysis is the tendency to skip unjustified conclusions concerning causal relationships. Researchers usually obtain evidence that two variables are highly correlated; however, that does not prove that one variable causes another. Finding relationships among correlations and causation need in-depth expertise and experience regarding MDD platforms and their concepts, as such links are mainly qualitative. Additionally, benchmarking and statistical analysis are likely to become outdated soon and continuously kept up to date, which is a high-cost process.

As aforementioned, finding the best MDD platform for an organization is a decision-making process that deals with evaluating several alternatives and criteria. Accordingly, the selected platform should address the concerns and priorities of the decision-makers. Conversely to MCDM approaches, studies based on "Benchmarking" and "Statistical Analysis" principally offer generic results and comparisons and do not consider individual decision-maker needs and preferences. A variety of MCDM approaches have been introduced by researchers recently.

The majority of the MCDM techniques (Davies & Reeves, 2010; Delgado et al., 2015; Hendriks et al., 2017; Mejri et al., 2015; Şen et al., 2018; Štemberger et al., 2009) define domain-specific quality attributes to evaluate the alternatives. Such studies are mainly appropriate for specific case studies. Furthermore, MCDM approaches are valid for a specified period; therefore, the results of such studies will be outdated by MDD platforms' advances. Note that, in our proposal, this is also a challenge, and we propose a solution for keeping the knowledge base up to date in section 6.6. Some of the methods, such as FTOPSIS and AHP, are not scalable, so in modifying the list of alternatives or criteria, the evaluation process should be redone. Therefore, these methods are costly and applicable to only a small number of criteria and alternatives. This study has considered 151 criteria and 30 alternatives to building a decision model for the MDD platform selection problem.

In contrast to the named approaches, the cost of creating, evaluating, and applying the proposed decision model in this study is not penalized exponentially by the number of criteria and alternatives. It is an evolvable and expandable approach that splits down the decision-making process into four maintainable phases (Farshidi et al., 2018c). Moreover, we introduce several parameters to measure the values of non-Boolean criteria, such as the maturity level and market popularity of the MDD platforms. The proposed decision model addresses main knowledge management issues, including capturing, sharing, and maintaining knowledge. Furthermore, it uses the ISO/IEC 25010 (ISO, 2011) as a standard set of quality attributes. This quality standard is a domain-independent software quality model and provides reference points by defining a top-down standard quality model for software systems.

Recently, we built five decision models based on the framework to model the selection of Database Management Systems (Farshidi et al., 2018c), Cloud Service Providers (Farshidi et al., 2018a), Blockchain Platforms (Farshidi et al., 2020c), Software Architecture Patterns (Farshidi et al., 2020e), and Programming Languages. In all five studies, case studies were conducted to evaluate the DSS's effectiveness and usefulness in addressing MCDM problems. The results confirmed that the DSS performed well to solve the mentioned problems in software production. We believe that the framework can be employed as a guideline to build decision models for MCDM problems in software production.

6.8 Conclusion

In this study, the selection process of the model-driven development platforms is modeled as a multi-criteria decision-making problem that deals with evaluating a set of alternatives and taking into account a set of decision criteria (Triantaphyllou et al., 1998). Moreover, we presented a decision model for the selection problem based on the technology selection framework (Farshidi et al., 2018a). The novelty of the approach provides knowledge about model-driven development platforms to support uninformed decision-makers while contributing a sound decision model to knowledgeable decision-makers. Furthermore, it incorporates deeply embedded software engineering concepts, such as the ISO software quality standards and the MoSCoW prioritization technique, besides knowledge engineering theories, to develop the decision model. We conducted four industry case studies to evaluate the decision model's usefulness and effectiveness to address the decision problem. We find that while organizations are typically tied to particular ecosystems by extraneous factors, they can benefit significantly from our DSS by evaluating their decisions, exploring more potential alternative solutions, and analyzing an extensive list of features. The case studies show that this article's decision model also provides a foundation for future work on MCDM problems. We intend to build trustworthy decision models to address the *Programming Language* selection problem as our (near) future work.

Part III: Decision-Making in Pattern-Driven Design

CHAPTER 7

Capturing Software Architecture Knowledge

Context: Software architecture is a knowledge-intensive field. One mechanism for storing architecture knowledge is the recognition and description of architectural patterns. Selecting architectural patterns is a challenging task for software architects, as knowledge about these patterns is scattered among a wide range of literature.
Method: We report on a systematic literature review, intending to build a decision model for the architectural pattern selection problem. Moreover, twelve experienced practitioners at software-producing organizations evaluated the usability and usefulness of the extracted knowledge.
Results: An overview is provided of 29 patterns and their effects on 40 quality attributes. Furthermore, we report in which systems the 29 patterns are applied and in which combinations. The practitioners confirmed that architectural knowledge supports software architects with their decision-making process to select a set of patterns for a new problem. We investigate the potential trends among architects to select patterns.
Conclusion: With the knowledge available, architects can more rapidly select and eliminate combinations of patterns to design solutions. Having this knowledge readily available supports software architects in making more efficient and effective design decisions that meet their quality concerns.

keywords- architectural patterns; architectural styles; quality attributes; design decisions; knowledge acquisition

This chapter is partially based on the following publication:

Farshidi, S., S. Jansen & J. M. van der Werf (2020e), “Capturing software architecture knowledge for pattern-driven design”, *Journal of Systems and Software*.

7.1 Introduction

Software architecture plays an indispensable role in the success or failure of any software system, as it deals with the base structure, subsystems, and interactions among these subsystems (Clements et al., 2003). Software architecting can be viewed as a decision-making process: software architects consider a set of alternative solutions that could solve a system design problem, and select the set that is evaluated as the optimal (Lago & Avgeriou, 2006). Software architecture decisions are design decisions that address system requirements, including both functional and quality requirements. In this article, we present the results from an SLR that intends to support architects in the decision process, by linking quality attributes to software patterns[1].

Software architecture design decisions, such as the selection of architectural patterns and software design patterns, are typically made in the early phases of the software development life cycle. In the following paragraphs, we define architectural patterns, styles, and tactics (Shaw, 1995).

Architectural patterns are universal and reusable solutions to commonly occurring problems in software architecture (Buschmann et al., 2007a). Each architectural pattern describes high-level structures and behaviors of software systems and addresses a particular recurring problem within a given context in software architecture design. Architectural patterns aim to satisfy several functional and quality attribute requirements. In literature, sometimes the terms “architectural patterns” and “architectural styles” are used interchangeably, since they are, in essence, the same concepts and only differ in their description forms (Avgeriou & Zdun, 2005).

Software design patterns are experience-based standard solutions applied by developers to solve common problems when implementing a software system (Hussain et al., 2017). Note, a *software design pattern* is not a finished design that can be transformed directly into source or machine code. Architectural patterns are similar to software design patterns but have a broader scope. In this study, we focus on *architectural patterns*, and for the sake of brevity, we use *patterns* to refer to them.

software architecture tactics are design decisions that improve individual quality attribute concerns (Harrison & Avgeriou, 2010). Tactics that are implemented in existing architectures can have significant impacts on the patterns in the system. In other words, tactics are reusable architectural building blocks that provide generic solutions to address issues about quality attributes that patterns have impacts on.

Pattern descriptions contain knowledge about quality attributes, and software architects rely on that knowledge to make effective design decisions, so increasing such knowledge means increasing the role of patterns in satisfying quality attributes (Me et al., 2016). Patterns and quality attributes are not independent and have significant interaction with each other. Such interactions can be observed as trade-offs between quality attributes. Software architects need to select and employ an optimal set of patterns to satisfy quality concerns. For instance, some studies assert that *Reusability* is a strength (Qin et al., 2008; Sabagh & Al-Yasiri, 2011) and *Scalability* is a liability (Galster et al., 2010; Majidi et al., 2010) of the *Layers* pattern. If an architect is

[1]The knowledge base of this study, including the primary studies and extracted knowledge, is available as a technical report on the following web page: http://swapslr.com

looking for both qualities, she has two options: choose another (set of) pattern(s) or use *software architecture tactics* to improve *Scalability*.

Software architects are making the design decisions that have long-lasting impacts on quality attributes of a software-intensive system (Kruchten, 2008). Software architects define the architecture of the system, maintain the architectural integrity of the system, assess technical risks, perform risk mitigation strategies, participate in project planning, consult with design and implementation teams, and assist product marketing (Kruchten, 1999). Therefore, software architects make high-level design decisions every day (Tyree & Akerman, 2005). Software architects engage in processes of creation, perfection, and destruction on a daily basis. Their work consists of setting standards for developers, designing and implementing new parts of a system's architecture, developing shells around and interfaces to legacy systems, monitoring quality attributes, and occasionally creative destruction to make way for significant renovations. Pattern selection is a process that happens organically during the process of architecting a system.

Generally speaking, functional requirements define what a system does, whereas quality requirements explain how well those functions are performed (Blaine & Cleland-Huang, 2008). Quality requirements tend to present trade-offs that must be thoroughly negotiated and resolved (Chung et al., 2000). For instance, a software architect might want to design a system to be both highly secure and available, or she might want a system to respond quickly and support thousands of users simultaneously. Therefore, she has to design an architectural solution that supports these conflicting quality requirements to optimize the delivered system's value. Quality requirements are often more challenging to measure and track than their functional counterparts. Whereas functional requirements are either present or not present in a system, quality requirements tend to be achieved at various levels along a continuum (Blaine & Cleland-Huang, 2008).

System quality is best exposed in production, independent of whether system quality has been made explicit. Note, it is essential to recall those well-known authors, such as Wiegers & Beatty (2013), classify quality attributes as external (exposed at the run time/in production, e.g., performance) and internal (exposed at design time, e.g., modifiability). If architects do not think about performance, the system will still expose its performance in the field. The knowledge around the quality of a system under design is hard to gather without *in the field* experiences; however, experience with similar patterns in other systems provides invaluable insight into the inherent qualities of a new system. The rationale behind this article is that patterns exhibit similar quality behaviors when purely implemented (without tactics) in different systems and that this knowledge can be used by architects to make informed design decisions.

In this study, we followed a mixed research method, a combination of qualitative and quantitative research, to systematically capture architectural knowledge and make it available in a reusable and extendable format. First, we conducted a Systematic Literature Review (SLR). The SLR has been carried out following the steps and guidelines of Kitchenham (2004) to identify common lists of patterns and quality attributes, besides strengths and liabilities, application domains, combinations, and trends of the patterns. Next, a serious of expert interviews, based on Bogner et al.

(2009), has been conducted to evaluate the usefulness and reusability of the extracted knowledge. Note, the knowledge is summarized in this article, and we propose three ways of disseminating the knowledge to the architect: education, tool support, and pattern quality impact reporting. The practitioners who participated in this research confirmed that the extracted knowledge supports software architects with their daily decision-making process.

7.2 Background

7.2.1 Patterns in Software Architecture

Several definitions exist that explain Software Architecture. It is both seen as the set of structures of software elements, and their relations and properties to reason over a software system (Bass et al., 2013), and as the set of principal design decisions (Jansen et al., 2008).

In this paper, we consider the former definition, the set of structures, as the outcome of the latter, i.e., software architecture is the outcome of a set of principle design decisions. This is reflected in the meta-model, depicted in Figure 7.1, which is based on the ISO/IEC/IEEE standard 42010 (ISO, 2011). Architectural decisions may depend on other decisions, pertains to one or more concerns of stakeholders, and should contain some rationale to justify it. The outcome of the decision affects the architecture description. Besides, the decision may raise new concerns. Concerns include both functional requirements as well as quality attributes (Bass et al., 2013).

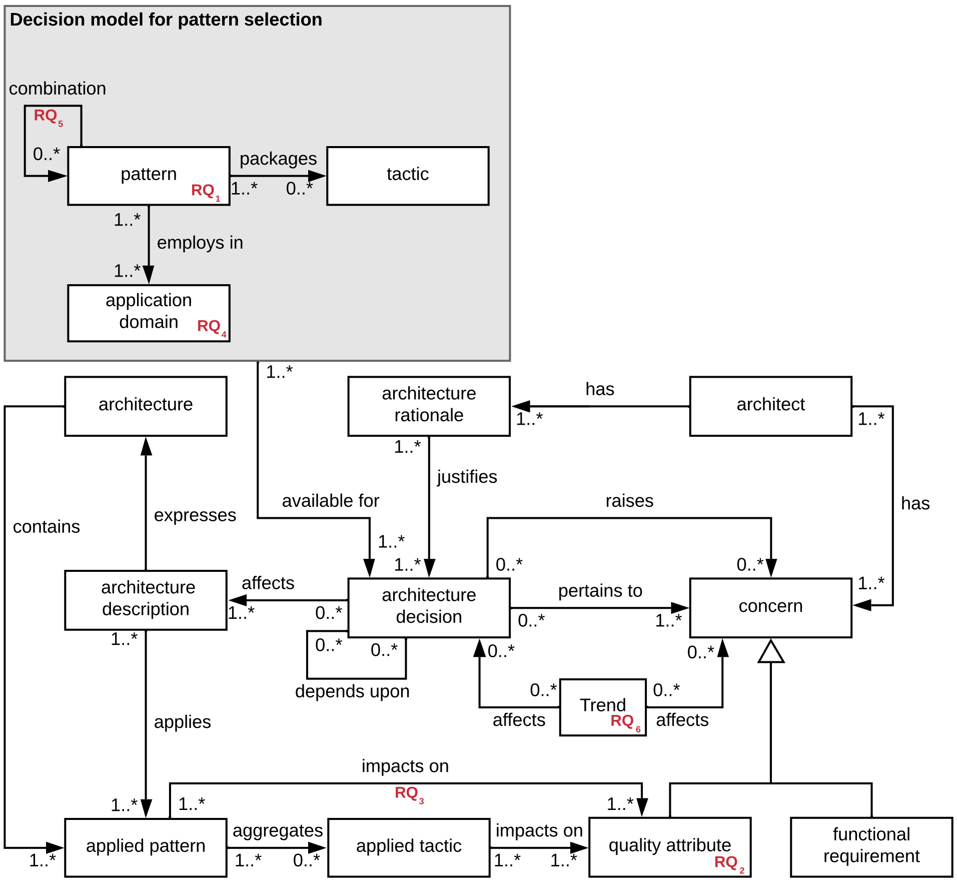
An architectural pattern expresses a fundamental structural organization schema for software systems (Rozanski & Woods, 2012). A closely related term in literature is “architectural style”. As there is no widely accepted definition for both terms in literature, we refer to both as “architectural pattern”. An architectural pattern differs from software patterns, also referred to as design patterns, in that a software pattern provides a solution for a general design problem (Hussain et al., 2017), whereas an architectural pattern describes the organizational schema of a software system.

Table 7.1 outlines the definitions of the foundational concepts for the SLR. Please note that many of the definitions were handpicked from the plethora of definitions available because we needed to make sure that the definitions fit the meta-model in Figure 7.1.

7.2.2 Decision Process

Building a software architecture can be regarded as a decision-making process (Lago & Avgeriou, 2006): a software architect considers several alternative solutions (design decisions) that could solve the design problem statement, and subsequently chooses one of the solutions that optimally addresses the problem. The software architecture design decision, such as the selection of architectural patterns, is formulated as follows: (1) a software architect runs into a design problem, (2) she looks for actual features she thinks can solve this problem, such as “distribute data over multiple servers”, (3) she goes through the description of several patterns and identifies several candidates, (4) she identifies an optimum pattern for her problem

Figure 7.1: This figure shows a meta-model, based on the ISO/IEC/IEEE standard 42010 (ISO, 2011), for decision-making in software architecture. The essential included elements are the architect, the architecture, the knowledge base, and the quality attributes.



and goes through tactics to make sure it works in the context. The decision model for the pattern selection problem can be used in steps 2 and 3 to facilitate the decision-making process for software architects.

Figure 7.1 represents a meta-model for decision-making in architecture. It shows in general terms how patterns, quality attributes, and tactics are related to each other, and how they are linked to the architecture. It provides a structure for discussion of the specific ways that applied tactics affect the patterns used. It also provides a foundation for the description of the impact of applied patterns and tactics on the software architect's quality concerns. Note that we distinguished the applied patterns and tactics in the architecture from the potential set of design decisions (patterns and tactics that are available in the knowledge base of software architects).

Table 7.1: List of terms and their definitions used in this article. Please note that all terms except for Functional Requirement can be preceded by the words “Software Architecture”.

Term	Definition	Refs
Software Architecture	Software architecture is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.	Clements et al. (2003)
Pattern	universal and reusable solutions to commonly occurring problems in software architecture.	Buschmann et al. (2007a)
Tactic	design decisions that improve individual quality attribute concerns	Harrison & Avgeriou (2010).
Quality	The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value.	ISO (2011)
Architect	person, team, or organization responsible for systems architecture	ISO (2017)
Rationale	captures the knowledge and reasoning that justify the resulting design, and its primary goal is to support designers by providing means to record and communicate the argumentation and reasoning behind the design process.	Horner & Atwood (2006) and Tang et al. (2006)
Decision	A decision is consisting of a restructuring effect on the components and connectors that make up the software architecture, design rules imposed on the architecture and resulting system as a consequence, design constraints imposed on the architecture, and a rationale explaining the reasoning behind the decision.	Bosch (2004)
Functional Requirement	condition or capability that must be met or possessed by a system, system component, product, or service to satisfy an agreement, standard, specification, or other formally imposed documents	ISO (2017)
Concern	is any interest in the system. The term is derived from the phrase "separation of concerns" as in Software Engineering. One or more stakeholders may hold a concern. Concerns involve system considerations such as performance, reliability, security, availability, and scalability.	ISO (2011)

The pattern selection process is challenging for software architects, as knowledge about patterns is scattered among a wide range of literature. Knowledge regarding patterns has to be collected, organized, stored, and quickly retrieved when it needs to be employed. There exists a need for a decision support system that intelligently supports software architects in selecting suitable patterns according to their requirements.

7.2.3 Related Studies

It is becoming increasingly common in software engineering to synthesize results through SLRs, even though that is a relatively recent phenomenon (Brereton et al., 2007). In software, architecture research SLRs are also increasingly common (Uzun & Tekinerdogan, 2018; Weinreich & Groher, 2016) and generally serve the purpose of mapping out particular research challenges in the domain. Our SLR was conducted because we lacked a near-to-complete source of evidence to create a reliable decision model for architects. Our study distinguishes itself from such studies as it synthesizes literature intending to collect data for practitioners and evaluates the collected data with practitioners themselves. The study also contributes overviews of commonly discussed patterns and quality attributes, providing a basis for new research. It is notable, for instance, that many of the quality attributes found in our study are not present in the well known ISO standards.

The software architecture field has evolved over the last four decades (Clements & Shaw, 2009; Shaw & Clements, 2006) from the early fundamental concepts from the mid-80s to the ubiquitous proliferation of roles of software architects in contemporary industrial practice (Capilla et al., 2016). Architectural knowledge, such as the impacts of patterns on quality attributes, has been widely addressed in the literature. However, the knowledge is fragmented over a wide range of heterogeneous studies (Buchgeher et al., 2016; Me et al., 2016; Tang et al., 2011b), so a sound methodology is required to capture and aggregate this knowledge systematically. The data collection is an empirical study that can be quantitative or qualitative (Runeson & Höst, 2009). Quantitative data comprises numbers and classes, while qualitative data involves descriptions and explanations of phenomena. Quantitative data is analyzed using statistics, while qualitative data is analyzed using expert interviews or/and case studies to provide a more detailed and more in-depth explanation. However, a combination of qualitative and quantitative data often provides a better understanding of the studied phenomenon (Seaman, 1999) (Mixed research).

Research methods are classified based on their data collection techniques (interview, observation, literature, etc.), inference techniques (taxonomy, protocol analysis, statistics, etc.), research purpose (evaluation, exploration, description, etc.), units of analysis (individuals, groups, process, etc.), and so forth. Multiple research methods are combined to achieve a fuller picture and a more in-depth understanding of the studied phenomenon by connecting complementary findings that conclude from the use of methods from the different methodological traditions of qualitative and quantitative investigation (Johnson & Onwuegbuzie, 2004).

In this study, we considered a systematic literature review and expert interviews as a mixed data collection method to identify frequent mentioning sets of patterns and quality attributes that were discussed widely in academic publications. Then, we highlighted 29 patterns and 40 quality attributes than were mentioned in more than three selected primary studies. Moreover, we extracted potential strengths and liabilities of the patterns to map the patterns to the quality attributes and calculate the impacts of the patterns on the quality attributes based on fuzzy logic. Additionally, we realized that the authors of the selected primary studies employed the patterns in particular types of systems and applications so that we considered them as the potential application domains of the patterns. Furthermore, we tracked the publications' years of the studies and their mentioned patters to imply a trendy manner among academics to employ patterns and research them.

Table 7.2 positions this study among a subset of selected primary studies. This table shows that none of the selected primary studies employed qualitative and quantitative data collection methods to evaluate a significant number of patterns. Note, the research results of all of the selected primary studies have been included in the knowledge base of the SLR (See Section 7.3.7).

Table 7.2: This table shows a subset of studies in literature. The first six columns indicate the selected study (Study), the publication type (PT) (including Research Paper (RP), Book, and Chapter (Chp)), the publication year (Year), and the data collection method (DCM), the research purpose (Purpose), and data collection type (Type) of the corresponding selected primary studies, respectively. The seventh and eighth columns (#P and #QA) denote the number of considered patterns and quality attributes in the selected primary studies. The last three columns identify whether the selected primary studies investigated on the potential domains of patterns, possible trends of utilizing patterns, impacts of patterns on quality attributes, or not.

Study	**PT**	**Year**	**DCM**	**Purpose**	**Type**	#P	#QA	**Domain**	**Trend**	**Impact**
This Study	RP	2020	SLR Interview	Evaluation	Mixed	29	40	Yes	Yes	Yes
Pramod Mathew Jacob (2018)	RP	2018	Experiment	Evaluation	Quantitative	4	8	Yes	No	Yes
Haoues et al. (2017)	RP	2017	Survey	Evaluation	Quantitative	3	27	No	No	Yes
Me et al. (2016)	RP	2016	SLR	Evaluation	Quantitative	8	15	No	No	Yes
Richards (2015)	Book	2015	Case Study	Evaluation	Qualitative	5	6	Yes	No	Yes
Buyya et al. (2013)	Chp	2013	Case Study	Description	Qualitative	15	15	Yes	No	Yes
Yang et al. (2012)	RP	2012	Case Study	Evaluation	Qualitative	7	11	Yes	No	Yes
Bode & Riebisch (2010)	RP	2010	Case Study	Evaluation	Mixed	9	15	No	No	Yes
Harrison & Avgeriou (2008a)	RP	2010	Statistics	Description	Quantitative	20	4	Yes	No	No
Ahmad et al. (2010)	RP	2010	Case Study	Description	Qualitative	5	9	No	No	Yes
Qin et al. (2008)	Chp	2008	Case Study	Description	Qualitative	7	15	Yes	No	Yes
Harrison & Avgeriou (2007)	RP	2007	Statistics	Evaluation	Quantitative	7	8	No	No	Yes
Avgeriou & Zdun (2005)	RP	2005	Literature	Description	Qualitative	24	10	No	No	Yes
Buschmann et al. (1996)	Book	1996	Case Study	Description	Qualitative	8	20	Yes	No	Yes
Garlan & Shaw (1993)	Chp	1994	Case Study	Description	Qualitative	6	5	Yes	No	Yes

Note, an extensive list of studies addresses the impacts of patterns on quality attributes. Each study considered different sets of patterns and quality attributes (Columns $\#P$ and $\#QA$). Moreover, we perceived that some patterns have conflicting impacts on a particular quality attribute. For instance, some studies (Ahmad et al., 2010; Harrison & Avgeriou, 2008b) expressed that *Performance efficiency* is a key strength of *Client-Server*, however, some other studies (Elahi & Babamir, 2015; Jacob & Mani, 2018) stated that *Performance efficiency* is a key liability of *Client-Server*. The majority of studies in the literature reported some potential domains of patterns. However, we realized that different studies suggested different domains. For example, Yang et al. (2012) stated that *Pipe and Filters* can be used in *Operating Systems*, and Buyya et al. (2013) asserted this pattern can be employed in *Compiler design* as well.

7.3 Systematic Literature Review

Recently, we designed a framework (Farshidi et al., 2018c) and implemented a Decision Support System (DSS) (Farshidi et al., 2018b) for supporting software developers and architects (decision-makers) with their multi-criteria decision-making (MCDM) problems in software production. An MCDM problem deals with evaluating a set of alternatives and considers a set of decision criteria (Triantaphyllou et al., 1998). The framework applies the six-step decision-making process (Majumder, 2015) to build maintainable and evolvable decision models for MCDM problems in software production. Moreover, the framework provides a guideline for decision-makers to build decision models for MCDM problems in software production. Based on the framework, we built decision models for the selection of Database Management Systems (Farshidi et al., 2018c), Cloud Service Providers (Farshidi et al., 2018a), and Blockchain Platforms (Farshidi et al., 2020c)[2].

In order to capture knowledge systematically regarding patterns and build a decision model, based on the framework, for the pattern selection problem (as future work), the following research questions have been formulated to guide our study:

RQ_1**:** Which patterns are frequently employed by architects since the emergence of the field?

RQ_2**:** Which quality attributes are commonly utilized by architects to evaluate patterns?

RQ_3**:** What are strengths and liabilities of patterns reported in literature?

RQ_4**:** What are the possible application domains of patterns mentioned in literature?

RQ_5**:** Which combinations of patterns are available in literature?

RQ_6**:** Do architects select patterns based on trends?

RQ_1: A set of patterns among an extensive list of patterns should be considered. Note, patterns can be alternatives to each other, for example, *Interpreter*, *Rule-Based System*, and *Virtual Machine* (Avgeriou & Zdun, 2005).

RQ_2: By increasing knowledge about patterns, it is possible to make better-

[2]The decision models and modeling studio are available on the DSS website: www.dss.amuse-project.org.

informed decisions, avoid failures, and better satisfy quality attributes and achieve system-wide quality targets (Me et al., 2016). A set of quality attributes should be defined in the decision model. Quality attributes are characteristics of the system that are intrinsically non-functional. One of the primary purposes of the architecture of a system is to create a system design to satisfy the quality attributes (Harrison & Avgeriou, 2007). It is essential to find quality attributes that are widely mentioned by other researchers to identify the characteristics of patterns.

RQ_3: Part of the software architects' concerns are those requirements that have impacts on quality attributes of software-intensive systems (Kazman et al., 1994). Quality requirements are the horizontal cross-cutting concerns that impact a system, such as performance, security, and usability. Software architects should be aware of any requirement or design decision that impacts one of these concerns and should elicit requirements that allow for the measurement of quality attributes. Therefore, to build a beneficial and powerful decision model for the pattern selection problem, it must be achievable to find which patterns impact specific quality attributes, compare and contrast impacts, and highlight their interactions.

RQ_4: Application-generic and application-specific knowledge are two types of architectural knowledge (Lago & Avgeriou, 2006). Application-generic knowledge refers to knowledge that software architects have implicitly in their heads, from their former experience. Moreover, application-specific knowledge involves all the decisions taken during the architecting process of a particular system and the architectural solutions that implemented the decisions. In other words, application-generic knowledge is used to make decisions for a single application and thus construct application-specific knowledge. Therefore, knowledge regarding application domains, in which candidate patterns are already employed, can help software architects make informed decisions.

RQ_5: Patterns tend to be combined to provide greater support for the reusability during the software design process (That et al., 2013). A pattern can be blended with, connected to, or included in another pattern. For instance, the *Broker* pattern can be connected to the *Client-Server* pattern to form the combined *Client-Server-Broker* pattern (Harrison & Avgeriou, 2010).

RQ_6: Software architecture has experienced considerable growth over the past decades, and it promises to continue that growth for the foreseeable future. Although the architectural design has matured into an engineering discipline that is broadly recognized and practiced, some significant challenges will need to be addressed. Such challenges are expected to arise as a natural outcome of dissemination and maturation of the well-known architectural practices and technologies (Garlan, 2014). Software developers and architects should be aware of technology advancements, standards, and trends that affect potential architecture decisions and concerns. The last research question investigates any potential trends among architects that attract them to use a particular pattern.

Systematic Literature Review is one of the most broadly accepted research methods of evidence-based software engineering (Kitchenham et al., 2004). An SLR provides a prescribed process for identifying, evaluating, and interpreting all available evidence relevant to a particular research question or topic (Petersen et al., 2008). In this study, the SLR functioned as a knowledge acquisition process to capture knowledge about patterns and ultimately making it available in forms of reusable knowledge. The SLR has been carried out following the steps and guidelines of Kitchenham (2004): reasoning the necessity of the SLR, defining research questions, searching relevant studies, applying inclusion/exclusion criteria, assessing the quality of studies, extracting knowledge, analyzing the results.

7.3.1 Data sources and search strategy

In this study, the search strategy has two search methods: *manual search* and *automatic search*. These search methods are complementary to each other. In the *manual search*, we investigated published studies in reputable journals and conferences in the software architecture domain. This search method guarantees that we explore relevant studies, but it consumes a significant amount of time and effort in judging many irrelevant studies.

In the *automatic search*, we defined a search query to retrieve results from scientific search engines. Firstly, the search query was built based on the generic keywords extracted during the *manual search* process. In other words, the search query only contained generic keywords to avoid possible biased search results; for instance, we did not consider any standard titles of patterns (such as Layers and Client-Server) and quality attributes (such as performance and availability) explicitly. Secondly, we tested the query on the selected scientific search engines to find out whether the outcomes are compatible with the results of the *manual search*. Note, the query contains the concepts of the meta-model (see Figure 7.1), as it gives an overview of the decision-making process in designing architecture. In the automatic search (Zhang et al., 2011), we used the following query:

((“software architecture” OR “software architectural”) AND (“pattern” OR “style”)) AND (“selection” OR “evaluation” OR “quality attribute” OR “design decision” OR “decision-making”)

Figure 7.2 demonstrates the stages of the search process and the numbers of primary studies in each stage. Moreover, Table 7.3 shows the journals and conference proceedings considered in the manual search besides the scientific search engines in the automatic search. Note, Google Scholar was not involved in the automatic search since it offers many irrelevant studies. Moreover, it has substantial overlap with the other digital libraries considered in this SLR.

7.3.2 Inclusion and exclusion criteria

The inclusion and exclusion criteria were applied to the selected publications at different rounds of the search process, as illustrated in Figure 7.2. The studies were included in the SLR if they were peer-reviewed, written in English, available, and discussed patterns. Furthermore, the abstracts or titles of the primary studies had to explicitly state that the articles were on the topic of architectural patterns. The articles were published mainly as journal papers, conference papers, theses, technical reports, or books.

The peer-reviewed articles relevant to the topic of interest were published from 1990 to the first half of 2019. Note, we did not limit the SLR to this period. However, we did not find any qualified primary studies before 1990 to add to the SLR’s knowl-

Figure 7.2: This figure illustrates the phases of the search process and the number of primary studies in each phase of the SLR. The corresponding number of primary studies in each step of the search process for manual search and automatic search is signified in red and blue, respectively.

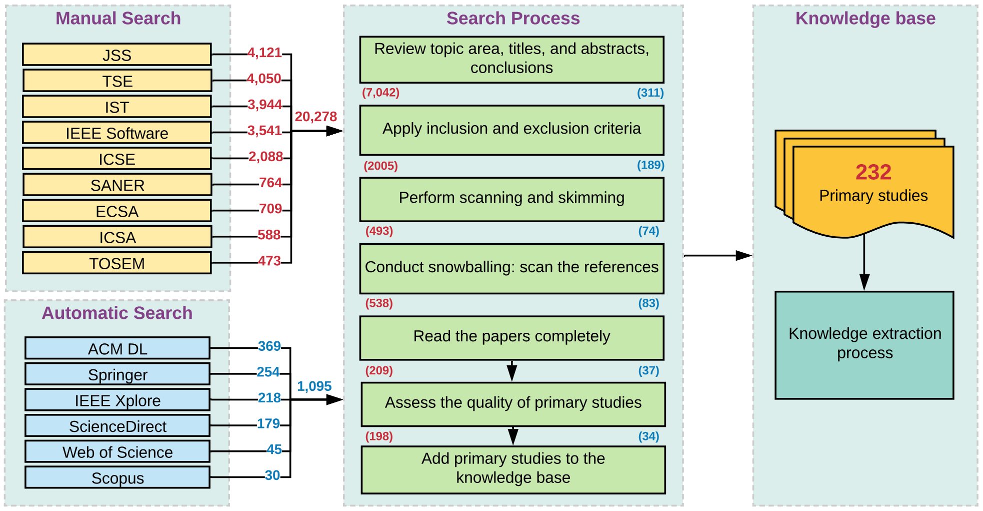


Table 7.3: Selected journals and conference proceedings in the manual and automatic searches.

Source	Acronym
Journal of Systems and Software	JSS
IEEE Transactions on Software Engineering	TSE
Information and Software Technology	IST
IEEE Software, International Conference on Software Engineering	ICSE
IEEE International Conference on Software Analysis, Evolution and Reengineering.	SANER
European Conference on Software Architecture	ECSA
International Conference on Software Architecture	ICSA
ACM Transactions on Software Engineering and Methodology	TOSEM
ACM Digital Library	ACM DL
Springer Publishing	Springer
IEEE Xplore Digital Library	IEEE Xplore
ScienceDirect	-
Web of Science	-
Elsevier's Scopus	Scopus

edge base. Editorials, position papers, keynotes, reviews, tutorial summaries, and panel discussions were excluded from the SLR. Moreover, all duplicated publications, studies with inadequate validation (i.e., no evidence), and on other platforms instead of computer-based patterns (e.g., Computer Networks, Electronics) were not considered in the SLR. A publication was only selected for knowledge extraction when it had at least a proof of concept (such as a case study or an experiment). The less mature one was excluded if two publications addressed the same topic and were published in different conferences or journals. The journals and conference proceedings in the manual search besides the primary studies in the automatic search were reviewed by four researchers (including a principal investigator, a junior researcher, and two research assistants).

7.3.3 Quality assessment

In addition to the inclusion and exclusion criteria, it is essential to assess the quality of primary studies (Kitchenham, 2004). The quality assessment of primary studies comes up with more detailed inclusion and exclusion criteria, guides the interpretation of findings and determines the strength of inferences, and offers recommendations for further research. Recording the strengths and weaknesses of primary studies indicates whether aspects of study design or conduct have biased the results (substantially the extent to which the study results can be "believed") (Khan et al., 2001).

Dybå & Dingsøyr (2008) introduced three main issues (Rigour, Credibility, and Relevance) regarding the quality of primary studies that should be taken into account when assessing primary studies in an SLR. *Rigour* indicates whether a thorough and appropriate approach has been applied to research methods in the study. *Credibility* signifies whether the findings are well-presented and meaningful. *Relevance* denotes whether the results are useful to the software industry and the research community. Dyba and Dingsoyr presented 11 quality assessment questions to cover the three main issues that have been used in our assessment.

Both the first and second authors determined quality assessment criteria independently. Discrepancies arose in around 10% of the articles, and these were discussed collaboratively to come to a final judgment. The questions provide a measure of the extent to which we can be confident that primary study findings can make a valuable contribution to the review. The grading of each of the 11 quality assessment questions was done on a dichotomous ("yes" or "no") scale. Table 7.4 shows the result of the quality assessment questions for the primary studies in the SLR.

7.3.4 Search process

The number of primary studies at each stage of the search process in this paper is presented in Figure 7.2. First, we found *20,278* articles as a result of the manual search. Due to the considerable amount of retrieved publications in this step, the first round of selection was performed (Review topic area, titles, abstracts, and conclusions). Some publications were not easy to select based only on their titles and keywords, so such publications were preserved for the next round of selection (*7,042* publications). At the end of the second step, *2,005* publications met the inclusion criteria in the manual search process. Next, by scanning and skimming the text of the selected

Table 7.4: Quality assessment: each primary study in the SLR has been assessed based on these qualities. This table shows the percentages of the "yes/no" answers to the quality assessment question based on the 232 selected primary studies in the SLR.

quality assessment question	Yes (%)	No (%)
Is the paper based on research (or is it merely a "lessons learned" report based on expert opinion)?	98.71	1.29
Is there a clear statement of the aims of the research?	98.29	1.72
Is there an adequate description of the context in which the research was carried out?	96.12	3.88
Was the research design appropriate to address the aims of the research?	75.3	24.57
Was the recruitment strategy appropriate to the aims of the research?	90.95	9.05
Was there a control group with which to compare treatments?	10.34	89.66
Was the data collected in a way that addressed the research issue?	86.64	13.36
Was the data analysis sufficiently rigorous?	85.78	14.22
Has the relationship between researcher and participants been considered to an adequate degree?	46.98	53.02
Is there a clear statement of findings?	100	0.00
Is the study of value for research or practice?	100	0.00

publications, *493* relevant publications were identified. After that, snowballing was performed to scan the references of the selected publications to explore and identify *43* more studies in the manual search process. In the last round of selection, if a publication met all the inclusion and exclusion criteria, it was included. After reading the primary studies thoroughly, *209* publications were selected. The quality of the primary studies was reevaluated according to the quality assessment questions to exclude the low-quality publications (*11* publications were removed).

Next, the query was built according to the extracted keywords from the primary studies of the manual search. After performing the automatic search, *1,095* publications were found. In the first round of review, *311* primary studies were selected according to their topic areas, titles, abstracts, and conclusions. Afterward, inclusion and exclusion criteria were applied to refine the primary studies, so *189* articles were moved to the next stage. Based on the scanning and skimming of the primary studies, *74* papers were considered for performing snowballing. Subsequently, *9*, more studies were added to the knowledge base of the SLR. After reading the primary studies completely, *37* primary studies were selected. The quality of the primary studies was reevaluated according to the quality assessment questions to exclude the low-quality publications (*3* publications were removed).

Eventually, *232* high-quality primary studies (*198 + 34*) promoted to the knowledge base[3] of the SLR for performing the knowledge extraction process.

7.3.5 Knowledge extraction process

A structured coding procedure is employed to extract knowledge from the selected primary studies. Structured coding captures a conceptual area of the research interest (Saldaña, 2015). The extracted knowledge has been classified into six categories: *Patterns*, *Quality Attributes*, *Impacts*, *Application domains*, *Combinations*, and *Trends*. The rest of this study reports the results of data analysis with a descriptive approach.

[3]The knowledge base of this study, including the primary studies and extracted knowledge, is available as a technical report on the following web page: http://swapslr.com

7.3.6 Threats to validity

The validity assessment is an essential part of any empirical study, including SLRs (Zhou et al., 2016). The validity frequently involves Construct Validity, Internal Validity, External Validity, and Conclusion Validity. Other types of validity, such as Theoretical validity and Interpretive validity, were rarely considered in the field of software architecture, so they are not discussed in this paper.

Construct validity refers to whether an accurate operational measure or test has been used for the concepts being studied. In this study, a meta-model (see Figure 7.1), based on the ISO/IEC/IEEE standard 42010 (ISO, 2011), was built to represent the decision-making process in designing software architecture. The essential elements of the meta-model are utilized to formulate the research questions. The meta-model guarantees that the research questions cover all potential publications regarding patterns. The query in the automatic search was built based on the meta-model, so we tried to obtain more relevant studies as much as possible.

Internal validity attempts to verify claims about the cause-effect relationships within the context of a study. In other words, it determines whether the study is sound or not. In order to ensure that the paper selection process was unbiased as far as possible, the quasi-gold standard (QGS) (Zhang & Babar, 2010; Zhang et al., 2011) was adopted. The QGS systematically integrates manual and automated search strategies and suggests a relatively accurate search performance evaluation in terms of sensitivity and precision. Although we searched six online digital libraries, they are believed to cover the majority of the high-quality publications in software architecture. To capture as many publications as possible, however, we also employed the snowballing as the complementary search to diminish the possibility of missing relevant publications. The journals and conference proceedings in the manual search and the primary studies in the automatic search were reviewed by four researchers, including a principal investigator, a junior researcher, and two research assistants. Moreover, the practitioner evaluation sections reflect the usefulness and effectiveness of the SLR findings from real-world software architects' perspectives.

External validity defines the domain to which the research findings can be generalized to real-world applications. External validity is sometimes employed interchangeably with generalizability (feasibility of applying the results to other research settings). In this study, we selected publications that include a discussion about patterns from 1990 to 2019. The excluded studies and inaccessible studies may affect the generalizability of the SLR. However, as less than 3% was not accessible to us, we do not expect that data was missed that would significantly influence our results. The reusable extracted knowledge available through this study can help both academics and practitioners develop new theories and methods for future challenges.

Conclusion validity verifies whether the methods of a study such as the data collection method can be reproduced, with similar results. We captured knowledge from the selected publications regarding *Patterns*, *Quality Attributes*, *Impacts*, *Application domains*, *Combinations*, and *Trends*. The accuracy of the extracted knowledge was guaranteed through the protocol that was developed to define the knowledge extraction strategy and format. The review protocol was proposed and reviewed by the authors. We defined a data extraction form to obtain consistent extraction of relevant knowledge and checked whether the acquired knowledge would address the research questions. Both the first and second authors determined quality assessment criteria independently. Moreover, the crosscheck was necessary among the reviewers, and again we had at least two researchers extracting data independently.

7.3.7 Analysis and Results

Patterns

Patterns offer universal and reusable solutions to commonly occurring problems in software architecture design (Avgeriou & Zdun, 2005). Finding the most common set of patterns helps software architects to have a better understanding of design decision problems and potential solutions to solve such problems.

Figure 7.3 provides an overview of the number of studies that considered each pattern as one of their design decisions or pattern alternatives. The primary studies that discuss the patterns are spread across the early years of the emergence of software architecture (1990) (Kruchten et al., 2006) to the present (2019). Figure 7.3 shows the distribution of theses primary studies over the 29 years. To prevent potential biases, we only considered the patterns mentioned in at least three primary studies. Each selected publication was at least relevant to a particular pattern and discussed its characteristics (such as liabilities, strengths, components, connections, and typologies) and domains (see Section 7.3.7). Consequently, *29* patterns[4] satisfied the constraints and were included in this study.

The number of primary studies from the year 2005 has increased significantly. Furthermore, more than 20 percent of the primary studies were published in the years 2010 and 2011. As the academic literature is merely a reflection of the multitude of patterns that are being used in the industry, we must note that occurrence in academic literature does not necessarily mean occurrence in the industry. Figure 7.3 shows that *Client-Server*, *Layers*, *Pipes and Filters*, *Service-Oriented Architecture (SOA)*, and *Model-View-Controller (MVC)* are the top 5 architectural patterns that were investigated in the primary studies.

Quality Attributes

One of the fundamental concepts in software architecture specification is identifying required levels of measurement of software quality attributes or system qualities such as *performance*, *security*, *available*, and *reusability*.

In the literature, patterns are described according to the functionality they deliver, and their strengths or liabilities are shown concerning several quality attributes (Me et al., 2016). Strengths and liabilities assess the importance of the impact of patterns on quality attributes (Harrison & Avgeriou, 2007). Therefore, patterns and quality attributes are not independent and have significant explicit/implicit interactions (Harrison & Avgeriou, 2010). Such interactions can be represented as reusable

[4]A textual definition of each of the patterns is available in the technical report on the following web page: http://swapslr.com

Figure 7.3: This figure demonstrates the number of primary studies per year (1990-2019) that were relevant to a particular pattern. The bottom of the figure indicates the total number of primary studies that were relevant to the patterns. For example, *90* publications in the knowledge base of this study discussed the *Client-Server* pattern. The right side of the figure shows the number of primary studies per year. Some of the studies discussed more than one pattern. Hence the sum of numbers in the bottom row exceeds the total number of studies found. For instance, we found 87 publications in the year 2010.

	Patterns																													
Year	CLIENT-SERVER	LAYERS	PIPES AND FILTERS	SOA	MVC	COMPONENT-BASED	BLACKBOARD	PUBLISH-SUBSCRIBE	C2	IMPLICIT INVOCATION	BROKER	SHARED REPOSITORY	SPACE-BASED	PEER-TO-PEER	MICROSERVICE	PAC	MICROKERNEL	RPC	VIRTUAL MACHINE	REFLECTION	INTERPRETER	RULE-BASED SYSTEM	EXPLICIT INVOCATION	MASTER-SLAVE	BATCH SEQUENTIAL	INDIRECTION LAYER	INTERCEPTOR	MESSAGE QUEUING	CQRS	# studies per year
2019				1											1															2
2018	4	2	1	3		1		2					3	1	5											1		1	1	25
2017	1			7	2	1				1			2		9				1										1	25
2016	5	6	4	3	6	1	1	2	1	1	3	2	3	1	2	1	1			2		1							1	47
2015	7	6	3	3	3	4	2	1		1	3		5		1		1				1			1		1	1			44
2014	4	3	1	8	2	2		2		1	1	1	3			1	1	2										1		33
2013	2	1	2	4		1	1			1		2	1	1				1	1			1	1					1		21
2012	2	4	1	6	4		1						2			1					1		1							23
2011	5	6	5	6	3	3	3	2	1	1	1	3	1	3		1	1	1	1	1			2		1					51
2010	11	15	9	3	6		6	5	5	3	4	5		1		4	3	1	2	2	1			1						87
2009	1	1	4	5	1	1		2	3	1				2										1						22
2008	5	6	6	1	6		5	3	3	2	4	2		2		3	1		2		2		1			1	1			56
2007	5	4	2	4	3	3	1	3	1		1			2		2	2	1		1										35
2006	7	4	3	2	1	2		1	4		1			3				1		1					1		1			32
2005	5	3	1		2	2	1	2	2	1	1	2		3		1	1	1	1	1	1	1	1		1	1	1	1		37
2004	4	2			1	4		1	2		2	1																		17
2003	4		1		1	1		1								1								1						10
2002	3		1	1		1			1																					7
2001	1	1	2		1		1	1			1					1	2	1		1				1						14
2000	2	2	1		1		1			1	1					1	1	1		1										13
1999	3	3	4		1	4	1		3	2	1	1				1		1				1			1					27
1998	1		1							1																				3
1997	1		1						1																					3
1996		2	2		1	1	1		1	2								1	1		1									13
1995	6	3	5			1	3			3		1						2	1		1									26
1994			1							1																				2
1993		1	1				1			1		1							1		1				1					8
1992																						1								1
1991		1					1												1		1	1								5
1990	1																													1
# studies (patterns)	90	76	62	57	45	33	30	28	28	24	24	21	20	19	18	18	14	14	12	10	10	6	6	5	5	4	4	4	3	
# Experts	12	12	8	12	12	12	6	12	3	6	12	4	8	12	12	2	6	12	12	5	7	9	5	9	9	3	4	12	9	

knowledge elements (Me et al., 2016). For instance, selecting the *Layers* pattern involves a trade-off between efficiency and maintainability, where the second quality attribute is better fit (Harrison & Avgeriou, 2007).

We tried to identify the most widespread quality concerns that were considered in the literature. Figure 7.4 indicates the quality attributes that were explicitly mentioned in at least three primary studies. We encountered *40* relevant quality attributes. According to the results of the analysis (see Figure 7.4), *Reusability*, *Flex-*

ibility, *Performance efficiency*, *Scalability*, and *Maintainability* are the top five software quality attributes that were investigated and reported on in more than *30* primary studies.

Figure 7.4 shows that *Characteristics* of the ISO/IEC 25010 standard (such as *Reliability*, *Performance efficiency*, *Usability*, and *Maintainability*) were considered as quality concerns in the primary studies. However, *Subcharacteristics* of the ISO/IEC 25010 standard (such as *Operability* and *Accountability*) were less discussed in the primary studies. Note, the quality attributes printed in black are based on the ISO/IEC 25010 standard (ISO, 2011), and the rest of them (printed in blue) are not mentioned in the ISO standard[5]. Each cell of the matrix contains two rows. The first row is a triple (L|N|H), including the numbers of studies that reported a particular quality attribute as a Liability (L), Neutral (N), and Strength (H) for its corresponding pattern. The decimal numbers in the second rows of the stained cells show the results of the fuzzy calculation for the impacts.

Impacts

Every architecture decision is made with a rationale. A strength or liability is an argument to utilize or to avoid a pattern in a particular situation (Me et al., 2016). Therefore, the degree to which patterns impact quality attributes determines architectural decisions (i.e., adopting or avoiding a pattern for a given design problem).

When architects have to make architecture decisions, an understanding of the impacts of patterns on quality attributes is needed. The solution space from which an architect must select one design is far more extensive than an architect can oversee (Sabry, 2015). Our observation that further illustrates this problem is that it is not uncommon in industry to hire an architect with experience and expertise with a particular pattern. As such, software architects need better decision support tooling, to help them make their decisions with the right knowledge at hand.

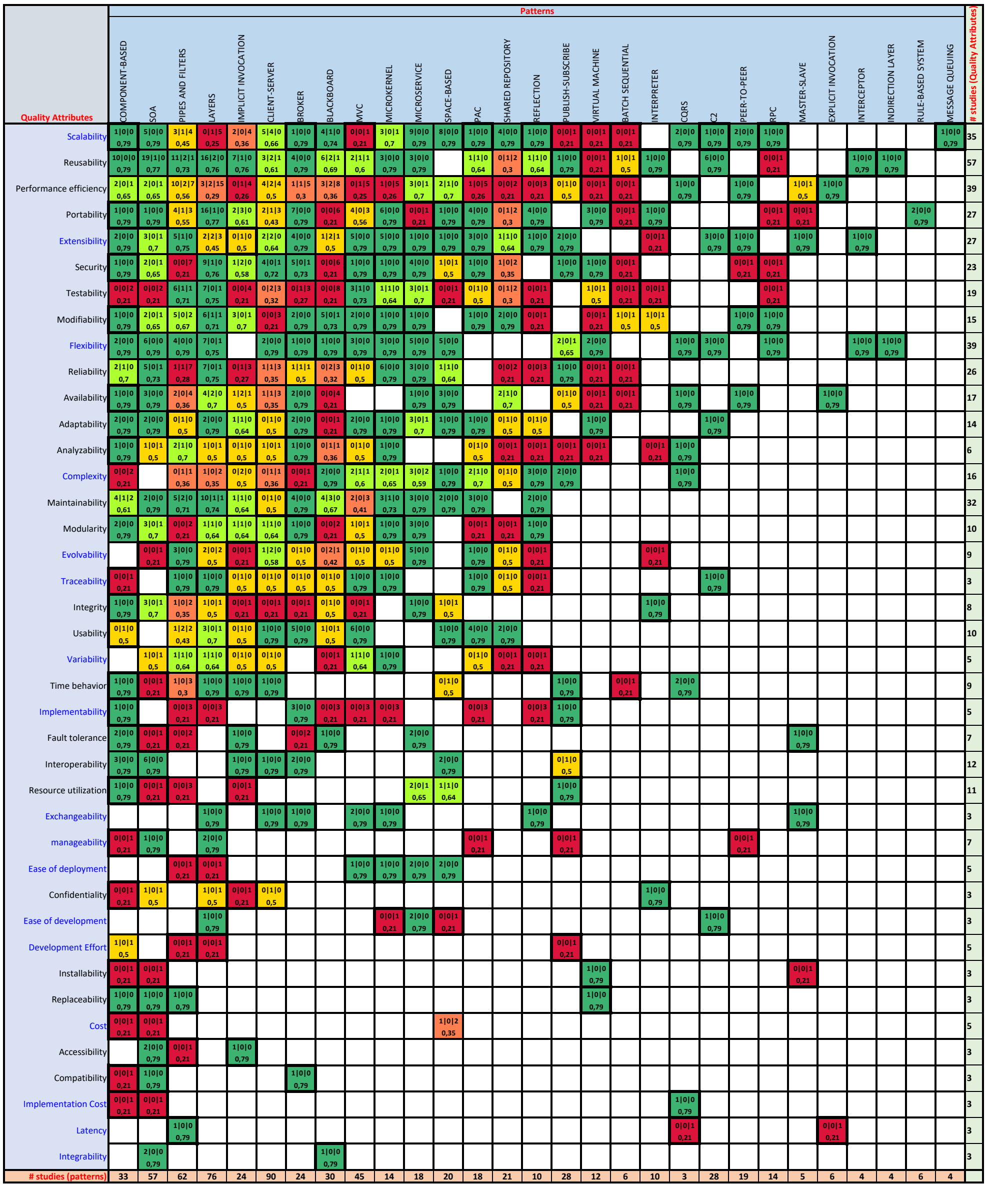
Identifying the impacts of patterns on quality attributes requires analysis of a considerable amount of knowledge regarding patterns (Harrison & Avgeriou, 2010). Missing the impacts of patterns on quality attributes at architecture design time leads to additional liabilities. Because quality attributes are system-wide capabilities, they generally cannot be evaluated entirely until the whole system can be evaluated (Burnstein, 2006).

In the knowledge extraction phase of this study, we realized some inconsistencies regarding the observed impacts of patterns on quality attributes. Some studies reported conflicting impacts of a particular pattern on a quality attribute. For instance, (Harrison & Avgeriou, 2008a; Qin et al., 2008; Sharma et al., 2015) stated that *efficiency* is a strength of the *Pipe and Filter* pattern, however, (Vogel et al., 2011) expressed that *efficiency* is a liability for this pattern. Therefore, *efficiency* can be considered as both strength and liability of the *Pipes and Filters* pattern.

Quantifying the impact of a particular pattern on the quality attributes is complicated because quality attributes are system-wide capabilities. Generally, they cannot be evaluated entirely until the whole system can be evaluated. In this study, we applied fuzzy logic as a method to aggregate the extracted knowledge regarding

[5]The definitions of the quality attributes are entirely available in the technical report on the following web page: Http://swapslr.com

Figure 7.4: This figure shows the Quality Impact Matrix. Liabilities (red cells), Strengths (green cells), Neutrals (yellow cells), and Unknown (white cells) are shown based on the cell colors. Furthermore, the color intensity is an indicator of agreement among studies as well as the numbers in the cells. This table provides the relationships between patterns and quality attributes. Note, as this figure is hard to read, a larger version is available from http://swapslr.com.



	Patterns																													
Quality Attributes	COMPONENT-BASED	SOA	PIPES AND FILTERS	LAYERS	IMPLICIT INVOCATION	CLIENT-SERVER	BROKER	BLACKBOARD	MVC	MICROKERNEL	MICROSERVICE	SPACE-BASED	PAC	SHARED REPOSITORY	REFLECTION	PUBLISH-SUBSCRIBE	VIRTUAL MACHINE	BATCH SEQUENTIAL	INTERPRETER	CQRS	C2	PEER-TO-PEER	RPC	MASTER-SLAVE	EXPLICIT INVOCATION	INTERCEPTOR	INDIRECTION LAYER	RULE-BASED SYSTEM	MESSAGE QUEUING	# studies (Quality Attributes)
Scalability	1|0|0 0,79	5|0|0 0,79	3|1|4 0,45	0|1|5 0,25	2|0|4 0,36	5|4|0 0,66	1|0|0 0,79	4|1|0 0,74	0|0|1 0,21	3|0|1 0,7	9|0|0 0,79	8|0|0 0,79	1|0|0 0,79	4|0|0 0,79	1|0|0 0,79	0|0|1 0,21	0|0|1 0,21	0|0|1 0,21		2|0|0 0,79	1|0|0 0,79	2|0|0 0,79	1|0|0 0,79						1|0|0 0,79	35
Reusability	10|0|0 0,79	19|1|0 0,77	11|2|1 0,73	16|2|0 0,76	7|1|0 0,76	3|2|1 0,61	4|0|0 0,79	6|2|1 0,69	2|1|1 0,6	3|0|0 0,79	3|0|0 0,79		1|1|0 0,64	0|1|2 0,3	1|1|0 0,64	1|0|0 0,79	0|0|1 0,21	1|0|1 0,5	1|0|0 0,79		6|0|0 0,79		0|0|1 0,21			1|0|0 0,79	1|0|0 0,79			57
Performance efficiency	2|0|1 0,65	2|0|1 0,65	10|2|7 0,56	3|2|15 0,29	0|1|4 0,26	4|2|4 0,5	1|1|5 0,3	3|2|8 0,36	0|1|5 0,25	1|0|5 0,26	3|0|1 0,7	2|1|0 0,7	1|0|5 0,26	0|0|2 0,21	0|0|3 0,21	0|1|0 0,5	0|0|1 0,21	0|0|1 0,21		1|0|0 0,79		1|0|0 0,79		1|0|1 0,5	1|0|0 0,79					39
Portability	1|0|0 0,79	1|0|0 0,79	4|1|3 0,55	16|1|0 0,77	2|3|0 0,61	2|1|3 0,43	7|0|0 0,79	0|0|6 0,21	4|0|3 0,56	6|0|0 0,79	0|0|1 0,21	1|0|0 0,79	4|0|0 0,79	0|1|2 0,3	4|0|0 0,79		3|0|0 0,79	0|0|1 0,21	1|0|0 0,79				0|0|1 0,21	0|0|1 0,21				2|0|0 0,79		27
Extensibility	2|0|0 0,79	3|0|1 0,7	5|1|0 0,75	2|2|3 0,45	0|1|0 0,5	2|2|0 0,64	4|0|0 0,79	1|2|1 0,5	5|0|0 0,79	5|0|0 0,79	1|0|0 0,79	1|0|0 0,79	3|0|0 0,79	1|1|0 0,64	1|0|0 0,79	2|0|0 0,79			0|0|1 0,21		3|0|0 0,79	1|0|0 0,79		1|0|0 0,79		1|0|0 0,79				27
Security	1|0|0 0,79	2|0|1 0,65	0|0|7 0,21	9|1|0 0,76	1|2|0 0,58	4|0|1 0,72	5|0|1 0,73	0|0|6 0,21	1|0|0 0,79	1|0|0 0,79	4|0|0 0,79	1|0|1 0,5	1|0|0 0,79	1|0|2 0,35		1|0|0 0,79	1|0|0 0,79	0|0|1 0,21				0|0|1 0,21	0|0|1 0,21							23
Testability	0|0|2 0,21	0|0|2 0,21	6|1|1 0,71	7|0|1 0,75	0|0|4 0,21	0|2|3 0,32	0|1|3 0,27	0|0|8 0,21	3|1|0 0,73	1|1|0 0,64	3|0|1 0,7	0|0|1 0,21	0|1|0 0,5	0|1|2 0,3	0|0|1 0,21		1|0|1 0,5	0|0|1 0,21	0|0|1 0,21				0|0|1 0,21							19
Modifiability	1|0|0 0,79	2|0|1 0,65	5|0|2 0,67	6|1|1 0,71	3|0|1 0,7	0|0|3 0,21	2|0|0 0,79	5|0|1 0,73	2|0|0 0,79	1|0|0 0,79	1|0|0 0,79		1|0|0 0,79	2|0|0 0,79	0|0|1 0,21		0|0|1 0,21	1|0|1 0,5	1|0|1 0,5			1|0|0 0,79	1|0|0 0,79							15
Flexibility	2|0|0 0,79	6|0|0 0,79	4|0|0 0,79	7|0|1 0,75		2|0|0 0,79	1|0|0 0,79	1|0|0 0,79	3|0|0 0,79	3|0|0 0,79	5|0|0 0,79	5|0|0 0,79				2|0|1 0,65	2|0|0 0,79			1|0|0 0,79	3|0|0 0,79		1|0|0 0,79			1|0|0 0,79	1|0|0 0,79			39
Reliability	2|1|0 0,7	5|0|1 0,73	1|1|7 0,28	7|0|1 0,75	0|1|3 0,27	1|1|3 0,35	1|1|1 0,5	0|2|3 0,32	0|1|0 0,5	6|0|0 0,79	3|0|0 0,79	1|1|0 0,64		0|0|2 0,21	0|0|3 0,21	1|0|0 0,79	0|0|1 0,21	0|0|1 0,21												26
Availability	1|0|0 0,79	3|0|0 0,79	2|0|4 0,36	4|2|0 0,7	1|2|1 0,5	1|1|3 0,35	2|0|0 0,79	0|0|4 0,21			1|0|0 0,79	3|0|0 0,79		2|1|0 0,7		0|1|0 0,5	0|0|1 0,21	0|0|1 0,21		1|0|0 0,79		1|0|0 0,79			1|0|0 0,79					17
Adaptability	2|0|0 0,79	2|0|0 0,79	0|1|0 0,5	2|0|0 0,79	1|1|0 0,64	0|1|0 0,5	2|0|0 0,79	0|0|1 0,21	2|0|0 0,79	1|0|0 0,79	3|0|1 0,7	1|0|0 0,79	1|0|0 0,79	0|1|0 0,5	0|1|0 0,5		1|0|0 0,79				1|0|0 0,79									14
Analyzability	1|0|0 0,79	1|0|1 0,5	2|1|0 0,7	1|0|1 0,5	0|1|0 0,5	1|0|1 0,5	1|0|0 0,79	0|1|1 0,36	0|1|0 0,5	1|0|0 0,79			0|1|0 0,5	0|0|1 0,21	0|0|1 0,21	0|0|1 0,21	0|0|1 0,21		0|0|1 0,21	1|0|0 0,79										6
Complexity	0|0|2 0,21		0|1|1 0,36	1|0|2 0,35	0|2|0 0,5	0|1|1 0,36	0|0|1 0,21	2|0|0 0,79	2|1|1 0,6	2|0|1 0,65	3|0|2 0,59	1|0|0 0,79	2|1|0 0,7	0|1|0 0,5	3|0|0 0,79	2|0|0 0,79				1|0|0 0,79										16
Maintainability	4|1|2 0,61	2|0|0 0,79	5|2|0 0,71	10|1|1 0,74	1|1|0 0,64	0|1|0 0,5	4|0|0 0,79	4|3|0 0,67	2|0|3 0,41	3|1|0 0,73	3|0|0 0,79	2|0|0 0,79	3|0|0 0,79		2|0|0 0,79															32
Modularity	2|0|0 0,79	3|0|1 0,7	0|0|2 0,21	1|1|0 0,64	1|1|0 0,64	1|1|0 0,64	1|0|0 0,79	0|0|2 0,21	1|0|1 0,5	1|0|0 0,79	3|0|0 0,79		0|0|1 0,21	0|0|1 0,21	1|0|0 0,79															10
Evolvability		0|0|1 0,21	3|0|0 0,79	2|0|2 0,5	0|0|1 0,21	1|2|0 0,58	0|1|0 0,5	0|2|1 0,42	0|1|0 0,5	0|1|0 0,5	5|0|0 0,79		1|0|0 0,79	0|1|0 0,5	0|0|1 0,21				0|0|1 0,21											9
Traceability	0|0|1 0,21		1|0|0 0,79	1|0|0 0,79	0|1|0 0,5	0|1|0 0,5	0|1|0 0,5	0|1|0 0,5	1|0|0 0,79	1|0|0 0,79			1|0|0 0,79	0|1|0 0,5	0|0|1 0,21						1|0|0 0,79									3
Integrity	1|0|0 0,79	3|0|1 0,7	1|0|2 0,35	1|0|1 0,5	0|0|1 0,21	0|0|1 0,21	0|0|1 0,21	0|1|0 0,5	0|0|1 0,21		1|0|0 0,79	1|0|1 0,5							1|0|0 0,79											8
Usability	0|1|0 0,5		1|2|2 0,43	3|0|1 0,7	0|1|0 0,5	1|0|0 0,79	5|0|0 0,79	1|0|1 0,5	6|0|0 0,79			1|0|0 0,79	4|0|0 0,79	2|0|0 0,79																10
Variability		1|0|1 0,5	1|1|0 0,64	1|1|0 0,64	0|1|0 0,5	0|1|0 0,5		0|0|1 0,21	1|1|0 0,64	1|0|0 0,79			0|1|0 0,5	0|0|1 0,21	0|0|1 0,21															5
Time behavior	1|0|0 0,79	0|0|1 0,21	1|0|3 0,3	1|0|0 0,79	1|0|0 0,79	1|0|0 0,79						0|1|0 0,5				1|0|0 0,79		0|0|1 0,21		2|0|0 0,79										9
Implementability	1|0|0 0,79		0|0|3 0,21	0|0|3 0,21			3|0|0 0,79	0|0|3 0,21	0|0|3 0,21	0|0|3 0,21			0|0|3 0,21		0|0|3 0,21	1|0|0 0,79														5
Fault tolerance	2|0|0 0,79	0|0|1 0,21	0|0|2 0,21		1|0|0 0,79		0|0|2 0,21	1|0|0 0,79			2|0|0 0,79													1|0|0 0,79						7
Interoperability	3|0|0 0,79	6|0|0 0,79			1|0|0 0,79	1|0|0 0,79	2|0|0 0,79					2|0|0 0,79				0|1|0 0,5														12
Resource utilization	1|0|0 0,79	0|0|1 0,21	0|0|3 0,21		0|0|1 0,21						2|0|1 0,65	1|1|0 0,64				1|0|0 0,79														11
Exchangeability				1|0|0 0,79		1|0|0 0,79	1|0|0 0,79		2|0|0 0,79	1|0|0 0,79					1|0|0 0,79									1|0|0 0,79						3
manageability	0|0|1 0,21	1|0|0 0,79		2|0|0 0,79									0|0|1 0,21			0|0|1 0,21						0|0|1 0,21								7
Ease of deployment			0|0|1 0,21	0|0|1 0,21					1|0|0 0,79	1|0|0 0,79	2|0|0 0,79	2|0|0 0,79																		5
Confidentiality	0|0|1 0,21	1|0|1 0,5		1|0|1 0,5	0|0|1 0,21	0|1|0 0,5													1|0|0 0,79											3
Ease of development				1|0|0 0,79						0|0|1 0,21	2|0|0 0,79	0|0|1 0,21									1|0|0 0,79									3
Development Effort	1|0|1 0,5		0|0|1 0,21	0|0|1 0,21												0|0|1 0,21														5
Installability	0|0|1 0,21	0|0|1 0,21															1|0|0 0,79							0|0|1 0,21						3
Replaceability	1|0|0 0,79	1|0|0 0,79	1|0|0 0,79														1|0|0 0,79													3
Cost	0|0|1 0,21	0|0|1 0,21										1|0|2 0,35																		5
Accessibility		2|0|0 0,79	0|0|1 0,21		1|0|0 0,79																									3
Compatibility	0|0|1 0,21	1|0|0 0,79					1|0|0 0,79																							3
Implementation Cost	0|0|1 0,21	0|0|1 0,21																		1|0|0 0,79										3
Latency			1|0|0 0,79																	0|0|1 0,21					0|0|1 0,21					3
Integrability		2|0|0 0,79						1|0|0 0,79																						3
# studies (patterns)	33	57	62	76	24	90	24	30	45	14	18	20	18	21	10	28	12	6	10	3	28	19	14	5	6	4	4	6	4	

the potential impacts of patterns on quality attributes.

Fuzzy Logic Calculations - we employed fuzzy logic (Chen, 1998) as a method for aggregating individual fuzzy opinions into a group fuzzy consensus pinion. Suppose each primary study as an individual expert, where expert $E_i(i = 1, 2, ..., n)$ constructs a positive trapezoidal fuzzy number R_i with membership functions $M_{R_i}(x)$ to represent his/her opinion on a particular impact. In this study, we defined the following trapezoidal fuzzy numbers for Liability (L), Neutral (N), and Strength (H):

$$L = (0.0, 0.1428, 0.2856, 0.4286)$$

$$N = (0.2856, 0.4286, 0.5712, 0.7140)$$

$$H = (0.5712, 0.7140, 0.8568, 1.0)$$

Suppose $R_1 = (a_1, b_1, c_1, d_1)$ and $R_2 = (a_2, b_2, c_2, d_2)$ are two trapezoidal numbers that represent two experts' opinion in fuzzy space, then the similarity $S(R_1, R_2)$ between these R_1 and R_2 is defined as follows (Chen, 1998):

$$S(R_1, R_2) = 1 - \frac{|a_1 - a_2| + |b_1 - b_2| + |c_1 - c_2| + |d_1 - d_2|}{4}$$

The degree of agreement $A(E_i)$ of expert E_i is calculated based on the following equation:

$$A(E_i) = \frac{1}{n-1} \sum_{j=1 \,\wedge\, i \neq j}^{n} S(R_i, R_j); i = 1, 2, .., n$$

The relative degree of agreement $RA(E_i)$ of expert E_i is defined as follows:

$$RA(E_i) = \frac{A(E_i)}{\sum_{i=1}^{n} A(E_i)}; i = 1, 2, .., n$$

Finally, the aggregation of fuzzy opinion is calculated based on the following equation (Chen, 1998):

$$R = RA(E_1) \otimes R_1 \oplus RA(E_2) \otimes R_2 \oplus ... \oplus RA(E_n) \otimes R_n$$

Note, in this study, we used Mean of Maxima (MoM) as a method of deffuzification, so that, $MoM(L) = 0.21$, $MoM(N) = 0.50$, and $MoM(H) = 0.79$.

Figure 7.4 presents the impacts of the patterns on the quality attributes. Note, the impacts have been reported as *Liabilities* (red cells), *Strengths* (green cells), *Neutrals* (yellow cells), or *Unknown* (white cells). The *Unknown* impacts mean that we did not find any information about them. Note, the cells with thick borders signify singleton impacts, which means that we found only one study that has been discussed those impacts. The coloring codes are the results of the calculated fuzzy logic (the decimal number in the second row of each colored cell) to gain a consensus among studies. Therefore the color intensity indicates the agreements among studies on particular impacts. In other words, the color intensity can help decision-makers to have a better understanding of existing knowledge in the literature concerning the reported impacts. For instance, we found *20* studies regarding the impact of *Layers* pattern on *Reusability*, so that, *17* studies considered *Reusability* as a key strength, *2* studies mentioned some *Reusability* challenge, and only one study asserted that *Reusability* is a key liability for the *Layers* pattern. Therefore, the dark green color can be interpreted that *Reusability* is a key liability for the *Layers* pattern; however, some *Reusability* challenges reported in the literature regarding this impact.

Application domains

By increasing knowledge about patterns, it is possible to make better-informed decisions, avoid failures, and better satisfy quality attributes and achieve system-wide quality targets (Me et al., 2016).

Application-generic and application-specific knowledge are two types of architectural knowledge (Lago & Avgeriou, 2006). Application-generic knowledge refers to knowledge that software architects have implicitly in their heads, from their former experience in working in one or more domains. Moreover, application-specific knowledge involves all the decisions taken during the architecting process of a particular system and the architectural solutions that implemented the decisions. Therefore, application-generic knowledge is used to make decisions for a single application and thus construct application-specific knowledge.

The application domains, in which the observed patterns are used, support software architects in selecting appropriate patterns for their problem domain. Figure 7.5 shows the application domains of the identified patterns. We categorized the observed application domains based on the suggested software taxonomy by (Forward & Lethbridge, 2008).

Combinations

Despite an extensive list of patterns documented in the literature, patterns are infrequently applied in a system design in their original form, and they must be combined with other patterns to address different design decisions of the system (Buschmann et al., 2007a). In other words, a particular pattern provides the missing ingredient needed by another pattern or conflicts with another one by providing an alternative solution to a related problem. The goal of combining patterns is to make the resulting design more complete and balanced (Buschmann et al., 2007b).

In general, not all potential combinations of patterns are useful. However, because each pattern description is self-contained and independent of the others, it is difficult to extract the useful combinations from the individual pattern descriptions (Schmidt et al., 2013). The combinations of patterns are more than aggregates of their elements (Kamal & Avgeriou, 2010). Unfortunately, individual patterns descriptions are not always explicit on "how" to combine them with consistent patterns. For instance, the Layers pattern can be combined with the Client-Server pattern, or the C2 and Publish-Subscribe patterns can be used as a paired pattern (Kamal & Avgeriou, 2010).

Suppose PAT is the set of frequently used patterns bu software architects in the SLR, P_1 and P_2 are two patterns, where $P_1, P_2 \in PAT$. When building a solution for a particular problem addressed by P_1, one sub-problem is similar to a problem addressed by P_2. Consequently, the pattern P_1 utilizes the pattern P_2 in its solution.

Figure 7.5: This figure illustrates possible applications of the architectural patterns according to the SLR. The numbers in the cells show the number of studies that discussed the corresponding application domain of an architectural pattern. Note, in the first column, cells in the dark blue indicate the categories of the application domains.

	Patterns																													
Application domains	SOA	PIPES AND FILTERS	LAYERS	SHARED REPOSITORY	COMPONENT-BASED	CLIENT-SERVER	BLACKBOARD	MVC	MICROSERVICE	IMPLICIT INVOCATION	RPC	SPACE-BASED	MICROKERNEL	PEER-TO-PEER	BROKER	PUBLISH-SUBSCRIBE	C2	VIRTUAL MACHINE	PAC	EXPLICIT INVOCATION	REFLECTION	INTERPRETER	MASTER-SLAVE	MESSAGE QUEUING	CQRS	RULE-BASED SYSTEM	BATCH SEQUENTIAL	INDIRECTION LAYER	INTERCEPTOR	# studies
Design and Engineering Software	10	8	8	2	14	2	1	11	1	5	1		2	1	2	2	3	4	6		2	3	1							89
Interactive System								9		2							2		6											19
Compiler Design		5	1							1	1							4				3								15
Case & Related Developer Tools			2												2															4
Commercial-Off-The-Shelf (Cots)	6				8																									14
Data Base Systems			2	2		2				2						1														9
Context-Aware Systems	2	2	2		2		1	1						1									1							12
System Families	1	1														1														3
Adaptable Systems													2				1				2									5
Software Product Line (Spl)	1		1		4			1	1																					8
Distributed Computing	24	3	6		2	11	2	2	8		3	4	1	2	5		3			1			1	1	2		1			82
Distributed Systems	18	2	3		2	11	2	1	5		3	1	1	2	5		3			1			1		2		1			64
Cloud Computing Applications	3	1							2			3												1						10
Mobile Applications	3		3					1	1																					8
Web Applications / Services	10	3	6		2	13		4	6	1	2	4			3					1										55
Web-Based Systems	4	3	6		1	12		4	2	1		3			3					1										40
Web Services	3				1	1			2		2	1																		10
Service-Based Systems	3								2																					5
Systems Software	1	16	9	1	4	5	2	1		1	1		7			2		1			1	1								53
Operating Systems	1	14	7	1	4	3	1	1		1	1		4			2														40
Network and Communication Systems			2			2																								4
Multi-Processors Environment		2					1																							3
Plug-and-Play Environment													3					1			1	1								6
Strategic and Operations Analysis	27								1	1		1																		30
Enterprise Service Bus (ESB)	17								1	1																				19
Enterprise Application Integration (EAI)	7																													7
Customer Relationship Management (CRM)	3											1																		4
Information Management and Decision Support Systems	5		4	3	1		2	3							2			2	2						2	3				29
Expert System				1			2											1								3				7
Management Information Systems	5		4	2	1			3							2			1	2						2					22
Control-Dominant Software	2	3	2	2	3	3				1	1	1	2	1		1			2	1										25
Embedded Systems	1	2	2	2	2	2				1			1						2	1										16
Real-Time Systems	1	1			1						1		1																	5
Internet Of Things (Iots)						1						1		1		1														4
Computation-Dominant Software		5		2			9																							16
Speech Recognition				1			6																							7
Signal Processing		4					2																							6
Pattern Recognition		1		1			1																							3
Data-Dominant Software				2							1			6										1						10
File-Sharing Applications				1										3																4
Exchange Data And Information				1							1													1						3
Skype Network														3																3
Games / Entertainment		2					2	3									1													8
Gaming Systems		2					2	3									1													8
Transaction Processing	1			1		1																								3
Banking System	1			1		1																								3
Coverage (%)	57	43	39	37	35	33	33	33	26	26	26	22	22	20	17	15	15	15	13	13	8.7	8.7	8.7	8.7	8.7	4.3	4.3	0	0	

Note, a typical combination of patterns is the combination of P_1 and P_2 (e.g, a software architect can employ the Microservice pattern besides Rule-based patterns). In contrast to "P_1 employs P_2", P_1 does not employ P_2 in its solution.

Figure 7.6 illustrates the combinations of the pattern that we found during the SLR. The observed combinations in Figure 7.6 are based on the "P_1 employs P_2" relationships. For example, *17* primary studies stated that the *Client-Server* pattern employs the *Broker* pattern. The broker is responsible for receiving all messages, filtering the messages, deciding who is the owner of each message, and sending the message to the correct clients.

Figure 7.6: This figure demonstrates the observed combinations of patterns while performing SLR. Please note that we only identified couples, so the figure should be read as that we encountered the broker pattern combined with the Layers pattern five times in the literature. Moreover, the combinations are not symmetric; for instance, Broker-Layers is not the same as Layers-Broker. Architects can use this figure to decide whether a combination they are planning to make, has been made before. Note, each cell in the last row (compatibility) indicates the percentage of the patterns that can be combined with the corresponding pattern based on the selected primary studies. For instance, the "Client-Server" can be combined with 66% of the other patterns in the list.

	Patterns																													
Potential combinations	CLIENT-SERVER	LAYERS	PIPES AND FILTERS	SOA	MVC	SPACE-BASED	MICROKERNEL	PUBLISH-SUBSCRIBE	C2	PEER-TO-PEER	COMPONENT-BASED	PAC	EXPLICIT INVOCATION	MICROSERVICE	VIRTUAL MACHINE	BROKER	INDIRECTION LAYER	SHARED REPOSITORY	BLACKBOARD	INTERPRETER	BATCH SEQUENTIAL	REFLECTION	INTERCEPTOR	CQRS	RULE-BASED SYSTEM	IMPLICIT INVOCATION	RPC	MASTER-SLAVE	MESSAGE QUEUING	# studies
CLIENT-SERVER		18	1	6	1	5	1	2			1	1					1	1						1			1			40
BROKER	17	6		2	3		1	1		2	1		1					2								1				37
LAYERS	9		1	7	3	1	1		1	1	1	1		1	2	2	1	1		1		1								35
SHARED REPOSITORY	3	5	5	1	2	1			2	2	1	2				1			1											26
COMPONENT-BASED	4	10	2	2	1		1	1	1								1													23
PUBLISH-SUBSCRIBE	6		1	3	1				3	2			3			1														20
RPC	9			3			1				2		1	1					1	1										19
PIPES AND FILTERS	2	6				1		1			2								1											13
MVC	2	6	3									1		1																13
BLACKBOARD	2	4	3																		1	1								11
PAC	3	4	2		1																									10
SOA	2	4				2					1																			9
C2	2	1		1				1		2			2																	9
INTERPRETER		2	1		1		1								4															9
INDIRECTION LAYER	1						1		1						2					1			1		1					8
IMPLICIT INVOCATION	1							3	1	1						1														7
EXPLICIT INVOCATION	2				1				3	1																				7
MESSAGE QUEUING	1			1		1		1					1	1							1									7
MICROKERNEL	1	5																												6
INTERCEPTOR	3			1																										4
SPACE-BASED				2										1																3
PEER-TO-PEER	2					1																								3
MICROSERVICE						3																								3
RULE-BASED SYSTEM			1		1										1															3
VIRTUAL MACHINE							1																							1
REFLECTION																							1							1
BATCH SEQUENTIAL			1																											1
CQRS						1																								1
MASTER-SLAVE																														0
Compatibility (%)	66	41	38	38	34	31	28	24	24	24	24	14	17	17	14	14	10	10	10	10	6,9	6,9	6,9	3,4	3,4	3,4	3,4	0	0	

Trends

The possibility of existing trends among researchers in selecting patterns has been investigated in this SLR. As aforementioned, the primary studies that discuss the patterns are spread across the early years of the emergence of software architecture (1990) (Kruchten et al., 2006) to the present (2019).

Although numerous software systems have succeeded by employing patterns consciously, there have also been failures, due in substantial part to common misinterpretations about patterns, i.e., what they are and what they are not, what characteristics and purpose they have, their target audience, and the various strengths and liabilities of applying them (Buschmann et al., 2007b). The pattern community has long been interested in understanding the underlying theories, forms, and methodologies of patterns, pattern languages, and associated concepts to codify knowledge about, understanding of, and application domains of patterns.

Figure 7.3 shows the distribution of theses primary studies over the 29 years. To prevent potential biases, we only considered the patterns that were mentioned at the minimum of three primary studies. We observe that *SOA*, *Cloud computing architecture (Spaced-based)*, and *Microservices* gained more attention in recent years. Moreover, some patterns such as *C2*, *Presentation-abstraction-control (PAC)*, *Remote Procedure Call (RPC)* and *Batch sequential* patterns are not discussed widely in academic literature.

7.3.8 Discussion

This section summarizes the observed answers to the research questions and identifies several lessons learned. We end the discussion with an interesting question: how can creativity be preserved when an architecture decision is simplified to a limited decision model?

Addressing Research Questions

In this subsection, we reflect on each of the proposed research questions based on the SLR.

To answer the first research question (RQ_1) that aims at identifying the frequently employed patterns since the emergence of the field (1990), we found 29 patterns (see figure 7.3.7) that discussed at more than three primary studies.

The second research question (RQ_2) is the most frequent quality attributes that software architects are mainly concerned about. We found *40* quality attributes (see Figure 7.3.7) that explicitly mentioned in the primary studies as liabilities and strengths of the patterns.

The answer to the third research question (RQ_3) reveals the impacts of the patterns on the quality attributes based on the aggregation of liabilities and strengths reported in primary studies (see Figure 7.4). Such impacts lead to a deeper understanding of the patterns, identify the potential risks of employing a particular pattern, facilitate generating a quality attribute utility tree for a system, improve architecture documentation, and assist software architects with the pattern selection process.

To answer the fourth research question (RQ_4) that aims at finding common application domains, we observed 35 application domains and classified them into 11 categories (see Figure 7.5). With such knowledge regarding the application domains, software architects can determine whether similar patterns have been chosen in their domains.

To answer the fifth research question (RQ_5), we collected a set of suitable combinations of patterns observed in the primary studies (see Figure 7.6). Such combinations can address common sub-problems in patterns, such as solving the communication problem in the Client-Server pattern by the Broker pattern. Note, each paired pattern (e.g., Client-Server-Broker) can have entirely different characteristics from its constituent patterns.

The sixth research question (RQ_6) asks whether trends can be observed in pattern selection among software architects. Figure 7.3 demonstrates the distribution of the observed patterns this SLR from 1990 up to 2019. We realized that some patterns were trending for a period, and other patterns gained more attention after several years. For instance, the *C2* was trending before 2010; moreover, the *Microservice* pattern gained attention in recent years. However, the *Client-Server*, *Layers*, *Pipe and Filters*, *Component based* patterns were almost always considered as primary alternatives in the pattern selection process.

Lessons learned

Knowledge about software patterns and their impacts on quality attributes is spread throughout decades of scientific reporting on pattern observation in practice. Architects must continuously align quality requirements, patterns, tactics, and application domains. It is non-trivial for both practitioners and academics to answer questions such as "what kind of effect does the introduction of *Microservices* have on the variability of a system for end-users?". Software architects typically neglect to sufficiently document their design decisions because they do not appreciate the advantages of documentation of such design decisions (Harrison & Avgeriou, 2010). This lack of accurate documentation can significantly impact future design decisions. Furthermore, it is problematic for the actual architecture in practice.

We can revert to traditional building architecture for several lessons learned. First, we must accept that we will not find a comprehensive set of patterns: technological innovations will continually introduce more complex and specific patterns. Analog to how the elevator has enabled us to build taller buildings, new innovative patterns such as *CQRS* enable us to create larger and more scalable systems. This continuous innovation remains a responsibility of the academic community to consolidate and present architecture knowledge to the practitioner community continuously.

It is possible to identify trends in pattern usage. We hypothesize that software architects are biased towards trending patterns in their architecture design decisions. Over time, quality requirements of systems change because of advances in technology that address particular quality concerns of software architects. Software architects need to have a more explicit awareness of software architecture trends and evaluate them in the context of the system requirements. If we look at traditional building architecture again, it does not come as a surprise that architects are sensitive to trends: patterns may introduce new possibilities that provide end-users with more efficient and satisfying structures.

In this research, we only focus on individual patterns that solve particular parts of a design problem. Patterns, however, have several types of relationships with each other. (1) Patterns can be alternatives to each other, for example, *Interpreter*, *Virtual Machine*, and *Rule-based system*. (2) Patterns can also be complementary and easily combined. For instance, the combination of *Client-Server* and *Broker* is valid and mentioned in some studies in the SLR. (3) Patterns may also be incompatible. For instance, we did not find any combinations of *Pipes and Filters* and *Broker*.

Besides reporting, academics have a responsibility to define what architects need to make explicit. The majority of the primary studies focus on a limited set of patterns and quality attributes (see Figures 7.3 and 7.4), and they were more concerned with generic quality attributes, such as the quality attributes of the ISO/IEC 25010 standard. According to the ISO/IEC 25010 standard description (ISO, 2011), the *Characteristics* are broken down into *Subcharacteristics*. The *Characteristics* are conceptually more generic quality attributes, and conversely, the *Subcharacteristics* have more concrete definitions. Several studies considered a *Characteristics* and its *Subcharacteristics* as two separate quality attributes (For example, *Maintainability* and *Modifiability*). Architects and researchers need to be more accurate in defining the patterns, their usage of them, and the quality attributes they measure them by.

Patterns promoting similar quality attributes sometimes have common characteristics. For instance, both *Layers* and *C2* support *flexibility* and *separation of concerns*, and there is a significant implementation overlap between them. While the similarity of patterns is a reliable indicator of potentially reusable code, it often has the opposite effect on the compositionality of those patterns. Experience shows that the similar patterns (e.g., C2 and Layers) cannot or are not typically composed together (Malek et al., 2010). Our main observation here is that essential relationships with other patterns also characterize patterns. The ability to rapidly compose patterns in this manner opens up new avenues of research to study the compatibility of patterns with one another and to develop new hybrid and domain-specific patterns. One of the most significant threats to this study's validity is that we take the academic reporting of patterns as a representative overview of the industry. In the future, we aim to solve this by also including grey literature in the study. Furthermore, we identify a need for a comprehensive view of patterns, where a curated set of patterns is regularly published as a reference for architects, similar to other industry-specific catalogs.

Software architecture tactics are a sub-class of design decisions and focus on the improvement of particular quality attributes (Harrison & Avgeriou, 2010). For instance, *Ping/Echo* and *Heartbeat* are two tactics that can be selected to improve *Reliability*. If selecting and applying sets of patterns without consideration impede some of the quality attributes, these tactics can be employed to improve a system's quality attributes. A future research challenge is to support architects in this fine-tuning of a selected set of patterns using particular tactics.

Our hypothesis remains that an optimal initial set of patterns will require less use of tactics at a later stage in a system's development. We define an optimal set of patterns as the theoretical set of patterns that best addresses the requirements of the software project, including features (e.g., provides an API), quality (e.g., up to current security standards), and project requirements (feasible to implement with allotted resources). We acknowledge that identifying this set perfectly is impossible, for instance, due to the use of tactics, but in software design, we must strive towards such an optimal set.

Stifling creativity. A relevant question is whether the data provided in this article stifles the architect's creativity: the article could be used to discourage particular new pattern combinations, for instance. We believe that the benefits of having overviews such as the most common combinations, such as in Table 7.3.7 of this article, can inspire architects to work with a broader set of knowledge than they would have before. Following that hypothesis, the information in this article should broaden the architects' knowledge instead of stifling them into set rules.

7.4 Practitioner Evaluation

We followed Myers and Newman guidelines (Myers & Newman, 2007) to conduct a series of qualitative semi-structured interviews with twelve senior software architects to explore expert knowledge regarding architectural patterns and evaluate the outcomes of the SLR.

We developed a role description before contacting potential experts in order to ensure the right target group. We contacted 43 architects in the Netherlands through email using the role description and information about our research topic. Overall, twelve senior software architects at different software producing organizations in the Netherlands participated in this research. The experts were pragmatically and conveniently selected according to their expertise and experience that they mentioned on their *LinkedIn* profile. The experts had, on average, more than ten years of experience with designing architectures. Each of the interviews followed a semi-structured interview protocol and lasted between 60 and 90 minutes.

According to Runeson et al. (2012), we discuss the four threats: construct validity, internal validity, external validity, and reliability. We used open questions to elicit as much information as possible from the experts minimizing prior bias. All interviews were done in person and recorded with the interviewees' permission, then coded for further analysis to decrease a threat to construct validity. In order to mitigate a possible threat to internal validity, we consider a set of expert evaluation criteria (including "Years of experience", "Expertise", "Skills", "Education", and "Level of expertise") to select the experts. This study's relatively small number of interviewees highlights the issue of generalization and the external validity of the research results. However, the diversity of the interviewees, who were working at twelve different software development companies, lead to unbiased and generalize results. The interview protocol and coding were reviewed by two authors of this paper to minimize a threat to reliability.

Patterns: The domain experts were familiar with most of the selected patterns in this study. However, some experts asserted that particular patterns, such as *C2* and *Indirection Layer*, are not as well-known as the rest of the patterns. Moreover, two experts mentioned that *Master-Slave* is not frequently used in software architecture. The last row in Figure 7.3 shows the number of experts that were familiar with each pattern. Note, all twelve experts were familiar with well-known patterns, such as "Client-Server", "Layers", "SOA", "MVC", "Component-based", and "Microservices".

Quality Attributes: The domain experts were familiar with the reported quality attributes, i.e., the qualities in the ISO standard (see Figure 7.4). They mentioned that software architects mostly consider a limited set of quality attributes to evaluate real-world software systems. Furthermore, they asserted that some of the quality attributes in our list are semantically close to each other and can be combined. For instance, one of the experts asserted that terms such as "response time", "capacity", "latency", "throughput", and "execution speed" are linked to "Performance"; moreover, quality attributes such as "modifiability" and "stability" are connected to "Maintainability".

Based on the IEEE Standard Glossary of Software Engineering Terminology (Committee et al., 1998; Samadhiya et al., 2010), the quality of software products is the degree to which a system, component or process meets specified requirements (such as functionality, performance, security, and maintainability) and the extent to which a system, component or process meets the needs or expectations of a user. It is necessary to find quality attributes that are widely recommended by other researchers to measure the characteristics of the system.
The result of the SLR confirmed that researchers do not agree upon a set of conventional quality attributes (See Figure 7.4). Additionally, we realized that their suggested quality attributes were mainly applied to specific domains to address different research questions. Moreover, quality attributes such as "Security" and "Confidentiality", "Availability" and "Fault-tolerance", "Testability" and "Traceability", "Maintainability" and "Manageability", etc. can be considered as synonym terminologies. However, we observed that some authors distinguished and categorized quality attributes conceptually. For instance, Yang et al. (2012) stated that "Confidentiality", "Integrity", "Accountability", "Authenticity" are sub characteristics of "Security". Similarly, Bode & Riebisch (2010) stated that "Testability" and "Traceability" are sub characteristics of "Evolvability". Consequently, a set of nonexclusive and domain-independent quality attributes is needed to evaluate software products.
The ISO/IEC 25010 (ISO, 2011) presents best practice recommendations on the base of a quality assessment model. The quality model defines which quality characteristics should be considered when assessing the qualities of a software product. The key rationale behind using such software quality models is that they are a standardized way of measuring a software product (Haoues et al., 2017). In figure 7.4, the quality attributes printed in black are based on the ISO/IEC 25010 standard (ISO, 2011), and the rest of them (printed in blue) are not mentioned in the ISO standard[6].

Strength and Liabilities: The domain experts asserted that Figure 7.4 provides an extensive analysis regarding the impacts. They confirmed that such analysis is useful for software architects and can assist them with their decision-making process to select the best fitting set of patterns according to their quality concerns. The experts expressed that in real-world scenarios, software architects employ tactics to improve individual quality concerns. Tactics are mainly implemented in the source code so that their implementation can be easier or more difficult based on the nature of the system they are implemented in.

Application Domains The experts asserted that they had almost similar experiences with selecting and employing patterns in particular domains. One of the experts confirmed that some patterns are well-known candidates in particular domains, such as a combination of CQRS, Microservices, Layers, and Client-Server, which are all commonly used in ERP software. The practitioners stated that knowledge about application domains could be helpful for software architects and support them to identify

[6]The definitions of the quality attributes are entirely available in the technical report on the following web page: Http://swapslr.com

the initial set of patterns based on the similarity between their application domains and the observed domains based on other architects' experiences.
It is interesting to highlight that the knowledge regarding a limited set of patterns can lead to a cognitive bias (Montibeller & Winterfeldt, 2015) that forces practitioners an over-reliance on the patterns that they are familiar with. For instance, we noticed that some experts during the interviews had emphasized more on a particular set of patterns that they have mentioned as their expertise and skills in their LinkedIn profiles.

Combinations: The practitioners stated that in real-world architectures, they manipulate and combine patterns with meeting their requirements. Furthermore, they employ combinations of patterns besides software architecture tactics as architecture strategies to achieve particular quality attribute goals (e.g., improving security or performance). The practitioners confirmed that such knowledge about combinations are useful to them and can provide guidelines to select patterns and practical combinations. The practitioners also reconfirmed that pattern combinations can exist in many configurations. This presents a new challenge. For example, if a microservice uses CQRS independently, CQRS does not influence the total microservice architecture. However, if CQRS is used in an event-based architecture, those two patterns need to be developed in lock-step, as they influence each other heavily. For now, we recognize a dichotomy: combinations of patterns can be made that influence each other, while it is also possible to have combinations of patterns that do not influence each other at all. In future work, such relationships should be made explicit and specified in more detail.

Trends: The practitioners asserted that it is a well-known phenomenon that any technology is trend sensitive due to new insights and rapid advancements. Consequently, software architects have to be informed about the advancements in the technology industry and trends that can benefit their business in the future. Software architects sometimes have to select a particular set of patterns because of legacy technology choices. Sometimes vendor lock-in makes a customer dependent on a vendor for products and services, unable to use another vendor without substantial switching costs. An example of a pattern that has been trending in recent years is the *Microservices* pattern. *Microservices* advantages can tempt software architects to consider it as a hammer and convert every problem (design decision) into a nail. In other words, software architects tend to consider a set of patterns that are *trending*. For instance, one of the experts mentioned that software architects prefer to use *Publish-Subscribe* instead of *RPC* as a communication mechanism. Furthermore, *MVC*, as a pattern that facilitates the design of user-interfaces, is more popular than its alternatives, *C2* and *PAC*. In our research, we need to be cognizant of these trends, while not becoming dogmatic. In engineering, new tools have led to some of the greatest advances, and we expect the job of the software architect to remain an engineering job primarily for a long time.

7.5 Conclusion

Knowledge about architectural patterns is scattered among studies in the literature. In this study, we capture and aggregate knowledge about architectural patterns and make it available through this paper and a web site as reusable knowledge for architects. The amount of data collected from academic literature surpasses other studies in terms of a number of patterns studied and quality impacts identified. We also identify possible trends and application domains of architectural patterns.

The practitioners who participated in this research confirmed that the provided knowledge in this study could support researchers and practitioners with selecting the best fitting sets of architectural patterns for designing pattern-driven architecture according to their quality concerns and application domains.

The lack of sufficient knowledge regarding patterns and their impacts on quality attributes, plus their application domains in literature, impedes progress in the software architecture field and leads to unreliable decisions by software architects. This research serves several purposes. First, it is an explicit call to action for all architects and researchers to document their pattern usage, the quality attributes they meet, the tactics used to optimize those quality attributes, and the application domains they best apply. Second, we use this work as a source for designing more extensive decision support system (Farshidi et al., 2018c) that can support architects in finding the right combination of patterns for any software system. We plan to evaluate the decision support system in expert sessions with seasoned software architects. As the knowledge base of the decision support system also functions as a knowledge-sharing platform, it may become the first up to date and maintained pattern catalog.

CHAPTER 8

Decision Support for Pattern-Driven Architecture

The selection process of architectural patterns is challenging for software architects, as knowledge about patterns is scattered among a wide range of literature. Knowledge about architectural patterns must be collected, organized, stored, and quickly retrieved when it needs to be employed. In this tool paper, we introduce a decision support system that uses a decision model for supporting software architects with the pattern selection problem according to their requirements, including functional and quality requirements. The decision model is built based on a technology selection framework for modeling multi-criteria decision-making problems in software production. Twenty-four software architects in the Netherlands have evaluated the tool. They confirm that the tool supports them with their daily decision-making process.

keywords- Software architecture patterns; Pattern-driven software architecture; multi-criteria decision-making; decision support system; decision model;

This chapter is based on the following publication:

Farshidi, S. & S. Jansen (2020a), "A decision support system for pattern-driven software architecture", in: *Proceedings of the 14th European Conference on Software Architecture, ECSA 2020,* vol. 1, ACM, pp. 1–12.

8.1 Introduction

Software architecture is fundamental for the development of a software product and plays an indispensable role in its success or failure as software architecture deals with the base structure, subsystems, and interactions among these subsystems (Clements et al., 2003). Software architecture design can be viewed as a decision-making process: software engineers consider a set of alternative solutions that could solve a system design problem, and select the set that is evaluated as the optimal (Lago & Avgeriou, 2006).

Software architecture is the composition of a set of architectural design decisions, concerns, variation points, features, and usage scenarios that address various system requirements, including functional and quality requirements (Bosch, 2004). Each architectural design decision is made with a design rationale (Dutoit et al., 2007), which represents the knowledge that provides the answers to questions about the design decision or the process followed to make that decision.

An architectural pattern describes high-level structures and behaviors of software systems and addresses a particular recurring problem within a given context in software architecture design (Buschmann et al., 1996). Architectural patterns aim to satisfy several requirements and help to document the architectural design decisions (Avgeriou & Zdun, 2005). So that selecting architectural patterns is a subset of architectural design decisions (Zimmermann, 2010), and it is a challenging task for software architects, as knowledge about patterns, such as their application domains and their interactions with quality attributes, is scattered among a wide range of literature (Tang et al., 2011b). Thus, a decision support system (DSS) is needed to support software architects with architectural pattern selection intelligently.

In this article, we present a DSS for Pattern-Driven Architecture, which assists software architects in selecting the best fitting set of patterns. The DSS asks architects for their requirements in terms of functional requirements and quality concerns. Accordingly, several sets of architectural patterns are returned that match these requirements. Subsequently, architects can start tweaking the requirements to find the most suitable set of patterns for their design. The DSS is based on several well-known software engineering concepts, such as the ISO/IEC software quality models and the MoSCoW prioritization technique. Architects will indicate their preferences using primary selections such as 'The application *must have* high availability' and 'The application *could have* accessibility'. Using a literature study, we have assessed how patterns perform on these quality criteria. The DSS bundles this knowledge and provides architects with an interactive and collaborative decision tool.

We regard building a software architecture as a decision-making process (Rozanski & Woods, 2012): (1) Stakeholders with their requirements are engaged. (2) Scenarios are captured. (3) Architectural patterns are identified to address requirements. (4) Potential combinations of patterns are explored. (5) Architects evaluate the combinations of patterns (alternative solutions). If the alternative solutions do not meed the requirements, they are reworked and requirements revisited. (6) An architecture is drafted using the identified patterns (alternative solutions), viewpoints, and perspectives. (7) Different architectural alternatives for refining the draft are explored, and architectural decisions are made to select among them. (8) The architecture is evaluated with stakeholders. Finally, if the architecture does not fulfill stakeholder requirements, the architecture design is reworked and requirements possibly revisited (see Fig. 8.1). While this process has been a reliable method for producing architectures, it strongly depends on the architect's limited knowledge and experience, who may have experience with only a small number of patterns. Thus, we envision a process where the architect is supported by tools to enhance her knowledge of the patterns available for particular design problems.

Recently, we designed a framework (Farshidi et al., 2018c) for supporting software developers and architects (decision-makers) with their multi-criteria decision-making (MCDM) problems in software production. An MCDM problem deals with evaluating a set of alternatives and considers a set of decision criteria (Majumder, 2015). In this tool paper, we introduce a decision model, based on the framework, for the patterns selection problem. The DSS employed the decision model to support software architects with the pattern selection problem. Accordingly, we believe that the DSS can be used in steps (1-5) to facilitate the decision-making process for software architects (see Fig. 8.1).

The rest of this tool paper is organized as follows. Section 8.2 outlines a brief description of the DSS components and explains the constituent parts of the decision model. Section 8.3 presents the DSS and its application through a real-world example. Section 8.4 positions the DSS, among other tools and MCDM approaches, in the literature. Finally, Section 8.6 presents the evaluation of the DSS and summarizes this tool paper.

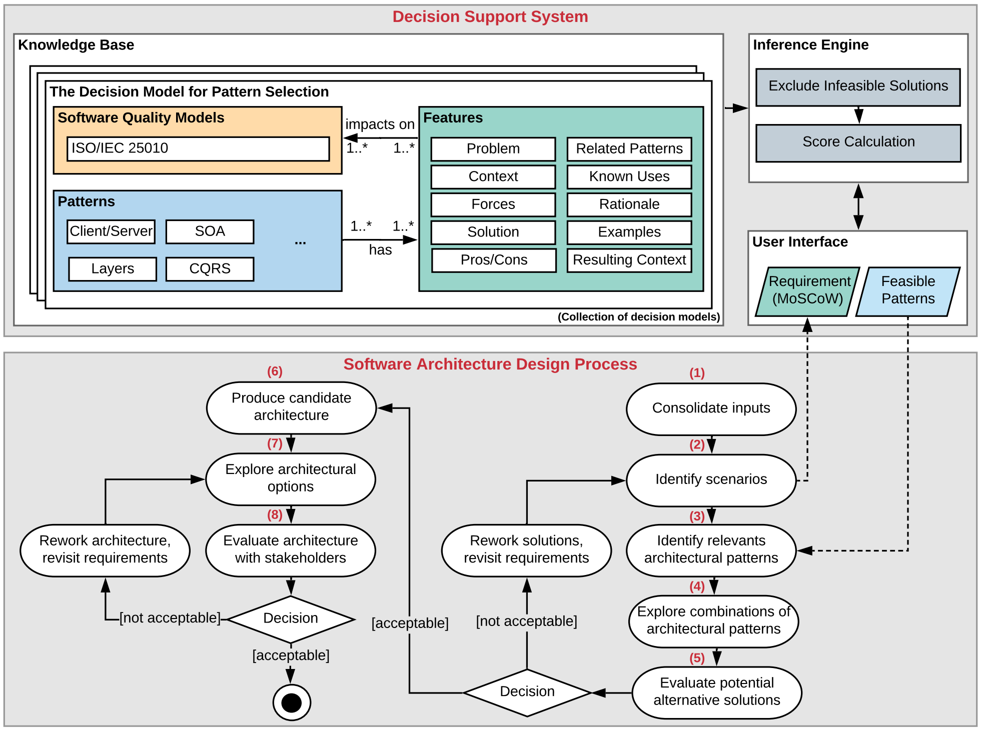
8.2 Decision Support System

A DSS is an information system that comprises domain-specific knowledge and decision models to assist decision-makers by offering knowledge about a set of alternatives (Wang, 1997). In this tool paper, the DSS integrates key aspects of knowledge-driven and model-driven DSSs (Power, 2008a) to store and organize the extracted knowledge regarding architectural patterns systematically facilitate the decision-making process. Note, for the sake of simplicity, we use *patterns* to refer "architectural patterns".

Additionally, we follow the framework (Farshidi et al., 2018c) for modeling decision problems in software production as MCDM problems. The framework applies the six-step decision-making process (Majumder, 2015) to build decision models for MCDM problems. The knowledge base of the DSS is a collection of decision models for different MCDM problems (Farshidi et al., 2018a; Farshidi et al., 2018b; Farshidi et al., 2020c; Farshidi et al., 2020e). According to the framework, the decision model for the pattern selection problem contains three sets (including Patterns, Software Quality Models, and Features) besides the mapping among the elements of these sets (see Fig. 8.1).

- Patterns: Patterns are the building blocks that, when assembled, can provide complete solutions for an architect's problem (see Fig. 8.1). Patterns have relationships to each other. For example, patterns can be alternatives to each other, for example, *Interpreter*, *Rule-Based System*, and *Virtual Machine* (Avgeriou & Zdun, 2005); Moreover, some patterns can also be complementary and combined. For instance, the

Figure 8.1: This figure shows that the DSS can be deployed in the the software architecture design process to support the software architects with the pattern selection problem (Farshidi et al., 2018a; Rozanski & Woods, 2012).


Decision Support System
Knowledge Base
The Decision Model for Pattern Selection
Software Quality Models
ISO/IEC 25010
impacts on
1..* 1..*
Features
Problem
Related Patterns
Context
Known Uses
Forces
Rationale
Solution
Examples
Pros/Cons
Resulting Context
Patterns
Client/Server
SOA
...
Layers
CQRS
1..* 1..*
has
(Collection of decision models)
Inference Engine
Exclude Infeasible Solutions
Score Calculation
User Interface
Requirement (MoSCoW)
Feasible Patterns
Software Architecture Design Process
(1) Consolidate inputs
(2) Identify scenarios
(3) Identify relevants architectural patterns
(4) Explore combinations of architectural patterns
(5) Evaluate potential alternative solutions
Decision
[not acceptable]
Rework solutions, revisit requirements
[acceptable]
(6) Produce candidate architecture
(7) Explore architectural options
(8) Evaluate architecture with stakeholders
Decision
[not acceptable]
Rework architecture, revisit requirements
[acceptable]

Client-Server pattern can be combined with the *Broker* pattern (Harrison & Avgeriou, 2010).
- Software Quality Models: A set of quality attributes, such as Availability and Security, should be defined in the decision model. We employed the *ISO/IEC 25010* standard (ISO, 2011) as a domain-independent quality model. The key rationale behind using this software quality model is that it is a standardized way of assessing a software product's quality. Moreover, it describes how easily and reliably a software product can be used.
- Features: Each pattern has a set of features, for instance, "centralized governance" is a feature of the "Client-Server". We identified the following types of features through a Systematic Literature Review (SLR) (Farshidi et al., 2020e). We reviewed *21,373* articles, and finally, *232* high-quality primary studies have been selected for performing the knowledge extraction process. Note, such feature types can be found in most patterns, even with different titles. (1: Problem) Descriptions of the problems indicating the intent in applying patterns. (2: Context) The preconditions under which patterns are applicable. (3: Forces) Descriptions of the allied forces and constraints. (4: Solution) Static structures and dynamic behaviors of patterns. (5: Resulting Context)

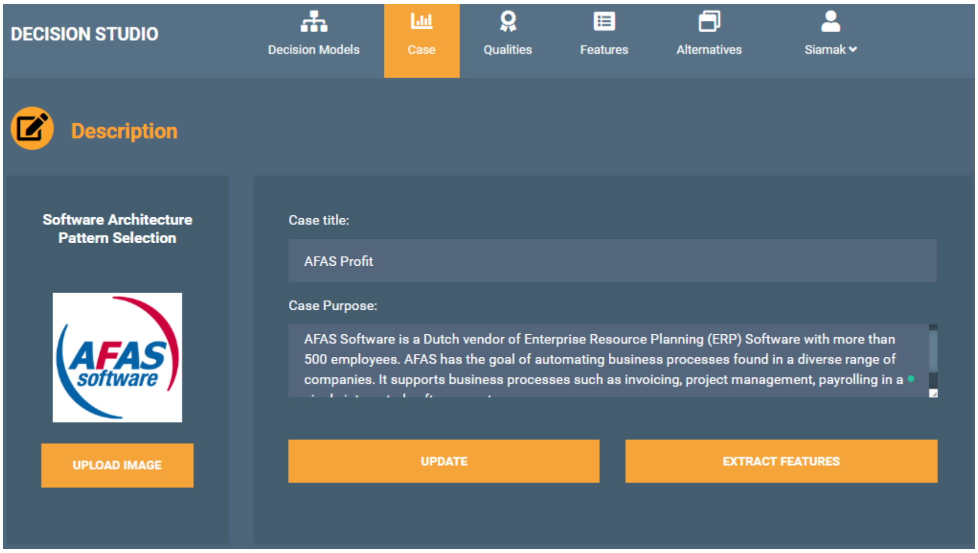
The post-conditions after a pattern has been applied. (6: Examples) Some sample applications of patterns. (7: Rationale) An explanation/justification of each pattern as a whole. (8: Related Patterns) The relationships among patterns. (9: Known Uses) Known applications of patterns within existing systems. (10: Pros/Cons) Pros and cons of employing patterns.
- Mappings: We identified the impacts of 29 patterns on 40 quality attributes based on a series of expert interviews with twelve senior software architects at different software producing organizations in the Netherlands (Farshidi et al., 2020e). Moreover, The mapping between the patterns and the features was investigated with the SLR and the experts.
Decision-Makers, such Software architects and developers, prioritize their requirements based on the MoSCoW prioritization technique (DSDM consortium and others, 2014), and then they send the requirements through the user interface of the DSS to the inference engine. Figure 8.3 shows the user interface of the DSS.
Inference engine: The DSS has an inference engine that receives inputs from the user interface. Next, it excludes all infeasible solutions, those that do not support "Must-Have" features or those that support "Won't-Have" features, and then it ranks the feasible solutions based on the number of "Should-Have" and "Could-Have" features that they support. In other words, requirements with *Must-Have* or *Won't-Have* priorities act as hard-constraints and requirements with *Should-Have* and *Could-Have* priorities act as soft-constraints. The inference engine assigns a non-negative score to each alternative solution based on the well-known Sum of Weights Method (Farshidi et al., 2018a), and finally, it returns a shortlist of feasible patterns (solutions) to the user interface.

8.3 A Practical Running Example

The DSS is accessible through the following link: (`https://dss-mcdm.com`). After login to the system, a software architect should select the "Software Architecture Pattern Selection" to create an instance of the decision model.

This section presents a real-world example of the pattern selection process. We asked a software architect at AFAS Software, a software producing organization in the Netherlands, to define their software architecture from a high-level of abstraction; then, we used the DSS and the decision model to capture the architect's concerns and requirements; next, the DSS generated a set of solutions accordingly.
Case Description - The software architect described AFAS software as follows: *AFAS Software is a Dutch vendor of Enterprise Resource Planning (ERP) Software with more than 500 employees. AFAS has the goal of automating business processes found in a diverse range of companies. It supports business processes such as invoicing, project management, payrolling in a single integrated software system. The current AFAS product, called AFAS Profit, is a traditional client-server application with a relational database for storing and retrieving customers' management data, such as business models and ontologies. AFAS Profit is a complete, integrated ERP system used by more than 10000 small and medium-sized enterprises. For example, Ernst & Young, Kwik-Fit, LeasePlan, Oad Reizen, Sandd, and Wibra, are already employing AFAS Profit to automate their business processes. Fig. 8.2 shows the description of the decision-making problem in terms of*

Figure 8.2: The architects describe their case in the context description screen. The tool uses text matching to automatically extract a subset of features from the description to get the architect started.



Software Architecture Pattern Selection

Case title:

AFAS Profit

Case Purpose:

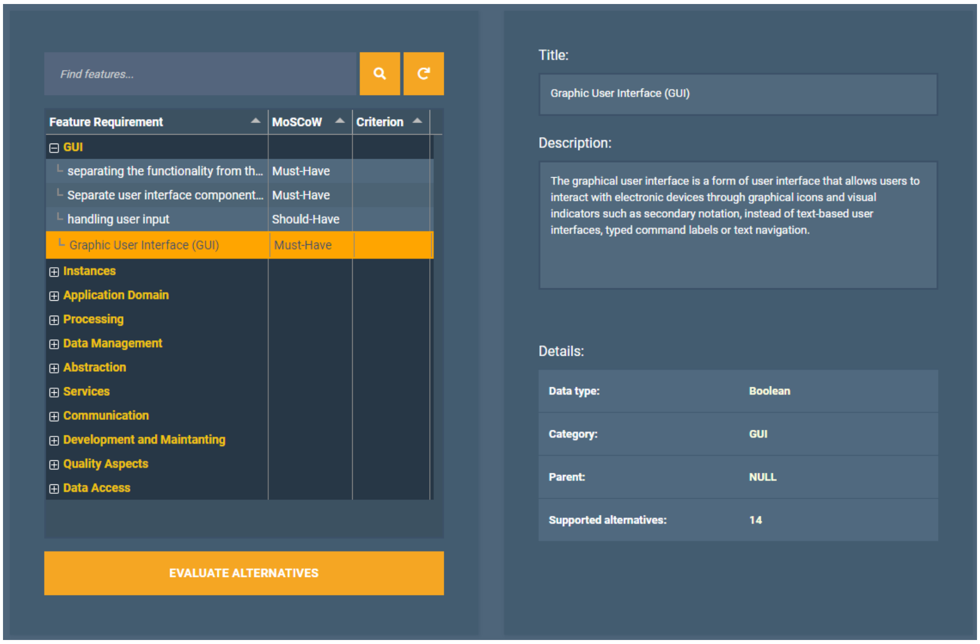
AFAS Software is a Dutch vendor of Enterprise Resource Planning (ERP) Software with more than 500 employees. AFAS has the goal of automating business processes found in a diverse range of companies. It supports business processes such as invoicing, project management, payrolling in a

the case title and description; moreover, the logo of the company can be attached to the "case description".

Case Definition - The software architect defined AFAS Profit as *a web-based solution that is consistent with the user experience of the windows client but feels web-native to customers. AFAS Profit is configurable by customers in their styling to match their logo and business style.* AFAS Profit has the following characteristics: (1) It is a combination of a client or frontend portion that interacts with the user and a server or back-end portion that interacts with the shared resource. The client process contains solution-specific logic and provides the interface between the user and the rest of the application system. The server process acts as a software engine that manages shared resources. (2) All data are centralized on a single server, simplifying security checks, including updates of data and software. (3) It supports a higher degree of flexibility and security, compared to the previous solution. (5) Its performance has increased significantly, compared to the previous solution, as tasks are shared between servers.

The architect stated that *"Functional Correctness", "Resource Utilization", "Configurability", "Accessibility", "Reliability", "Availability", and "Scalability" are the main quality concerns. Additionally, "technology agnostic", "modern web application", and "reusability of the business logic" are the key requirements of AFAS profit. Fig. 8.3 shows the "case definition" of AFAS Profit. The software architect assigned the MoSCoW priorities (Must-Have, Should-Have, Could-Have, and Won't-Have) to the requirements. Note, the data type the features can be either Boolean or Non-Boolean. For instance, "handling user input" is a Boolean feature, which means that a pattern either supports it or*

Figure 8.3: represents how a decision-maker can define the requirements based on the MoSCoW prioritization technique.



Find features...

Feature Requirement	MoSCoW	Criterion
⊟ GUI		
└ separating the functionality from th...	Must-Have	
└ Separate user interface component...	Must-Have	
└ handling user input	Should-Have	
└ Graphic User Interface (GUI)	Must-Have	
⊞ Instances		
⊞ Application Domain		
⊞ Processing		
⊞ Data Management		
⊞ Abstraction		
⊞ Services		
⊞ Communication		
⊞ Development and Maintanting		
⊞ Quality Aspects		
⊞ Data Access		

EVALUATE ALTERNATIVES

Title:

Graphic User Interface (GUI)

Description:

The graphical user interface is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, instead of text-based user interfaces, typed command labels or text navigation.

Details:

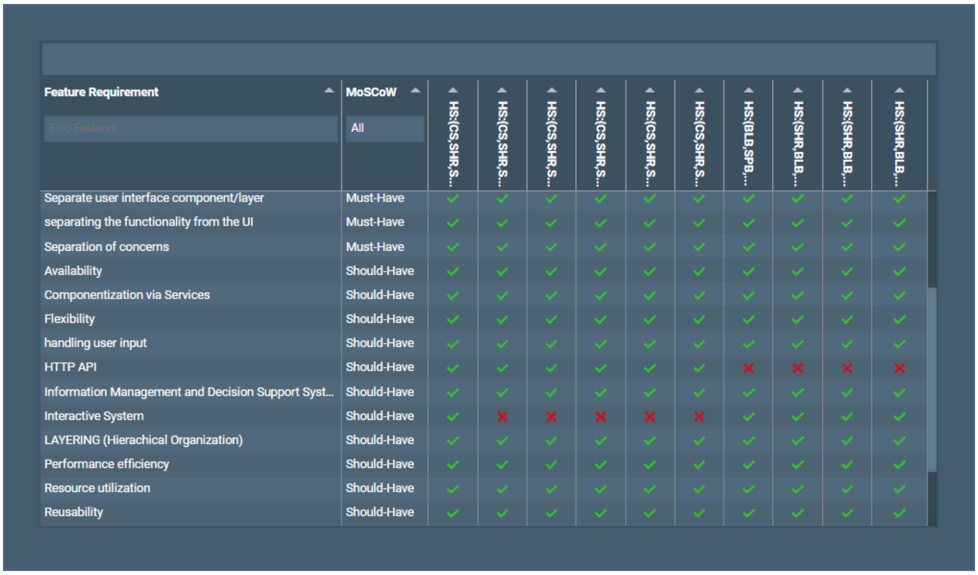
Data type:	Boolean
Category:	GUI
Parent:	NULL
Supported alternatives:	14

not. However, the level of support of "Availability" or "Scalability", as two Non-Boolean features, of a pattern can be "High", "Medium", or "Low".

Case Evaluation - The software architect stated that *AFAS Profit architecture is based on a combination of the "Client-Server", "Publish-Subscribe", and "Layers" patterns. The main rationales behind these design decisions are (1) the frontend can be easily replaced or upgraded, and every module of the business logic, in the back-end, can be reused. (2) The web client communicates over HTTP with the server, so it is possible to choose different technologies for the web client. (3) They can implement a Content Management System (CMS) to make the web client configurable in style and layout. (4) While the data is requested through communication with the server, preventing stale data, the CMS parts are published with some delay, making it possible to cache the style and layout for fast retrieval.*

The inference engine gets the requirements and evaluates the alternative patterns in its knowledge base accordingly. As each pattern supports only a limited set of features, the inference engine has to generate feasible solutions (combinations of patterns). Note, finding a subset of patterns that support all hard-constraints can be formulated as the set cover problem. The DSS uses an algorithm based on the set cover problem to generate several feasible solutions when all patterns in its knowledge base do not support the entire list of hard-constraints of a decision-maker. For instance, Fig. 8.4 shows that the DSS could not find any patterns that address all the

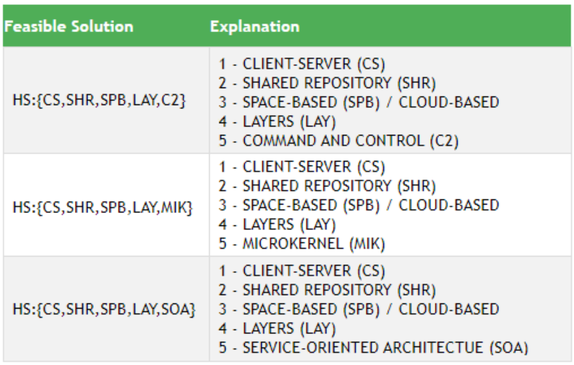
Figure 8.4: illustrates part of the case evaluation by the DSS. Ticks (✓) in a row signify that the feature is supported by the corresponding patterns, and crosses (✗) symbolize that the patterns do not support the feature.



Feature Requirement	MoSCoW	HS:{CS,SHR,S…	HS:{CS,SHR,S…	HS:{CS,SHR,S…	HS:{CS,SHR,S…	HS:{CS,SHR,S…	HS:{CS,SHR,S…	HS:{BLB,SPB,…	HS:{SHR,BLB,…	HS:{SHR,BLB,…	HS:{SHR,BLB,…
Find Features...	All										
Separate user interface component/layer	Must-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
separating the functionality from the UI	Must-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Separation of concerns	Must-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Availability	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Componentization via Services	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Flexibility	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
handling user input	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HTTP API	Should-Have	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗
Information Management and Decision Support Syst…	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Interactive System	Should-Have	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓
LAYERING (Hierachical Organization)	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Performance efficiency	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Resource utilization	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Reusability	Should-Have	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

AFAS Profit requirements so that it generated a set of solutions consist of multiple patterns.

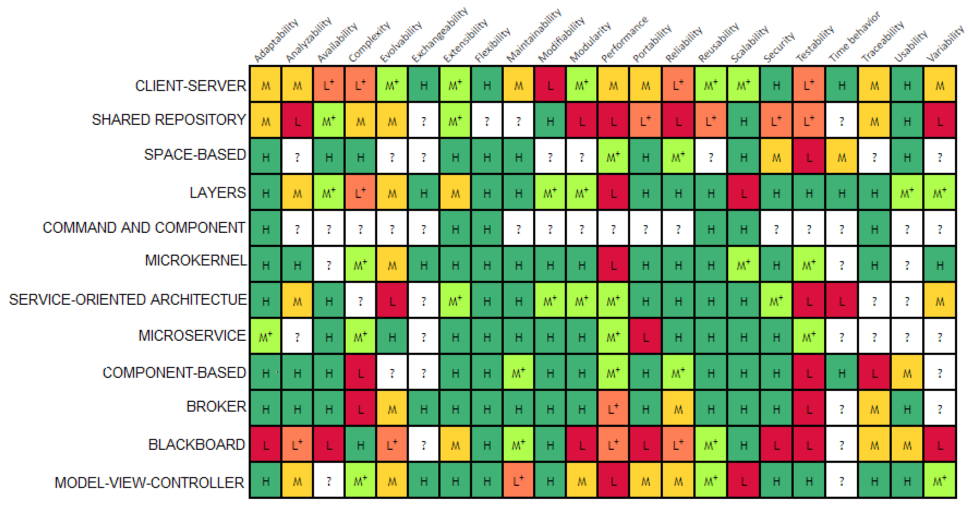
Figure 8.5: shows top-3 solutions for AFAS Profit.



Feasible Solution	Explanation
HS:{CS,SHR,SPB,LAY,C2}	1 - CLIENT-SERVER (CS) 2 - SHARED REPOSITORY (SHR) 3 - SPACE-BASED (SPB) / CLOUD-BASED 4 - LAYERS (LAY) 5 - COMMAND AND CONTROL (C2)
HS:{CS,SHR,SPB,LAY,MIK}	1 - CLIENT-SERVER (CS) 2 - SHARED REPOSITORY (SHR) 3 - SPACE-BASED (SPB) / CLOUD-BASED 4 - LAYERS (LAY) 5 - MICROKERNEL (MIK)
HS:{CS,SHR,SPB,LAY,SOA}	1 - CLIENT-SERVER (CS) 2 - SHARED REPOSITORY (SHR) 3 - SPACE-BASED (SPB) / CLOUD-BASED 4 - LAYERS (LAY) 5 - SERVICE-ORIENTED ARCHITECTUE (SOA)

Patterns tend to be combined to provide greater support for the reusability during the software design process (That et al., 2013). A pattern can be blended with, connected to, or included in another pattern. For instance, the *Broker* pattern can be connected to the *Client-Server* pattern to form the combined *Client-Server-Broker* pattern (Harrison & Avgeriou, 2010). Fig. 8.5 shows top-3 solutions for AFAS profit. The solutions support all requirements with Must-Have priorities and do not support Won't have requirements (hard-constraints). Note, the DSS generated almost similar solutions that the experienced software architects at AFAS came up with. Note that the DSS sorts its suggestions based on their scores so that top-3 solutions can be considered the most valuable suggestions.

Figure 8.6: show a subset of the mapping between features and patterns used by the DSS to generate solutions for AFAS profit. The primary source of knowledge to build this mapping is the SLR. We employed Fuzzy logic to gain some agreement among the selected studies to calculate the values (Farshidi et al., 2020e). Note: High (H), Medium (M), Low (L), Unknown (?).

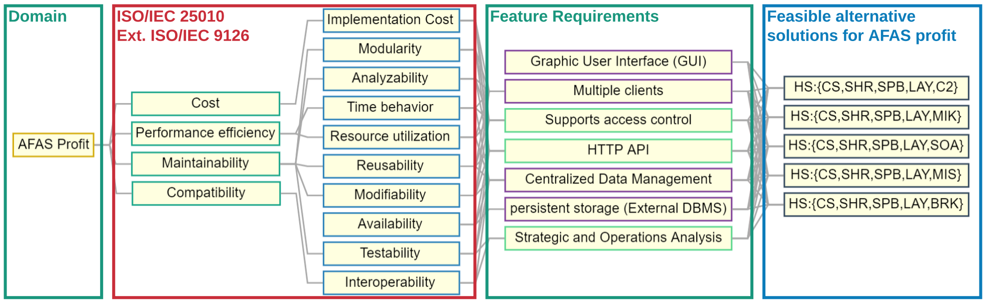


	Adaptability	Analyzability	Availability	Complexity	Evolvability	Exchangeability	Extensibility	Flexibility	Maintainability	Modifiability	Modularity	Performance	Portability	Reliability	Reusability	Scalability	Security	Testability	Time behavior	Traceability	Usability	Variability
CLIENT-SERVER	M	M	L^+	L^+	M^+	H	M^+	H	M	L	M^+	M	M	L^+	M^+	M^+	H	L^+	H	M	H	M
SHARED REPOSITORY	M	L	M^+	M	M	?	M^+	?	?	H	L	L	L^+	L	L^+	H	L^+	L^+	?	M	H	L
SPACE-BASED	H	?	H	H	?	?	H	H	H	?	?	M^+	H	M^+	?	H	M	L	M	?	H	?
LAYERS	H	M	M^+	L^+	M	H	M	H	H	M^+	M^+	L	H	H	H	L	H	H	H	H	M^+	M^+
COMMAND AND COMPONENT	H	?	?	?	?	?	H	H	?	?	?	?	?	?	H	H	?	?	?	H	?	?
MICROKERNEL	H	H	?	M^+	M	H	H	H	H	H	H	L	H	H	H	M^+	H	M^+	?	H	?	H
SERVICE-ORIENTED ARCHITECTUE	H	M	H	?	L	?	M^+	H	H	M^+	M^+	M^+	H	H	H	H	M^+	L	L	?	?	M
MICROSERVICE	M^+	?	H	M^+	H	?	H	H	H	H	H	M^+	L	H	H	H	H	M^+	?	?	?	?
COMPONENT-BASED	H	H	H	L	?	?	H	H	M^+	H	H	M^+	H	M^+	H	H	H	L	H	L	M	?
BROKER	H	H	H	L	M	H	H	H	H	H	H	L^+	H	M	H	H	H	L	?	M	H	?
BLACKBOARD	L	L^+	L	H	L^+	?	M	H	M^+	H	L	L^+	L	L^+	M^+	H	L	L	?	M	M	L
MODEL-VIEW-CONTROLLER	H	M	?	M^+	M	H	H	H	L^+	H	M	L	M	M	M^+	L	H	H	?	H	H	M^+

The DSS Reports - In the knowledge extraction phase for building the decision model, we observed multiple inconsistencies regarding the impacts of patterns on quality attributes. Some studies reported adverse impacts of a particular pattern on a quality attribute. For instance, *efficiency* can be considered as both strength and liability of the *Pipes and Filters* pattern. We applied fuzzy logic to aggregate the extracted knowledge regarding the potential impacts of patterns on quality attributes. In the implementation of the score calculation (trade-off) phase of the DSS, the impact values range from -2 to 2+. Accordingly, the patterns with more liabilities score lower than those that have more strengths. Note, quantifying the impact of a particular pattern on the quality attributes is complicated because quality attributes are system-wide capabilities. Generally, they cannot be evaluated entirely until the whole system can be evaluated. The DSS evaluates alternative solutions according to decision-makers' quality concerns. Fig 8.6 shows the impacts of the single solutions for AFAS profit on a subset of quality attributes.

Fig. 8.7 illustrates a decision structure based on AFAS profit requirements. The

Figure 8.7: shows part of the decision structure for the AFAS profit that was generated by the DSS. The domain of the decision-making process is "Finding the best fitting set of patterns for AFAS profit". The qualities are based on the ISO/IEC 25010 (ISO, 2011) quality model. The software architect (decision-maker) defined the feature requirements. The DSS suggested feasible alternative solutions for AFAS profit (last level). Note, the mapping between the qualities and the features was based on domain experts' knowledge; moreover, the relationships among features and patterns were determined based on the SLR (Farshidi et al., 2020e).



DSS automatically generates such decision structures according to the requirements of decision-makers. The first level of the decision structure (Domain) indicates the goal of the decision-making process. The second level denotes the relevant quality attributes that impact the prioritized requirements, which are signified in the third level (requirements). The last level (Feasible Solutions) shows a list of feasible patterns for the decision domain.

8.4 Related Work

In the SLR (Farshidi et al., 2020e), we reviewed selected *232* high-quality primary studies for performing the knowledge extraction process. The knowledge base of the SLR, including the primary studies and extracted knowledge, is available as a technical report on the following web page: http://swapslr.com. We realized that researchers introduced a variety of tools and MCDM techniques to address the pattern selection problem. Notably, there are few tools available for software architects. Architecting is a knowledge-intensive practice, so it can be hard to find the best way to support architects with the right knowledge at the right time. A subset of tools for supporting software architects with their design decisions are presented as follows: *Archium* (www.archium.io) is a visualization tool that produces a view on the functional dependencies between architectural design decisions. It is not an automatic pattern detection or selection, but visualizing the dependencies can help software architects identify such patterns. *ArchReco* (www.cs.ucy.ac.cy/ sielis) provides a design environment that software architects can draw diagrams with pre-defined shapes that exist in a palette. The description of the shapes is part of a contextual element set that ArchReco's processes suggest the most suitable context-based recommended design patterns. Such Design Patterns are retrieved from several data sources and filtered according to the contextual information that is processed when software architects request recommendations.*Sirius* (www.obeodesigner.com/en/product/sirius) is a tool that enables software architects to graphically design complex systems while keeping the corresponding data consistent (architecture, component properties, etc.). *AKB* (www.se.jku.at/akb-knowledge-sharing) is an implementation and extension of the Architecture Haiku concept, a one-page design description. AKB supports software architects with capturing and sharing of architectural knowledge based on architecture profiles.

The DSS enables software architects to document their drawings and design rationales. We implemented a design studio based on the Unified Modeling Language concepts to store design decisions while the decision-making process. The main difference between the DSS and such tools is that it supports software architects with their decision-making process. In other words, the DSS provides a discussion and negotiation platform to enable software architects to make group decisions. Furthermore, the DSS can be used over the full life-cycle and can co-evolve its advice based on evolving requirements. Software architects can prioritize their functional requirements and quality concerns using the MoSCoW prioritization technique through the user interface of the DSS. Then, the DSS generates a set of feasible solutions that address the requirements.

In contrast to existing MCDM approaches in the literature (see (Farshidi et al., 2020e)), the cost of creating, evaluating, and applying the decision model is not penalized exponentially by the number of criteria and alternatives, because it is an evolvable and expandable approach that splits down the decision-making process into four maintainable phases (Farshidi et al., 2018c). The decision model addresses main knowledge management issues, including capturing, sharing, and maintaining knowledge.

Recently, we built five more decision models based on the framework for the selection process of database Management Systems (Farshidi et al., 2018c), Cloud-Service Providers (Farshidi et al., 2018a), Blockchain Platforms (Farshidi et al., 2020c), Model-Driven Software Development Platforms, and Programming Languages. Case studies and expert interviews were conducted to evaluate the DSS's effectiveness and usefulness in addressing these MCDM problems. The results confirmed that the DSS performed well to solve the mentioned problems in software production.

8.5 Evaluation

We carried out a study with 24 software architects and developers in the Netherlands to assess the user acceptance of the decision support system and the decision model based on the Technology Acceptance Model. Firstly, we formed 12 groups of two individuals according to their expertise and the companies that they were working with. Next, we introduced the decision model within the DSS portal and presented some of its applications. Then, we assigned the problem definitions of two real-world software architectures to the groups and asked them to design two solutions for the problems. The groups used the decision model within the DSS platform to help them with (1) defining the requirements based on the MoSCoW prioritization technique, and (2) finding the best fitting set of patterns. The group sessions lasted between

45 to 60 minutes. At the end of the sessions, we ask all of the participants to fill out a TAM-based questionnaire; Next, we collected their feedback and opinion about the decision model. The participants highlighted that the decision model, in terms of reusable knowledge regarding the patterns, was a useful tool that can support them to explore more patterns while designing real-world software architectures. They asserted that the decision model assists them in finding liabilities and strength of patterns, their features, and potential application domains that they have employed in.

The DSS assists software architects in the requirements elicitation activity by offering a list of essential features of patterns. Moreover, software architects have different perspectives on their requirements in different phases of the Software Development Life-Cycle. They might want to consider generic domain features in the early phases of the life-cycle, whereas they are interested in more technical features as their development process matures. Therefore, the DSS might come up with various solutions for a software architect in different phases of its software development life-cycle. As the choices of a decision-maker are stored in the DSS knowledge base, it does not cost a significant amount of time to rerun the decision-making process. In a typical scenario, an architect will tweak her decisions and values to assess her choices have on the desired set of patterns. Software architects sometimes have to select a particular set of patterns because of legacy technology choices. Sometimes vendor lock-in makes a customer dependent on a vendor for products and services, unable to use another vendor without substantial switching costs. An example of a pattern that has been trending in recent years is the *Microservices* pattern (see (Farshidi et al., 2020e)). *Microservices* advantages can tempt architects to consider it as a hammer and convert every design decision into a nail.

Patterns and quality attributes are not independent and have significant interaction with each other. Such interactions can be observed as trade-offs between quality attributes. Software architects need to select and employ an optimal set of patterns to satisfy quality concerns. For instance, some studies assert that *Reusability* is a strength and *Scalability* is a liability of the *Layers* pattern (see (Farshidi et al., 2020e)). If an architect is looking for both qualities, she has two options: choose another (set of) pattern(s) or use *tactics* to improve *Scalability*. System quality is best exposed in production, independent of whether system quality has been made explicit. We recall that well-known authors, such as Wiegers & Beatty (2013), classify quality attributes as external (exposed at the run time/in production, e.g., performance) and internal (exposed at design time, e.g., modifiability). If architects do not think about performance, the system will still expose its performance in the field. The knowledge around the quality of a system under design is hard to gather without *in the field* experiences; however, experience with similar patterns in other systems provides invaluable insight into the inherent qualities of a new system. The DSS recommends patterns that exhibit similar quality behaviors when purely implemented (without tactics) in different systems and that this knowledge can be used by architects to make informed design decisions. We consider it future work to further explore these relationships between patterns and the way in which these communicating properties are best communicated to architects, having to choose from a set of complex solutions.

The tool has been designed using the .Net framework. While it has been optimized somewhat, the tool will sometimes still perform slowly, with end-user wait times of around 5 seconds, which is workable, but not ideal. One of the challenges is the solution space: for recommending solutions (combinations of patterns), the problem's search space is huge, consisting of 29 patterns and 188 features. For instance, for a solution with three patterns, the problem's search space is found to contain $\sim 29 \times 28 \times 27 \times 188$ possible problem states.

8.6 Conclusion

In this tool paper, we present a DSS besides a decision model for architectural pattern selection. The DSS suggests feasible patterns for particular cases based on the quality concerns and functional requirements of decision-makers. The DSS[1] is accessible through the following link: (https://dss-mcdm.com). We consider it future work to ensure that the knowledge base remains up to date, for instance, through a wiki-mechanism. Thus, software architects can consider the DSS as a source of knowledge and reliable assistance while making decisions regarding the best-fitting set of patterns for their software architectures. Additionally, we should enhance the DSS with a learning module that improves its learnability aspect in the future.

It is presently impossible to assess which patterns are compatible and frequently used in combination, even though practically all systems implement more than one pattern. The knowledge base of the DSS contains individual patterns that solve particular parts of a design problem. The inference engine uses an algorithm based on the set cover problem to generate several feasible solutions when all patterns in its knowledge base do not support the entire list of hard-constraints of a decision-maker.

In our studies, we have dealt with different kinds of architectures, with a slight bias towards enterprise resource planning systems. We consider it as future work to apply the tool to problems in other domains, such as Internet of Things, gaming, or media systems.

[1]Please watch a demo video of the DSS through this link: `https://youtu.be/AhfGYpwpJSQ`

CHAPTER 9

Design Decisions in Pattern-Driven Architecture

Context: Software architecture design is a decision-making process. Software architects make design decisions continuously, based on their design rationales and tacit knowledge, while designing software architectures. A subclass of design decisions is selecting architectural patterns, which is a challenging process, as knowledge about them is fragmented over a wide range of various studies. Hence, a decision model is required to analyze architectural patterns using systematic identification and evaluation of potential alternatives. *Method:* We have developed a decision model and tool to support software architects in architectural pattern selection using the Technology Acceptance Model. We have evaluated potential adoption with 24 software architects in the Netherlands. *Results:* The decision model was perceived as useful for architectural pattern selection, and the participants also described it as a form of knowledge that they needed in their work. Additionally, we received constructive feedback to improve the tool. *Conclusion:* Having reusable knowledge of the decision model readily available supports software architects in making design decisions that meet their requirements and priorities. Additionally, the lessons learned can be employed by other researchers to set up their studies, to evaluate their decision support tools, and to gain feedback from software architects.

keywords- Architectural Patterns; Design Decisions; Decision Model; Decision Support System; Technology Acceptance Model

This chapter is based on the following publication:

Farshidi, S. & S. Jansen (2020b), "Evaluating architect adoption of a decision support tool", *(Submitted)*.

9.1 Introduction

Decision-making is an inevitable portion of software development, and a considerable number of decisions are made during the software development life cycle regarding processes, products, tools, methods, and techniques (Ruhe, 2002). Software architecture is the composition of a set of architectural design decisions, concerns, variation points, features, and usage scenarios that address various system requirements, including functional and quality requirements (Bosch, 2004). Design decisions are concerned with the system's application domain, architectural patterns employed in the system, or other infrastructure selections as well as other aspects needed to satisfy all requirements (Bosch, 2004). Each architectural design decision is made with a design rationale (Dutoit et al., 2007), which represents the knowledge that provides the answers to questions about the design decision or the process followed to make that decision. Software architecture decisions are made in the early stage of the software development life cycle and have a significant impact on shaping the analysis of the problem and the expression of the design (Shaw, 1995). Software architects make decisions that have long-lasting impacts on quality attributes of a software-intensive system (Kruchten, 2008).

An architectural pattern describes high-level structures and behaviors of software systems and addresses a recurring problem within a given context in software architecture design (Buschmann et al., 1996). Architectural patterns aim to satisfy several functional and quality attribute requirements and help document architectural design decisions (Avgeriou & Zdun, 2005). Selecting architectural patterns is a subset of architectural design decisions (Zimmermann, 2010), and it is a challenging task for software architects, as knowledge about patterns is scattered among a wide range of literature (Farshidi et al., 2020e; Me et al., 2016). Note, for the sake of brevity, we use *patterns* to refer to *architectural patterns*.

Knowledge regarding patterns has to be collected, organized, stored and quickly retrieved when it needs to be employed. There exists a need for a decision model to support software architects in selecting suitable patterns according to their requirements. The decision-making process in software architecture can be modeled as a Multi-Criteria Decision Making (MCDM) technique that provides more formal and quantitative reasoning. Finding the best fitting solution for a particular subset of design decisions can also be modeled as an MCDM problem that deals with evaluating a set of alternatives and takes into account a set of decisions criteria (Triantaphyllou et al., 1998). Recently, we introduced a framework (Farshidi et al., 2018a) to build decision models for MCDM problems in software production. Moreover, we designed and implemented a Decision Support System (DSS) (Farshidi et al., 2018b) for supporting software architects with their such MCDM problems.

In this study, we follow the framework (Farshidi et al., 2018a) to build a decision model for the pattern selection problem. Next, we carried out a study with 24 software practitioners in the Netherlands to assess the decision model within DSS's knowledge base in addressing the decision-making problem. This study is structured as follows: Section 9.2 describes multiple research methods that we have combined to achieve a fuller picture and a more in-depth understanding of patterns by connecting complementary findings that conclude from the use of methods from the different methodological traditions of qualitative and quantitative investigations. Moreover, it highlights barriers to the knowledge acquisition and decision-making process and argues how we have minimized the threats to the validity of the results. Section 9.3 formulates the pattern selecting as an MCDM problem, afterward, it outlines the main building blocks of the decision model; Additionally, Section 9.4 presents the research setting and estimates background knowledge of the software practitioners on patterns. Furthermore, it assesses the user acceptance of the decision model based on qualitative analysis of the Technology Acceptance Model (TAM). Section 9.5 analyzes the results and discusses lessons learned from the study. Section 9.6 positions our work among the other pattern selection techniques in the literature. Finally, Section 9.7 summarizes the proposed approach, defends its novelty, and offers directions for future studies.

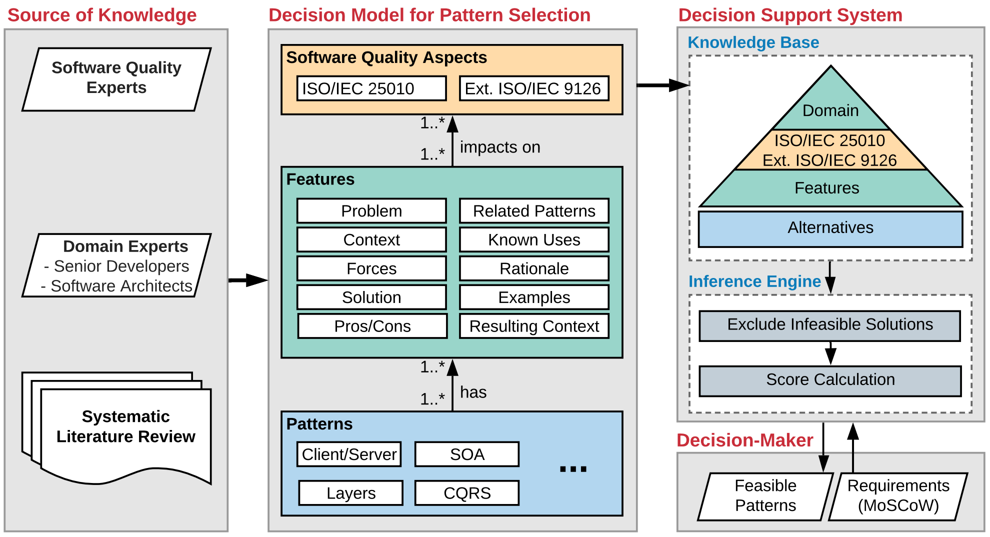
9.2 Research method

We regard building a software architecture as a decision-making process (Lago & Avgeriou, 2006): a software architect considers several alternative solutions that could solve the design problem, and subsequently chooses one of the solutions that optimally addresses the problem. The software architecture design decision, such as the selection of patterns, is formulated as follows (Farshidi et al., 2020e). (1) a software architect runs into a design problem; (2) she looks for actual features she thinks can solve this problem, such as "distribute data over multiple servers";(3) she goes through the description of several patterns and identifies several candidates; (4) she identifies an optimum pattern for her problem and applies tactics to make sure it works in the context. We assume that the proposed decision model in this study can be employed in steps (2) and (3) to facilitate the decision-making process for software architects (see Fig. 9.1). To guide our research, we defined the following research questions: (RQ_1) How can we transform architectural knowledge regarding patterns into a decision model to support software architects with the pattern selection process? (RQ_2) Do software architects find the decision model useful and easy to use in the pattern selection process? (RQ_3) How do software architects use the decision model within the DSS knowledge base to find the best fitting set of patterns?

The framework (Farshidi et al., 2018a), which we use in this study, provides a guideline for software architects to build decision models for MCDM problems in software production following six-step of the decision-making process (Majumder, 2015). Fig. 9.1 represents the main building blocks of the DSS besides the proposed decision model. In this study, we considered the following knowledge sources to answer the first research question and build and evaluate a decision model for the pattern selection problem.

1) Structured Literature Review (SLR): The decision model for the pattern selection problem requires knowledge regarding patterns in terms of potential alternatives (candidate solutions) and their features. As architectural knowledge about patterns is scattered among a wide range of literature (Farshidi et al., 2020e; Me et al., 2016), we conducted an SLR, following the steps and guidelines of Kitchenham (2004), to capture this architectural knowledge systematically. The SLR functioned as a knowl-

Figure 9.1: shows the decision model for the pattern selection problem. Moreover, it represents the main building blocks of the DSS that is adapted from our previous study (Farshidi et al., 2018a). Note, the DSS and the decision model is accessible through the following link: `https://dss-mcdm.com`



edge acquisition process to capture knowledge regarding a potential list of patterns and their features. Eventually, *232* high-quality primary studies were selected for performing the knowledge extraction process (Farshidi et al., 2020e).

2) Expert Interview: We followed the Myers and Newman guidelines (Myers & Newman, 2007) to conduct a series of qualitative semi-structured interviews with twelve senior software architects, including freelancers and employees of different software producing organizations, to explore expert knowledge regarding architectural patterns and build the decision model. We developed a role description before contacting potential experts in order to ensure the right target group. We contacted 43 architects in the Netherlands through email using the role description and information about our research topic. Overall, twelve senior software architects in the Netherlands participated in this research. The experts were pragmatically and conveniently selected according to their expertise and experience that they mentioned on their *LinkedIn* profile. The experts had, on average, more than ten years of experience with designing software architectures. Each of the interviews followed a semi-structured interview protocol and lasted between 60 and 90 minutes.

We used open questions to elicit as much information as possible from the experts minimizing prior bias. All interviews were done in person and recorded with the interviewees' permission; The interviews were coded for further analysis to decrease a threat to construct validity. In order to mitigate a possible threat to internal validity, we consider a set of expert evaluation criteria (including "Years of experience", "Expertise", "Skills", "Education", and "Level of expertise") to select the experts. The

relatively small number of interviewees for this study highlights the issue of generalization and the external validity of the research results. However, the diversity of the interviewees, who were working at or with a variety of software development companies, led to unbiased and generalized results. The interview protocol and coding were reviewed by two authors of this paper to minimize a threat to reliability.
3) Evaluation with Practitioners: To answer the last two research questions, we conducted a study with a group of software practitioners, including 24 software architects and developers in the Netherlands. The software practitioners were selected according to their expertise and years of experience. On average, the participants had more than seven years of experience designing and developing real-world software-intensive systems. We asked them to use the decision model through the DSS to design two real-world software architecture. Finally, we assessed the user acceptance of the decision model based on qualitative analysis of the TAM (Davis, 1989).

Biases, such as motivational and cognitive (Montibeller & Winterfeldt, 2015), arise because of shortcuts or heuristics that software architects use to solve problems and perform tasks. The Hawthorne effect (Jones, 1992), which is the tendency for software architects to change their behavior when they are being observed, is a form of cognitive bias. The participants might have been more careful in the observational setting than they would be in the real setting because they are being observed by scientists judging their assessment questionnaires. Moreover, the Bandwagon effect (Nadeau et al., 1993), which is the tendency to do or believe things because many other software architects do or believe the same, is another form of cognitive bias. The Bandwagon effect typically shows up in group decisions. To mitigate the Hawthorne and Bandwagon effects, we formed 12 groups of two individuals according to their expertise and the companies that they were working with. In order to keep the balance between the groups, the teammates, including one senior practitioner beside one junior practitioner, were selected from different software companies.

9.3 Architectural pattern selection as an MCDM

In this study, we formulated the pattern selection problem as an MCDM problem. Let $Patterns = \{p_1, p_2, \dots p_{|Patterns|}\}$ be a set of patterns (i.e., "SOA", "Layers", and "CQRS"). Moreover, $Features = \{f_1, f_2, \dots f_{|Features|}\}$ be a set of features (i.e., known uses, context, and forces) of the patterns, and each pattern p, where $p \in Patterns$, has a subset of the $Features$. The goal is finding the suitable pattern p which meets a set of requirements ($Requirements$), where $Requirements \subseteq Features$. In other words, the pattern p is a feasible pattern (or a combination of patterns) that addresses the requirements and concerns of the software architects. For instance, when a software architect is looking for a pattern, or a set of patterns, to build a highly scalable distributed system, she can employ "SOA" as an alternative solution, as it supports "decentralized data management" and "decentralized governance", moreover, "Scalability" is a strength of this pattern (Farshidi et al., 2020e).

We follow the framework (Farshidi et al., 2018a) as a guideline to build the decision model, so the following building blocks should be defined:
- Patterns: They are universal and reusable solutions to commonly occurring problems in software architecture. In literature, sometimes the terms "architectural

patterns" and "architectural styles" are used interchangeably, since they are, in principle, the same concepts and only differ in their description forms (Avgeriou & Zdun, 2005). *Design patterns* are experience-based approved solutions employed by developers to solve common problems when implementing a software system (Hussain et al., 2017). Patterns are comparable to design patterns but have a more extensive scope. After reviewing the primary studies in the SLR, we identified 29 patterns mentioned in at least three studies. Table 9.1 shows the complete list of the patterns that we have considered in the decision model.
- Features: Several different compositions are used in the literature for defining patterns, and no single composition has achieved widespread acceptance (Haren, 2011). However, there is a comprehensive agreement on the types of features that a pattern should contain (Buschmann et al., 1996). During the SLR, we tried to identify the following types of features. Note, such feature types can be found in most patterns, even with different titles. (1: Problem) Descriptions of the problems indicating the intent in applying patterns. (2: Context) The preconditions under which patterns are applicable. (3: Forces) Descriptions of the allied forces and constraints. (4: Solution) Static structures and dynamic behaviors of patterns. (5: Resulting Context) The post-conditions after a pattern has been applied. (6: Examples) Some sample applications of patterns. (7: Rationale) An explanation/justification of each pattern as a whole. (8: Related Patterns) The relationships among patterns. (9: Known Uses) Known applications of patterns within existing systems. (10: Pros/Cons) Advantages and disadvantages of employing patterns. In this study, we have identified 188 features[1] regarding the patterns. Table 9.1 shows a subset of the features that we have considered in the decision model.

- The mapping between features and patterns: Patterns are described based on the functionality they deliver, besides their strengths or liabilities are shown concerning several quality attributes (Me et al., 2016). A strength or liability is an argument to employ or to avoid a pattern in a particular situation. Therefore, the degree to which patterns impact quality attributes supports software architects selecting the best fitting pattern (s), i.e., adopting or avoiding a pattern for a given design problem. The notion of "forces", as a feature type, equates in many ways to "quality attributes" that software architects try to optimize and the concerns they attempt to address in designing architectures.

The impacts of the patterns on the quality attributes, such as availability and security, are key reasons to select a particular pattern. In the knowledge extraction phase of the SLR, we realized some inconsistencies regarding the observed impacts of patterns on quality attributes. Some studies reported adverse impacts of a particular pattern on a quality attribute. For instance, Qin et al. (2008) and Sharma et al. (2015) stated that *efficiency* is a strength of the *Pipe and Filter* pattern, however, Vogel et al. (2011) expressed that *efficiency* is a liability for this pattern. Therefore, *efficiency* can be considered as both strength and liability of the *Pipes and Filters* pattern.

Quantifying the impact of a particular pattern on the quality attributes is complex.

[1]Note, the complete list of the features is available on the DSS website: <The link has been removed due to the double-blind review process>

Table 9.1: shows a subset of the mapping between the features and the patterns. Note, the complete lists of definitions of the patterns and features besides the mapping among them is available on the DSS website: `https://dss-mcdm.com`

Feature-Pattern	Patterns																												
	SOA	MICROSERVICE	SPACE-BASED	CLIENT-SERVER	C2	MVC	BLACKBOARD	SHARED REPOSITORY	MICROKERNEL	PUBLISH-SUBSCRIBE	LAYERS	PIPES AND FILTERS	COMPONENT-BASED	BROKER	PEER-TO-PEER	PAC	IMPLICIT INVOCATION	MASTER-SLAVE	BATCH SEQUENTIAL	INDIRECTION LAYER	CQRS	RPC	INTERCEPTOR	MESSAGE QUEUING	RULE-BASED SYSTEM	REFLECTION	EXPLICIT INVOCATION	INTERPRETER	VIRTUAL MACHINE
Centralized data management			✓	✓			✓	✓											✓						✓				
Decentralized data management	✓	✓	✓	✓											✓				✓										
Data conversion/transformation								✓				✓							✓		✓								
Multiple clients			✓	✓	✓	✓	✓	✓							✓														
Centralized governance	✓		✓	✓			✓	✓										✓							✓				
Decentralized governance	✓	✓	✓												✓		✓		✓										
Single access point			✓	✓			✓	✓										✓											
Parallel processing		✓	✓	✓	✓		✓	✓		✓	✓	✓	✓			✓		✓			✓			✓					
Multi stage process	✓										✓	✓	✓						✓										
Sequential processing							✓					✓							✓										
Thread processes			✓						✓			✓						✓											
Scalability	↑	↑	↑	↗	↑	↓	↑	↑	↑	↓	↓	→	↑	↑	↑	↑	↘		↓		↑	↑		↑		↑			↓
Reusability	↑	↑		↗	↑	↗	↑	↓	↑	↑	↑	↑	↑	↑		↗	↑		→	↑		↓	↑			↗		↑	↓
Performance efficiency	↗	↑	↑	→		↓	↘	↓	↓	→	↓	↗	↗	↓	↑	↓	↓	→	↓		↑					↓	↑		↓
Testability	↓	↑	↓	↓		↑	↓	↓	↗		↑	↑	↓	↓		→	↓		↓			↓				↓		↓	→
Modifiability	↗	↑		↓		↑	↑	↑	↑		↑	↗	↑	↑	↑	↑	↑		→			↑				↓		→	↓

We applied fuzzy logic to aggregate the extracted knowledge regarding the potential impacts of patterns on quality attributes (Farshidi et al., 2020e). Table 9.1 shows a subset of the mapping between the features and the patterns that we have considered in the decision model.

- Impacts of features: We employed the *ISO/IEC 25010* standard (ISO, 2011) and *extended ISO/IEC 9126* standard (Carvallo & Franch, 2006) as two domain-independent quality models to analyze the impacts of the features on the software quality aspects. The key rationale behind using these Software quality models is that they are a standardized way of measuring a software product. Moreover, they describe how easily and reliably a software product can be used.

The mapping between the sets *Qualities*, which includes "Software Quality Aspects", and "*Features*" was identified based on domain experts' knowledge. The domain experts, who participated in this phase of the research, mapped the features to the software quality aspects based on a Boolean adjacency matrix ($Qualities \times Features \rightarrow Boolean$). For instance, they stated that "Decentralized Governance" as a feature influences the "Time behavior" of a software product. The experts did not enforce a feature to present in a single quality aspect, as typically, the features have impacts on multiple quality aspects, which means that the cardinality of the mapping is many-to-many. For example, "Centralized Governance" as a feature might impact on "Resource utilization" and "Analyzability" of a software product.

- Feature Requirements: Software architects should prioritize their requirements using a set of weights according to the definition of the *MoSCoW* prioritization technique and send them to the inference engine of the DSS. Feature requirements with *Must-Have* or *Won't-Have* priorities act as hard constraints and feature requirements with *Should-Have* and *Could-Have* priorities act as soft constraints. The DSS excludes all infeasible patterns which do not support features with *Must-Have* and support features with *Won't-Have* priorities. Then, it assigns non-negative scores to feasible patterns according to the number of features with *Should-Have* and *Could-Have* prioritizes (Farshidi et al., 2018a) that they have.

Figure 9.2: illustrates the participants while working together on their assignments.

9.4 Empirical Evidence

A group of software practitioners, including 24 software architects and developers in the Netherlands, have participated in this research to assess the user acceptance of the decision model based on the TAM. Firstly, we tried to estimate the participants' background knowledge with the 29 patterns, so we asked them to fill out a questionnaire and indicate whether they were familiar with the patterns. Then, we asked them to determine which patterns had employed before in their software design projects. Afterward, we formed 12 groups of two individuals according to their expertise and the companies that they were working with. Next, we introduced the decision model within the DSS knowledge base and presented some of its applications. Afterward, we assigned the problem definitions of two real-world software architectures to the groups and asked them to design two solutions for the problems. The groups used the DSS as an assistant to help them with 1) defining the requirements based on the MoSCoW prioritization technique, and 2) finding the best fitting set of patterns. The group sessions lasted between 45 to 60 minutes. At the end of the sessions, we ask all of the participants to fill out a TAM-based questionnaire. Next, we collected their feedback and opinion about the decision model. Fig. 9.2 illustrates the software architects while working on designing the assigned software architectures.

Table 9.2: shows the software practitioners' background knowledge regarding patterns. They indicated the patterns that they were familiar with or had employed in their projects.

Architectural patterns	Familiar	Employed
Client-Server	22.22%	77.78%
Service-Oriented Architecture	33.33%	61.11%
Model-View-Controller	22.22%	72.22%
Component-Based	33.33%	55.56%
Microservice	44.44%	44.44%
Layers	22.22%	55.56%
Message Queuing	38.89%	38.89%
Virtual Machine	27.78%	44.44%
Publish-Subscribe	33.33%	33.33%
Event-Based / Implicit Invocation	27.78%	33.33%
Broker	33.33%	27.78%
Command Query Responsibility Segregation	38.89%	22.22%
Space-Based / Cloud-Based	33.33%	16.67%
Batch Sequential	22.22%	27.78%
Peer-To-Peer	33.33%	16.67%
Shared Repository	22.22%	22.22%
Remote Procedure Call	16.67%	27.78%
Master-Slave	16.67%	27.78%
Pipes and Filters	16.67%	22.22%
Reflection	16.67%	22.22%
Interpreter	22.22%	16.67%
Command and Control	22.22%	11.11%
Rule-Based System	5.56%	22.22%
Presentation-Abstraction-Control	22.22%	0.00%
Interceptor	11.11%	11.11%
Blackboard	16.67%	0.00%
Microkernel	5.56%	11.11%
Explicit Invocation	16.67%	0.00%
Indirection Layer	5.56%	11.11%

9.4.1 Assessment of Background Knowledge

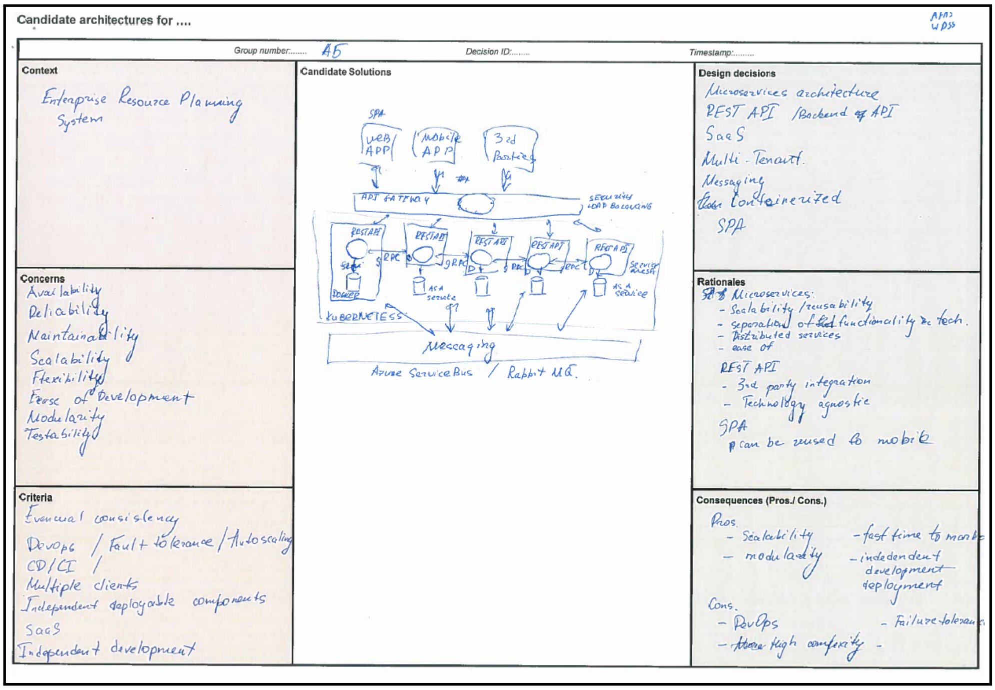
After reviewing and extracting knowledge from *232* high-quality primary studies in the SLR (Farshidi et al., 2020e), we identified 29 patterns that were mentioned in at least three primary studies. Next, we considered this set of patterns as the potential alternative solutions in the decision model.

In this study, we asked the 24 software practitioners who participated in our research to indicate their background knowledge regarding the patterns. They had to signify the patterns that they were familiar with or had employed in their projects. Table 9.2 presents the results of the analysis of the software practitioners' responses. The results show that *Client-Server*, *Service-Oriented Architecture (SOA)*, *Model-View-Controller (MVC)*, *Component-Based*, and *Microservice* are top-5 patterns that were indicated as the most well-known patterns among the participants. Additionally, none of the participants had employed *Explicit Invocation*, *Blackboard*, and *Presentation-*

Abstraction-Control in their projects. Moreover, we observed that the practitioners had limited knowledge regarding *Indirection Layer*, *Microkernel*, and *Interceptor*.

It is interesting to highlight that almost half of the participants were not informed about such a potential list of patterns. Accordingly, they requested to keep the hard copy of the descriptions and applications of the patterns that we attached to the questionnaires.

Figure 9.3: illustrates one of the questionnaires, including a candidate architecture for AFAS software architecture, that was filled out by a group of participants.



Candidate architectures for ....

AFAS
WDSS

Group number:....... A5	Decision ID:........	Timestamp:.........

Context

Enterprise Resource Planning System

Candidate Solutions

SPA
web APP
Mobile APP
3rd Parties
API GATEWAY
SECURITY
LOAD BALANCING
REST API
REST API
REST API
REST API
REST API
RPC
gRPC
RPC
RPC
DOCKER
AS A service
AS A service
Service mesh
KUBERNETESS
Messaging
Azure ServiceBus / Rabbit MQ.

Design decisions

Microservices architecture
REST API /Backend ~~of~~ API
SaaS
Multi-Tenant.
Messaging
~~[illegible]~~ Containerized
SPA

Concerns

Availability
Reliability
Maintainability
Scalability
Flexibility
Ease of Development
Modularity
Testability

Rationales

~~[illegible]~~ Microservices:
- Scalability /reusability
- separation of ~~[illegible]~~ functionality & tech.
- Distributed services
- ease of

REST API
- 3rd party integration
- Technology agnostic

SPA
~~p~~ can be reused to mobile

Criteria

Eventual consistency
Devops / Fault tolerance / Autoscaling
CD/CI /
Multiple clients
Independent deployable components
SaaS
Independent development

Consequences (Pros./ Cons.)

Pros.
- Scalability
- modularity
- fast time to market
- indedendent development ~~deployment~~

Cons.
- DevOps
- ~~More~~ High complexity
- Failure tolerance
-

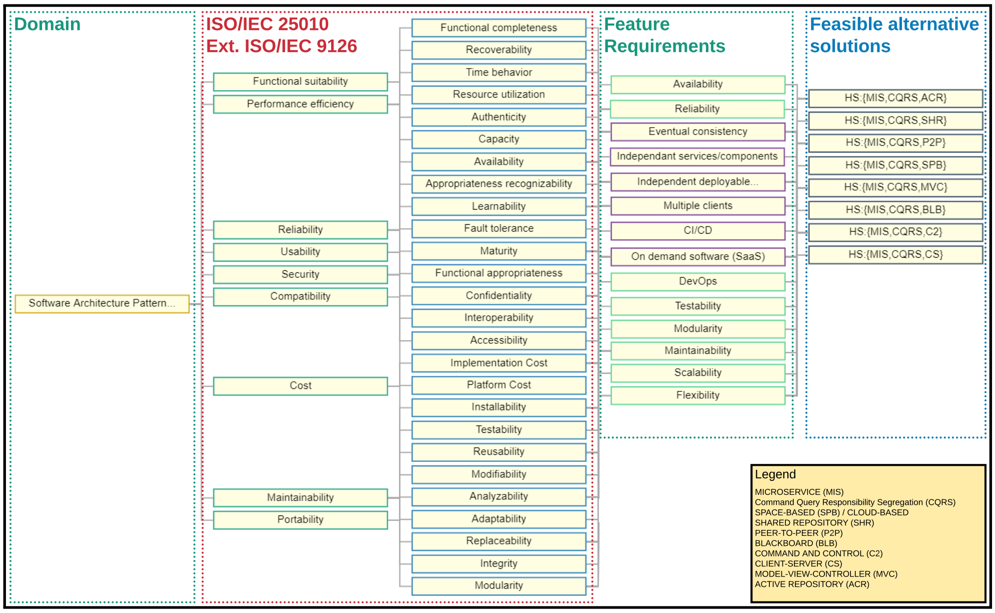
9.4.2 Candidate Architectures

Two experienced software architects from *AFAS software* and *Eijsink booq* have participated in our research and formulated their software architectures in the form of two real-world problems definitions. A brief description of the problem definitions is presented as follows.

AFAS software is an Enterprise Resource Planning (ERP) software product that automates and integrates business processes such as invoicing, project management, and payrolling in different enterprises. The AFAS software architecture should be a suite of modular, independent, technology agnostic, and scalable services. The service should be deployed on the cloud and be able to communicate through standard and well-defined interfaces. Additionally, the development of the services should be handle by multiple parallel software development teams. Eventual consistency should be appropriately addressed in the architecture design, as it offers low latency at the risk of returning stale data.

Eijsink booq is a streamlined ordering process that integrates payment terminals. Eijsink booq architecture should be a collection of components that directly responsible for the sales process, such as EFT (pin) terminals or a hotel system ("charge this to my room"). Components should control assortment and pricing; furthermore, they should work after the sale for bookkeeping and invoicing. The architecture should be highly secure, reliable, and available as it works with sensitive financial transactions. The main leading principle is "an interruptible sales process with a reliable management process". *Immutability:* Once an event is considered "finished" and is recorded, it cannot change. Recorded event data is immutable. *Unidirectionality:* data always travels in one direction, never bidirectionally. The responsible component stores and distributes the data, but will never receive the same data from other components. *Robustness through decoupling:* components will not have undue dependencies on each other. Decoupling and eventual consistency should be preferred over tight coupling.

Figure 9.4: shows one of the suggested set of alternative solutions by the DSS to a group of participants.



We asked the 24 software practitioners to design at least one solution per group to address the design problems. We used Rozanski and Woods guidelines (Rozanski & Woods, 2012) to develop the following sections in the questionnaire for candidate architectures. (1) *Context* refers to the environment of the software system. (2) *Concerns* often translate into requirements on quality attributes, which are variously called non-functional requirements, extra-functional requirements, behavioral requirements, or quality attribute requirements. (3) *Criteria* represent some significant, central functionality of the system. (4) *Candidate solutions* refer to the architecture and designing the system components that meet the system requirements. (5) *Design decisions* indicate a set of architectural patterns/styles. (6) *Rationale* reflect the rationale used for the decision-making process and form the natural bridge between the rationale and the resulting architecture. (7) *Consequences (Pros./ Cons.)* involve selecting different tradeoffs using the pros and cons of the solutions as arguments to rationalize the selection of a particular solution.

The participants employed the DSS as an assistant to elicit requirements, including functional requirements and quality concerns. Then, they filled out the questionnaires based on the suggested alternative solutions by the DSS and their tacit knowledge. Figure 9.3 illustrates one of the candidate architectures for AFAS software that was suggested by one of the groups. Figure 9.4 shows one of the suggested set of alternative solutions by the DSS used by one of the groups.

9.4.3 Technology Acceptance Model

The TAM is a robust framework applied to predict how and when individuals will adopt and use a new technology (Davis, 1989). The TAM has been widely applied in technology assessment, producing reliable results when users have worked with the technology for some time. The TAM suggests that "Perceived Usefulness" and "Perceived Ease of Use" are two significant determinants of users' attitudes towards employing information technology. "Perceived Ease of Use" is the degree to which a user believes that the technology is challenging to use, based on the cognitive resources required to work with a particular system. "Perceived Usefulness" is interpreted as how much a user believes that the system will enhance her performance on a task. According to the TAM, "Perceived **U**sefulness" and "Perceived **E**ase of Use" are firmly associated with **S**elf-predicted future usage, i.e., the intention to employ a tool. For each of the three fundamental concepts of the TAM, there are sets of statements that measure the concepts. We adopted the statements presented by (Babar et al., 2007b) to form the following TAM-based statements in the questionnaire:
Perceived Usefulness (Ui): Work more Quickly (U1): using the decision model in my job would enable me to accomplish tasks more quickly; Improve Performance (U2): using the decision model would improve my job performance; Increase Productivity (U3): using the decision model in my job would increase my productivity; Effectiveness (U4): using the decision model would enhance my effectiveness on the job; Makes Job Easier(U5): using the decision model would make it easier to do my job; Useful (U6): I would find the decision model useful in my job;
Perceived Ease of Use (Ei): Easy to Learn (E1): learning to operate the decision model would be easy for me; Easy to Perform (E2) I would find it easy to get the decision model to do what I want it to do; Clear and Understandable (E3): my interaction with the decision model would be clear and understandable; Easy to become Skilful (E4): I would find the decision model to be flexible to interact with; Easy to Remember (E5): it would be easy for me to become skillful at using the decision model; Easy to Use (E6): I would find the decision model easy to use;
Self-predicted future use (Si): Actual Usage (S1): I predict that I will regularly use the decision model in the future; Prefer Paper-Based Format (S2): I would prefer using the decision model to paper-based forms for performing inspections;

Table 9.3: shows the results of the assessment in terms of “descriptive statistics”, “factor analysis”, and “summary of responses”.

		Descriptive Statistics			Factor Analysis			Summary of Responses						
								1	2	3	4	5	6	7
		Mean	Standard Deviation	Cronbach’s α	Perceived Usefulness	Perceived Ease of Use	Self-predicted future usage	extremely likely	quite likely	slightly likely	neither	slightly unlikely	quite unlikely	exterely unlikely
Work more Quickly	U1:	3.75	1.39		**0.89**	0.36	0.15	0.00%	16.67%	41.67%	12.50%	8.33%	20.83%	0.00%
Improve Performance	U2:	3.42	1.29		**0.84**	-0.04	0.16	0.00%	29.17%	33.33%	12.50%	16.67%	8.33%	0.00%
Increase Productivity	U3:	3.79	1.58		**0.93**	0.31	0.12	0.00%	20.83%	37.50%	12.50%	8.33%	12.50%	8.33%
Effectiveness	U4:	3.42	1.41		**0.87**	0.17	0.03	0.00%	33.33%	29.17%	16.67%	4.17%	16.67%	0.00%
Makes Job Easier	U5:	2.96	1.14		**0.82**	0.26	0.04	8.33%	29.17%	33.33%	16.67%	12.50%	0.00%	0.00%
Useful	U6:	2.83	0.94		**0.82**	0.33	0.11	4.17%	37.50%	33.33%	20.83%	4.17%	0.00%	0.00%
Easy to Learn	E1:	1.96	0.61		0.25	0.39	0.63	16.67%	75.00%	4.17%	4.17%	0.00%	0.00%	0.00%
Easy to Perform	E2:	2.71	1.10		0.49	**0.81**	0.18	12.50%	29.17%	41.67%	12.50%	0.00%	4.17%	0.00%
Clear and Understandable	E3:	2.75	1.20		0.45	**0.89**	0.21	12.50%	33.33%	33.33%	12.50%	4.17%	4.17%	0.00%
Easy to become Skilful	E4:	2.63	0.75		0.24	**0.89**	0.29	4.17%	41.67%	41.67%	12.50%	0.00%	0.00%	0.00%
Easy to Remember	E5:	2.42	0.86		0.07	**0.85**	0.33	12.50%	45.83%	29.17%	12.50%	0.00%	0.00%	0.00%
Easy to Use	E6:	2.63	1.03		0.18	**0.89**	0.25	8.33%	50.00%	16.67%	20.83%	4.17%	0.00%	0.00%
Actual Usage	S1:	3.33	1.57		0.81	0.44	0.39	4.17%	33.33%	29.17%	12.50%	4.17%	12.50%	4.17%
Prefer Paper-Based Format	S2:	2.42	1.26		0.04	0.15	**0.89**	29.17%	25.00%	29.17%	12.50%	0.00%	4.17%	0.00%
Perceived Usefulness	Ui:	20.17	6.76	0.93	1.00	0.27	0.13							
Perceived Ease of Use	Ei:	15.26	4.59	0.89	0.27	1.00	0.27							
Self-predicted future usage	Si:	5.75	2.22	N/A	0.13	0.27	1.00							

The participants used a seven-point Likert scale to respond to each statement of the questionnaire regarding their degree of agreement or disagreement. They had to choose one of seven responses: extremely likely (1), quite likely (2), slightly likely (3), neither (4), slightly unlikely (5), quite unlikely (6), extremely unlikely (7). A numerical value was assigned to each statement so that the sum of those values meant the user’s attitudes towards employing the decision model. Additionally, we considered an open-end remark section following each set of statements related to each concept to gain more feedback from the participants. Table 9.3 shows the results of the assessment in terms of “descriptive statistics”, “factor analysis”, and “summary of responses”.

Descriptive Statistics: The numerical results correspond to the Likert scale show that the participants on average respond cautiously positive to the statements, as the mean values (column Mean) are mainly between 2 (quite likely) and 4 (neither). Many participants responded positively; however, a number of the participants were not convinced about the usefulness and ease of use of the decision model’s support for the pattern selection process. One reason for such responses might be the consequences of the low level of experience with the decision model within the DSS knowledge base (See the “Summary of Responses” for more detail). The results of the “Self-predicted future usage” factors show that, on average, the participants intended to use the decision model regularly if it is available.

The reliability analysis verifies the internal validity and consistency of the statements related to each concept of the TAM. Generally speaking, reliability is the degree to which one would obtain the same result if she carried out the study again to

the same participants under the same conditions (Laitenberger & Dreyer, 1998). The most widely accepted measure of reliability is Cronbach's alpha. Cronbach's alpha reveals how much each measured statement is correlated with every other statement, the consistency of the assessment model, i.e., the degree to which high responses correlate with highs, and low responses correlate with lows across all statements. A Cronbach's Alpha reliability level that exceeds a threshold of 0.8 indicates a reliable measure (Carmines & Zeller, 1979). In this study, the Alpha values were 0.93 and 0.89 for "Perceived Usefulness" and "Perceived Ease of Use" statements, respectively (See Fig. 9.4). Thus, the results confirm that the questionnaire was consistent.
Factor Analysis: This phase of the analysis reports the relationships among statements of the concepts. Factor analysis clusters the statements and assigns them to one of the three fundamental concepts of the TAM. The factor loadings indicate the correlation of the statements with the concepts so that they range from -1, a perfect negative association, through 0, no relation, to +1, a perfect positive correlation with the concepts. A statement is typically meant to a concept if it has a loading of at least 0.7 (Kim et al., 1978). Table 9.3 represents the adapted factor loading. The results of the usefulness statements (U1...U6) show a high correlation with "Perceived **U**sefulness", as their factor loadings are higher than 0.7. Additionally, the results of the ease of use statements (E1 ... E6) indicate an acceptable correlation with the second concept. Note, even lower values than 0.7 are sometimes considered significant for a particular concept (Kim et al., 1978). For instance, E1 and S1 have values (0.39) slightly below 0.7.

According to the theory of reasoned action, usefulness and ease of use are strongly correlated to self-predicted future (S1), the intention of actually using the tool, if it is available (Babar et al., 2007b). The last three rows of the "factor analysis" section of Table 9.3 show the concept correlation among these three concepts: Self-predicted future use is indeed correlated to both usefulness and ease of use, while usefulness and ease of use have some, correlation, which is consistent with reports in literature (Babar et al., 2007b). Hence, usefulness and ease of use are essential determinants that influence self-predicted future usage. Note, the majority of the participants stated that they prefer using a paper-based format of the decision model in the future, as the performance and the user interface of the DSS were not satisfactory from their perspective.
Summary of Responses: The distribution of the responses regarding each statement is shown in the summary of the responses' section (see Fig. 9.4). For instance, 41.67 percent of the participants believed that it is "slightly likely" that using the decision model in their job would enable them to accomplish tasks more quickly (U1); moreover, 20.83 percent of them stated that the statement is "quite unlikely" in their case. As aforementioned, to have a deeper understanding concerning the numerical responses, we asked the participants to elaborate their responses in open-end remarks (R_i) for each concept of the TAM. Overall, we received the following sixteen unique responses:
- *Perceived Usefulness:* (R_1)*"I do not make a lot of similar decisions in my daily job."*, (R_2)*"I would use the DSS as a second opinion rather than basing my entire decisions on it."*, (R_3)*"I prefer to use the DSS as an evaluation tool to check my decisions after the design."*, (R_4)*"Although The DSS needs some improvements, it can help me to give*

a border perspective.", (R_5)"*I have to use the DSS on my projects to check if it has the same conclusions as myself. I guess some specific things might be missing. However, when it concludes the same as me, it will help to convince others. Additionally, it can help me to explore more options.*", (R_6)"*I think during the design phase, the DSS might be confusing and make the phase slower. However, after the initial design, it could be useful for evaluating and improving the design.*", (R_7)"*I like the idea of suggesting the best fitting patterns. It can be used to generate architecture descriptions.*", and (R_8)"*two participants asserted that they could not tell us anything about the usefulness of the DSS, as they do not have enough experience with it.*"
- *Perceived Ease of Use:* (R_9)"*The performance of the DSS should be improved.*", (R_{10})"*I need to have some more experience with the DSS to answer the questions regarding its ease of use.*", (R_{11})"*three participants expressed that the user interface of the DSS should be improved for actual usage.*", (R_{12})"*I need more time to use it and see how it works. Nevertheless, at first glance, it is not easy to use.*", and (R_{13})"*It looks easy to use.*".
- *Self-predicted future use:* (R_{14})"*I would like to use the DSS besides the traditional decision-making process.*", (R_{15})"*I will try to convince others to try the DSS, and I want to use to once more and share my experience with it.*", (R_{16})"*I will try it again.*".

9.5 Discussion and Analysis

To answer the first research question (RQ_1), we followed the framework to capture knowledge regarding the patterns from the literature and software architects' experience. It is essential to admit that we cannot find a comprehensive set of patterns: technological innovations will continually introduce more complex and specific patterns. Analog to how the elevator has enabled us to build taller buildings, new innovative patterns such as *CQRS* enable us to create more extensive and more scalable systems. Because of this continuous innovation, it remains a responsibility of the academic community to consolidate and present architectural knowledge to the practitioner community continuously.
(RQ_2) The participants highlighted that the decision model, in terms of reusable knowledge regarding the patterns, was a useful tool that can support them to explore more patterns while designing real-world software architectures. They asserted that the decision model itself could assist them in finding liabilities and strength of patterns, their features, and potential application domains that they have employed in. After the session, most of the participants asked us to send the hard copy of the decision model in the form of tables.

Although we received significant positive numerical responses regarding the usefulness of the decision model (U1...U6) and oral feedback, the open-end remarks revealed that the DSS needed improvements in its performance and user interface to support software architects. Moreover, according to the responses and during the session, we realized that the DSS was not an intuitive tool, and the participants required some training to be able to use it effectively. The session's time limit was an issue that affected the evaluation of the usefulness of the decision model.

The responses regarding the "self-predicted future use" (S1 and S2) also show that the participants were interested in using the decision model and tended to use it as a support tool that supports them with the pattern selection process.

Table 9.4: shows a comparison of selected studies from the literature that addresses the pattern selection problem. The first and second columns (Studies and Years) refer to the studies and their publication years. The third column (DMA) indicates the decision-making approach that the studies have employed to address the problem. The fourth column (MCDM) denotes whether the corresponding decision-making technique is an MCDM approach. The fifth column (PC) indicates whether the MCDM approach applied pairwise comparison as a weight calculation method or not. The sixth column (QA) determines the type of quality attributes. The seventh and eighth columns (#C and #P) signify the number of criteria and patterns that were considered in the selected studies.

Studies	Years	DMA	MCDM	PC	QA	#C	#P
This study		DSS	Yes	No	ISO/IEC 25010 EX. ISO/IEC 9216	188	29
Jacob & Mani (2018)	2018	Benchmarking	No	N/A	Domain Specific	8	4
Haoues et al. (2017)	2017	Benchmarking	No	N/A	ISO/IEC 25010	39	5
Me et al. (2016)	2016	Benchmarking	No	N/A	Domain Specific	15	11
Nawaz et al. (2015)	2015	WSM	Yes	No	Domain Specific	38	2
Richards (2015)	2015	Benchmarking	No	N/A	Domain Specific	6	5
Galster et al. (2010)	2010	AHP	Yes	Yes	Domain Specific	29	5
Moaven et al. (2008)	2008	Fuzzy logic	Yes	Yes	Domain Specific	3	3
Avgeriou & Zdun (2005)	2005	Benchmarking	No	N/A	Domain Specific	10	24
Garlan & Shaw (1993)	1994	Benchmarking	No	N/A	Domain Specific	5	6

(RQ_3) Almost all participants drew the draft versions of their designs on pieces of paper and made their design decisions accordingly. In other words, drawing the design was a way for them to convert their tacit knowledge to explicit knowledge to communicate, brainstorm, and make decisions (see Figure 9.3). Afterward, they started to use the DSS as a tool to evaluate what they had designed (see Figure 9.4). Thus, we realized that the DSS requires an option to document their drawings and design rationales to prevent knowledge vaporization. We implemented a design studio based on the Unified Modeling Language concepts to collect such drawings while the decision-making process.

9.6 Related work

Benchmarking and documentation are typically time-consuming approaches and mainly applicable to a limited set of patterns and criteria, as they require in-depth knowledge of patterns and concepts. Furthermore, such studies should be kept up to date continuously, which involves a high-cost process. Table 9.4 shows a subset of such studies.

MCDM Techniques have introduced by researchers to address the pattern selection problem in the literature. A subset of selected MCDM methods, such as the Weighted Sum Model (WSM) and The Analytic Hierarchy Process (AHP), is shown in Table 9.4. The majority of the MCDM techniques in literature define domain-specific quality attributes to evaluate the alternatives. Such studies are appropriate for specific case studies. Furthermore, MCDM approaches are valid for a specified period; therefore, the results of such studies will be outdated. Some of the methods, such as AHP, are not scalable, so in modifying the list of alternatives or criteria, the whole process of evaluation should be redone. Therefore, these methods are costly and applicable to only a small number of criteria and alternatives. According to the SLR (Farshidi et al., 2020e), we considered 188 criteria and 29 patterns to build a decision model for the pattern selection problem. Recently, we built five more decision models based on the framework for the selection process of database Management Systems (Farshidi et al., 2018c), Cloud-Service Providers (Farshidi et al., 2018a), Blockchain Platforms (Farshidi et al., 2020c), Model-Driven Software Development Platforms, and Programming Languages. Case studies and expert interviews were conducted to evaluate the DSS's effectiveness and usefulness in addressing these MCDM problems. The results confirmed that the DSS performed well to solve the mentioned problems in software production.

9.7 Conclusions

In this study, the architectural pattern selection problem in pattern-driven software design is modeled as a multi-criteria decision-making problem that deals with evaluating a set of alternatives (patterns) and taking into account a set of decision criteria (Triantaphyllou et al., 1998). Moreover, we presented a decision model for the decision-making problem based on our framework (Farshidi et al., 2018a). In order to build the decision model, we conducted an SLR besides a set of expert interviews. Additionally, we carried out a study with a group of software practitioners in the Netherlands to assess the user acceptance of the decision model within the knowledge base of a decision support system (Farshidi et al., 2018b) based on the technology acceptance model. The results show that software practitioners tend to employ the decision model as a decision support tool to evaluate their design decisions in pattern-driven software design. It is essential to highlight that we have received constructive feedback from the participants to improve the decision model in the decision support system's knowledge base to facilitate the decision-making process. Academics and practitioners can use the lesson learned in this study to set up their studies for evaluating decision support tools in software architecture.

Concluding the Research

CHAPTER 10

Conclusion

In this dissertation, we developed a theoretical framework to assist software engineers with decision-making in MCDM problems in software production. The framework provides a guideline for software engineers to capture knowledge systematically from different sources of knowledge to build decision models for MCDM problems in software production. Knowledge has to be captured and organized when it is needed. We designed and implemented a decision support system for software production, called SoProDSS, that utilized our decision models to facilitate decision-making and support software engineers with their MCDM problems.

The rest of this chapter is organized as follows: Section 10.1 addresses the research questions that are defined in the introduction chapter (see Section 1.7.1). Section 10.2 discusses the threats to the validity of the research. Next, reflections on this research are provided by placing research results and implications in a broader context, and directions for future research are described in Section 10.3. Finally, Section 10.4 presents the limitations of the main limitations of our work and gives an agenda for future research to continue improving the SoProDSS and supporting software engineers with their decision-making problems in software production.

10.1 Contributions and observations

This section discusses the strengths and liabilities of MCDM techniques in literature and positions the MCDM framework among them. Furthermore, it addresses the research questions which are defined in Section 1.7.1 and explains our main observations and contributions to the field of software production.

10.1.1 Strengths and Liabilities of MCDM Techniques

Decision analysis, which is the study of decision making for problems with multiple objectives, has been developed and widely employed in solving complex decision-making problems. Over the past few years, various methods and underlying theories have been introduced for solving decision-making problems in software production. According to the literature studies in Sections 2.3, 3.3, 4.3, 5.7, 6.6, and 7.2.3, we observed that the majority of the multi-criteria decision-making (MCDM) techniques in literature define domain-specific quality attributes to evaluate alternatives. Such techniques are mainly appropriate for specific case studies.

The results of MCDM approaches are valid for a specified period because of technological advances. Note that, in our proposal, this is also a challenge, and we propose a solution for keeping the knowledge base up to date, in Section 5.6. Additionally, a pairwise comparison is typically considered as the main method to assess the weight of criteria in MCDM techniques. For a problem with n number of criteria $\frac{n(n-1)}{2}$ number of comparisons are needed (Saaty, 1990). It means that the pairwise comparison is a time-consuming process, and gets exponentially more complicated as the number of criteria increases (Ribeiro et al., 2011). A subset of MCDM approaches, such as AHP and TOPSIS, are not scalable (Ibriwesh et al., 2018; Khari & Kumar, 2013), so in modifying the list of alternatives or criteria, the whole process of evaluation should be redone. Therefore, these methods are costly to maintain, inflexible to change, and applicable to only a small number of criteria and alternatives (see Tables 2.1, 3.1, 4.1, 5.6, 6.6, and 7.2).

In contrast to the named approaches in the literature, the cost of creating, evaluating, and applying the decision models, including Database Management System (Chapter 2), Cloud Service Provider (Chapter 3), Blockchain Platform (Chapter 4), Programming Language Ecosystem (Chapter 5), Model-Driven Software Development Platform (Chapter 6), are not penalized exponentially by the number of criteria and alternatives, because they are evolvable and expandable approaches that split down the decision-making process into four maintainable phases (see Chapter 2). Moreover, we introduce several parameters to measure non-Boolean criteria' values, e.g., the maturity level and popularity of technology alternatives. The decision models address the main knowledge management issues, such as capturing, sharing, and maintaining knowledge. Furthermore, they use the ISO/IEC 25010 (ISO, 2011) as a standard set of quality attributes. This quality standard is a domain-independent software quality model and provides reference points by defining a top-down standard quality model for software systems (see Section 6.4.3).

10.1.2 Research Questions

Software engineering is a knowledge-intensive field (Pressman, 2005), so software engineers need to spend a significant portion of their time collecting data regarding their design decisions and the domains of the decisions-making problems (Meyer et al., 2019). Thus, the main research question of this dissertation is formulated as follows:

MRQ — ***How can software engineering knowledge be captured and organized systematically to support software engineers with software production decision-making?***

We hypothesized that software engineers lack the needed knowledge to make significant decisions in software production. To answer the main question of this dissertation and gain more insight into the decision-making process in software production, we formulated six research questions in the introduction chapter and investigated them in Chapters 2 to 9. This section answers the questions and briefly evaluates the findings that correspond to each of the questions.

RQ_1 — ***How do software engineers make decisions in software production?***

We conducted interviews with a set of domain experts at different software producing organizations to answer this research question. Additionally, we tracked the case study participants' behavior while defining their requirements and making-decisions and observed the following points:
1.1 - Software engineers typically consider a limited set of criteria and solutions and make decisions based on their tacit knowledge. Based on the case studies that we have conducted in this dissertation, we realized that software engineers continuously improve and reevaluate their decisions in different phases of the software development life-cycle. In Chapters 2, 3, and 4, we observed that software engineers typically focus on more generic domain features of alternatives in the early phases of the life-cycle, whereas they are interested in more detailed and specific domain features as their development process matures.
1.2 - In Chapter 9, we find that software engineers draw draft versions of their designs on pieces of paper and make their design decisions accordingly. In other words, drawing the design is a way for them to convert their tacit knowledge to explicit knowledge to communicate, brainstorm, and make decisions.
1.3 - From the expert interviews reported in Chapters 2 and 5, we notice that experience in using technology provides invaluable knowledge when selecting suitable technology. In other words, software engineers typically prefer to select technology solutions that they have employed before and have experience with. The main factor is the cost of adding a new technology solution. Hiring new software engineers, changing the infrastructure, and learning the best practices are costly for software producing organizations. There are many risks associated with the decision-making process, as a software producing organization could end up being stuck in a legacy

technology for which there is no longer a demand (Khadka et al., 2014) or sufficient support.
1.4 - Software engineers are biased towards trending technology solutions in their design decisions (see Chapter 7). Over time, quality requirements of systems change because of advances in technology that address particular quality concerns of software engineers. It is possible to identify trends in technology usage.
1.5 - In Chapter 6, we describe how we observed that the decision-making process is different for small organizations versus large ones. Looking at the IT landscape, we notice a difference in the selection process because the requirements of small and large organizations are different. Small enterprises typically start purchasing a unique technology solution to solve multiple problems; they cannot invest in various solutions for performing different tasks because of financial constraints. Larger enterprises can invest in employing various solutions for different tasks, and thus experience more flexibility in their decision processes.
1.6 - The interviewed domain experts who participated in our research were mainly software consultants (see Sections 2.2, 3.2, 4.2, 5.2, 6.3, and 9.2). Accordingly, we realized that software producing organizations are willing to pay significant amounts for advice from domain experts, such as software architects, cloud specialists, and database experts.

Observation I
Software engineers typically consider a limited set of criteria and solutions and make decisions based on their tacit knowledge. They continuously improve and reevaluate their decisions in different phases of the software development life-cycle. Software engineers prefer to select technology solutions that they have employed before and have experience with. Additionally, they are biased towards trending technology solutions. Software producing organizations typically invest more in their decision-making process by hiring software consultants.

RQ_2 — ***How can a framework be developed that serves as a reference framework for decision problems in software production?***

The development of software products, systems, and services typically results in complex decision support models and decision-making processes (Badampudi et al., 2018). In literature, designers have proposed a variety of theory development approaches, with slightly different iterative structure and logic. A subset of the selected studies is presented as follows.

Seven guidelines were introduced by Hevner et al. (2004) to align information science DSR with real-world production experience. They stated that a business needs to motivate the development of validated artifacts that meet those needs. The development of justified theories about these artifacts produces knowledge that supports design scientists with their design decisions. Meredith et al. (1989) plus Cooper & Emory (1995) suggested that all DSR investigations involve a continuous, repetitive cycle of *description*, *explanation*, and *testing* (through prediction). Thus, proposing knowledge (explanation) and validating knowledge (testing) simply are two stages in the ongoing cycle of research. The nature of the design process was described by Simon (1996) as a Generate/Test Cycle. The design process in the Generate/Test Cycle involves, first, the generation of alternatives and, then, the testing of these alternatives against a whole array of requirements and constraints. Van Strien (1997) suggested a regulative cycle that begins with an investigation of a practical problem, then determines a set of possible solutions, validates them, selects one of them, and implements the solution chosen; the outcome of which can then be evaluated, which could be the start of a new cycle of the regulative cycle. Cross & Roy (1989) introduced a four-stage model (exploration, generation, evaluation, and communication) for designing products.

In this dissertation, we develop a theoretical framework called the MCDM framework (see Figure 1.4), in an iterative process for supporting software engineers with decision problems in software production.

2.1 - Theory development is a process of gradual change (Baxter, 2004). In other words, the development process of theories in DSR is an act of iterative interpretation. In a theory development process, designers make comparable design decisions in a particular domain. Such design decisions and their corresponding design rationales should be grouped and considered as repeatable design decisions for building similar theories in a particular domain. The captured knowledge from the development process of a theory provides an overview of the theory's design decisions and rationales. With such overviews, scientists can systematically develop and report their theories in DSR. Figure 1.1 shows the design process that we have followed to develop the MCDM framework for building decision models for MCDM problems in software production.

2.2 - The framework should be instantiated in each cycle of the MCDM framework development process to build a decision model for a particular MCDM problem in software production. In this dissertation, we instantiated the MCDM framework to build six decision models, including Database Management System (Chapter 2), Cloud Service Provider (Chapter 3), Blockchain Platform (Chapter 4), Programming Language Ecosystem (Chapter 5), Model-Driven Software Development Platform (Chapter 6), and Architectural Pattern selection problems (Chapter 9). The evaluation of each decision model leads to partial validation of the framework as a reference guideline that can be employed to address decision-making problems in software production.

2.3 - The applicability and validity of the framework should be tested by conducting multiple deductive case studies in each iteration of the theory development process. In this dissertation, we conducted 21 real-world case studies at different software-producing organizations (see Sections 2.5.3, 3.5.3, 4.5, 5.4, and 6.5). Case study participants had their functional requirements and quality concerns. Accordingly, they identified a set of alternative solutions based on their internal meetings before participating in the research. Afterward, the SoProDSS generated a shortlist of ranked feasible solutions according to their requirements. We defined the results' success when they, in part, aligned with the case study participants' shortlist of solutions and provided new suggestions that were identified as being of interest to the case study participants.

2.4 - In each cycle of the MCDM framework development process, (new) constructs

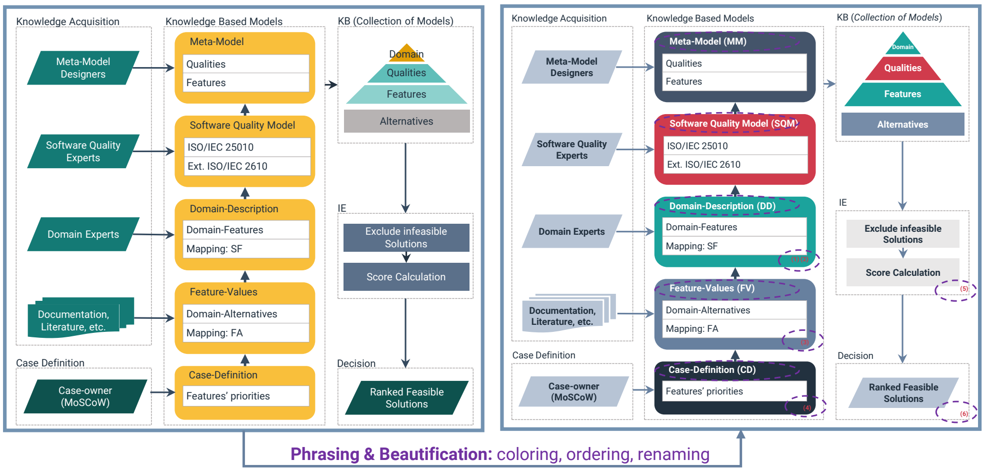


Figure 10.1: illustrates two revisions of the MCDM Framework. Please note the changes in captions, coloring, and ordering of the constructs.

and relationships are defined or revised accurately to keep consistency among components of the theory (see Section 1.8). We revised the MCDM framework during the theory development process and recorded the design decisions and rationales. In the early stages of the research, we assumed that technology alternatives have only Boolean features, meaning that a technology alternative supports a functionality or does not support it. According to the expert interviews in Chapter 2, we realized that technology alternatives have non-Boolean features, such as the total cost of ownership and popularity in the market. Consequently, we had to define new concepts and propositions.
2.5 - Multiple knowledge representation formalisms (Sloman, 1985) can be utilized to represent a specific theory and improve the depth of understanding of the theory. We considered the graphical representation and mathematical formalization besides ontological modeling as three knowledge representation formalisms to represent the design decisions and rationales of the MCDM framework (see Chapter 3). We realized that sometimes making design decisions in a particular knowledge representation formalism can lead to inconsistencies in the others. Such inconsistencies, in turn, lead to making either phrasing or beautification design decisions in the theory development process.

Figure 10.1 illustrates two revisions of the graphical representation of the MCDM framework; moreover, the figure pinpoints design decisions that we made besides their design rationales while building the framework. The improvement of the appearance (beautification) of a knowledge representation formalism, such as intuitive naming and proper coloring, leads to a higher level of understanding. Note, we changed the graphical representation and the mathematical formalization of the framework simultaneously to keep the consistency between them.
2.6 - Design decisions and rationales can be completed in one stage of the development process, for instance, "Textual improvement by capitalization and graphical

beautifications (coloring)" was a design rationale of two design decisions (phrasing and beautification) in the description stage of the development process of the MCDM framework, and led to a revision of the framework.

Design decisions and rationales can be initiated and improved along with design decisions and partially finalized in more than one stage of the framework. For example, identifying primary constructs and statements of relationships started and progressed in parallel within the description and explanation stages of the framework's development process and led to its first revision. In this dissertation, the constituent components of the framework have not changed since their definitions (see Figures 2.1, 3.1, 4.1, 5.1, 6.1, and 9.1).

2.7 - The number of design rationales and design decisions has decreased significantly in the last revisions of the framework. In this dissertation we used six figures (including Figures 2.1, 3.1, 4.1, 5.1, 6.1, and 9.1) to represented the main building blocks of the graphical representation of the MCDM framework. Each figure represents a decision model based on the framework for a particular decision-making problem in software production. A quick comparison of these six figures shows that the number of differences among the first three figures (Chapters 2, 3, and 4) is more than the last three ones (Chapters 5, 6, and 9). Thus, we conclude that the development process of the MCDM framework is converged to a maturity level that it can be employed to address different decision-making problems in software production.

Contribution I
The development process of an MCDM framework is an act of iterative interpretation. In each cycle of the development process, (new) constructs and relationships should be defined or revised to keep consistency among the framework's components. The framework should then be instantiated in each cycle of the development process to build a decision model for a particular MCDM problem in software production. The evaluation of each decision model leads to partial validation of the framework as a reference guideline that can be employed to address decision-making problems in software production. The framework's applicability and validity can be tested by conducting multiple deductive case studies in each iteration of the theory development process.
The number of design rationales and design decisions will be decreased significantly in the framework's final revisions, as it will be reached an acceptable level of maturity.

RQ_3 — ***Which sources of knowledge should be used to build decision models in software production?***

3.1 - When comparing technology alternatives, data must be collected from a variety of unstructured digital documents, such as webpages, whitepapers, scientific articles, fact sheets, technical reports, product wikis, product forums, product videos, and webinars can be used to form initial hypotheses. In Chapters 2, 3, 4, 5, 6, and 9, we perceived that initial sets of domain features and alternatives, as our initial hypotheses, concerning decision problems can be extracted based on exploring literature and existing documentation.

3.2 - Tacit knowledge (such as observations, opinions, prejudices, and ideas) is one of the primary sources of organizational knowledge (Nonaka & Von Krogh, 2009) to build the decision models in this dissertation (see Figure 1.1). In Chapter 6, we observed that tacit knowledge is deeply rooted in software engineers' practices and experiences. The biased and intuitive nature of tacit knowledge makes it challenging to process or transfer the captured knowledge in any systematic or logical way. The study of heuristics-and-biases has investigated various decision-making shortcuts and has documented their inferior performance (Kahneman et al., 1982; Tversky & Kahneman, 1974). However, these uncomplicated heuristics can be viewed as smart approaches to save time so that a decision-maker can respond immediately (Gigerenzer & Selten, 2002). Applying simple rules is sometimes an answer to complexity (Simon, 1955). When faced with a problem that is highly complex to solve optimally, the decision-maker falls back on a simple rule that makes sense based on what is understood. Fast-and-frugal heuristics can perform well in certain domains (Gigerenzer & Todd, 1999) to find the best fitting alternatives based on a limited set of criteria, for instance, background knowledge and experience of the decision-maker. For tacit knowledge to be communicated, it should be transformed into explicit knowledge (e.g., words, models, or numbers) so that anyone can understand it (Mohamed, 2010).

3.3 - Software quality models serve in this dissertation as the basis for the evaluation models of technology alternatives. Based on the IEEE Standard Glossary of Software Engineering Terminology (Committee et al., 1998; Samadhiya et al., 2010), the quality of software products is the degree to which a system, component or process meets specified requirements (such as functionality, performance, security, and maintainability) and the extent to which a system, component or process meets the needs or expectations of a user. It is necessary to find quality attributes widely recommended by other researchers to measure the system's characteristics. The literature study results showed that researchers do not agree upon a set of conventional criteria, including quality attributes and domain features, to evaluate technology alternatives. Additionally, we realized that their suggested criteria were mainly applied to specific domains to address different research questions. Consequently, a set of nonexclusive and domain-independent criteria is needed to evaluate technology alternatives. The ISO/IEC 25010 (ISO, 2011) presents best practice recommendations on the base of a quality assessment model. The quality model defines which quality characteristics should be considered when assessing the qualities of a software product. In this dissertation, we used the *ISO/IEC 25010* standard (ISO, 2011) and *extended ISO/IEC 9126* standard (Carvallo & Franch, 2006) as two domain-independent quality models to analyze domain features based on their impact on quality attributes of technology alternatives (see Chapter 2).

Observation II
The primary sources of knowledge for building decision models should be extracted based on exploring literature and existing documentation. Moreover, tacit knowledge (such as observations, opinions, prejudices, and ideas) should be considered as another source of knowledge to build decision models. Additionally, standard software quality models should be used for the evaluation models of technology alternatives, as they provide a foundation for reasoning, and it would be inefficient to recreate such lists of qualities independently.

RQ_4 — ***How should domain knowledge for building a decision model be extracted and categorized?***

4.1 - Expert Interview is an essential knowledge acquisition technique (Chen, 2004) in eliciting knowledge about software production decisions. We followed the guidelines from Myers & Newman (2007) to conduct 92 qualitative semi-structured interviews with senior software engineers to explore expert knowledge regarding the decision-making problems and evaluate the outcomes of our study so that we used the tacit knowledge of domain experts to identify the right sets of domain features and alternatives (see Sections 2.2, 3.2, 4.2, 5.2, 6.3, 7.4, and 9.2).
4.2 - Document analysis is a systematic procedure for reviewing or evaluating documents, including text and images that have been recorded without a researcher's intervention (Bowen et al., 2009). Document analysis is one of the analytical methods in qualitative research that requires data investigation and interpretation to elicit meaning, gain understanding, and develop empirical knowledge (Corbin & Strauss, 2014). According to the MCDM framework, the mapping between features and alternatives is defined based on documentation and expert interviews. In the knowledge acquisition phases of the decision models, we realized that one of the principal challenges is the lack of standard terminology among technology providers. Different vendors refer to the same concept by different names, or even worse, the same name might stand for different concepts in different alternatives. Discovering conflicts in the mapping is essential to prevent semantic mismatches throughout the selection process. In order to tackle this issue and prevent potential mismatches, we have initially collected the critical features of each alternative and then, based on cross peer reviews and expert interviews, tried to identify the similarities and mismatches (see Chapter 2 and 7).
4.3 - Systematic Literature Review is one of the most broadly accepted research methods of evidence-based software engineering (Kitchenham et al., 2004). As a significant part of the architectural knowledge is scattered, incoherent, and incomplete (Tang et al., 2011b), a sound methodology is required to capture this knowledge systematically. The data collection is an empirical study that can be quantitative or qualitative (Runeson & Höst, 2009). Quantitative data comprises numbers and classes, while qualitative data involves descriptions and explanations of phenomena. Quantitative data is analyzed using statistics, while qualitative data is analyzed using expert interviews or/and case studies to provide a more detailed and more in-depth explanation. However, a combination of qualitative and quantitative data often pro-

vides a better understanding of the studied phenomenon (Seaman, 1999). An SLR provides a prescribed process for identifying, evaluating, and interpreting all available evidence relevant to a particular research question or topic (Petersen et al., 2008). In Chapter 7, the SLR applied as a knowledge acquisition process to capture knowledge about architectural patterns and ultimately making it available in forms of reusable knowledge.

4.4 - In this dissertation, a structured coding procedure has been employed to extract knowledge from the selected sources of knowledge. Structured coding captures a conceptual area of the research interest (Saldaña, 2015). The extracted knowledge, which correspond to the elements of a decision problem, has been categorized into five categories: *quality attributes*, *alternatives*, *domain features*, *impacts of domain features on quality attributes*, and *supportability of domain features by alternatives*. Next, the extracted knowledge has been employed to build a decision model for an MCDM problem (see Chapters 2, 3, 4, 5, 6, and 9).

4.5 - In Sections 2.3, 3.3, 4.3, 5.7, 6.6, and 7.2.3, we realized that a significant number of MCDM techniques in the literature use pairwise comparison as the main method to assess the weight of criteria. For a problem with n number of criteria $\frac{n(n-1)}{2}$ number of comparisons are needed (Saaty, 1990). It means that the pairwise comparison is a time-consuming process, and gets exponentially more complicated as the number of criteria increases. Moreover, most MCDM methods are not scalable, so in modifying the list of alternatives or criteria, the evaluation process must be repeated. Traditional methods are costly and applicable to a small number of criteria and alternatives. In most cases, a unique optimal solution for an MCDM problem does not exist, so it is necessary to use decision-makers' preferences to differentiate between and prioritize solutions (Majumder, 2015). Therefore, decision-makers, as one of the sources of knowledge in their decision-making process, should prioritize and classify domain features based on their requirements and priorities.

Observation III
Expert interviews, document analysis, and systematic literature review are knowledge acquisition techniques employed to extract knowledge from different sources of knowledge. A structured coding procedure can be employed to extract knowledge from the selected sources of knowledge. The extracted knowledge, which correspond to the elements of a decision problem, can be categorized into five categories: *quality attributes*, *alternatives*, *domain features*, *impacts of domain features on quality attributes*, and *supportability of domain features by alternatives*.
Additionally, as one of the sources of knowledge in the decision-making process, decision-makers need to prioritize and classify domain features based on their requirements and priorities.

RQ_5 — ***How should the extracted knowledge for building a decision model be organized for facilitating the decision-making process?***

5.1 - Software production is a suitable domain to deploy decision support systems that intelligently support decision-makers with the decision-making process. The decision-making process gets more complicated as the numbers of decision-makers, alternatives and criteria increase (Majumder, 2015). DSS experts in Chapters 2 and 3 stated that a DSS is a tool that can be used over the software development life-cycle and can co-evolve its advice based on evolving requirements. In this dissertation, we designed and implemented the SoProDSS that integrates key aspects of Knowledge-Driven and Model-Driven DSSs to systematically organize the extracted knowledge regarding decision models to facilitate the decision-making process, and to support software engineers with their design decisions in software production (see Chapter 8). Figure 10.2 illustrates the MCDM framework and the main building blocks of the SoProDSS.

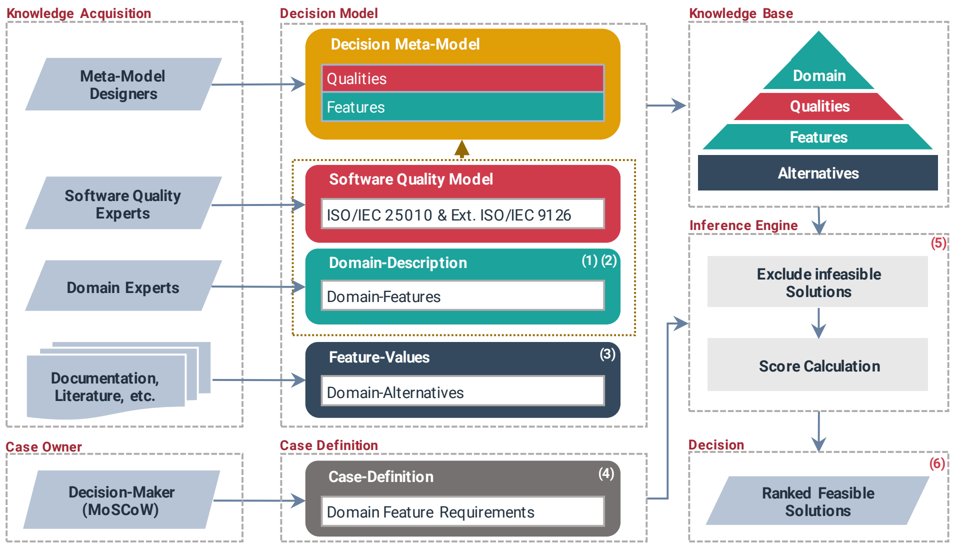


Figure 10.2: The MCDM framework that we follow to build decision models for MCDM problems in software production.

5.2 - The decision-making process in multi-criteria decision making is defined aptly by Majumder (2015) to have six steps: (1) identifying the objective, (2) selection of the criteria, (3) selection of the alternatives, (4) selection of the weighing method, (5) applying the method of aggregation, and (6) decision-making based on the aggregation results. In this dissertation, we followed this decision-making process to build maintainable and evolvable decision models for MCDM problems, and to make the knowledge acquisition more reliable and trustful (see Chapter 1).
5.3 - A decision model in the knowledge base of the SoProDSS as it is a collection of facts and rules of an MCDM problem. In other words, a decision model defines a decision structure to solve a specific MCDM problem (see Figure 4.2). Here we describe the components that we defined in this work:
5.3.1 - The *Decision Meta-Model* is a simplified view of decision models and highlights the fundamental structure of decision models in the knowledge base. Furthermore, it provides ontological descriptions of MCDM problems. The Decision Meta-Model has two sets, namely, *Qualities* and *Features*. Software quality attributes such as inter-

operability, maturity, and performance of technology alternatives are kept in the set *Qualities*. Additionally, domain features should be listed in the set *Features*.
5.3.2 - The *Domain-Description* determines the first and second steps, indicated by *Identifying the objective* and *Selection of the features*, of the decision-making process. As aforementioned, the initial set of domain features is extracted from online documentation of technology alternatives. Then, the list of features should be reviewed and validated by a set of selected domain experts. Next, the experts define the mapping between the sets *Qualities* and *Features* based on a Boolean adjacency matrix ($Qualities \times Features \rightarrow Boolean$).
5.3.3 The *Feature-Value* represents the third step, shown by *Selection of the alternatives*, of the decision-making process. A list of technology alternatives should be defined based on well-known vendors, websites, related forums, and domain experts' knowledge (see Figures 4.2 and 5.1). Domain features can be either Boolean or non-Boolean. A Boolean domain feature ($Feature^B$) is a feature that its supportability by technology alternatives is investigated. Figures 4.3, 5.2, and 6.1 are three samples of the mapping between Boolean domain features and technology alternatives ($Mapping^{Boolean} : Feature^B \times Alternatives \rightarrow \{0,1\}$). Moreover, a non-Boolean domain feature ($Feature^N$) assigns a non-Boolean value to a particular technology alternative. Figures 4.4, 5.3, and 6.3 present a subset of mappings between non-Boolean domain features and technology alternatives ($Mapping^{non-Boolean} : Feature^N \times Alternatives \rightarrow \{High, Medium, Low\}$).
5.3.4 - The *Case-Definition* defines the fourth step, denoted by *Selection of the weighing method*, of the decision-making process. The MoSCoW prioritization technique is a tool in management, business analysis, project management, and software engineering to reach a common understanding with decision-makers on the importance they place on each domain feature requirement; According to the MCDM framework, decision-makers should prioritize their domain feature requirements using the *MoSCoW* technique.

Suppose $W_{MoSCoW} = \{w_{Must}, w_{Should}, w_{Could}, w_{Won't}\}$ is the set of priority weights according to the definition of the *MoSCoW* (DSDM consortium and others, 2014). Domain feature requirements with *Must Have* or *Won't Have* priorities act as hard constraints and domain feature requirements with *Should Have* and *Could Have* priorities act as soft constraints. Note, we could have used other prioritization techniques, but we wanted to keep it simple (see Tables 2.2, 3.4, and 4.5 besides Figures 5.4 and 6.4).
5.3.5 - The *Inference Engine* comprises two steps: the fifth step, i.e., *Applying the method of aggregation* and the sixth step, i.e., *Decision making based on the aggregation results*, of the decision-making process. Decision-makers define and prioritize their domain feature requirements based on *MoSCoW* prioritization technique, and then the Inference Engine of the SoProDSS receives them (see Figure 8.3). The Inference Engine infers candidate solutions using the rules and facts of the decision models that it has in its knowledge base. In other words, the Inference Engine excludes infeasible solutions and assigns scores to the feasible ones, and then offers a ranked shortlist of feasible solutions to the decision-makers (see Tables 2.3, 3.5, 4.6, 5.5, 6.5, and 8.5).

Contribution II
We designed and implemented the SoProDSS to systematically organize the extracted knowledge regarding decision models, facilitate the decision-making process, and support software engineers with their design decisions in software production. The decision models that we have built, following the MCDM theory and the six-step of the decision-making process, should be uploaded in the knowledge base of the SoProDSS. Then, decision-makers should define their domain feature requirements based on the MoSCoW prioritization technique. Accordingly, the SoProDSS inference engine excludes infeasible solutions and offers a ranked shortlist of feasible solutions.

RQ_6 **— *Will software engineers be willing to use the decision models within the decision support system to perform their tasks?***

6.1 - In Chapter 9, we carried out a study with 24 software engineers in the Netherlands to assess the user acceptance of the SoProDSS based on the Technology Acceptance Model (TAM). We ask the participants to fill out a TAM-based questionnaire and state their opinions about the SoProDSS. The participants highlighted that SoProDSS is a useful tool that can help them explore more alternatives while designing real-world software products. They asserted the SoProDSS itself could assist them in finding liabilities and strength of alternatives, their features, and potential application domains that they have employed in.
6.2 - The captured knowledge in the knowledge base of the SoProDSS is useful for software engineers. It can assist them with their decision-making process to select the best fitting set of alternatives according to their concerns, as declared by the software engineers in our research described in Chapter 7.
6.3 - In Section 2.5.3, 3.5.3, 4.5, 5.4, and 6.5, the case study participants mentioned that the SoProDSS provides an effective shortlist of alternative solutions to help software engineers in their initial decision-making process. In other words, the SoProDSS recommended nearly the same solutions as the case study participants suggested to their companies after extensive analysis and discussions. However, the SoProDSS offers a shortlist of feasible alternatives; therefore, software producing organizations should perform further investigations, such as performance testing, to find the best fitting alternative for their software products.

The case study participants asserted that the updated and validated version of the SoProDSS is useful in finding the shortlist of feasible solutions. Finally, it reduces the time and cost of the decision-making process.
6.4 - The SoProDSS provides a discussion and negotiation platform to enable requirements engineers to make group decisions (see Chapter 4). It detects and highlights the conflicts in the assigned priorities to the domain feature requirements by decision-makers and asks them to resolve disagreements. Thus, the SoProDSS supports software engineers in the requirements verification and validation activity by avoiding conflict between domain feature requirements and generating feasible solutions according to the requirements. Moreover, the SoProDSS can be considered as a communication tool among the decision-makers to facilitate the requirements

specification activity.

Contribution III
The software engineers, who assessed the SoProDSS and the decision models, highlighted that the SoProDSS is a useful tool that can help them explore more alternatives while designing real-world software products. They asserted the SoProDSS itself could assist them in finding liabilities and strength of alternatives, their features, and potential application domains that they have employed in. Moreover, they mentioned that the SoProDSS provides an effective shortlist of alternative solutions to help them in their initial decision-making process. They declared that the updated and validated version of the SoProDSS is useful and valuable in finding the shortlist of feasible solutions. Finally, it reduces the time and cost of the decision-making process.

We formulated six research questions to answer the following main question of the dissertation:

*MRQ — **How can software engineering knowledge be captured and organized systematically to support software engineers with software production decision-making?***

MRQ.1 - We presented the development process of the MCDM framework (See Figure 1.1) in an iterative process. In each cycle of the MCDM framework development process (or in each chapter of the dissertation), we instantiated the framework to build a decision model for a particular MCDM problem in software production. Accordingly, we built and evaluated six decision models for the following decision problems in software production: (1) Database Management System (Chapter 2), (2) Cloud Service Provider (Chapter 3), (3) Blockchain Platform (Chapter 4), (4) Programming Language Ecosystem (Chapter 5), (5) Model-Driven Software Development Platform (Chapter 6), and (6) Architectural Pattern (Chapter 9) selection problems.
MRQ.2 - We showed that the MCDM framework is *a theory for design and action* (Gregor, 2006) for building decision models in the context of MCDM problems in software production. The MCDM framework shows that HOW software engineers can efficiently select the best fitting alternative solutions based on their requirements and priorities. Table 1.2 outlines the MCDM framework as a *theory for design and action* in DSR.
MRQ.3 - We designed and implemented the SoProDSS that integrates key aspects of Knowledge-Driven and Model-Driven DSSs to organize the extracted knowledge regarding decision models systematically, to facilitate the decision-making process, and to support software engineers with their design decisions in software production (see Chapter 8).
MRQ.4- Decision-makers define and prioritize their domain feature requirements based on *MoSCoW* prioritization technique (DSDM consortium and others, 2014), and then send them to the Inference Engine of the SoProDSS. The Inference Engine

infers candidate solutions using the rules and facts of the decision models that it has in its knowledge base. In other words, the Inference Engine excludes infeasible solutions and assigns scores to the feasible ones, and then offers a ranked shortlist of feasible solutions to the decision-makers (see Section 2.5.3, 3.5.3, 4.5, 5.4, and 6.5).

The main contribution

Software engineers should follow the MCDM framework to systematically capture and organize knowledge regarding an MCDM problem in software production. Additionally, they should employ the SoProDSS that we created to define their feature requirements and quality concerns based on the MoSCoW prioritization technique. With the SoProDSS in hand, software engineers make decisions more rapidly and efficiently with a richer set of information than without it.

10.2 Threats to Validity

The validity assessment is an essential part of any empirical study. Validity discussions typically involve Construct Validity, Internal Validity, External Validity, and Conclusion Validity. Other types of validity, such as Theoretical validity and Interpretive validity, are rarely considered in software engineering, so they are not discussed in this dissertation. Table 10.1 shows the tactics that we used to mitigate the threats to the validity of this dissertation.

Construct validity — refers to whether an accurate operational measure or test has been used for the concepts being studied. Developing a theory is an incremental process (Simon, 1996) and requires making correct design decisions. Moreover, the development process involves a continuous and repetitive cycle.

In literature, decision-making is typically defined as a process or a set of ordered activities concerning stages of problem identifying, data collection, defining alternatives, selecting a shortlist of alternatives as feasible solutions with the ranked preferences (Fitzgerald et al., 2017; Kaufmann et al., 2012). Majumder (2015) defines the following six steps of a decision-making process as a multi-criteria decision-making: (1) identifying the objective, (2) selection of the criteria, (3) selection of the alternatives, (4) selection of the weighing method, (5) applying the method of aggregation, and (6) decision-making based on the aggregation results.

In this dissertation, we developed a theoretical framework (MCDM framework) based on the MCDM theory and the six-step of a decision-making process to model decision-making problems in software production (see Chapter 1.6). The MCDM framework contains the following constructs: *Domain* of the problem, *Domain Features*, *Alternatives*, *Software Quality Model* to indicate the impacts of domain features on alternatives, *Decision-Maker*, *MoSCoW* prioritization technique as the weighing method, *Domain Feature Requirements*, the *Weighting Sum Model (WSM)* as the method of aggregation, the *Inference Engine* of the decision support system to suggest feasible solutions, *Ranked Feasible Solutions*.

The framework has instantiated in each cycle of the MCDM framework development process to build a decision model for a particular MCDM problem in software production. We designed and implemented the SoProDSS to organize decision models to facilitate decision-making.

To mitigate the threats to the construct validity, we employed different knowledge acquisition techniques to capture knowledge from domain experts, case studies, literature studies to (re)define the constructs and their relationships in each cycle of the theory development process (see Sections 2.2, 3.2, 4.2, 5.2, 6.3, and 9.2).

The SoProDSS and the decision models have been evaluated through 21 real-world case studies at different software-producing organizations in the Netherlands and Iran. The case study participants asserted that the approach and tooling provide significantly more insight into their selection process, provide a richer prioritized option list than if they had done their research independently, and reduce the time and cost of the decision-making process. However, we also asserted that it is not easy to implement, adopt, and maintain such a system as its knowledge base must be updated regularly. Moreover, software engineers' strong opinions surrounding technology alternatives make it somewhat more complicated to find consensus in the data. We followed the guidelines from Myers & Newman (2007) to conduct 92 qualitative semi-structured interviews with senior software engineers to explore expert knowledge about the decision-making problems, decision models, and the outcomes of our study.

Internal validity — attempts to verify claims about the cause-effect relationships within the context of a study. In other words, it determines whether the study is sound or not.

In this dissertation, we instantiated the MCDM framework to build six decision models, including Database Management System (Chapter 2), Cloud Service Provider (Chapter 3), Blockchain Platform (Chapter 4), Programming Language Ecosystem (Chapter 5), Model-Driven Software Development Platform (Chapter 6), and Architectural Pattern selection problems (Chapter 9). Then, we conducted a set of case studies to evaluate each decision model.

To mitigate the threats to the internal validity of each decision model, we defined the case study results success when they, in part, aligned with the case study participants' shortlist and when they provided new suggestions that were identified as being of interest to the case study participants—using the case study participants' opinion as a measurement instrument was risky, as they may not had sufficient knowledge to make a valid judgment. We countered this risk by conducting more than one case study, assuming that the case study participants were handling in their interest and applying the SoProDSS to other problem domains in software production.

In Chapter 2, 3, 4, 5, 6, and 9, we observed that biases, such as motivational and cognitive (Montibeller & Winterfeldt, 2015), arise because of shortcuts or heuristics that decision-makers use to solve problems and perform tasks. The Hawthorne effect (Jones, 1992), which is the tendency for decision-makers to change their behavior when they are being observed, is a form of cognitive bias. The case study participants might have been more careful in the observational setting than they would be in the real setting because they are being observed by scientists judging their selected domain feature requirements and priorities. Moreover, the Bandwagon effect (Nadeau et al., 1993), which is the tendency to do or believe things because many other decision-makers do or believe the same, is another form of cognitive bias. The Bandwagon effect typically shows up in group decisions. To mitigate the Hawthorne and Bandwagon effects, individual *and* group interviews based on a set of predefined interview protocols have been conducted.

External validity — concerns the domain to which the research findings can be generalized. External validity is sometimes used interchangeably with generalizability (feasibility of applying the results to other research settings). Wieringa & Daneva (2015) state that "the scope of a theory is the set of phenomena to which it is applicable. The scope of a generalization is the set of phenomena for which it is true, and the scope of a model is the set of phenomena to which it can be applied". In other words, a theoretical framework is general if it can be applied to many phenomena. The more general a framework, the larger the set of phenomena to which it is applicable. A theoretical generalization is a form of generalizations that theorizes the findings of a case study according to critical realism consists of postulated constructs and relationships in the real domain (Tsang, 2014).

The majority of the case study participants and domain experts who participated in the research originated from software producing organizations in the Netherlands (see Sections 2.5.3, 3.5.3, 4.5, 5.4, and 6.5). In other words, we evaluated the decision models mainly in the context of Dutch software producing organizations. To mitigate threats to the research's external validity, we captured knowledge from different sources of knowledge without any regional limitations to define the constructs and build the decision models. Accordingly, we hypothesize that the research results can be generalized to all software engineers worldwide who face uncertainty in software production decision-making problems.

Another question is whether the approach and software tools can be applied to other problem domains as well. The problem domains were selected opportunistically and pragmatically, but we are convinced that there are still many decision problems to which the approach can be applied. The categories of problems to which the MCDM approach and toolset can be applied successfully can be summed up as follows: (1) the problem regards a technology decision in system design with long-lasting consequences, (2) there is copious scientific, industry, and informal knowledge publicly available to software engineers, and (3) the (team of) software engineer(s) is not knowledgeable in the field but very knowledgeable about the system requirements.

A challenge for this dissertation is that the qualities and features that we have identified with the support of a limited set of experts can vary wildly with the perception of the expert (see Sections 2.2, 3.2, 4.2, 5.2, 6.3, and 9.2). While we are convinced that the experts have added a significant amount of extra knowledge to the decision models, one might argue we need a large number of experts per technology alternative to reach consensus on each feature. The main objective of expert interviews, as a knowledge acquisition method, is to efficiently and thoroughly extract **rules** and **facts** for the knowledge base of a DSS from domain experts (Hu, 2013). In each cycle of the MCDM framework development process to build a decision model for a particular MCDM problem, interviewing with experts had continued until the extracted knowledge had been converged (data saturation) (Legard et al., 2003). We should also be aware of the strong opinions surrounding technology alternatives, as that makes it somewhat more complicated to find consensus in the data. A potential solution to this validity threat is building a community around the SoProDSS to use feedback from decision-makers to improve its knowledge base.

Conclusion validity — verifies whether the methods of a study such as the data collection method can be reproduced, with similar results. We captured knowledge from the sources of knowledge following the MCDM framework. The accuracy of the extracted knowledge was guaranteed through the protocol that was developed to define the knowledge extraction strategy and format (see Sections 2.2, 3.2, 4.2, 5.2, 6.3, and 9.2). To mitigate the threats to the research's conclusion validity, a set of review protocols were proposed and applied by multiple researchers, including bachelor and master students. We defined a structured coding procedure to keep consistency in the knowledge extraction process and check whether the acquired knowledge addresses the research questions (see Section 7.3.3). Moreover, we crosschecked the captured knowledge to assess the quality of the results, and we had at least two assistants extracting data independently.

Table 10.1: The tactics that we used to mitigate the threats to the dissertation validity.

Tactics	Construct validity	Internal validity	External validity	Conclusion validity
Following the six-step of the decision-making process	✓			
Following the MCDM theory	✓		✓	✓
An extensive literature study	✓		✓	
Conducting 21 case studies	✓	✓	✓	
Interviewing 92 domain experts	✓		✓	
Employing interview protocols		✓		✓
Capturing knowledge without any regional limitations			✓	
Following a structured coding procedure				✓
Quality assessment based on cross-check reviews				✓

Software engineering and design science are two fields where methodological approaches have matured rapidly. In this dissertation, we have followed the guidelines from those fields to ensure that we derived correct conclusions from our work and that the artifacts we created are proven to contribute to the field. In Table 10.1, we show that we diligently defend against common validity threats to strengthen this line of reasoning and hope that this will remain the standard of scrutiny applied in our maturing domain.

10.3 Reflections

In this section, we reflect on the research project in general and, more specifically, on the research process, methods applied, and industry observations.

10.3.1 Reflections on the Research Process

Expert interview

Expert interview is an essential knowledge acquisition technique (Chen, 2004) in qualitative research. We followed the Myers & Newman (2007) guidelines to conduct a series of qualitative semi-structured interviews with senior software engineers to explore expert knowledge regarding the decision-making problems and evaluate the outcomes of our study. Thus, expert interviews are by no means just "data-gathering meetings" employed essentially for capturing knowledge. To clarify any misunderstandings: expert interviews are not only a popular knowledge acquisition technique, but they are also a reliable research method (Bogner et al., 2009) (see Sections 2.2, 3.2, 4.2, 5.2, 6.3, and 9.2).

The first step in conducting expert interviews is identifying the right set of domain experts. Experts are "agents having specific expertise within an organizational or institutional context," who "(re)present solutions to problems and decision-making processes" (Meuser & Nagel, 1989). Experts do not – as in the research by Meuser and Nagel – belong to an organization's management elite, however very often hail from the mid- and lower ranks of organizations. Generally, they are highly educated people who are deeply knowledgeable about their status and accustomed to presenting themselves favorably, tackling challenging situations, and elaborating on complex contexts (Bogner et al., 2009).

During the phase of conducting the expert interview, it is necessary to follow a predefined protocol as a systematic approach to ask questions for specific information related to the aims of the study (Patton, 2015). In other words, the interview questions should be aligned with the research questions. Maxwell (2012) stated that the research questions formulate what we want to understand; the interview questions are what we want to ask experts to gain that understanding. The development of suitable interview questions needs creativity and insight, rather than an automatic translation of the research questions into an interview protocol. It depends basically on the researchers' understanding of the research context and how the interview questions and observational strategies will operate in practice.

Burke & Miller (2001) suggests a well-structured interview guideline, which, apart from open questions, also provides closed ones. They stated that the answers to open questions function as an explanation horizon for the answering behavior with closed questions: "Ensure you have a mix of open-ended and close-ended questions. It is helpful to have some questions where people respond, for example, in a specific Likert scale fashion (that is, close-ended response options), so that you have some easy-to-score data. The open-ended questions will then provide you with the rich filler to elaborate upon such responses."

In this dissertation, each of the interview series followed a semi-structured interview protocol and lasted mainly between 60 and 90 minutes. Acquired knowledge during each interview typically propagated to the next to validate the captured knowledge incrementally. Finally, the findings were sent to the interview participants afterward for final confirmation (see Sections 2.2, 3.2, 4.2, 5.2, 6.3, and 9.2).

Interviewers or scientists who conduct expert interviews should behave as neutral as possible. Still, they cannot always act neutrally in interviews (Bogner et al., 2009), while they must attempt to extract unbiased tacit knowledge of interviewees. It is essential to highlight that, during the expert interviews, we realized that the way of talking and behaving, for example, a proper body language in face-to-face conversations, have significant impacts on the motivation of the interviewees to elaborate on detail and tell more about the context. Therefore, we tried to dress appropriately, avoid prolonged eye contact, be confident and friendly, be aware of our body language, and be prepared for what to say.

Sometimes interviewers are not interested enough to transfer their tacit knowledge, known as the iceberg-effect (Vogel, 1995), caused by a variety of reasons. Perhaps the interviewee is not a "real" expert on the topic studied, or the expert is no longer knowledgeable in the domain. For instance, we faced several situations where the interviewees were full-blooded experts, but the conversations were awkward and tiring, and getting them to talk was tough. Meuser & Nagel (1989) recommended to terminate such discussions. Please note, to a certain degree, this effect is typically happening at the beginning of all interviews, as a situation of trust should be built up first (Bogner et al., 2009). For example, we face the following case in an interview with an expert at a software producing organization: The expert was very suspicious and asked me what I wanted to know exactly and how I wanted to use it afterward. At first, he refused to record his voice, but after my explanation and guarantee to anonymously use the extracted knowledge, he allowed me to do so. In the future, we will use consent forms to gain permission to share, use, or distribute the captured knowledge (anonymously) before conducting the interviews.

Case study

Case study is an empirical research method (Jansen, 2009) that investigates a phenomenon within a particular context in the domain of interest (Yin, 2017). Case studies can describe, explain, and evaluate a hypothesis. Researchers are free to carry out an empirical study in any way, as long as it takes place within a realistic context. A case study can be employed to collect data regarding a particular phenomenon or to apply a tool and then evaluate its efficiency and effectiveness using expert interviews (see Sections 2.2, 3.2, 4.2, 5.2, 6.3, and 9.2).

Planning and scoping a case study research project that addresses research questions appropriately and adequately can be challenging. Additionally, the data collection phase can be time-consuming and tiresome, and typically ends with large amounts of data (Cavaye, 1996). Furthermore, the availability of suitable case study companies may be restricted, as software producing organizations are not always enthusiastic about participating in case study research. The reporting of the results can also be complicated, as the validity of findings should be investigated, and the conclusions reached should be established (Yin, 2017).

The unit of analysis explains what forms a "case", and a complete set of data for one study of the unit of analysis constitutes a single case (Darke et al., 1998). The unit of analysis can be an individual, a group, an organization, or an event or another phenomenon. It is typically defined based on initial research questions and the expected level to address the research questions (Yin, 2017). In other words, the unit of analysis indicates the breadth and depth of the data collection process to answer the research question adequately.

The number of cases to be studied depends on the focus of the research question. As discussed earlier, single cases provide for in-depth investigation and full description. Multiple case designs allow accurate or theoretical replication and cross-case comparison. The right number of cases for qualitative research is not indicated in the literature. Yin (2017) recommends that more replications lead to a higher level of certainty. Eisenhardt (1989) states that between four and ten cases are acceptable for theory building using case study research. Darke et al. (1998) acknowledge that both single- and multiple case designs can be applied in exploratory research. Where explanatory research is undertaken, an individual case may provide the foundation for developing explanations of why a phenomenon occurs, and these may then lead to further investigation by applying them to more cases in other settings. In this dissertation, to evaluate each decision model, we selected multiple cases in the context of different software producing organizations to explore the research questions and theoretical evolution much broader and prevent potential biases.

Theory development process

Based on the findings, we extract the following lessons for young researchers. First, we observe that theory development in design science requires tenacity: the many versions of even the most straightforward theoretical framework indicate that one must not stop developing a theory until a consensus is reached between a team of researchers, no more inconsistencies and gaps are identified, and (parts of) the framework is evaluated. Secondly, we must train young researchers to be accurate in their definitions of concepts: if their primary constructs are inaccurate, the definition relationships become impossible or even erroneous. Thirdly, we must make junior researchers intimately familiar with the research activities needed for process-based theory development, such as literature research, academic discourse, and different theory representations. Finally, we must carefully use the word "theory", as it is used to indicate both a set of untested hypotheses and a well-established, tested, and accepted theory.

10.3.2 Reflection on the Outcomes

The development of software products, systems, and services typically results in complex decision models and decision-making processes (Badampudi et al., 2018). Software engineers need a decision support system in software production as (1) decision problems are often inadequately understood and described, (2) decisions are made at the last moment and under time pressure, (3) decisions are not relying on empirically evaluated decision models, best knowledge and experience and a sound methodology, and (4) decisions are made without considering the perspectives of all the involved stakeholders (Ruhe, 2002).

In this dissertation, we observe that software engineers perform much better and make informed decisions with the right knowledge at the right time (See Sections 2.6, 3.6, 4.6, 5.6, 6.6, 7.4, and 9.5). Generally speaking, decision support systems can support decision-makers to transfer and organize knowledge. Efficient decision support provides software engineers more independence to analyze data and documents to acquire knowledge systematically in terms of facts and results, as they need them.

The complexity of the software engineering process and its socio-technical nature as the main barriers to the adoption of decision support systems (Donzelli, 2006). Additionally, software engineers are, like most intelligent professionals, opinionated, moody, and convinced of their tacit knowledge. In order to mitigate the impacts of tacit expert knowledge, domain interpretation, and of overlearned professional practices, software engineering knowledge needs to be systematically captured and organized when it is required.

From the results of our analysis (see Sections 2.6, 3.6, 4.6, 5.6, 6.6, 7.4, and 9.5), we realized that is a lot of resistance from software engineers to adopt decision support systems because everyone is an expert. In other words, software engineers typically do not want to change their working style and how they are performing tasks. Cummings & Worley (2014) stated that change is often seen as a personal threat by those involved in a transition. Unfamiliarity with the new ways of working can lead to discomfort that spontaneously rises software engineers' resistance (Bridges, 2009).

Adopting new tools also challenge software engineers to advance their skills and knowledge, gain new ones, and learn the tools and techniques – all additional threats to the individual that actively contribute to increasing their resistance to change (Serour & Henderson-Sellers, 2005). Consequently, software engineers can develop a resistance to change their working style, which itself can become a critical obstacle to adopt a DSS to evaluate software engineering design decisions.

The inescapable reality is that people are different and therefore act and react to changes independently. Indeed, occasionally, even the same software engineer can behave in a different manner (Serour & Henderson-Sellers, 2005). Moreover, resistance could be a silent request for assistance, more information, or a statement of different priorities. Resistance can be considered to be an opportunity to gather information and learn more about the current and desired state (Bamberger, 2002). Accordingly, we will continue to introduce and improve the SoProDSS and the decision models, and look for different use cases and potential decision-making problems in the software engineering domain along the road.

In this dissertation, we designed and implemented the SoProDSS that integrates key aspects of Knowledge-Driven and Model-Driven DSSs to store and organize the extracted knowledge regarding decision models systematically, to facilitate the decision-making process, and to support software engineers with their design decisions in software production (see Figure 1.4). The SoProDSS is somewhere between a real product and a proof of concept. We are planning to establish a startup based on the decision-making concepts and the MCDM framework to fill the gap in decision-making approaches between industry and academia and evaluate the SoProDSS in real-world scenarios.

Software engineering is a knowledge-intensive field (Pressman, 2005), and software engineers spend a significant portion of their time collecting data regarding their daily tasks (Meyer et al., 2019). As decision support systems support software engineers with their jobs by facilitating their decision-making processes and recommending design decisions, the principal concepts of DSSs should become an elemental part of software engineering education.

10.3.3 Personal Reflections

Doing a Ph.D. for me was an incredible journey into science and engineering. As a researcher and sometimes as a developer, I have experienced incredible, joyful moments, along with some painful lessons.

I have developed a tool to support the community of people who I care about most, and then I have improved it based on the feedback that I have received from them. I have spent hours discussing the tool's outcomes with the case study participants, domain experts, and my colleagues. I have learned how to present myself and tell the story about my research for different people with different backgrounds.

The most time-consuming and troublesome part of the work was the data collection phase. However, I have always been so excited and curious to see the results and then analyze them. Moreover, finding the right set of domain experts and case study companies was challenging. On average, one-third of the selected experts replied to my emails, and in the end, half of them agreed to participate in the research.

Finally, I believe that my future career choices are influenced by the research I have done and the people I have met during the last four years. Now, I know that I want to work on knowledge engineering and management for the rest of my career.

10.4 Limitations and Future Work

In this final section, we discuss the main limitations of our work and identify some directions for future research.

(1) Automated domain feature extraction — Mapping domain features to alternatives in a decision problem is a time-consuming process, as currently, we analyze potential documents manually. In order to keep the knowledge base of the SoProDSS up-to-date, we need to check the documentation of the alternatives periodically. Traditional methods of feature extraction, such as natural language processing techniques (Bakar et al., 2015; Riloff, 1996), require handcrafted features in a training set. Deep learning enables us to extract feature automatically from big data, instead of adopting handcrafted features, which mainly depends on our prior knowledge and initial hypotheses regarding the feature set.

(2) Sentiment analysis — Mapping non-Boolean domain features to alternatives require analyzing documents and expert opinions regarding alternatives. The data collection phase is a time-consuming process that involves finding a set of parameters to estimate feelings about alternatives. Additionally, the validity of the estimation mainly relies on domain expert opinions that can be biased toward their tacit knowledge. Sentiment analysis (Feldman, 2013; Liu, 2012; Pang & Lee, 2008) can be employed to interpret and classify emotions (positive, negative, and neutral) within the documentation regarding alternatives, such as popularity in the market, using natural language processing techniques.

(3) A community of users — Software producing organizations are under pressures that force them to react instantly to evolving conditions in business environments.

They need to be innovative in how they operate and be agile in making strategic, tactical, and operational decisions. Building a community of users around the SoProDSS lets us be closer to decision-makers in real-time, understand their requirements and preferences, support them quickly with making rational decisions, and use their feedback to improve the SoProDSS.

(4) Knowledge-as-a-Service (KaaS) — Researchers and practitioners in the software engineering field face fundamental challenges introduced by fragmented knowledge from heterogeneous, autonomous sources with complicated and uncertain relations in particular research domains. Additionally, the exponential growth rate of knowledge in a domain surpasses human experts' current ability to formalize and capture tacit and explicit knowledge effectively. Thus, a Knowledge-as-a-Service should be designed and implemented to automate the knowledge acquisition based on artificial intelligence approaches, integrate the captured knowledge, and deliver consistent knowledge to researchers and practitioners (Zhao et al., 2012; Zhao et al., 2019).

(5) Creativity in design decisions — Software production is primarily a complex problem-solving activity, which requires creativity (Glass, 2006). In real-world scenarios, software engineers deal with the following two types of decision problems. (1) routine decision problems that they need to make decisions among a set of standard alternative solutions. (2) non-routine decision problems that they use their tacit knowledge to make creative and unique design decisions. In this dissertation, we mainly focused on routine decision problems and excluded the art creation and human characteristics, including creativity. In order to support software engineers with their design decisions in software production, non-routine decision problems should be investigated and modeled as well (Howard et al., 2008; Maiden et al., 2010; Shneiderman et al., 2006).

Figure 10.3: CherryPickInc is the name of the startup company, that we are planning to establish based on the MCDM concepts of this dissertation.

(6) Valorization — It is necessary to create value the acquired knowledge regarding the decision model in this dissertation and make the SoProDSS suitable and available for commercial and societal use and translating it into a competitive decision support tool and entrepreneurial activity. We are planning to establish a startup based on the decision-making concepts and the MCDM framework to fill the gap in decision-making approaches between industry and academia and evaluate the SoProDSS in real-world scenarios. We believe that the SoProDSS will be significantly improved if it employs in daily practices of software engineers and gain feedback from them. It can then be used as a powerful tool to support software producing organizations with their decision problems in software production.

Bibliography

Ahmad, R., A. Nadeem, T.-h. Kim, et al. (2010), "Isare: an integrated software architecture reuse and evaluation framework", in: *International Conference on Advanced Software Engineering and Its Applications*, Springer, pp. 174–187.

Asadi, M. & R. Ramsin (2008), "Mda-based methodologies: an analytical survey", in: *European Conference on Model Driven Architecture-Foundations and Applications*, Springer, pp. 419–431.

Avgeriou, P., P. Kruchten, P. Lago, P. Grisham & D. Perry (2007), "Sharing and reusing architectural knowledge–architecture, rationale, and design intent", in: *29th International Conference on Software Engineering (ICSE'07 Companion)*, IEEE, pp. 109–110.

Avgeriou, P. & U. Zdun (2005), "Architectural patterns revisited-a pattern language", *European Conference on Pattern Languages of Programs*.

Babar, M. A., R. C. de Boer, T. Dingsoyr & R. Farenhorst (2007a), "Architectural knowlege management strategies: approaches in research and industry", in: *Second Workshop on Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent (SHARK/ADI'07: ICSE Workshops 2007)*, IEEE, pp. 2–2.

Babar, M. A., T. Dingsøyr, P. Lago & H. Van Vliet (2009), *Software architecture knowledge management*, Springer.

Babar, M. A. & P. Lago (2009), "Design decisions and design rationale in software architecture", *Journal of Systems and Software*, vol. 82, no. 8, pp. 1195–1197.

Babar, M. A., D. Winkler & S. Biffl (2007b), "Evaluating the usefulness and ease of use of a groupware tool for the software architecture evaluation process", in: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, IEEE, pp. 430–439.

Badampudi, D., K. Wnuk, C. Wohlin, U. Franke, D. Smite & A. Cicchetti (2018), "A decision-making process-line for selection of software asset origins and components", *Journal of Systems and Software*, vol. 135, pp. 88–104.

Bakar, N. H., Z. M. Kasirun & N. Salleh (2015), "Feature extraction approaches from natural language requirements for reuse in software product lines: a systematic literature review", *Journal of Systems and Software*, vol. 106, pp. 132–149.

Bamberger, J (2002), "Managing resistance–techniques for managing change and improvement", in: *Asia Pacific Software Engineering Process Group (SEPG) Conference Handbook and CD-ROM, Hong Kong, 30pp.*

Baskerville, R. & J. Pries-Heje (2010), “Explanatory design theory”, *Business & Information Systems Engineering*, vol. 2, no. 5, pp. 271–282.

Bass, L., P. Clements & R. Kazman (2013), *Software Architecture in Practice*, Addison Wesley.

Baumeister, J. & A. Striffler (2015), “Knowledge-driven systems for episodic decision support”, *Knowledge-Based Systems*, vol. 88, pp. 45–56.

Baxter, L. A. (2004), “A tale of two voices: relational dialectics theory”, *Journal of Family Communication*, vol. 4, no. 3-4, pp. 181–192.

Beck, K., M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. (2001), “Principles behind the agile manifesto”, *Agile Alliance*, pp. 1–2.

Becker, C., M. Kraxner, M. Plangg & A. Rauber (2013), “Improving decision support for software component selection through systematic cross-referencing and analysis of multiple decision criteria”, in: *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, IEEE, pp. 1193–1202.

Bhattacharya, P. & I. Neamtiu (2011), “Assessing programming language impact on development and maintenance: a study on c and c++”, in: *Proceedings of the 33rd Int. Conference on Software Engineering*, ACM, pp. 171–180.

Bissyandé, T. F., F. Thung, D. Lo, L. Jiang & L. Réveillère (2013), “Popularity, interoperability, and impact of programming languages in 100,000 open source projects”, in: *2013 IEEE 37th annual computer software and applications conference*, IEEE, pp. 303–312.

Blaine, J. D. & J. Cleland-Huang (2008), “Software quality requirements: how to balance competing priorities”, *IEEE Software*, vol. 25, no. 2, pp. 22–24.

Bode, S. & M. Riebisch (2010), “Impact evaluation for quality-oriented architectural decisions regarding evolvability”, in: *European Conference on Software Architecture*, Springer, pp. 182–197.

Bogner, A., B. Littig & W. Menz (2009), *Interviewing experts*, Springer.

Bosch, J. (2004), “Software architecture: the next step”, in: *European Workshop on Software Architecture*, Springer, pp. 194–199.

Bowen, G. A. et al. (2009), “Document analysis as a qualitative research method”, *Qualitative research journal*, vol. 9, no. 2, p. 27.

Brahimi, L., L. Bellatreche & Y. Ouhammou (2016), “A recommender system for dbms selection based on a test data repository”, in: *East European Conference on Advances in Databases and Information Systems*, Springer, pp. 166–180.

Brambilla, M., J. Cabot & M. Wimmer (2017), “Model-driven software engineering in practice”, *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207.

Brereton, P., B. A. Kitchenham, D. Budgen, M. Turner & M. Khalil (2007), “Lessons from applying the systematic literature review process within the software engineering domain”, *Journal of systems and software*, vol. 80, no. 4, pp. 571–583.

Bridges, W. (2009), *Managing transitions: Making the most of change*, Da Capo Press.

Bright, T. J., A. Wong, R. Dhurjati, E. Bristow, L. Bastian, R. R. Coeytaux, G. Samsa, V. Hasselblad, J. W. Williams, M. D. Musty, et al. (2012), “Effect of clinical decision-support systems: a systematic review”, *Annals of internal medicine*, vol. 157, no. 1, pp. 29–43.

Brown, A. W. (2004), "Model driven architecture: principles and practice", *Software and Systems Modeling*, vol. 3, no. 4, pp. 314–327.

Brownsword, L., T. Oberndorf & C. A. Sledge (2000), "Developing new processes for cots-based systems", *IEEE software*, vol. 17, no. 4, pp. 48–55.

Buchgeher, G., R. Weinreich & T. Kriechbaum (2016), "Making the case for centralized software architecture management", in: *International Conference on Software Quality*, Springer, pp. 109–121.

Burge, J. E., J. M. Carroll, R. McCall & I. Mistrik (2008), *Rationale-based software engineering*, Springer.

Burke, L. A. & M. K. Miller (2001), "Phone interviewing as a means of data collection: lessons learned and practical recommendations", in: *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, vol. 2, 2.

Burnstein, I. (2006), *Practical software testing: a process-oriented approach*, Springer Science & Business Media.

Buschmann, F, R Meunier, H Rohnert, P Sommerlad & M Stal (1996), "Pattern-oriented software architecture-a system of patterns", *Advances in software engineering and knowledge engineering*, vol. 1, pp. 1–487.

Buschmann, F., K. Henney & D. C. Schmidt (2007a), *Pattern-oriented software architecture, on patterns and pattern languages*, vol. 5, John wiley & sons.

Buschmann, F., K. Henney & D. C. Schmidt (2007b), "Past, present, and future trends in software patterns", *IEEE software*, vol. 24, no. 4, pp. 31–37.

Büyüközkan, G. & S. Güleryüz (2016), "A new integrated intuitionistic fuzzy group decision making approach for product development partner selection", *Computers & Industrial Engineering*, vol. 102, pp. 383–395.

Buyya, R., C. Vecchiola & S. T. Selvi (2013), *Mastering cloud computing: foundations and applications programming*, Newnes.

Cai, X., M. R. Lyu, K.-F. Wong & R. Ko (2000), "Component-based software engineering: technologies, development frameworks, and quality assurance schemes", in: *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*, IEEE, pp. 372–379.

Capilla, R., A. Jansen, A. Tang, P. Avgeriou & M. A. Babar (2016), "10 years of software architecture knowledge management: practice and future", *Journal of Systems and Software*, vol. 116, pp. 191–205.

Carando, P. (Dec. 1989), "Shadow: fusing hypertext with ai", *IEEE Expert: Intelligent Systems and Their Applications*, vol. 4, no. 4, 65–78.

Carmines, E. G. & R. A. Zeller (1979), *Reliability and validity assessment*, vol. 17, Sage publications.

Carvallo, J. P. & X. Franch (2006), "Extending the iso/iec 9126-1 quality model with non-technical factors for cots components selection", in: *Proceedings of the 2006 international workshop on Software quality*, ACM, pp. 9–14.

Cavaye, A. L. (1996), "Case study research: a multi-faceted research approach for is", *Information systems journal*, vol. 6, no. 3, pp. 227–242.

Cechich, A., M. Piattini & A. Vallecillo (2003), *Component-based software quality: methods and techniques*, vol. 2693, Springer.

Ceri, S., M. Brambilla & P. Fraternali (2009), "The history of webml lessons learned from 10 years of model-driven development of web applications", in: *Conceptual modeling: Foundations and applications*, Springer, pp. 273–292.

Chen, S.-M. (1998), "Aggregating fuzzy opinions in the group decision-making environment", *Cybernetics & Systems*, vol. 29, no. 4, pp. 363–376.

Chen, W. K. (2004), *The electrical engineering handbook*, Elsevier.

Chung, L, B Nixon, E Yu & J Mylopoulos (2000), "Non-functional requirements in software engineering–kluwer academic publishers", *Massachusetts, USA*.

Clements, P., R. Kazman, M. Klein, D. Devesh, S. Reddy & P. Verma (2007), "The duties, skills, and knowledge of software architects", in: *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, IEEE, pp. 20–20.

Clements, P., R. Kazman, M. Klein, et al. (2003), *Evaluating software architectures*, Tsinghua University Press Beijing.

Clements, P. & M. Shaw (2009), "" the golden age of software architecture" revisited", *IEEE software*, vol. 26, no. 4, pp. 70–72.

Cochran, J. K. & H.-N. Chen (2005), "Fuzzy multi-criteria selection of object-oriented simulation software for production system analysis", *Computers & operations research*, vol. 32, no. 1, pp. 153–168.

Committee, S. E. S. et al. (1998), "Ieee standard for software maintenance", *IEEE Std*, pp. 1219–1998.

Cooper, R. & W. Emory (1995), *Business Research Methods 5th et. London, Richard D Irwin*.

Corbin, J. & A. Strauss (2014), *Basics of qualitative research: Techniques and procedures for developing grounded theory*, Sage publications.

Costanza, P., C. Herzeel & W. Verachtert (2019), "A comparison of three programming languages for a full-fledged next-generation sequencing tool", *BMC bioinformatics*, vol. 20, no. 1, p. 301.

Cross, N (1999), "Evidence from protocol and other formal studies of design activity", *Proceedings of Knowing and Learning to Design*, pp. 27–29.

Cross, N. & R. Roy (1989), *Engineering design methods*, vol. 4, Wiley New York.

Cummings, J. (2003), "Knowledge sharing: a review of the literature".

Cummings, T. G. & C. G. Worley (2014), *Organization development and change*, Cengage learning.

Darke, P., G. Shanks & M. Broadbent (1998), "Successfully completing case study research: combining rigour, relevance and pragmatism", *Information systems journal*, vol. 8, no. 4, pp. 273–289.

Davies, I. & M. Reeves (2010), "Bpm tool selection: the case of the queensland court of justice", in: *Handbook on Business Process Management 1*, Springer, pp. 339–360.

Davis, F. D. (1989), "Perceived usefulness, perceived ease of use, and user acceptance of information technology", *MIS quarterly*, pp. 319–340.

De Boer, R. C. & R. Farenhorst (2008), "In search ofarchitectural knowledge"', in: *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*, pp. 71–78.

De Boer, R. C. & H. Van Vliet (2009), "Quont: an ontology for the reuse of quality criteria", in: *2009 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, IEEE, pp. 57–64.

Dean Jr, J. W. & M. P. Sharfman (1993), "Procedural rationality in the strategic decision-making process", *Journal of management Studies*, vol. 30, no. 4, pp. 587–610.

Dear Developers: Coding Languages That Will Set You Apart (2019), `https://hired.com/blog/candidates/data-reveals-hottest-coding-languages/`, Pratini, Napala, (visited on 02/09/2020).

Delgado, A., D. Calegari, P. Milanese, R. Falcon & E. García (2015), "A systematic approach for evaluating bpm systems: case studies on open source and proprietary tools", in: *IFIP International Conference on Open Source Systems*, Springer, pp. 81–90.

Desanctis, G. & R. B. Gallupe (1987), "A foundation for the study of group decision support systems", *Management science*, vol. 33, no. 5, pp. 589–609.

Developer Survey Results (2019), `https://insights.stackoverflow.com/survey/2019`, StackOverflow, (visited on 02/09/2020).

Dhar, V. & R. Stein (1997), *Intelligent Decision Support Methods: The Science of Knowledge Work*, USA: Prentice-Hall, Inc., ISBN: 0135199352.

Dhiman, H. S. & D. Deb (2020), *Decision and Control in Hybrid Wind Farms*, Springer.

Dinh, T. T. A., J. Wang, G. Chen, R. Liu, B. C. Ooi & K.-L. Tan (2017), "Blockbench: a framework for analyzing private blockchains", in: *Proceedings of the 2017 ACM International Conference on Management of Data*, ACM, pp. 1085–1100.

Do You Speak Code? (2019), `https://www.codingame.com/work/resources/codingame-2019-developer-survey/programming-languages/`, codingame, (visited on 02/09/2020).

Dodgson, J. S., M. Spackman, A. Pearman & L. D. Phillips (2009), *Multi-criteria analysis: a manual*, Department for Communities and Local Government: London.

Donzelli, P. (2006), "Decision support system for software project management", *IEEE software*, vol. 23, no. 4, pp. 67–75.

Dougherty, D. & A. Robbins (1997), *sed & awk: UNIX Power Tools*, " O'Reilly Media, Inc."

Doumpos, M. & E. Grigoroudis (2013), "Multicriteria decision aid and artificial intelligence", *Whiley (UK)*.

Doval, D., S. Mancoridis & B. S. Mitchell (1999), "Automatic clustering of software systems using a genetic algorithm", in: *STEP'99. Proceedings Ninth International Workshop Software Technology and Engineering Practice*, IEEE, pp. 73–81.

DSDM consortium and others (2014), "The dsdm agile project framework handbook", *Ashford, Kent, UK: DSDM Consortium*.

Dunie, R., W. Schulte, M Cantara & M. Kerremans (2019), "Magic quadrant for intelligent business process management suites", *Gartner Inc*.

Dutoit, A. H., R. McCall, I. Mistrík & B. Paech (2007), *Rationale management in software engineering*, Springer Science & Business Media.

Dvořák, O., R. Pergl & P. Kroha (2018), "Affordance-driven software assembling", in: *Enterprise Engineering Working Conference*, Springer, pp. 39–54.

Dybå, T. & T. Dingsøyr (2008), "Empirical studies of agile software development: a systematic review", *Information and software technology*, vol. 50, no. 9-10, pp. 833–859.

Eight Top Programming Languages and Frameworks of (2019), `https://hackernoon.com/8-top-programming-languages-frameworks-of-2019-2f08d2d21a1`, hackernoon, (visited on 02/09/2020).

Eisenhardt, K. M. (1989), “Building theories from case study research”, *Academy of management review*, vol. 14, no. 4, pp. 532–550.

Elahi, A. & S. M. Babamir (2015), “Evaluating software architectural styles based on quality features through hierarchical analysis and fuzzy integral (fahp)”, in: *Information and Knowledge Technology*, IEEE, pp. 1–6.

Embley, D. W., S. W. Liddle & O. Pastor (2011), “Conceptual-model programming: a manifesto”, in: *Handbook of Conceptual Modeling*, Springer, pp. 3–16.

Falessi, D., G. Cantone, R. Kazman & P. Kruchten (2011), “Decision-making techniques for software architecture design: a comparative survey”, *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, pp. 1–28.

Farenhorst, R. & H. Van Vliet (2009), “Understanding how to support architects in sharing knowledge”, in: *2009 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, IEEE, pp. 17–24.

Farshidi, S. & S. Jansen (2020a), “A decision support system for pattern-driven software architecture”, in: *Proceedings of the 14th European Conference on Software Architecture, ECSA 2020*, vol. 1, ACM, pp. 1–12.

– (2020b), “Evaluating architect adoption of a decision support tool”, *(Submitted)*.

Farshidi, S., S. Jansen, R. De Jong & S. Brinkkemper (2018a), “A decision support system for cloud service provider selection problems in software producing organizations”, in: *2018 IEEE 20th Conference on Business Informatics (CBI)*, vol. 1, IEEE, pp. 139–148.

– (2018b), “Multiple criteria decision support in requirements negotiation”, in: *the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018)*, vol. 2075, pp. 100–107.

Farshidi, S., S. Jansen & M. Deldar (2020a), “A decision model for programming language ecosystem selection”, *(Submitted)*.

– (2020b), *A Decision Model for Programming LanguageEcosystem Selection: Seven Industry Case Studies*, `http://dx.doi.org/10.17632/5tc6v6zkzf.1`, Utrecht University, (visited on 05/15/2020).

Farshidi, S., S. Jansen, S. España & J. Verkleij (2020c), “Decision support for blockchain platform selection: three industry case studies”, *IEEE Transactions on Engineering Management*.

Farshidi, S., S. Jansen & S. Fortuin (2020d), *Model-Driven Development Platform Selection: Four Industry Case Studies*, `http://dx.doi.org/10.17632/fbg29x5vkk.1`, Utrecht University, (visited on 07/06/2020).

– (2021), “Model-driven development platform selection: four industry case studies”, *Software and Systems Modeling*.

Farshidi, S., S. Jansen, R. de Jong & S. Brinkkemper (2018c), “A decision support system for software technology selection”, *Journal of Decision Systems*.

Farshidi, S., S. Jansen & J. M. van der Werf (2020e), “Capturing software architecture knowledge for pattern-driven design”, *Journal of Systems and Software*.

Fedorowicz, J. (1993), “A technology infrastructure for document-based decision support systems”, in: *Decision support systems (3rd ed.) putting theory into practice*, pp. 125–136.

Feldman, R. (2013), “Techniques and applications for sentiment analysis”, *Communications of the ACM*, vol. 56, no. 4, pp. 82–89.

Feraud, M. & S. Galland (2017), “First comparison of SARL to other agent-programming languages and frameworks”, *Procedia Computer Science*, vol. 109, pp. 1080–1085.

Fischer, G., A. C. Lemke, R. McCall & A. I. Morch (1991), “Making argumentation serve design”, *Human–Computer Interaction*, vol. 6, no. 3-4, pp. 393–419.

Fitzgerald, B. & K.-J. Stol (2014), “Continuous software engineering and beyond: trends and challenges”, in: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pp. 1–9.

Fitzgerald, D. R., S. Mohammed & G. O. Kremer (2017), “Differences in the way we decide: the effect of decision style diversity on process conflict in design teams”, *Personality and Individual Differences*, vol. 104, pp. 339–344.

Floudas, C. A. & P. M. Pardalos (2008), *Encyclopedia of optimization*, Springer Science & Business Media.

Fortus, D., J. Krajcik, R. C. Dershimer, R. W. Marx & R. Mamlok-Naaman (2005), “Design-based science and real-world problem-solving”, *International Journal of Science Education*, vol. 27, no. 7, pp. 855–879.

Forward, A. & T. C. Lethbridge (2008), “A taxonomy of software types to facilitate search and evidence-based software engineering”, in: *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, ACM, p. 14.

Franch, X. & J. P. Carvallo (2003), “Using quality models in software package selection”, *IEEE software*, vol. 20, no. 1, pp. 34–41.

Frauenthaler, P., M. Borkowski & S. Schulte (2019), “A framework for blockchain interoperability and runtime selection”, *arXiv preprint arXiv:1905.07014*.

Frederiksen, N. (1986), “Toward a broader conception of human intelligence.”, *American Psychologist*, vol. 41, no. 4, p. 445.

Fu, L., L. Shi, Y. Yang & B. Yu (2010), “The selection of project management software by FAHP and FMCDM in automobile r&d process”, in: *Networking and Digital Society (ICNDS), 2010 2nd International Conference on*, vol. 1, IEEE, pp. 66–69.

Galster, M., A. Eberlein & M. Moussavi (2010), “Systematic selection of software architecture styles”, *Iet Software*, vol. 4, no. 5, pp. 349–360.

García-Borgoñon, L., M. A. Barcelona, J. A. García-García, M Alba & M. J. Escalona (2014), “Software process modeling languages: a systematic literature review”, *Information and Software Technology*, vol. 56, no. 2, pp. 103–116.

Garg, R., R Sharma & K. Sharma (2017), “Mcdm based evaluation and ranking of commercial off-the-shelf using fuzzy based matrix method”, *Decision Science Letters*, vol. 6, no. 2, pp. 117–136.

Garg, S. K., S. Versteeg & R. Buyya (2011), “Smicloud: a framework for comparing and ranking cloud services”, in: *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, IEEE, pp. 210–218.

Garlan, D. (2014), “Software architecture: a travelogue”, in: *Proceedings of the on Future of Software Engineering*, ACM, pp. 29–39.

Garlan, D. & M. Shaw (1993), “An introduction to software architecture”, in: *Advances in software engineering and knowledge engineering*, World Scientific, pp. 1–39.

Gerrity, T. P. (1971), “Design of man-machine decision systems: an application to portfolio management”, in:

Ghosh, T. (2004), “Creating incentives for knowledge sharing draft”.

Gigerenzer, G. & R. Selten (2002), *Bounded rationality: The adaptive toolbox*, MIT press.

Gigerenzer, G. & P. M. Todd (1999), *Simple heuristics that make us smart*, Oxford University Press, USA.

Gil-Aluja, J. (2013), *Handbook of management under uncertainty*, vol. 55, Springer Science & Business Media.

Glass, R. L. (2006), *Software Creativity 2.0*, developer.* Books.

Godse, M. & S. Mulik (2009), “An approach for selecting software-as-a-service (saas) product”, in: *Cloud Computing, 2009. CLOUD’09. IEEE International Conference on*, IEEE, pp. 155–158.

Graaf, B., M. Lormans & H. Toetenel (2003), “Embedded software engineering: the state of the practice”, *IEEE software*, vol. 20, no. 6, pp. 61–69.

Gregor, S. (2006), “The nature of theory in information systems”, *MIS Quarterly*, vol. 30, no. 3, pp. 611–642.

Gruber, T. R. (1989), “Automated knowledge acquisition for strategic knowledge”, in: *Knowledge Acquisition: Selected Research and Commentary*, Springer, pp. 47–90.

Guo, J., J. White, G. Wang, J. Li & Y. Wang (2011), “A genetic algorithm for optimized feature selection with resource constraints in software product lines”, *Journal of Systems and Software*, vol. 84, no. 12, pp. 2208–2221.

Hailpern, B. & P. Tarr (2006), “Model-driven development: the good, the bad, and the ugly”, *IBM systems journal*, vol. 45, no. 3, pp. 451–461.

Halabi, T. & M. Bellaiche (2017), “Evaluation and selection of cloud security services based on multi-criteria analysis MCA”, in: *Computing, Networking and Communications (ICNC), 2017 International Conference on*, IEEE, pp. 706–710.

Haoues, M., A. Sellami, H. Ben-Abdallah & L. Cheikhi (2017), “A guideline for software architecture selection based on ISO 25010 quality related characteristics”, *Int. Journal of System Assurance Engineering and Management*, vol. 8, no. S2, pp. 886–909.

Haren, V. (2011), “Togaf version 9.1 a pocket guide”.

Harrison, N. B. & P. Avgeriou (2007), “Leveraging architecture patterns to satisfy quality attributes”, in: *European conference on software architecture*, pp. 263–270.

– (2008a), “Analysis of architecture pattern usage in legacy system architecture documentation”, in: *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, IEEE, pp. 147–156.

Harrison, N. B. & P. Avgeriou (2008b), “Incorporating fault tolerance tactics in software architecture patterns”, in: *International Workshop on Software Engineering for Resilient Systems*, New York, New York, USA: ACM Press, p. 9.

Harrison, N. B. & P. Avgeriou (2010), "How do architecture patterns and tactics interact? a model and annotation", *Journal of Systems and Software*, vol. 83, no. 10, pp. 1735–1758.

Heineman, G. T. & W. T. Councill (2001), "Component-based software engineering", *Putting the pieces together, addison-westley*, p. 5.

Hendriks, D., S. Hoppenbrouwers & P van Bommel (2017), "The selection process of model based platforms", MA thesis, the Netherlands: Radboud University Nijmegen.

Hettiarachchi, C. & H. Do (2019), "A systematic requirements and risks-based test case prioritization using a fuzzy expert system", in: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, pp. 374–385.

Hevner, A. R., S. T. March, J. Park & S. Ram (2004), "Design science in information systems research", *MIS quarterly*, pp. 75–105.

– (2008), "Design science in information systems research", *Management Information Systems Quarterly*, vol. 28, no. 1, p. 6.

Hileman, G. & M. Rauchs (2017), *Global blockchain benchmarking study*.

Holsapple, C., S. Shen & A. Whinston (1982), "A consulting system for data base design", *Information Systems*, vol. 7, no. 3, pp. 281 –296.

Holtz, N. M. & W. J. Rasdorf (1988), "An evaluation of programming languages and language features for engineering software development", *Engineering with Computers*, vol. 3, no. 4, pp. 183–199.

Horner, J. & M. E. Atwood (2006), "Effective design rationale: understanding the barriers", in: *Rationale management in software engineering*, Springer, pp. 73–90.

Hospers, J. (1956), "What is explanation?", *Essays in Conceptual Analysis*, pp. 94–119.

Howard, T. J., S. J. Culley & E. Dekoninck (2008), "Describing the creative design process by the integration of engineering design and cognitive psychology literature", *Design studies*, vol. 29, no. 2, pp. 160–180.

Hu, S. D. (2013), *Expert systems for software engineers and managers*, Springer Science & Business Media.

Hussain, S., J. Keung & A. A. Khan (2017), "Software design patterns classification and selection using text categorization approach", *Applied Soft Computing*, vol. 58, pp. 225–244.

Hutchinson, J., J. Whittle & M. Rouncefield (2014), "Model-driven engineering practices in industry: social, organizational and managerial factors that lead to success or failure", *Science of Computer Programming*, vol. 89, pp. 144–161.

Ibriwesh, I., S.-B. Ho & I. Chai (2018), "Overcoming scalability issues in analytic hierarchy process with redccahp: an empirical investigation", *Arabian Journal for Science and Engineering*, vol. 43, no. 12, pp. 7995–8011.

IEEE-SA (2000), *1471-2000-IEEE Recommended Practice for Architectural Description for Software-Intensive Systems*.

İmamoğlu, M. Y. & D. Çetinkaya (2017), "A rule based decision support system for programming language selection", in: *2017 2nd International Conference on Knowledge Engineering and Applications (ICKEA)*, IEEE, pp. 71–75.

ISO (2011), "Iec25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models", *International Organization for Standardization*, vol. 34, p. 2910.

ISO (2011), “Iec/ieee systems and software engineering: architecture description”, *ISO/IEC/IEEE 42010: 2011 (E)(Revision of ISO/IEC 42010: 2007 and IEEE Std 1471-2000)*.

– (2017), *iec/ieee international standard-systems and software engineering–vocabulary*, tech. rep., ISO/IEC/IEEE 24765: 2017 (E).

Jacob, P. M. & P. Mani (2018), “Software architecture pattern selection model for internet of things based systems”, *IET Software*.

Jadhav, A. S. & R. M. Sonar (2011), “Framework for evaluation and selection of the software packages: a hybrid knowledge based system approach”, *Journal of Systems and Software*, vol. 84, no. 8, pp. 1394–1407.

Jansen, A. & J. Bosch (2005), “Software architecture as a set of architectural design decisions”, in: *Working IEEE/IFIP Conference on Software Architecture*, IEEE, pp. 109–120.

Jansen, A., J. Bosch & P. Avgeriou (2008), “Documenting after the fact: recovering architectural design decisions”, *Journal of Systems and Software*, vol. 81, no. 4, pp. 536–557.

Jansen, S., M. Cusumano & K. M. Popp (2019), “Managing software platforms and ecosystems”, *IEEE Software*, vol. 36, no. 3, pp. 17–21.

Jansen, S. (2009), “Applied multi-case research in a mixed-method research project: customer configuration updating improvement”, in: *Information Systems Research Methods, Epistemology, and Applications*, IGI Global, pp. 120–139.

Jansen, S., S. Brinkkemper & A. Finkelstein (2013a), “Business network management as a survival strategy”, in: *Software Ecosystems*, Edward Elgar Publishing, 29—42.

Jansen, S., M. A. Cusumano & S. Brinkkemper (2013b), *Software ecosystems: analyzing and managing business networks in the software industry*, Edward Elgar Publishing.

Johnson, R. B. & A. J. Onwuegbuzie (2004), “Mixed methods research: a research paradigm whose time has come”, *Educational researcher*, vol. 33, no. 7, pp. 14–26.

Jones, S. R. (1992), “Was there a hawthorne effect?”, *American Journal of sociology*, vol. 98, no. 3, pp. 451–468.

Jusoh, Y. Y., K. Chamili, N. Che Pa & J. Yahaya (2014), “Open source software selection using an analytical hierarchy process (ahp)”, *American Journal of Software Engineering and Applications*, vol. 3, no. 6, pp. 83–89.

Kahneman, D., S. P. Slovic, P. Slovic & A. Tversky (1982), *Judgment under uncertainty: Heuristics and biases*, Cambridge university press.

Kamal, A. W. & P. Avgeriou (2010), “Mining relationships between the participants of architectural patterns”, in: *European Conference on Software Architecture*, Springer, pp. 401–408.

Kapteijns, T., S. Jansen, S. Brinkkemper, H. Houët & R. Barendse (2009), “A comparative case study of model driven development vs traditional development: the tortoise or the hare”, *From code centric to model centric software engineering: Practices, Implications and ROI*, vol. 22.

Karsak, E. E. & C. O. Özogul (2009), “An integrated decision making approach for erp system selection”, *Expert systems with Applications*, vol. 36, no. 1, pp. 660–667.

Kaufmann, L., S. Kreft, M. Ehrgott & F. Reimann (2012), “Rationality in supplier selection decisions: the effect of the buyer’s national task environment”, *Journal of Purchasing and Supply Management*, vol. 18, no. 2, pp. 76–91.

Kazman, R., L. Bass, G. Abowd & M. Webb (1994), “Saam: a method for analyzing the properties of software architectures”, in: *Proceedings of 16th International Conference on Software Engineering*, IEEE, pp. 81–90.

Khadka, R., B. V. Batlajery, A. M. Saeidi, S. Jansen & J. Hage (2014), “How do professionals perceive legacy systems and software modernization?”, in: *Proceedings of the 36th International Conference on Software Engineering*, pp. 36–47.

Khan, K. S., G. Ter Riet, J. Glanville, A. J. Sowden, J. Kleijnen, et al. (2001), *Undertaking systematic reviews of research on effectiveness: CRD’s guidance for carrying out or commissioning reviews*, 4 (2n, NHS Centre for Reviews and Dissemination.

Khari, M. & N. Kumar (2013), “Comparison of six prioritization techniques for software requirements”, *Journal of Global Research in Computer Science*, vol. 4, no. 1, pp. 38–43.

Kim, J.-O., O. Ahtola, P. E. Spector, C. W. Mueller, et al. (1978), *Introduction to factor analysis: What it is and how to do it*, vol. 13, Sage.

Kitchenham, B. (2004), “Procedures for performing systematic reviews”, *Keele, UK, Keele University*, vol. 33, no. 2004, pp. 1–26.

Kitchenham, B. A., T. Dyba & M. Jorgensen (2004), “Evidence-based software engineering”, in: *Proceedings of the 26th international conference on software engineering*, IEEE Computer Society, pp. 273–281.

Kochhar, P. S., D. Wijedasa & D. Lo (2016), “A large scale study of multiple programming languages and code quality”, in: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, IEEE, pp. 563–573.

Koens, T. & E. Poll (2018), “What blockchain alternative do you need?”, in: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, Springer, pp. 113–129.

Kohli, R. & S. K. Sehra (2014), “Fuzzy multi criteria approach for selecting software quality model”, *International Journal of Computer Applications*, vol. 98, no. 11.

Kruchten, P. (1999), “The software architect”, in: *Working Conference on Software Architecture*, Springer, pp. 565–583.

– (2006), “What do software architects do”, *Technical Report SEI online report*.

– (2008), “What do software architects really do?”, *Journal of Systems and Software*, vol. 81, no. 12, pp. 2413–2416.

Kruchten, P., H. Obbink & J. Stafford (2006), “The past, present, and future for software architecture”, *IEEE software*, vol. 23, no. 2, pp. 22–30.

Kuo, T.-T., H. Zavaleta Rojas & L. Ohno-Machado (2019), “Comparison of blockchain platforms: a systematic review and healthcare examples”, *Journal of the American Medical Informatics Association*, vol. 26, no. 5, pp. 462–478.

Lago, P. & P. Avgeriou (2006), “First workshop on sharing and reusing architectural knowledge.”, *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 5, pp. 32–36.

Laitenberger, O. & H. M. Dreyer (1998), “Evaluating the usefulness and the ease of use of a web-based inspection data collection tool”, in: *Int. Software Metrics Symposium. Metrics*, IEEE, pp. 122–132.

Lee, J. W. & S. H. Kim (2000), “Using analytic network process and goal programming for interdependent information system project selection”, *Computers & Operations Research*, vol. 27, no. 4, pp. 367–382.

Lee, S. & K.-K. Seo (2016), “A hybrid multi-criteria decision-making model for a cloud service selection problem using bsc, fuzzy delphi method and fuzzy ahp”, *Wireless Personal Communications*, vol. 86, no. 1, pp. 57–75.

Legard, R., J. Keegan & K. Ward (2003), “In-depth interviews”, *Qualitative research practice: A guide for social science students and researchers*, vol. 6, no. 1, pp. 138–169.

Lesani, S. H., B. Rouyendegh & B Erdebilli (2014), “Object-oriented programming language selection using fuzzy AHP method”, in: *annual meeting of the ISAHP*, vol. 29, pp. 1–17.

Li, Y.-F., M. Xie & T. Goh (2009), “A study of mutual information based feature selection for case based reasoning in software cost estimation”, *Expert Systems with Applications*, vol. 36, no. 3, pp. 5921–5931.

Lin, H.-Y., P.-Y. Hsu & G.-J. Sheen (2007), “A fuzzy-based decision-making procedure for data warehouse system selection”, *Expert systems with applications*, vol. 32, no. 3, pp. 939–953.

Little, J. D. (1970), “Models and managers: the concept of a decision calculus”, *Management science*, vol. 16, no. 8, B–466.

Liu, B. (2012), “Sentiment analysis and opinion mining”, *Synthesis lectures on human language technologies*, vol. 5, no. 1, pp. 1–167.

Liu, S., F. T. Chan & W. Ran (2016), “Decision making for the selection of cloud vendor: an improved approach under group decision-making with integrated weights and objective/subjective attributes”, *Expert Systems with Applications*, vol. 55, pp. 37–47.

Look At 5 of the Most Popular Programming Languages (2019), `https://stackify.com/popular-programming-languages-2018/`, PUTANO, BEN, (visited on 02/09/2020).

Macdonald, M, L Liu-Thorrold & R Julien (2017), “The blockchain: a comparison of platforms and their uses beyond bitcoin”, *COMS4507-Adv. Computer and Network Security*.

Maček, D. & D. Alagić (2017), “Comparisons of bitcoin cryptosystem with other common internet transaction systems by AHP technique”, *Journal of Information and Organizational Sciences*, vol. 41, no. 1, pp. 69–87.

Maiden, N., S. Jones, K. Karlsen, R. Neill, K. Zachos & A. Milne (2010), “Requirements engineering as creative problem solving: a research agenda for idea finding”, in: *2010 18th IEEE International Requirements Engineering Conference*, IEEE, pp. 57–66.

Majidi, E., M. Alemi & H. Rashidi (2010), “Software architecture: a survey and classification”, in: *Communication Software and Networks, 2010. ICCSN’10. Second International Conference on*, IEEE, pp. 454–460.

Majumder, M. (2015), “Multi criteria decision making”, in: *Impact of urbanization on water shortage in face of climatic aberrations*, Springer, pp. 35–47.

Malek, S., H. Ramnath Krishnan & J. Srinivasan (2010), “Enhancing middleware support for architecture-based development through compositional weaving of styles”, *Journal of Systems and Software*, vol. 83, no. 12, pp. 2513–2527.

Mannila, L. & M. de Raadt (2006), “An objective comparison of languages for teaching introductory programming”, in: *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pp. 32–37.

Maple, C. & J. Jackson (2018), “Selecting effective blockchain solutions”, in: *European Conference on Parallel Processing*, Springer, pp. 392–403.

Maranville, S. (1992), “Entrepreneurship in the business curriculum”, *Journal of Education for Business*, vol. 68, no. 1, pp. 27–31.

March, S. T. & V. C. Storey (2008), “Design science in the information systems discipline: an introduction to the special issue on design science research”, *MIS quarterly*, vol. 32, no. 4, pp. 725–730.

Marín, B., A. Salinas, J. Morandé, G. Giachetti & J. L. de la Vara (2014), “Main features for mdd tools: an exploratory study”, in: *International Conference on Model-Driven Engineering and Software Development*, Springer, pp. 183–196.

Maxwell, J. A. (2012), *Qualitative research design: An interactive approach*, vol. 41, Sage publications.

Me, G., C. Calero & P. Lago (2016), “A long way to quality-driven pattern-based architecting”, in: *European Conference on Software Architecture*, ed. by Tekinerdogan, B., Zdun, U. & Babar, A., Cham: Springer International Publishing, pp. 39–54.

Medvidovic, N. & R. N. Taylor (2010), “Software architecture: foundations, theory, and practice”, in: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, ACM, pp. 471–472.

Meidan, A., J. A. García-García, M. Escalona & I Ramos (2017), “A survey on business processes management suites”, *Computer Standards & Interfaces*, vol. 51, pp. 71–86.

Mejri, A., S. A. Ghanouchi & R. Martinho (2015), “Evaluation of process modeling paradigms enabling flexibility”, *Procedia Computer Science*, vol. 64, pp. 1043–1050.

Melle, W. van, E. H. Shortliffe & B. G. Buchanan (1984), “EMYCIN: a knowledge engineer’s tool for constructing rule-based expert systems”, *Rule-based expert systems: The MYCIN experiments of the Stanford Heuristic Programming Project*, pp. 302–313.

Melo, C. d. O., J. Moraes, M. Ferreira & R. M. d. C. Figueiredo (2017), “A method for evaluating end-user development technologies”, *ORGANIZATIONAL TRANSFORMATION and INFORMATION SYSTEMS (SIGORSA)*.

Meredith, J. (1993), “Theory building through conceptual methods”, *International Journal of Operations & Production Management*, vol. 13, no. 5, pp. 3–11.

Meredith, J. R., A. Raturi, K. Amoako-Gyampah & B. Kaplan (1989), “Alternative research paradigms in operations”, *Journal of operations management*, vol. 8, no. 4, pp. 297–326.

Meuser, M. & U. Nagel (1989), “Experteninterviews-vielfach erprobt, wenig bedacht: ein beitrag zur qualitativen methodendiskussion”.

Meyer, A., E. T. Barr, C. Bird & T. Zimmermann (2019), “Today was a good day: the daily life of software developers”, *IEEE Transactions on Software Engineering*.

Meyerovich, L. A. & A. S. Rabkin (2013), “Empirical analysis of programming language adoption”, in: *ACM SIGPLAN Notices*, vol. 48, ACM, pp. 1–18.

Mishra, A. R., A. Chandel & D. Motwani (2020), “Extended mabac method based on divergence measures for multi-criteria assessment of programming language with interval-valued intuitionistic fuzzy sets”, *Granular Computing*, vol. 5, no. 1, pp. 97–117.

Moaven, S., J. Habibi, H. Ahmadi & A. Kamandi (2008), “A Decision Support System for Software Architecture-Style Selection”, in: *2008 Sixth International Conference on Software Engineering Research, Management and Applications*, IEEE, IEEE, pp. 213–220, ISBN: 978-0-7695-3302-5, URL: `http://ieeexplore.ieee.org/document/4609428/`.

Mohamed, A. H. (2010), “Facilitating tacit-knowledge acquisition within requirements engineering”, in: *Proceedings of the 10th WSEAS international conference on Applied computer science*, pp. 27–32.

Mohemad, R., A. R. Hamdan, Z. A. Othman & N. M. M. Noor (2010), “Decision support systems (DSS) in construction tendering processes”, *arXiv preprint arXiv:1004.3260*.

Montibeller, G. & D. Winterfeldt (2015), “Cognitive and motivational biases in decision and risk analysis”, *Risk Analysis*, vol. 35, no. 7, pp. 1230–1251.

Morton, M. S. S. (1971), *Management decision systems: computer-based support for decision making*, Division of Research, Graduate School of Business Administration, Harvard ...

Most used programming languages among developers worldwide (2019), `https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/`, statista, (visited on 02/09/2020).

Musen, M. A., B. Middleton & R. A. Greenes (2014), “Clinical decision-support systems”, in: *Biomedical informatics*, Springer, pp. 643–674.

Myers, M. D. & M. Newman (2007), “The qualitative interview in is research: examining the craft”, *Information and organization*, vol. 17, no. 1, pp. 2–26.

Nadeau, R., E. Cloutier & J.-H. Guay (1993), “New evidence about the existence of a bandwagon effect in the opinion formation process”, *International Political Science Review*, vol. 14, no. 2, pp. 203–213.

Nawaz, F., A. Mohsin, S. Fatima & N. K. Janjua (2015), “Rule-based multi-criteria framework for saas application architecture selection”, in: *IFIP Int. Conf. on Artificial Intelligence in Theory and Practice*, Springer, pp. 129–138.

Nickerson, R. S. (1994), “The teaching of thinking and problem solving”, in: *Thinking and problem solving*, Elsevier, pp. 409–449.

Nonaka, I. & H. Takeuchi (1995), *The knowledge-creating company: How Japanese companies create the dynamics of innovation*, Oxford university press.

Nonaka, I. & G. Von Krogh (2009), “Perspective—tacit knowledge and knowledge conversion: controversy and advancement in organizational knowledge creation theory”, *Organization science*, vol. 20, no. 3, pp. 635–652.

Olariu, C., M. Gogan & F. Rennung (2016), “Switching the center of software development from it to business experts using intelligent business process management suites”, in: *Soft Computing Applications*, Springer, pp. 993–1001.

Onut, S. & T. Efendigil (2010), “A theorical model design for erp software selection process under the constraints of cost and quality: a fuzzy approach”, *Journal of Intelligent & Fuzzy Systems*, vol. 21, no. 6, pp. 365–378.

Oztaysi, B. (2014), “A decision model for information technology selection using ahp integrated topsis-grey: the case of content management systems”, *Knowledge-Based Systems*, vol. 70, pp. 44–54.

Pahl, C., N. El Ioini & S. Helmer (2018), “A decision framework for blockchain platforms for IoT and edge computing”, in: *International Conference on Internet of Things, Big Data and Security*.

Pang, B. & L. Lee (2008), “Opinion mining and sentiment analysis”, *Foundations and trends in information retrieval*, vol. 2, no. 1-2, pp. 1–135.

Parker, K. R., J. T. Chao, T. A. Ottaway & J. Chang (2006), “A formal language selection process for introductory programming courses”, *Journal of Information Technology Education: Research*, vol. 5, no. 1, pp. 133–151.

Pastor, O. & J. C. Molina (2007), *Model-driven architecture in practice: a software production environment based on conceptual modeling*, Springer Science & Business Media.

Patton, M. Q. (2015), “Qualitative research and methods: integrating theory and practice”, *Thousand Oaks, CA: SAGE Publications*.

Perkusich, M., L. C. e Silva], A. Costa, F. Ramos, R. Saraiva, A. Freire, E. Dilorenzo, E. Dantas, D. Santos, K. Gorgônio, H. Almeida & A. Perkusich (2020), “Intelligent software engineering in the context of agile software development: a systematic literature review”, *Information and Software Technology*, vol. 119, p. 106241.

Petersen, K., R. Feldt, S. Mujtaba & M. Mattsson (2008), “Systematic mapping studies in software engineering.”, *International Conference on Evaluation and Assessment in Software Engineering*.

Peyton Jones, S., R. Leshchinskiy, G. Keller & M. M. Chakravarty (2008), “Harnessing the multicores: nested data parallelism in haskell”, in: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Phillips Brooks, F. (1995), *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*, Pearson Education India.

Polanyi, M. (1966), *The Tacit Dimension*, London: Routledge and Kegan Paul.

Pour, G. (1998), “Component-based software development approach: new opportunities and challenges”, in: *Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No. 98EX176)*, IEEE, pp. 376–383.

Pour, G., M. Griss & J. Favaro (1999), “Making the transition to component-based enterprise software development: overcoming the obstacles-patterns for success”, in: *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No. PR00275)*, IEEE, pp. 419–419.

Power, D. J. (2000), “Web-based and model-driven decision support systems: concepts and issues”, *AMCIS 2000 Proceedings*, p. 387.

– (2008a), “Decision support systems: a historical overview”, in: *Handbook on decision support systems 1*, Springer, pp. 121–140.

– (2008b), “Understanding data-driven decision support systems”, *Information Systems Management*, vol. 25, no. 2, pp. 149–154.

Power, D. J. & R. Sharda (2007), “Model-driven decision support systems: concepts and research directions”, *Decision Support Systems*, vol. 43, no. 3, pp. 1044–1061.

Pramod Mathew Jacob, a. P. M. (2018), “Software architecture pattern selection model for internet of things based systems”, *IET Software*, vol. 12, 5, 390–396(6).

Pressman, R. S. (2005), *Software engineering: a practitioner’s approach*, Palgrave macmillan.

Programming Languages InfoQ Trends Report (2019), `https://www.infoq.com/articles/programming-language-trends-2019/`, Avram, Abel et al., (visited on 02/09/2020).

PYPL PopularitY of Programming Language (2019), `http://pypl.github.io/PYPL.html`, PYPL, (visited on 02/09/2020).

Qin, Z., X. Zheng & J. Xing (2008), “Architectural styles and patterns”, *Software Architecture*, pp. 34–88.

Ramsey, C. L. & V. R. Basili (1989), “An evaluation of expert systems for software engineering management”, *IEEE Transactions on Software Engineering*, vol. 15, no. 6, pp. 747–759.

Ravasan, A. Z., S. Rouhani & H. Hamidi (2014), “A practical framework for business process management suites selection using fuzzy topsis approach.”, in: *ICEIS (3))*, pp. 295–302.

Ray, B., D. Posnett, V. Filkov & P. Devanbu (2014), “A large scale study of programming languages and code quality in github”, in: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, pp. 155–165.

Razavian, M., A. Tang, R. Capilla & P. Lago (2016), “Reflective approach for software design decision making”, in: *2016 Qualitative Reasoning about Software Architectures (QRASA)*, IEEE, pp. 19–26.

Repschlaeger, J., T. Proehl & R. Zarnekow (2014), “Cloud service management decision support: an application of AHP for provider selection of a cloud-based it service management system”, *Intelligent Decision Technologies*, vol. 8, no. 2, pp. 95–110.

Ribeiro, R. A., A. M. Moreira, P. Van den Broek & A. Pimentel (2011), “Hybrid assessment method for software engineering decisions”, *Decision Support Systems*, vol. 51, no. 1, pp. 208–219.

Richards, M. (2015), *Software architecture patterns*, O’Reilly Media, Incorporated.

Richardson, C. & J. R. Rymer (2016), “Vendor landscape: the fractured, fertile terrain of low-code application platforms”, *FORRESTER, Janeiro*.

Riloff, E. (1996), “An empirical study of automated dictionary construction for information extraction in three domains”, *Artificial intelligence*, vol. 85, no. 1-2, pp. 101–134.

Rodriguez, A., F. Ortega & R. Concepción (2017), “An intuitionistic method for the selection of a risk management approach to information technology projects”, *IS*, vol. 375, pp. 202–218.

Rouhani, S. & A. Z. Ravasan (2015), “Multi-objective model for intelligence evaluation and selection of enterprise systems”, *International Journal of Business Information Systems*, vol. 20, no. 4, pp. 397–426.

Rozanski, N. & E. Woods (2012), *Software systems architecture: working with stakeholders using viewpoints and perspectives*, Addison-Wesley.

Ruhe, G. (2002), "Software engineering decision support–a new paradigm for learning software organizations", in: *International Workshop on Learning Software Organizations*, Springer, pp. 104–113.

Runeson, P. & M. Höst (2009), "Guidelines for conducting and reporting case study research in software engineering", *Empirical software engineering*, vol. 14, no. 2, p. 131.

Runeson, P., M. Host, A. Rainer & B. Regnell (2012), *Case study research in software engineering: Guidelines and examples*, John Wiley & Sons.

Rus, I. & J. S. Collofello (1999), "A decision support system for software reliability engineering strategy selection", in: *Proceedings. Twenty-Third Annual International Computer Software and Applications Conference (Cat. No. 99CB37032)*, IEEE, pp. 376–381.

Rus, I., M. Halling & S. Biffl (2003), "Supporting decision-making in software engineering with process simulation and empirical studies", *International Journal of Software Engineering and Knowledge Engineering*, vol. 13, no. 05, pp. 531–545.

Rymer, J, R. Koplowitz, C Mines, S Sjoblom & C Turley (2019), *The Forrester Wave™: Low-Code Development Platforms For AD&D Professionals*.

Saaty, T. L. (1990), "How to make a decision: the analytic hierarchy process", *European journal of operational research*, vol. 48, no. 1, pp. 9–26.

Saaty, T. L. & L. G. Vargas (2006), *Decision making with the analytic network process*, Springer.

Sabagh, A. A. & A. Al-Yasiri (2011), "An extensible framework for context-aware smart environments", in: *International Conference on Architecture of Computing Systems*, Springer, pp. 98–109.

Sabry, A. E. (2015), "Decision Model for Software Architectural Tactics Selection Based on Quality Attributes Requirements", *Procedia Computer Science*, vol. 65, pp. 422–431.

Sage, A. (1991), *Decision support systems engineering*, Wiley series in systems engineering, J. Wiley, ISBN: 9780471530008.

Sahay, B. & A. Gupta (2003), "Development of software selection criteria for supply chain solutions", *Industrial Management & Data Systems*.

Saldaña, J. (2015), *The coding manual for qualitative researchers*, Sage.

Samadhiya, D., S.-H. Wang & D. Chen (2010), "Quality models: role and value in software engineering", in: *2010 2nd International Conference on Software Technology and Engineering*, vol. 1, IEEE, pp. V1–320.

Sanchis, R., Ó. García-Perales, F. Fraile & R. Poler (2020), "Low-code as enabler of digital transformation in manufacturing industry", *Applied Sciences*, vol. 10, no. 1, p. 12.

Sattar, N. A. (2018), "Selection of low-code platforms based on organization and application type", MA thesis, Lappeenranta University of Technology, Finland: Business and Management.

Schmidt, D. C., M. Stal, H. Rohnert & F. Buschmann (2013), *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, vol. 2, John Wiley & Sons.

Schon, D. A. (1984), *The reflective practitioner: How professionals think in action*, vol. 5126, Basic books.

Seaman, C. B. (1999), "Qualitative methods in empirical studies of software engineering", *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572.

Şen, A. Y., N. Semiz, B. Güneş, D. Algül, Z. Gergin & N. D. Dönmez (2018), "The selection of a process management software with fuzzy topsis multiple criteria decision making method", in: *The International Symposium for Production Research*, Springer, pp. 150–167.

Sendall, S. & W. Kozaczynski (2003), "Model transformation: the heart and soul of model-driven software development", *IEEE software*, vol. 20, no. 5, pp. 42–45.

Serour, M. & B Henderson-Sellers (2005), "Resistance to adoption of an oo software engineering process: an empirical study", in: *European and Mediterranean Conference on Information Systems, EMCIS 2005*.

Sharda, R., S. H. Barr & J. C. MCDonnell (1988), "Decision support system effectiveness: a review and an empirical test", *Management science*, vol. 34, no. 2, pp. 139–159.

Sharma, A., M. Kumar & S. Agarwal (2015), "A complete survey on software architectural styles and patterns", *Procedia Computer Science*, vol. 70, pp. 16–28.

Shaw, M. (1995), "Making choices: a comparison of styles for software architecture", *IEEE Software*, vol. 12, no. 6, pp. 27–41.

Shaw, M. & P. Clements (2006), "The golden age of software architecture", *IEEE software*, vol. 23, no. 2, pp. 31–39.

Shneiderman, B., G. Fischer, M. Czerwinski, M. Resnick, B. Myers, L. Candy, E. Edmonds, M. Eisenberg, E. Giaccardi, T. Hewett, et al. (2006), "Creativity support tools: report from a us national science foundation sponsored workshop", *International Journal of Human-Computer Interaction*, vol. 20, no. 2, pp. 61–77.

Simon, H. A. (1955), "A behavioral model of rational choice", *The quarterly journal of economics*, vol. 69, no. 1, pp. 99–118.

Simon, H. A. (1996), *The Sciences of the Artificial (3rd Ed.)* Cambridge, MA, USA: MIT Press, ISBN: 0-262-69191-4.

Sloman, A. (1985), *Why we need many knowledge representation formalisms*, Citeseer.

Small place to discover languages in GitHub (2014), `https://githut.info/`, GitHut, (visited on 02/09/2020).

Sommerville, I. (1985), *Software Engineering (2nd Ed.)* USA: Addison-Wesley Longman Publishing Co., Inc., ISBN: 0201142295.

Sprague Jr, R. H. & H. J. Watson (1979), "Bit by bit: toward decision support systems", *California Management Review*, vol. 22, no. 1, pp. 60–68.

Staderini, M., E. Schiavone & A. Bondavalli (2018), "A requirements-driven methodology for the proper selection and configuration of blockchains", in: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, IEEE, pp. 201–206.

Staron, M. (2006), "Adopting model driven software development in industry–a case study at two companies", in: *International Conference on Model Driven Engineering Languages and Systems*, Springer, pp. 57–72.

Štemberger, M. I., V. Bosilj-Vukšić & M. I. Jaklić (2009), "Business process management software selection–two case studies", *Economic research-Ekonomska istraživanja*, vol. 22, no. 4, pp. 84–99.

Sujeeth, A. K., H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky & K. Olukotun (2011), “Optiml: an implicitly parallel domain-specific language for machine learning”, in: *ICML*.

Szyperski, C., D. Gruntz & S. Murer (2002), *Component software: beyond object-oriented programming*, Pearson Education.

Tang, A., M. A. Babar, I. Gorton & J. Han (2006), “A survey of architecture design rationale”, *Journal of systems and software*, vol. 79, no. 12, pp. 1792–1804.

Tang, A., T. de Boer & H. van Vliet (2011a), “Building roadmaps: a knowledge sharing perspective”, in: *Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge*, pp. 13–20.

Tang, A., P. Liang & H. Van Vliet (2011b), “Software architecture documentation: the road ahead”, in: *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, IEEE, pp. 252–255.

Tang, H., Y. Shi & P. Dong (2019), “Public blockchain evaluation using entropy and TOPSIS”, *Expert Systems with Applications*, vol. 117, pp. 204–210.

That, M. T. T., S. Sadou, F. Oquendo & I. Borne (2013), “Composition-centered architectural pattern description language”, in: *European Conference on Software Architecture*, Springer, pp. 1–16.

The RedMonk Programming Language Rankings (2019), `https://redmonk.com/sogrady/2019/07/18/language-rankings-6-19/`, O’Grady, Stephen, (visited on 02/09/2020).

The State of Developer Ecosystem (2019), `https://www.jetbrains.com/lp/devecosystem-2019/`, jetbrains, (visited on 02/09/2020).

The Top Programming Languages (2019), `https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019`, Cass, Stephen, (visited on 02/09/2020).

TIOBE Index (2020), `https://www.tiobe.com/tiobe-index/`, TIOBE, (visited on 02/09/2020).

Top Programming Languages Rankings (2019), `https://dzone.com/articles/top-programming-languages-rankings`, Zakrevsky, Alex, (visited on 02/09/2020).

Triantaphyllou, E., B Shu, S. N. Sanchez & T. Ray (1998), “Multi-criteria decision making: an operations research approach”, *Encyclopedia of electrical and electronics engineering*, vol. 15, no. 1998, pp. 175–186.

Tsang, E. W. (2014), “Case studies and generalization in information systems research: a critical realist perspective”, *The Journal of Strategic Information Systems*, vol. 23, no. 2, pp. 174–186.

Tversky, A. & D. Kahneman (1974), “Judgment under uncertainty: heuristics and biases”, *science*, vol. 185, no. 4157, pp. 1124–1131.

– (2000), *Choices, values, and frames*, Cambridge University Press.

Tyree, J. & A. Akerman (2005), “Architecture decisions: demystifying architecture”, *IEEE software*, vol. 22, no. 2, pp. 19–27.

Uzun, B. & B. Tekinerdogan (2018), “Model-driven architecture based testing: a systematic literature review”, *Information and Software Technology*, vol. 102, pp. 30 –48.

Van Den Berk, I., S. Jansen & L. Luinenburg (2010), “Software ecosystems: a software ecosystem strategy assessment model”, in: *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pp. 127–134.

Van Der Aalst, W. M. (2003), “Business process management demystified: a tutorial on models, systems and standards for workflow management”, in: *Advanced Course on Petri Nets*, Springer, pp. 1–65.

Van Strien, P. J. (1997), “Towards a methodology of psychological practice: the regulative cycle”, *Theory & Psychology*, vol. 7, no. 5, pp. 683–700.

Vincent, P., K. Iijima, M. Driver, J. Wong & Y. Natis (2019), “Magic quadrant for enterprise low-code application platforms”, *Gartner Inc.*

Visualizing Language Migration Over Time (2017), `https://www.i-programmer.info/news/98-languages/10943-visualizing-language-migration-over-time.html`, Swift, Janet, (visited on 02/09/2020).

Vogel, B. (1995), “Wenn der eisberg zu schmelzen beginnt–einige reflexionen über den stellenwert und die probleme des experteninterviews in der praxis der empirischen sozialforschung”, *Experteninterviews in der Arbeitsmarktforschung. Diskussionsbeiträge zu methodischen Fragen und praktischen Erfahrungen. Beiträge zur Arbeitsmarkt-und Berufsforschung*, vol. 191, pp. 73–83.

Vogel, O., I. Arnold, A. Chughtai & T. Kehrer (2011), “Architecture Means (WITH WHAT)”, in: *Software Architecture*, Springer Berlin Heidelberg, pp. 115–286.

Vugec, D. S., A.-M. Stjepić & L. Sušac (2019), “Business process management software functionality analysis: supporting social computing and digital transformation”, *ISSN 2671-132X Vol. 1 No. 1 pp. 1-876 June 2019, Zagreb*, p. 547.

Vujošević-Janičić, M. & D. Tošić (2008), “The role of programming paradigms in the first programming courses”, *The Teaching of Mathematics*, vol. 2, no. 21, pp. 63–83.

Vukšić, V. B., L. Brkić & M. Baranović (2016), “Business process management systems selection guidelines: theory and practice”, in: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, IEEE, pp. 1476–1481.

Walls, J. G., G. R. Widmeyer & O. A. El Sawy (1992), “Building an information system design theory for vigilant eis”, *Information systems research*, vol. 3, no. 1, pp. 36–59.

Wang, H. (1997), “Intelligent agent-assisted decision support systems: integration of knowledge discovery, knowledge analysis, and group decision support”, *Expert Systems with Applications*, vol. 12, no. 3, pp. 323–335.

Wang, Y., D. Liu & G. Ruhe (2004), “Formal description of the cognitive process of decision making”, in: *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.* IEEE, pp. 124–130.

Wasilewski, A. (2016), “Business process management suite (bpms) market changes 2009- 2015”, *Information Systems in Management*, vol. 5.

Weinreich, R. & I. Groher (2016), “Software architecture knowledge management approaches and their support for knowledge management activities: a systematic literature review”, *Information and Software Technology*, vol. 80, pp. 265 –286.

What Stats & Surveys Are Saying About Top Programming Languages (2019), `https://codinginfinite.com/top-programming-languages-2020-stats-surveys/`, codinginfinite, (visited on 02/09/2020).

Wiegers, K. & J. Beatty (2013), *Software requirements*, Pearson Education.

Wieringa, R. (2009), “Design science as nested problem solving”, in: *Proceedings of the 4th international conference on design science research in information systems and technology*, ACM, p. 8.

Wieringa, R. & M. Daneva (2015), “Six strategies for generalizing software engineering theories”, *Science of computer programming*, vol. 101, pp. 136–152.

Wong, W. & D. F. Radcliffe (2000), “The tacit nature of design knowledge”, *Technology Analysis & Strategic Management*, vol. 12, no. 4, pp. 493–512.

Wüst, K. & A. Gervais (2018), “Do you need a blockchain?”, in: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, IEEE, pp. 45–54.

Xu, L. & S. Brinkkemper (2007), “Concepts of product software”, *European Journal of Information Systems*, vol. 16, no. 5, pp. 531–541.

Yabo, P. (2016), *Comparison of Cryptocurrency Developments. Key Metrics of Blockchain Platforms*, CoinFabrik, URL: `https://docs.google.com/spreadsheets/d/1DQ770nGnHfJ0oRSqTLmIkhuVK5CAbs-Fgqb6UoGMfVM/edit#gid=0` (visited on 06/14/2016).

Yang, H., S. Zheng, W. C. Chu & C. Tsai (2012), “Linking functions and quality attributes for software evolution”, in: *2012 19th Asia-Pacific Software Engineering Conference*, vol. 1, pp. 250–259.

Yazgan, H. R., S. Boran & K. Goztepe (2009), “An erp software selection process with using artificial neural network based on analytic network process approach”, *Expert systems with applications*, vol. 36, no. 5, pp. 9214–9222.

Yıldızbaşı, A. & B. Daneshvar (2018), “Multi-criteria decision making approach for evaluation of the performance of computer programming languages in higher education”, *Computer Applications In Engineering Education*, vol. 26, no. 6, pp. 1992–2001.

Yin, R. K. (1981), “The case study as a serious research strategy”, *Knowledge*, vol. 3, no. 1, pp. 97–114.

– (2017), *Case study research and applications: Design and methods*, Sage publications.

Yoon, I., J. Kim & W. Lee (2016), “The analysis and application of an educational programming language (RUR-PLE) for a pre-introductory computer science course”, *Cluster Computing*, vol. 19, no. 1, pp. 529–546.

Zhang, H. & M. A. Babar (2010), “On searching relevant studies in software engineering”, in: *14th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pp. 1–10.

Zhang, H., M. A. Babar & P. Tell (2011), “Identifying relevant studies in software engineering”, *Information and Software Technology*, vol. 53, no. 6, pp. 625–637.

Zhao, Z., P. Grosso & C. de Laat (2012), “Oeirm: an open distributed processing based interoperability reference model for e-science”, in: *IFIP International Conference on Network and Parallel Computing*, Springer, pp. 437–444.

Zhao, Z., X. Liao, P. Martin, J. Maduro, P. Thijsse, D. Schaap, M. Stocker, D. Goldfarb & B. Magagna (2019), “Knowledge-as-a-service: a community knowledge base for research infrastructures in environmental and earth sciences”, in: *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642, IEEE, pp. 127–132.

Zhou, X., Y. Jin, H. Zhang, S. Li & X. Huang (2016), “A map of threats to validity of systematic literature reviews in software engineering”, in: *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, pp. 153–160.

Zimmermann, O. (2010), “Architectural decisions as reusable design assets”, *IEEE software*, vol. 28, no. 1, pp. 64–69.

Zolotas, C., K. C. Chatzidimitriou & A. L. Symeonidis (2018), “Restsec: a low-code platform for generating secure by design enterprise services”, *Enterprise Information Systems*, vol. 12, no. 8-9, pp. 1007–1033.

Summary

Decision making is an inevitable part of software engineering. Software engineers make a considerable number of decisions during the software development life cycle. Thus, as a subset of software engineering, software production can be considered a continuous decision-making process. The decision process refers to the steps involved in choosing and evaluating the best fitting alternative solution(s) for software engineers, as decision-makers, according to their preferences and requirements. Additionally, a software product is typically a long-living system to determine the future of the product and the costs associated with its development.

In order to make informed decisions, the decision-makers around a software product should either acquire knowledge themselves or hire external experts to support them with their decision-making process. The process gets more complicated as the number of decision-makers, alternatives, and criteria increases. Therefore, software production is a suitable domain to deploy decision support systems that intelligently support these decision-makers in the decision-making process. A decision model for each decision-making problem is required to externalize and organize knowledge regarding the selection context.

In this dissertation, we focus on pragmatically selected decision-making problems that software engineers face in software production. The following categories of software production decisions are discussed: (1) decision-making regarding COTS components for inclusion into software products. (2) decision problems related to software development technologies that deal with finding the best fitting technologies for developing a software product. (3) architectural design decisions concerning pattern-driven software design.

We developed a theoretical framework to assist software engineers with a set of Multi-Criteria Decision-Making (MCDM) problems in software production. The framework provides a guideline for software engineers to systematically capture knowledge from different knowledge sources to build decision models for MCDM problems in software production. Knowledge has to be collected, organized, and quickly retrieved when it is needed to be employed. We designed, implemented, and evaluated a decision support system (DSS) that utilizes such decision models to facilitate decision-making and support software engineers with their daily MCDM problems.

The framework and the decision support system have been used to model and support software engineers with the following decision-making problems:

1. **COTS component selection problems:**
    - Database Technology Selection
    - Cloud Service Provider Selection
    - Blockchain Platform Selection
2. **Software development technology selection problems:**
    - Programming Language Ecosystem Selection
    - Model-Driven Software Development Platform selection
3. **Decision-Making in Pattern-Driven Design:**
    - Software Architecture Pattern Selection

A broad study has been carried out based on qualitative and quantitative research to evaluate the DSS's efficiency and effectiveness and the decision models inside its knowledge base to support software engineers with their decision-making process in software production. The DSS and the decision models have been evaluated through 19 real-world case studies at different software-producing organizations located in the Netherlands and Iran. The case study participants asserted that the approach and tooling provide significantly more insight into their selection process, provide a richer prioritized option list than if they had done their research independently, and reduce the time and cost of the decision-making process. However, we also asserted that it is not easy to implement, adopt, and maintain such a system as its knowledge base must be updated regularly. Moreover, software engineers' strong opinions surrounding technology alternatives make it somewhat more complicated to find consensus in the data. We conducted 89 qualitative semi-structured interviews with senior software engineers to explore expert knowledge about the decision-making problems, decision models, and the outcomes of our study.

The dissertation concludes that software production decisions are best made with decision support systems but that the steps towards full adoption of such systems are hampered. First, gathering and maintaining appropriate knowledge in a centralized manner is relatively costly and requires more time investment than traditional decision methods. Secondly, software engineers are not used to using such technologies and find it challenging to adopt it into their daily practice.

Samenvatting

Het maken van beslissingen is een essentieel onderdeel van software engineering. Software engineers maken een significant aantal beslissingen tijdens de relatief lange levenscyclus van een software product. Deze beslissingen bepalen het succes van een software product en brengen vaak significante kosten en baten met zich mee. In dit proefschrift wordt het softwareproductieproces geduid als continu beslissingsproces. Het beslissingsproces in software productie wordt gedefinieerd als *de stappen die gevolgd worden bij het kiezen en evalueren van de best passende alternatieve oplossing voor en door software engineers, als besluitvormers, op basis van hun voorkeuren en vereisten*.

Om weloverwogen beslissingen te nemen, moeten de besluitvormers rondom een softwareproduct ofwel zelf kennis verwerven of externe experts inhuren bij het besluitvormingsproces. Het proces wordt ingewikkelder naarmate het aantal besluitvormers, alternatieven, en criteria toeneemt. In dit proefschrift poneren we dat softwareproductie een geschikt domein is voor de inzet van beslissingsondersteunende systemen die op intelligente wijze de software engineer ondersteunen. We onderkennen daarbij ook dat voor elk beslissingsprobleem een beslissingsmodel nodig is om kennis over de selectiecontext te verzamelen, organiseren, en te gebruiken.

In dit proefschrift richten we ons op besluitvormingsproblemen waarmee software engineers worden geconfronteerd bij de productie van software. De volgende categorieën beslissingen in softwareproductie worden besproken: (1) besluitvorming over componenten voor opname in softwareproducten; (2) beslissingsproblemen met betrekking tot software-ontwikkelingstechnologieën; en (3) architectonische ontwerpbeslissingen met betrekking tot patroongestuurd software-ontwerp.

We hebben een theoretisch raamwerk ontwikkeld om software engineers te helpen met een reeks Multi-Criteria Beslissingsproblemen (MCB) bij de productie van software. Het raamwerk maakt het voor software engineers mogelijk om systematisch kennis uit verschillende kennisbronnen te verzamelen voor de bouw van beslissingsmodellen voor MCB in softwareproductie. Kennis moet worden verzameld, geordend, opgeslagen en snel teruggevonden wanneer deze nodig is. We hebben een beslissingsondersteunend systeem ontworpen, geïmplementeerd, en geëvalueerd. Dit systeem neemt als invoer een beslissingsmodel en heeft als doel om de besluitvorming te vergemakkelijken en om software engineers te ondersteunen bij hun dagelijkse MCB.

Het raamwerk en het beslissingsondersteuningssysteem zijn gebruikt om software engineers te ondersteunen bij de volgende besluitvormingsprocessen:

1. **COTS Componentselectie:**
   - Database Technologie Selectie
   - Cloud Service Provider Selectie
   - Blockchain Platform Selectie
2. **Software ontwikkeltechnologie-selectie:**
   - Programmeertaal-ecosysteemselectie
   - Modelgedreven softwareontwikkelingsplatformselectie
3. **Besluitvorming voor patroongedreven ontwerp:**
   - Software Architectuur Patroonselectie

We hebben een studie uitgevoerd op basis van kwalitatief en kwantitatief onderzoek om de efficiëntie en effectiviteit van het beslissingsondersteuningsmodel en het beslissingsondersteuningssysteem te evalueren. Het beslossingsondersteuningsysteem en de beslismodellen zijn geëvalueerd aan de hand van 19 praktijkstudies bij verschillende software producenten in Nederland en Iran. De studiedeelnemers gaven aan dat de aanpak en tooling aanzienlijk meer inzicht geven in hun selectieproces. Daarnaast geeft het beslissingsondersteuningssysteem een rijkere lijst met geprioriteerde opties dan wanneer de deelnemers hun onderzoek onafhankelijk hadden gedaan. Tot slot geven de deelnemers aan dat het beslissingsondersteuningsmodel en -systeem tijd en kosten bespaart in het beslissingsproces.

We hebben echter ook ondervonden dat het niet eenvoudig is om een dergelijk systeem te implementeren, te adopteren, en te onderhouden, aangezien de kennis regelmatig moet worden bijgewerkt. Bovendien maken de sterke meningen van software engineers over technologie-alternatieven het ingewikkelder om consensus in de gegevens te vinden. We hebben 89 kwalitatieve semi-gestructureerde interviews gehouden met senior software engineers om kennis te vergaren over de besluitvormingsproblemen, besluitmodellen, en de resultaten van ons onderzoek.

Het proefschrift concludeert dat beslissingen in softwareproductie het beste worden genomen met beslissingsondersteunende systemen, maar dat de stappen naar volledige adoptie van dergelijke systemen nog worden belemmerd. Ten eerste is het verzamelen en onderhouden van geschikte kennis op een gecentraliseerde manier relatief duur en vergt het meer tijdinvestering dan traditionele besluitvormingsmethoden. Ten tweede zijn software-ingenieurs niet gewend om dergelijke technologieën te gebruiken en vinden ze het een uitdaging om deze in hun dagelijkse praktijk op te nemen. In de toekomst zullen we de het systeem verder ontwikkelen, bijvoorbeeld door de semi-automatische vergaring van kennis via machine learning. Daarnaast willen we werken aan het verder uitbreiden van de set van beslissingsmodellen.

Publication List

Farshidi, S., S. Jansen & J. M. van der Werf (2020e), “Capturing software architecture knowledge for pattern-driven design”, *Journal of Systems and Software*

Farshidi, S., S. Jansen, S. España & J. Verkleij (2020c), “Decision support for blockchain platform selection: three industry case studies”, *IEEE Transactions on Engineering Management*

Farshidi, S., S. Jansen & S. Fortuin (2021), “Model-driven development platform selection: four industry case studies”, *Software and Systems Modeling*

Farshidi, S. & S. Jansen (2020a), “A decision support system for pattern-driven software architecture”, in: *Proceedings of the 14th European Conference on Software Architecture, ECSA 2020,* vol. 1, ACM, pp. 1–12

Farshidi, S., S. Jansen, R. de Jong & S. Brinkkemper (2018c), “A decision support system for software technology selection”, *Journal of Decision Systems*

Farshidi, S., S. Jansen, R. De Jong & S. Brinkkemper (2018a), “A decision support system for cloud service provider selection problems in software producing organizations”, in: *2018 IEEE 20th Conference on Business Informatics (CBI)*, vol. 1, IEEE, pp. 139–148

Farshidi, S., S. Jansen, R. De Jong & S. Brinkkemper (2018b), “Multiple criteria decision support in requirements negotiation”, in: *the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018)*, vol. 2075, pp. 100–107

Curriculum Vitae

Siamak Farshidi was born on August 8th, 1988, in Tehran, Iran. From 2008 to 2001, he studied Software Engineering at Adiban Institute of Higher Education, where he received his Bachelor of Science degree. In the years that followed, he continued with the master of Software Engineering at Shiraz University, which resulted in a Master of Science degree in 2014. He started his Ph.D. research in May 2016 at the Department of Information and Computer Science at Utrecht University, focused on Multi-Criteria Decision-Making problems in software production. During his Ph.D., Siamak Farshidi coordinated and taught courses on data modeling, information systems, and software architecture to students of Business Informatics. Additionally, he established a startup company named CherryPickInc in 2020 based on his dissertation's decision-making concepts. The research and educational activities of Siamak Farshidi focus on Knowledge Engineering, Conceptual Modeling, Decision Support Systems, and Software Architecture.

SIKS Dissertation Series

2011 01 Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models

02 Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language

03 Jan Martijn van der Werf (TUE), Compositional Design and Verification of Component-Based Information Systems

04 Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference

05 Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.

06 Yiwen Wang (TUE), Semantically-Enhanced Recommendations in Cultural Heritage

07 Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction

08 Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues

09 Tim de Jong (OU), Contextualised Mobile Media for Learning

10 Bart Bogaert (UvT), Cloud Content Contention

11 Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective

12 Carmen Bratosin (TUE), Grid Architecture for Distributed Process Mining

13 Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling

14 Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets

15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval

16 Maarten Schadd (UM), Selective Search in Games of Different Complexity

17 Jiyin He (UVA), Exploring Topic Structure: Coherence, Diversity and Relatedness

18 Mark Ponsen (UM), Strategic Decision-Making in complex games

19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles

20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach

21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems

22 Junte Zhang (UVA), System Evaluation of Archival Description and Access

23 Wouter Weerkamp (UVA), Finding People and their Utterances in Social Media

24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior

25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics

26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots

27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns

28 Rianne Kaptein (UVA), Effective Focused Retrieval by Exploiting Query Context and Document Structure

29 Faisal Kamiran (TUE), Discrimination-aware Classification

30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions

31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality

32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science

33 Tom van der Weide (UU), Arguing to Motivate Decisions

34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations

35 Maaike Harbers (UU), Explaining Agent Behavior in Virtual Training

36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach

37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference

38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization

39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games

40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development

41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control

42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution

43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge

44 Boris Reuderink (UT), Robust Brain-Computer Interfaces

45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent
49 Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

---

2012 01 Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
02 Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
03 Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
04 Jurriaan Souer (UU), Development of Content Management System-based Web Applications
05 Marijn Plomp (UU), Maturing Interorganisational Information Systems
06 Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
07 Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
08 Gerben de Vries (UVA), Kernel Methods for Vessel Trajectories
09 Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms
10 David Smits (TUE), Towards a Generic Distributed Adaptive Hypermedia Environment
11 J.C.B. Rantham Prabhakara (TUE), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
12 Kees van der Sluijs (TUE), Model Driven Design and Data Integration in Semantic Web Information Systems
13 Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
14 Evgeny Knutov (TUE), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
15 Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
16 Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
17 Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance
18 Eltjo Poort (VU), Improving Solution Architecting Practices
19 Helen Schonenberg (TUE), What's Next? Operational Support for Business Process Execution
20 Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
21 Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
22 Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
23 Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
24 Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval
25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
26 Emile de Maat (UVA), Making Sense of Legal Text
27 Hayrettin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
28 Nancy Pascall (UvT), Engendering Technology Empowering Women
29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
31 Emily Bagarukayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure
32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning
33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)
34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications
35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics
36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes
37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation
38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms
39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks
40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia
41 Sebastian Kelle (OU), Game Design Patterns for Learning
42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning
43 Withdrawn
44 Anna Tordai (VU), On Combining Alignment Techniques
45 Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions
46 Simon Carter (UVA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation
47 Manos Tsagkias (UVA), Mining Social Media: Tracking Content and Predicting Behavior
48 Jorn Bakker (TUE), Handling Abrupt Changes in Evolving Time-series Data
49 Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions
50 Steven van Kervel (TUD), Ontologogy driven Enterprise Information Systems Engineering
51 Jeroen de Jong (TUD), Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

---

2013 01 Viorel Milea (EUR), News Analytics for Financial Decision Support

02 Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing
03 Szymon Klarman (VU), Reasoning with Contexts in Description Logics
04 Chetan Yadati (TUD), Coordinating autonomous planning and scheduling
05 Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
06 Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
07 Giel van Lankveld (UvT), Quantifying Individual Player Differences
08 Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators
09 Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications
10 Jeewanie Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
11 Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
12 Marian Razavian (VU), Knowledge-driven Migration to Services
13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
14 Jafar Tanha (UVA), Ensemble Approaches to Semi-Supervised Learning Learning
15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval
21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
22 Tom Claassen (RUN), Causal Discovery and Logic
23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning
25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning
27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience
29 Iwan de Kok (UT), Listening Heads
30 Joyce Nakatumba (TUE), Resource-Aware Business Process Management: Analysis and Support
31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications
32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development
33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
36 Than Lam Hoang (TUe), Pattern Mining in Data Streams
37 Dirk Börner (OUN), Ambient Learning Displays
38 Eelco den Heijer (VU), Autonomous Evolutionary Art
39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
41 Jochem Liem (UVA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
43 Marc Bron (UVA), Exploration and Contextualization through Interaction and Concepts

---

2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
02 Fiona Tuliyano (RUN), Combining System Dynamics with a Domain Modeling Method
03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
05 Jurriaan van Reijsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
06 Damian Tamburri (VU), Supporting Networked Software Development
07 Arya Adriansyah (TUE), Aligning Observed and Modeled Behavior
08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data Endpoints
09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
10 Ivan Salvador Razo Zapata (VU), Service Value Networks
11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support
12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
14 Yangyang Shi (TUD), Language Models With Meta-information
15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare
16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability

18 Mattijs Ghijsen (UVA), Methods and Models for the Design and Study of Dynamic Agent Organizations
19 Vinicius Ramos (TUE), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
21 Kassidy Clark (TUD), Negotiation and Monitoring in Open Environments
22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
24 Davide Ceolin (VU), Trusting Semi-structured Web Data
25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
26 Tim Baarslag (TUD), What to Bid and When to Stop
27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
30 Peter de Cock (UvT), Anticipating Criminal Behaviour
31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support
32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data
33 Tesfa Tegegne (RUN), Service Discovery in eHealth
34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.
35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach
36 Joos Buijs (TUE), Flexible Evolutionary Algorithms for Mining Structured Process Models
37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying
38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.
39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital
40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education
41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text
42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models
43 Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments
44 Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.
45 Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach
46 Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity
47 Shangsong Liang (UVA), Fusion and Diversification in Information Retrieval

---

2015 01 Niels Netten (UvA), Machine Learning for Relevance of Information in Crisis Response
02 Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls
03 Twan van Laarhoven (RUN), Machine learning for network data
04 Howard Spoelstra (OUN), Collaborations in Open Learning Environments
05 Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding
06 Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes
07 Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
08 Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
09 Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
10 Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning
11 Yongming Luo (TUE), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
12 Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
13 Giuseppe Procaccianti (VU), Energy-Efficient Software
14 Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
15 Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
16 Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork
17 André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
18 Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
19 Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
20 Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
21 Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning
22 Zhemin Zhu (UT), Co-occurrence Rate Networks
23 Luit Gazendam (VU), Cataloguer Support in Cultural Heritage
24 Richard Berendsen (UVA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection
26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure
27 Sándor Héman (CWI), Updating compressed colomn stores
28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO
29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains
30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning

31 Yakup Koç (TUD), On the robustness of Power Grids
32 Jerome Gard (UL), Corporate Venture Management in SMEs
33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources
34 Victor de Graaf (UT), Gesocial Recommender Systems
35 Jungxao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction

---

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UVA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime

02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation

03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines

04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store

05 Mahdieh Shadi (UVA), Collaboration Behavior

06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search

07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly

08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery

09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text

10 Robby van Delden (UT), (Steering) Interactive Play Behavior

11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment

12 Sander Leemans (TUE), Robust Process Mining with Guarantees

13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology

14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior

15 Peter Berck (RUN), Memory-Based Text Correction

16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines

17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution

18 Ridho Reinanda (UVA), Entity Associations for Search

19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval

20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility

21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)

22 Sara Magliacane (VU), Logics for causal inference under uncertainty

23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces

24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning

25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search

26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch

27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors

28 John Klein (VU), Architecture Practices for Complex Contexts

29 Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"

30 Wilma Latuny (UvT), The Power of Facial Expressions

31 Ben Ruijl (UL), Advances in computational methods for QFT calculations

32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives

33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity

34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics

35 Martine de Vos (VU), Interpreting natural science spreadsheets

36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging

37 Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy

38 Alex Kayal (TUD), Normative Social Applications

39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR

40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems

41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle

42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets

43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval

44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering

45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement

46 Jan Schneider (OU), Sensor-based Learning Support

47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration

48 Angel Suarez (OU), Collaborative inquiry-based learning

2018 01 Han van der Aa (VUA), Comparing and Aligning Process Representations

02 Felix Mannhardt (TUE), Multi-perspective Process Mining

03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction

04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks

05 Hugo Huurdeman (UVA), Supporting the Complex Dynamics of the Information Seeking Process

06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems

07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems

08 Rick Smetsers (RUN), Advances in Model Learning for Software Systems

09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
10 Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
11 Mahdi Sargolzaei (UVA), Enabling Framework for Service-oriented Collaborative Networks
12 Xixi Lu (TUE), Using behavioral context in process mining
13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
14 Bart Joosten (UVT), Detecting Social Signals with Spatiotemporal Gabor Filters
15 Naser Davarzani (UM), Biomarker discovery in heart failure
16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
17 Jianpeng Zhang (TUE), On Graph Sample Clustering
18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
20 Manxia Liu (RUN), Time and Bayesian Networks
21 Aad Slootmaker (OUN), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araujo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TUE), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (UVT), Emotion Recognition from Mandarin Speech
30 Wouter Beek, The "K" in "semantic web" stands for "knowledge": scaling semantics to the web

---

2019 01 Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
03 Eduardo Gonzalez Lopez de Murillas (TUE), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
05 Sebastiaan van Zelst (TUE), Process Mining with Streaming Data
06 Chris Dijkshoorn (VU), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
09 Fahimeh Alizadeh Moghaddam (UVA), Self-adaptation for energy efficiency in software systems
10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
12 Jacqueline Heinerman (VU), Better Together
13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
16 Guangming Li (TUE), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
17 Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
21 Cong Liu (TUE), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
22 Martin van den Berg (VU),Improving IT Decisions with Enterprise Architecture
23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
24 Anca Dumitrache (VU), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
25 Emiel van Miltenburg (VU), Pragmatic factors in (automatic) image description
26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
27 Alessandra Antonaci (OUN), The Gamification Design Process applied to (Massive) Open Online Courses
28 Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
29 Daniel Formolo (VU), Using virtual agents for simulation and training of social skills in safety-critical circumstances
30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
31 Milan Jelisavcic (VU), Alive and Kicking: Baby Steps in Robotics
32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
33 Anil Yaman (TUE), Evolution of Biologically Inspired Learning in Artificial Neural Networks
34 Negar Ahmadi (TUE), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
35 Lisa Facey-Shaw (OUN), Gamification with digital badges in learning programming
36 Kevin Ackermans (OUN), Designing Video-Enhanced Rubrics to Master Complex Skills
37 Jian Fang (TUD), Database Acceleration on FPGAs

	38	Akos Kadar (OUN), Learning visually grounded and multilingual representations
2020	01	Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour
	02	Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
	03	Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
	04	Maarten van Gompel (RUN), Context as Linguistic Bridges
	05	Yulong Pei (TUE), On local and global structure mining
	06	Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
	07	Wim van der Vegt (OUN), Towards a software architecture for reusable game components
	08	Ali Mirsoleimani (UL),Structured Parallel Programming for Monte Carlo Tree Search
	09	Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research
	10	Alifah Syamsiyah (TUE), In-database Preprocessing for Process Mining
	11	Sepideh Mesbah (TUD), Semantic-Enhanced Training Data AugmentationMethods for Long-Tail Entity Recognition Models
	12	Ward van Breda (VU), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment
	13	Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming
	14	Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases
	15	Konstantinos Georgiadis (OUN), Smart CAT: Machine Learning for Configurable Assessments in Serious Games
	16	Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling
	17	Daniele Di Mitri (OUN), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
	18	Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
	19	Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
	20	Albert Hankel (VU), Embedding Green ICT Maturity in Organisations
	21	Karine da Silva Miras de Araujo (VU), Where is the robot?: Life as it could be
	22	Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar
	23	Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
	24	Lenin da Nobrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots
	25	Xin Du (TUE), The Uncertainty in Exceptional Model Mining
	26	Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization
	27	Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
	28	Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality
	29	Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
	30	Bob Zadok Blok (UL), Creatief, Creatieve, Creatiefst
	31	Gongjin Lan (VU), Learning better – From Baby to Better
	32	Jason Rhuggenaath (TUE), Revenue management in online markets: pricing and online advertising
	33	Rick Gilsing (TUE), Supporting service-dominant business model evaluation in the context of business model innovation
	34	Anna Bon (MU), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
2021	01	Francisco Xavier Dos Santos Fonseca (TUD),Location-based Games for Social Interaction in Public Space