The Journal of Systems & Software 169 (2020) 110714

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Capturing software architecture knowledge for pattern-driven design

Siamak Farshidi [a,*], Slinger Jansen [a,b], Jan Martijn van der Werf [a]

[a] Department of Information and Computer Science, Utrecht University, The Netherlands
[b] Visiting Scientist, School of Engineering Science, LUT University, Finland

ARTICLE INFO

Article history:
Received 1 April 2020
Received in revised form 18 June 2020
Accepted 29 June 2020
Available online 3 July 2020

Keywords:
Architectural patterns
Architectural styles
Quality attributes
Design decisions
Knowledge acquisition

ABSTRACT

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.

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 and 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 and 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 and 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.

* Corresponding author.
E-mail addresses: s.farshidi@uu.nl (S. Farshidi), slinger.jansen@uu.nl (S. Jansen), j.m.e.m.vanderwerf@uu.nl (J.M. van der Werf).

[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.

https://doi.org/10.1016/j.jss.2020.110714

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., 2016a). 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 and Al-Yasiri, 2011) and *Scalability* is a liability (Majidi et al., 2010; Galster et al., 2010) of the *Layers* pattern. If an architect is 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 and 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 and 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 and 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 and 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.

2. Background

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 Fig. 1, which is based on the ISO/IEC/IEEE standard 42010 (ISO, 2011a). 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 and 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 1 outlines the definitions of the foundational concepts for the SLR. Please note that many of the definitions were hand-picked from the plethora of definitions available because we needed to make sure that the definitions fit the meta-model in Fig. 1.

2.2. Decision process

Building a software architecture can be regarded as a decision-making process (Lago and 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 and goes through tactics to make sure it works in the context. The decision

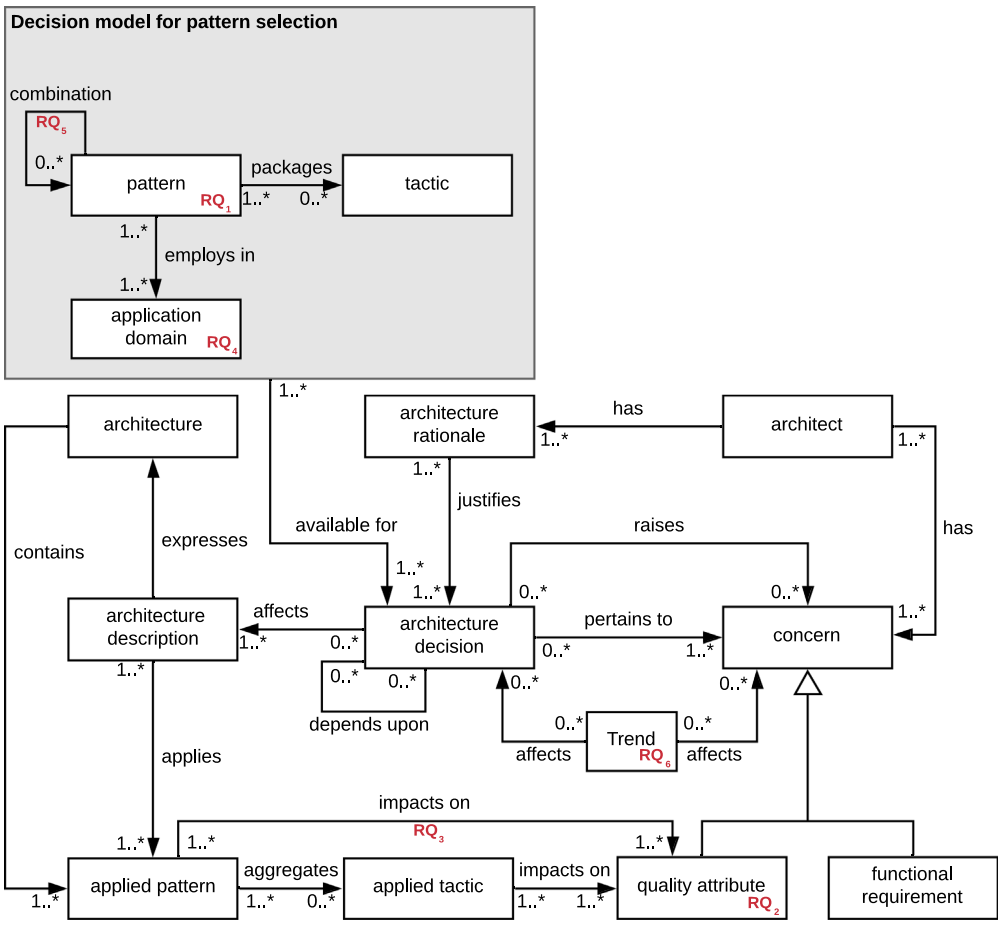

Fig. 1. shows a meta-model, based on the ISO/IEC/IEEE standard 42010 (ISO, 2011a), for decision-making in software architecture. The essential included elements are the architect, the architecture, the knowledge base, and the quality attributes.

model for the pattern selection problem can be used in steps 2 and 3 to facilitate the decision-making process for software architects.

Fig. 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).

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.

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 (Weinreich and Groher, 2016; Uzun and Tekinerdogan, 2018) 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 (Shaw and Clements, 2006; Clements and Shaw, 2009) 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 (Me et al., 2016b; Buchgeher et al., 2016; Tang et al., 2011), 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 and 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

Table 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 and 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 (2011b)
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.	Tang et al. (2006), Horner and Atwood (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 (2011a)

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 and 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 patterns to imply a trendy manner among academics to employ patterns and research them.

Table 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 3.7).

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 (Harrison and Avgeriou, 2008b; Ahmad et al., 2010) expressed that *Performance efficiency* is a key strength of *Client–Server*, however, some other studies (Elahi and Babamir, 2015; Jacob and 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.

Table 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. (2017a)	RP	2017	Survey	Evaluation	Quantitative	3	27	No	No	Yes
Me et al. (2016b)	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 and Riebisch (2010)	RP	2010	Case Study	Evaluation	Mixed	9	15	No	No	Yes
Harrison and 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 and Avgeriou (2007)	RP	2007	Statistics	Evaluation	Quantitative	7	8	No	No	Yes
Avgeriou and Zdun (2005)	RP	2005	Literature	Description	Qualitative	24	10	No	No	Yes
Bushchmann et al. (1996)	Book	1996	Case Study	Description	Qualitative	8	20	Yes	No	Yes
Garlan and Shaw (1994)	Chp	1994	Case Study	Description	Qualitative	6	5	Yes	No	Yes

3. Systematic literature review

Recently, we designed a framework (Farshidi et al., 2018a) 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., 2018a), Cloud Service Providers (Farshidi et al., 2018), and Blockchain Platforms (Farshidi et al., 2020).[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 and Zdun, 2005).

RQ_2: 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., 2016a). 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 and 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 and 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 and 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).

[2] The decision models and modeling studio are available on the DSS website: www.dss.amuse-project.org.

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.

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 Fig. 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”)*

Fig. 2 demonstrates the stages of the search process and the numbers of primary studies in each stage. Moreover, Table 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.

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 Fig. 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 knowledge 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).

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 and Dingsøyr (2008) introduced three main issues (Rigor, Credibility, and Relevance) regarding the quality of primary studies that should be taken into account when assessing primary studies in an SLR. *Rigor* 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 4 shows the result of the quality assessment questions for the primary studies in the SLR.

3.4. Search process

The number of primary studies at each stage of the search process in this paper is presented in Fig. 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 (*7042* publications). At the end of the second step, *2005* publications met the inclusion criteria in the manual search process. Next, by scanning and skimming the text of the selected publications, *493* relevant publications were identified. After that, snowballing was performed

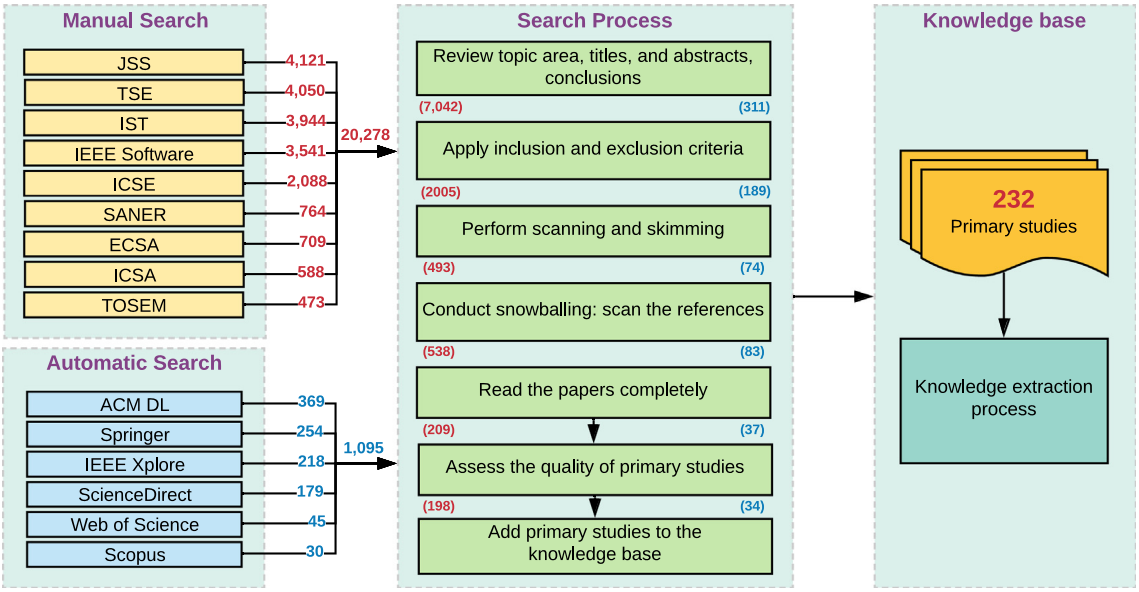

Fig. 2. illustrates the phases of the search process and the number of primary studies in each phase of the SLR. The corresponding number of primary studies in each step of the search process for manual search and automatic search is signified in red and blue, respectively.

Table 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

Table 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

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, *1095* 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.

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.

3.6. 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 Fig. 1), based on the ISO/IEC/IEEE standard 42010 (ISO, 2011a), 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 et al., 2011; Zhang and Babar, 2010) 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.

3.7. Analysis and results

3.7.1. Patterns

Patterns offer universal and reusable solutions to commonly occurring problems in software architecture design (Avgeriou and 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.

Fig. 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). Fig. 3 shows the distribution of these 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 3.7.4). 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. Fig. 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.

3.7.2. 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*.

[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.

[4] A textual definition of each of the patterns is available in the technical report on the following web page: http://swapslr.com.

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
	Patterns																													
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	

Fig. 3. 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.

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., 2016a). Strengths and liabilities assess the importance of the impact of patterns on quality attributes (Harrison and Avgeriou, 2007). Therefore, patterns and quality attributes are not independent and have significant explicit/implicit interactions (Harrison and Avgeriou, 2010). Such interactions can be represented as reusable knowledge elements (Me et al., 2016b). For instance, selecting the *Layers* pattern involves a trade-off between efficiency and maintainability, where the second quality attribute is better fit (Harrison and Avgeriou, 2007).

We tried to identify the most widespread quality concerns that were considered in the literature. Fig. 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 Fig. 4), *Reusability*, *Flexibility*, *Performance efficiency*, *Scalability*, and *Maintainability* are the top five software quality attributes that were investigated and reported on in more than *30* primary studies.

Fig. 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, 2011b), 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.

3.7.3. 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., 2016a). 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

[5] The definitions of the quality attributes are entirely available in the technical report on the following web page: Http://swapslr.com.

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	

Fig. 4. 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. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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 and 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, Sharma et al. (2015), Qin et al. (2008) and Harrison and Avgeriou (2008a) 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 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 $E_i (i = 1, 2, \ldots, n)$ 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:

$$A(E_i) = \frac{1}{n-1} \sum_{j=1 \wedge i \neq j}^{n} S(R_i, R_j); i = 1, 2, \ldots, n$$

The relative degree of agreement $RA(E_i)$ of expert E_i is defined as follows:

$$RA(E_i) = \frac{A(E_i)}{\sum_{i=1}^{n} A(E_i)}; i = 1, 2, \ldots, n$$

Finally, the aggregation of fuzzy opinion is calculated based on the following equation (Chen, 1998):

$$R = RA(E_1) \otimes R_1 \oplus RA(E_2) \otimes R_2 \oplus \cdots \oplus RA(E_n) \otimes R_n$$

Note, in this study, we used Mean of Maxima (MoM) as a method of defuzzification, so that, $MoM(L) = 0.21$, $MoM(N) = 0.50$, and $MoM(H) = 0.79$.

Fig. 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.

3.7.4. 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., 2016a).

Application-generic and application-specific knowledge are two types of architectural knowledge (Lago and 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. Fig. 5 shows the application domains of the identified patterns. We categorized the observed application domains based on the suggested software taxonomy by Forward and Lethbridge (2008).

3.7.5. 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 and 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 and Avgeriou, 2010).

Application domains	Patterns																												# studies
	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	
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	

Fig. 5. 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. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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. 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.

Fig. 6 illustrates the combinations of the pattern that we found during the SLR. The observed combinations in Fig. 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.

3.7.6. 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,

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	

Fig. 6. 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.

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.

Fig. 3 shows the distribution of these 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.

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?

3.8.1. 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 Fig. 3) 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 Fig. 4) 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 Fig. 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 Fig. 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 Fig. 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. Fig. 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.

3.8.2. 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 and 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.

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. 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 Figs. 3 and 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, 2011b), 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 gray 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 and 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 Section 3.7.5 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.

4. Practitioner evaluation

We followed Myers and Newman guidelines (Myers and 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 min.

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 Fig. 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 Fig. 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 (IEE, 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 Fig. 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 and 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, 2011b) 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., 2017b). In Fig. 4, the quality attributes printed in black are based on the ISO/IEC 25010 standard (ISO, 2011b), and the rest of them (printed in blue) are not mentioned in the ISO standard.[6]

Strength and Liabilities: The domain experts asserted that Fig. 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 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 and 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

[6] The definitions of the quality attributes are entirely available in the technical report on the following web page: Http://swapslr.com.

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.

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., 2018a) 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.

CRediT authorship contribution statement

Siamak Farshidi: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization, Supervision, Project administration. **Slinger Jansen:** Conceptualization, Methodology, Validation, Investigation, Writing - original draft, Writing - review & editing, Visualization, Supervision, Project administration, Funding acquisition. **Jan Martijn van der Werf:** Conceptualization, Methodology, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank the twelve experts that participated in this research. Furthermore, we thank the excellent support we have received from the journal editors and reviewers. Finally, we thank Sjaak Brinkkemper for comments on earlier versions of this article. This work is part of AMUSE Project and funded by NWO [project number 628.006.001].

References

Please note that the complete set of the selected primary studies and extracted knowledge is available as a technical report on the following web page: http://swapslr.com.

1998. Ieee standard for software maintenance. IEEE Std 1219-1998 1–56.
Ahmad, R., Nadeem, A., Kim, T.-h., 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.
Avgeriou, P., Zdun, U., 2005. Architectural patterns revisited-a pattern language. In: European Conference on Pattern Languages of Programs.
Bass, L., Clements, P., Kazman, R., 2013. Software Architecture in Practice. Addison Wesley.
Blaine, J.D., Cleland-Huang, J., 2008. Software quality requirements: How to balance competing priorities. IEEE Softw. 25 (2), 22–24.
Bode, S., Riebisch, M., 2010. Impact evaluation for quality-oriented architectural decisions regarding evolvability. In: European Conference on Software Architecture. Springer, pp. 182–197.
Bogner, A., Littig, B., Menz, W., 2009. Introduction: Expert interviews—An introduction to a new methodological debate. In: Interviewing Experts. Springer, pp. 1–13.
Bosch, J., 2004. Software architecture: The next step. In: European Workshop on Software Architecture. Springer, pp. 194–199.
Brereton, P., Kitchenham, B.A., Budgen, D., Turner, M., Khalil, M., 2007. Lessons from applying the systematic literature review process within the software engineering domain. J. Syst. Softw. 80 (4), 571–583.
Buchgeher, G., Weinreich, R., Kriechbaum, T., 2016. Making the case for centralized software architecture management. In: International Conference on Software Quality. Springer, pp. 109–121.
Burnstein, I., 2006. Practical Software Testing: A Process-Oriented Approach. Springer Science & Business Media.
Buschmann, F., Henney, K., Schmidt, D.C., 2007a. Pattern-Oriented Software Architecture, on Patterns and Pattern Languages, Vol. 5. John Wiley & Sons.
Buschmann, F., Henney, K., Schmidt, D.C., 2007b. Past, present, and future trends in software patterns. IEEE Softw. 24 (4), 31–37.
Bushchmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-Oriented Software Architecture-A System of Patterns. In: Advances in Software Engineering and Knowledge Engineering, vol. 1, Wiley and Sons Ltd, pp. 1–487.
Buyya, R., Vecchiola, C., Selvi, S.T., 2013. Principles of parallel and distributed computing. Master. Cloud Comput.
Capilla, R., Jansen, A., Tang, A., Avgeriou, P., Babar, M.A., 2016. 10 years of software architecture knowledge management: Practice and future. J. Syst. Softw. 116, 191–205.

Chen, S.-M., 1998. Aggregating fuzzy opinions in the group decision-making environment. Cybern. Syst. 29 (4), 363–376.
Chung, L., Nixon, B., Yu, E., Mylopoulos, J., 2000. Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, Massachusetts, USA.
Clements, P., Kazman, R., Klein, M., et al., 2003. Evaluating Software Architectures. Tsinghua University Press Beijing.
Clements, P., Shaw, M., 2009. "the golden age of software architecture" revisited. IEEE Softw. 26 (4), 70–72.
Dybå, T., Dingsøyr, T., 2008. Empirical studies of agile software development: A systematic review. Inf. Softw. Technol. 50 (9–10), 833–859.
Elahi, A., Babamir, S.M., 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.
Farshidi, S., Jansen, S., de Jong, R., Brinkkemper, S., 2018a. A decision support system for software technology selection. J. Decis. Syst.
Farshidi, S., Jansen, S., de Jong, R., Brinkkemper, S., 2018b. Multiple criteria decision support in requirements negotiation. In: International Conference on Requirements Engineering: Foundation for Software Quality.
Farshidi, S., Jansen, S., España, S., Verkleij, J., 2020. Decision support for blockchain platform selection: Three industry case studies. IEEE Transactions on Engineering Management 1–20. http://dx.doi.org/10.1109/TEM.2019.2956897.
Farshidi, S., Jansen, S., de Jong, R., Brinkkemper, S., 2018. A decision support system for cloud service provider selection problems in software producing organizations. In: IEEE International Conference on Business Informatics.
Forward, A., Lethbridge, T.C., 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.
Galster, M., Eberlein, A., Moussavi, M., 2010. Systematic selection of software architecture styles. IET Softw. 4 (5), 349–360.
Garlan, D., 2014. Software architecture: a travelogue. In: Proceedings of the on Future of Software Engineering. ACM, pp. 29–39.
Garlan, D., Shaw, M., 1994. An Introduction to Software Architecture. In: Advances in Software Engineering and Knowledge Engineering, vol. 12, pp. 1–39.
Haoues, M., Sellami, A., Ben-Abdallah, H., Cheikhi, L., 2017a. A guideline for software architecture selection based on ISO 25010 quality related characteristics. Int. J. Syst. Assur. Eng. Manag. 8 (S2), 886–909.
Haoues, M., Sellami, A., Ben-Abdallah, H., Cheikhi, L., 2017b. A guideline for software architecture selection based on ISO 25010 quality related characteristics. J. Syst. Assur. Eng. Manag. 8 (2), 886–909.
Harrison, N.B., Avgeriou, P., 2007. Leveraging architecture patterns to satisfy quality attributes. In: European Conference on Software Architecture. pp. 263–270.
Harrison, N.B., Avgeriou, P., 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., Avgeriou, P., 2008b. Incorporating fault tolerance tactics in software architecture patterns. In: International Workshop on Software Engineering for Resilient Systems. ACM Press, New York, New York, USA, p. 9.
Harrison, N.B., Avgeriou, P., 2010. How do architecture patterns and tactics interact? A model and annotation. J. Syst. Softw. 83 (10), 1735–1758.
Horner, J., Atwood, M.E., 2006. Effective design rationale: understanding the barriers. In: Rationale Management in Software Engineering. Springer, pp. 73–90.
Hussain, S., Keung, J., Khan, A.A., 2017. Software design patterns classification and selection using text categorization approach. Appl. Soft Comput. 58, 225–244.
ISO, I., 2011a. IEC/IEEE systems and software engineering: Architecture description. In: ISO/IEC/IEEE 42010: 2011 (E)(Revision of SO/IEC 42010: 2007 and IEEE Std 1471-2000). IEEE New York, NY.
ISO, 2011b. IEC25010: 2011 Systems and Software Engineering–Systems and Software Quality Requirements and Evaluation (SQuaRE)–System and Software Quality Models, Vol. 34. International Organization for Standardization, p. 2910.
ISO, I., 2017. IEC/IEEE International Standard-Systems and Software Engineering–Vocabulary. Technical Report, ISO/IEC/IEEE 24765: 2017 (E).
Jacob, P.M., Mani, P., 2018. Software architecture pattern selection model for Internet of Things based systems. IET Softw.
Jansen, A., Bosch, J., Avgeriou, P., 2008. Documenting after the fact: Recovering architectural design decisions. J. Syst. Softw. 81 (4), 536–557.
Johnson, R.B., Onwuegbuzie, A.J., 2004. Mixed methods research: A research paradigm whose time has come. Educ. Res. 33 (7), 14–26.
Kamal, A.W., Avgeriou, P., 2010. Mining relationships between the participants of architectural patterns. In: European Conference on Software Architecture. Springer, pp. 401–408.
Kazman, R., Bass, L., Abowd, G., Webb, M., 1994. SAAM: A method for analyzing the properties of software architectures. In: Proceedings of 16th International Conference on Software Engineering. IEEE, pp. 81–90.
Khan, K.S., Ter Riet, G., Glanville, J., Sowden, A.J., Kleijnen, J., et al., 2001. Undertaking Systematic Reviews of Research on Effectiveness: CRD's Guidance for Carrying Out or Commissioning Reviews, No. 4. NHS Centre for Reviews and Dissemination, 2n.
Kitchenham, B., 2004. Procedures for Performing Systematic Reviews, Vol. 33, No. 2004. Keele University, Keele, UK, pp. 1–26.
Kitchenham, B.A., Dyba, T., Jorgensen, M., 2004. Evidence-based software engineering. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, pp. 273–281.
Kruchten, P., 1999. The software architect. In: Working Conference on Software Architecture. Springer, pp. 565–583.
Kruchten, P., 2008. What do software architects really do? J. Syst. Softw. 81 (12), 2413–2416.
Kruchten, P., Obbink, H., Stafford, J., 2006. The past, present, and future for software architecture. IEEE Softw. 23 (2), 22–30.
Lago, P., Avgeriou, P., 2006. First workshop on sharing and reusing architectural knowledge. ACM SIGSOFT Softw. Eng. Notes 31 (5), 32–36.
Majidi, E., Alemi, M., Rashidi, H., 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 Singapore, Singapore, pp. 35–47.
Malek, S., Ramnath Krishnan, H., Srinivasan, J., 2010. Enhancing middleware support for architecture-based development through compositional weaving of styles. J. Syst. Softw. 83 (12), 2513–2527.
Me, G., Calero, C., Lago, P., 2016a. A long way to quality-driven pattern-based architecting. In: Tekinerdogan, B., Zdun, U., Babar, A. (Eds.), Software Architecture. Springer International Publishing, Cham, pp. 39–54.
Me, G., Calero, C., Lago, P., 2016b. A long way to quality-driven pattern-based architecting. In: European Conference on Software Architecture. pp. 39–54.
Montibeller, G., Winterfeldt, D., 2015. Cognitive and motivational biases in decision and risk analysis. Risk Anal. 35 (7), 1230–1251.
Myers, M.D., Newman, M., 2007. The qualitative interview in IS research: Examining the craft. Inf. Organ. 17 (1), 2–26.
Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M., 2008. Systematic mapping studies in software engineering. In: International Conference on Evaluation and Assessment in Software Engineering.
Pramod Mathew Jacob, a.P.M., 2018. Software architecture pattern selection model for Internet of Things based systems. IET Softw. 12 (6), 390–396.
Qin, Z., Zheng, X., Xing, J., 2008. Architectural styles and patterns. Softw. Archit. 34–88.
Richards, M., 2015. Software architecture patterns. O'Reilly Media, Incorporated.
Rozanski, N., Woods, E., 2012. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley.
Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14 (2), 131.
Runeson, P., Host, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples. John Wiley & Sons.
Sabagh, A.A., Al-Yasiri, A., 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 Comput. Sci. 65, 422–431.
Saldaña, J., 2015. The Coding Manual for Qualitative Researchers. Sage.
Samadhiya, D., Wang, S.-H., Chen, D., 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.
Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F., 2013. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Vol. 2. John Wiley & Sons.
Seaman, C.B., 1999. Qualitative methods in empirical studies of software engineering. IEEE Trans. Softw. Eng. 25 (4), 557–572.
Sharma, A., Kumar, M., Agarwal, S., 2015. A complete survey on software architectural styles and patterns. Procedia Comput. Sci. 70, 16–28.

Shaw, M., 1995. Making choices: A comparison of styles for software architecture. IEEE Softw. 12 (6), 27–41.
Shaw, M., Clements, P., 2006. The golden age of software architecture. IEEE Softw. 23 (2), 31–39.
Tang, A., Babar, M.A., Gorton, I., Han, J., 2006. A survey of architecture design rationale. J. Syst. Softw. 79 (12), 1792–1804.
Tang, A., Liang, P., Van Vliet, H., 2011. Software architecture documentation: The road ahead. In: 2011 Ninth Working IEEE/IFIP Conference on Software Architecture. IEEE, pp. 252–255.
That, M.T.T., Sadou, S., Oquendo, F., Borne, I., 2013. Composition-centered architectural pattern description language. In: European Conference on Software Architecture. Springer, pp. 1–16.
Triantaphyllou, E., Shu, B., Sanchez, S.N., Ray, T., 1998. Multi-criteria decision making: an operations research approach. Encyclopedia Electr. Electron. Eng. 15 (1998), 175–186.
Tyree, J., Akerman, A., 2005. Architecture decisions: Demystifying architecture. IEEE Softw. 22 (2), 19–27.
Uzun, B., Tekinerdogan, B., 2018. Model-driven architecture based testing: A systematic literature review. Inf. Softw. Technol. 102, 30–48.
Vogel, O., Arnold, I., Chughtai, A., Kehrer, T., 2011. Architecture means (WITH WHAT). In: Software Architecture. Springer Berlin Heidelberg, pp. 115–286.
Weinreich, R., Groher, I., 2016. Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review. Inf. Softw. Technol. 80, 265–286.
Wiegers, K., Beatty, J., 2013. Software Requirements. Pearson Education.
Yang, H., Zheng, S., Chu, W.C., Tsai, C., 2012. Linking functions and quality attributes for software evolution. In: 2012 19th Asia-Pacific Software Engineering Conference, Vol. 1. pp. 250–259.
Zhang, H., Babar, M.A., 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., Babar, M.A., Tell, P., 2011. Identifying relevant studies in software engineering. Inf. Softw. Technol. 53 (6), 625–637.
Zhou, X., Jin, Y., Zhang, H., Li, S., Huang, X., 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.

Siamak Farshidi is a Ph.D. candidate at the Department of Information and Computer Science at Utrecht University. His research focuses on Decision Support Systems for Multi-Criteria Decision-Making (MCDM) problems in software production.

Slinger Jansen is an assistant professor at the Department of Information and Computer Science at Utrecht University. His research focuses on software product management and software ecosystems, with a strong entrepreneurial component. Jansen received his Ph.D. in computer science from Utrecht University, based on the work entitled "Customer Configuration Updating in a Software Supply Network".

Jan Martijn van der Werf is an assistant professor at the Department of Computer Science at Utrecht University. His research focuses on modeling and analyzing behavior in software architecture of software products. Van der Werf received his Ph.D. from two universities: the Eindhoven University of Technology and the Humboldt Universität zu Berlin, on the dissertation entitled "Compositional Design and Verification of Component-based Information Systems".