All-inclusive Software Architecture Video Wall

Building a Prototype

Samantha van Dalen

Sjaak Brinkkemper

Jan Martijn van der Werf

Technical Report UU-CS-2017- 017
September 2017

Department of Information and Computing Sciences

Utrecht University

www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences

Utrecht University

P.O. Box 80.089

3508 TB Utrecht

The Netherlands

All-inclusive Software Architecture Video Wall
Building a Prototype

Samantha van Dalen, Sjaak Brinkkemper, Jan Martijn van der Werf

Utrecht University, Utrecht, The Netherlands
{S.vanDalen,S.Brinkkemper,J.M.E.M.vanderWerf}@uu.nl

Abstract. This technical report describes a study of software architecture models in literature, to combine their Viewpoints into one overview matrix (Viewpoint Matrix), connecting similar Viewpoints of different authors to each other. This report also describes the development of a prototype of a software architecting tool that incorporates the Viewpoint Matrix.

Keywords: Video Wall, Software Architecture, Viewpoint Matrix, Viewpoint, View.

Content

1 Introduction

Software architecture entails many different practices and methods. And, when done appropriately, it contains many different views of software. It remains a craft to know which views are required, and what the interplay between the different views is. Architecture erosion is therefore quickly on the lure. Software producing organizations therefore often cannot see the wood for the trees, meaning that due to the plethora of details, one does not know what is what anymore. To develop an application that appeals to all software architects practicing different architecting methods, a literature study is conducted to find these methods and combine them to one.

In this report, we describe the research methods followed in this project, including a short description about the context of the project. Next, the Viewpoint Matrix is explained, followed by the design decisions made. The paper ends with a discussion and an outlook on the future application and possible research topics.

2 Research method

To define the research method, we use the Design Science method described by Hevner et al. (2004) and Peffers et al. (2007).

2.1 Context

This project is part of the development of a main application for software architecting on a multi-touch video wall. This application has a multi-purpose:

1. It aid software architects to develop and maintain architectural models (hereafter named models) by giving the architects the means to draw and alter their models.
2. In discussions it is possible to alter and annotate the models to capture and document what is discussed in the session.
3. Visualization and management of software requirements.
4. Integration of the software requirements with the models.

In the future, the integration of the above mentioned purposes will result in an application that enables continuous architecting. Continuous architecting is a visionary term that embodies decision making that is performed on an up-to-date software architecture document. Research by De Feijter (de Feijter, 2017; van der Werf, de Feijter, Bex, & Brinkkemper, 2017) discusses how to envision continuous architecting in an application. The main point in this research explains how a continuous architecting tool can be used to capture the discussion between stakeholders by describing several scenarios. This research is also the foundation of this project. In this paper, we consider the main application to consist of two separate parts: Visualization of User Stories and Software Architecture Modelling.

2.2 Objectives

The main goal for this project is twofold:

1. A literature study to find differences and similarities between different architecting methods;
2. To build an interactive prototype that showcases the potential of the software architecture modelling side of the application.

The outcome of the literature study is input for the prototype. It will be a HTML website, supported by JavaScript to perform certain functionalities. The prototype will have a basic layout, static information and basic interactive functionalities.

2.3 Design and development

As mentioned in the previous section, the project consists of two parts: developing a prototype and a literature research. In this section, we describe shortly the design and development method of the prototype and of the literature research.

2.3.1 Prototype

The prototype is designed as a web-based tool. The layout and requirements are mostly based on a previous research by De Feijter (van der Werf, de Feijter, Bex, & Brinkkemper, 2017). In this research, PowerPoint was used to showcase the vision of the application. Combined with a fresh perspective for a layout, our prototype is gradually built from scratch. The development of the prototype is an iterative process. With each iteration a new version is made with new functionalities. In chapter 0, the design decisions for this prototype are explained.

2.3.2 Literature research

To find the differences and similarities of different architecting methods, several books and papers are compared. The books are selected from the body of knowledge in Software Architecture. In addition, an online search has been performed using the term 'software architecture model'. With restricted availability of books and time, we selected a top five most referenced books and papers:

1. Bass, Clements and Kazman (2013) identified four different views: Module, Component-and-Connector, Allocation, and Quality Views. These views are based on Clements, et al. (2001), with the addition of the Quality View. For these views ten different structures are identified, for example a concurrency, class (or generalization) and implementation structure.
2. Rozanski and Woods (2012) identified seven different viewpoints: Context, Functional, Information, Concurrency, Development, Deployment, and Operational. They also presented the most important models per view. This resulted in 19 models, for example a functional structure model, a state model and a migration model.
3. Taylor, Medvidović and Dashofy (2010) briefly mentioned five viewpoints: Logical, Physical, Deployment, Concurrency, and Behavioral. They described ten different modelling techniques, including Natural Language, UML and several Architecting Design Languages (ADLs).
4. Kruchten (1995) identifies in his 4+1 paper four views that are brought together with Scenarios. The views are: Logical, Process, Development and Physical. Per view a notation is presented.
5. Maier and Rechtin (2000) identified six different views: Purpose/objective, Form, Behavioral or Functional, Performance objective or Requirements, Data, and Managerial. Per view is explained what models can be used, resulting in six explicit model like block diagrams, data models, and data and event flow networks.

To gather the information from the resources, we created two overviews and the Viewpoint Matrix. The first overview, of Viewpoints (Appendix A), contains 26 viewpoints. The second overview presents Architectural Models containing 49 models, structures and techniques. Together, they form the basis for selected sections in the main application.

Overview of Viewpoints

Each book is scanned for information about views, viewpoints and models (modeling notations). To capture this information, we constructed an overview (Appendix A). In this overview, the following information is gathered, when available:

1. *Name of the viewpoint.* Dependent on the author, different terms are used for a similar concept. For this project, we use the term 'Viewpoint' to represent all these concept, based on the definition taken from the IEEE Standards that a viewpoint is a template, pattern, or specification for constructing a view (IEEE 1471-2000).
2. *Description of the viewpoint.* This can be the definition stated by the author or a description explaining the viewpoint.
3. *Additional information.* Information explaining more about the viewpoint, what can be expected in this view and important notes by the author.
4. *Elements & Models.* The corresponding models or elements that make up the viewpoint.

Viewpoint Matrix

The Viewpoint Matrix, extracted from the viewpoints, connects correlated viewpoints from one author to another based on similarity of concepts. The purpose of this matrix is to determine which Viewpoints model similar concepts of the architecture using different notations. The matrix is further explained in the next chapter.

Overview of Architectural Models

An additional overview is created to document different models, structures and notations (Appendix B). The purpose of this overview is to gather notations on how to create a model, so a list of notations can be implemented in the prototype.

Each book is scanned for information about models, structures and modeling notations. Each author described either a structure, a model and/or a modelling notation that can be used to model a certain viewpoint. The following information is gathered when available.

1. *Module/structure*. The name of the module, structure or technique.
2. *Purpose*. A description of the model and what purpose it serves.
3. *Elements*. The basic elements or concepts of the model, structure or technique.
4. *Notation*. The language or notation how the model is composed.

3 The Viewpoint Matrix

As mentioned in the previous chapter, the purpose of the Viewpoint Matrix is to determine the linking of similar models. The vision behind this matrix is that the user of the application can categorize or tag each model with a viewpoint. By linking similar viewpoints from different authors to each other, it creates a more global application where architects using different methods can select the same models using different selecting categorizations.

3.1 The matrix as a new concept

May (2005) briefly touch upon the correspondence between five viewpoint models when they are grouped into functional, dynamic and external structures. In their research, they primarily looked into five viewpoint models to determine the extent to which they cover the software architecture domain. Since only Kruchtens “4+1” View Model corresponds between Mays research and this project, and the difference between the grouping and linking of the models, the Viewpoint Matrix is a new concept.

Other comparisons on software architecture exist, but most researches focus on comparing the methods instead of the actual viewpoints. For instance, Shahin, Liang, & Babar (2014) performed a systematic review on the visualization of models, and Hofmeister et al. (2007) compared artifacts and activities.

3.2 Procedure for creating the Viewpoint Matrix

To create the Viewpoint Matrix (Figure 1), the following steps are taken.

Step 1: Gather viewpoints and remove duplicates.

Gather all the viewpoints of the Overview of Viewpoints (Appendix A) in a list. This results in duplicate viewpoints, for example both Kruchten and Rozanski & Woods mention the viewpoint Development. In order to create a compact overview, duplicated viewpoints are removed. For each duplicate viewpoint, the description is compared to check the similarities between the viewpoints.

Step 2: Determine the rows and columns.

The axes represent the viewpoints. Each viewpoint of the list from step 1 is copied to the matrix axes.

Step 3: Comparison of viewpoint between authors.

Per author (A) each viewpoints description is matched (when possible) with a viewpoint of the other author (B). Viewpoints are similar when:

a. Meaning of the description is similar.
b. Main terms in the description match or are synonyms.
c. Additional information confirms terms or description of the compared viewpoint.

Each similar viewpoint is written in the corresponding cell, linking the viewpoint of author A with one or multiple viewpoint(s) of author B.

Take for example Managerial view and Development view. The definition of the Managerial view is “the process by which the system is constructed and managed.” (Maier & Rechtin, 2000, p. 162). The Development viewpoint describes “the architecture that supports the software development process.” (Rozanski & Woods, 2012). Both descriptions mention the process to develop a system as their main concepts. The second condition stated above applies to these statements.

Step 4: Check if the matrix displays a mirror image at the diagonal line.

When the both sides display the same image, both viewpoints are found similar either as author A to B, as author B to A. When displaying a discrepancy between the images, corresponding viewpoint are checked again for similarities.

Step 5: Style matrix.

Black out the mirror image to give a one sided image of the matrix. This results in the matrix below.

From / To	Module	C&C	Allocation	Quality	Context	Functional	Information	Concurrency	Development	Deployment	Operational	Logical	Physical	Behavioral	Process	Scenarios	Purpose	Form	Performance objective	Data	Managerial
Module	-																				
C&C		-																			
Allocation			-																		
Quality				-																	
Context			■		-																
Functional	■	■				-															
Information		■					-														
Concurrency		■						-													
Development									-												
Deployment			■							-											
Operational											-										
Logical	■	■				■						-									
Physical		■											-								
Behavioral											■			-							
Process		■		■				■							-						
Scenarios					■											-					
Purpose/objective					■												-				
Form	■					■						■						-			
Performance objective				■											■				-		
Data		■					■													-	
Managerial									■												-

Figure 1 Viewpoint Matrix.

As mentioned in the beginning of this chapter, the matrix formed the base for the selecting of models by architects using different modelling methods. The implementation of this matrix in the prototype is discussed in the next chapter.

4 Design decisions

To implement the Viewpoint Matrix, an application is needed. In this chapter, we gradually build the application and explain the design decisions that have been made.

The layout of the application (Figure 2) consists of four main elements.

1. The menu strip. This menu contains the basic application options and menus.
2. The left side menu. This menu contains all the elements necessary for a software architect to operate the application.
3. The drawing ribbon. This ribbon contains all elements to draw or edit a model.
4. The canvas. This is where all the action takes place to visualize, build and alter models.

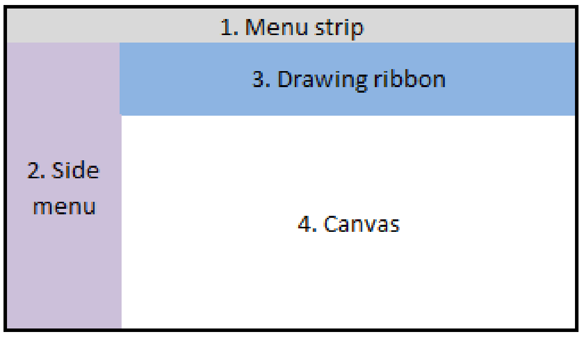


Figure 2 Layout of prototype.

4.1 Menu strip

The menu strip (Figure 3) is comparable to standard application options. The items are a combination of icons for quick access to options as Save, Undo and Redo, and text dropdown menu-for options as File, Edit and Help.

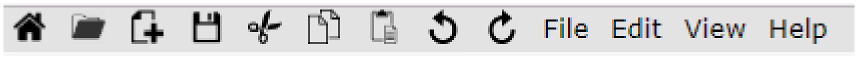


Figure 3 Layout of the Menu strip.

4.2 Side menu

The side menu consists of four menus-items and is expandable to more items in the future.

4.2.1 Select drawing elements

The first menu-item is the Select drawing elements (Figure 4). In this menu, the user can select which notations are visible in the drawing ribbon. A notation consists of drawing elements to draw a model. For example, the notation UML Use Case consists of elements as Actor, Use Case, and Dependency. By ticking one of the checkboxes, a JavaScript is executed to show the corresponding drawing elements in the ribbon.

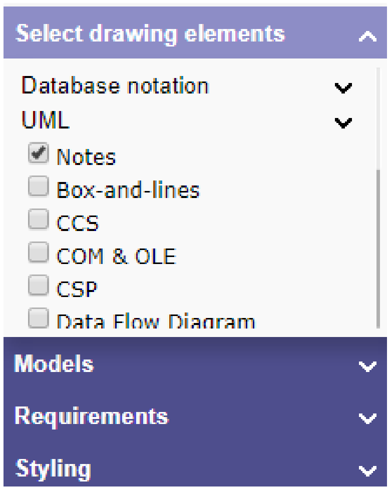
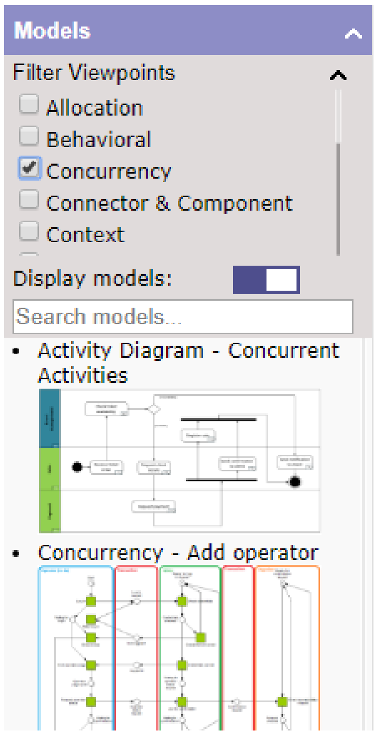


Figure 4 Layout of Selecting drawing elements

To minimalize the length of the list of notations, similar notation (as UML) are grouped together in a dropdown list and can be maximized when desired. The list is alphabetically sorted, with exception of Notes. Notes is an important option and can be easily overlooked in a list, and therefore it is placed at the top of the list.


Models

Filter Viewpoints

- [ ] Allocation
- [ ] Behavioral
- [x] Concurrency
- [ ] Connector & Component
- [ ] Context

Display models:

Search models...

- Activity Diagram - Concurrent Activities
- Concurrency - Add operator

Figure 6 Layout of Models.

4.2.2 Models

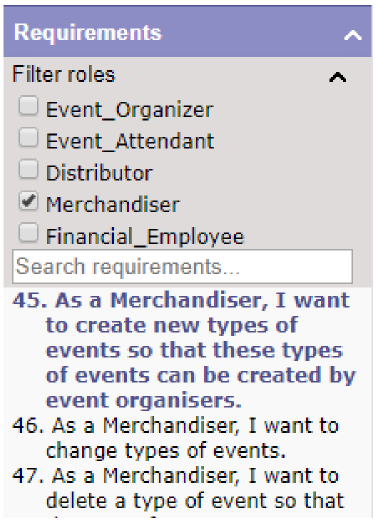
Models is the second menu-item (Figure 5). In this menu all the models are gathered so the user can find, filter and search form models. The first option is to filter the models per viewpoint. In this option the Viewpoint Matrix of the literature research is implemented. A JavaScript is executed by ticking one of the checkboxes. This script checks which options are active. Then per active option the relations between viewpoints from the matrix are checked. Then the corresponding models are shown in the list.

The second option is to Display models. This is a toggle function to show or hide the thumbnails of the models in the list. The third option is to search for a model. This is a real-time search function that updates the list based on what the user types.

The options are followed by the actual list of models. It contains the title of the model, followed by a thumbnail of the model. This thumbnail is also draggable into the canvas of the application. By clicking on one of the models, the font-weight and color is changed to indicate the focus on that model.

4.2.3 Requirements

Requirements is the third menu-item (Figure 6). This contains a list of User Stories. For this list is also a filter per role and search option available.


Requirements

Filter roles

- [ ] Event_Organizer
- [ ] Event_Attendant
- [ ] Distributor
- [x] Merchandiser
- [ ] Financial_Employee

Search requirements...

45. As a Merchandiser, I want to create new types of events so that these types of events can be created by event organisers.
46. As a Merchandiser, I want to change types of events.
47. As a Merchandiser, I want to delete a type of event so that

Figure 5 Layout of Requirements.

4.2.4 Styling

Styling is the last menu-item (Figure 7). In here the user will be able to style his model with different colors, lines and text. For this project a placeholder image is implemented that displays the future functionalities in this menu.

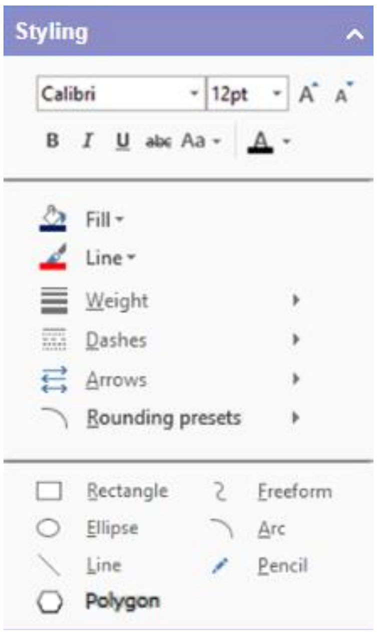


Figure 7 Layout of Styling.

4.2.5 Order of the menu-items

Selecting drawing elements is placed as top menu-item to minimize the distance between the ribbon and the selecting options. This way the coupling between the two elements is more naturally.

The focus of the application is on the models, therefore Models is the second menu-item. Another reason that we take into account is the size of the menu. If the Models menu is lower on the list, the chance to trigger a double scrollbar (one of the side menu and one of the list of all models) is bigger, than when Models is higher on the list. The same is applied to Requirements. Therefore Styling is the last option in the menu.

4.3 Drawing ribbon

The drawing ribbon (Figure 8) contains all the drawing elements an architect needs in able to draw models. As mentioned in the previous sector, the elements are grouped per notation and can be activated by selecting the desired notation.

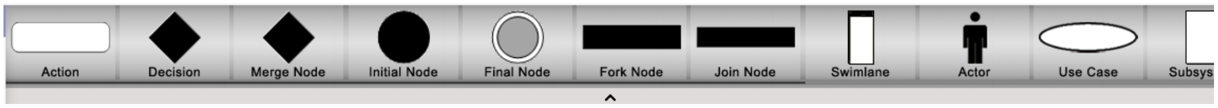


Figure 8 Layout of Drawing ribbon.

For accessibility from all positions in front of the video wall, the ribbon is placed at the top of the canvas. It has a similar placement as the ribbon in Microsoft applications. It is also possible to minimalize the ribbon when it is not needed. The default position is to hide this element, only to pop up when a notation is selected.

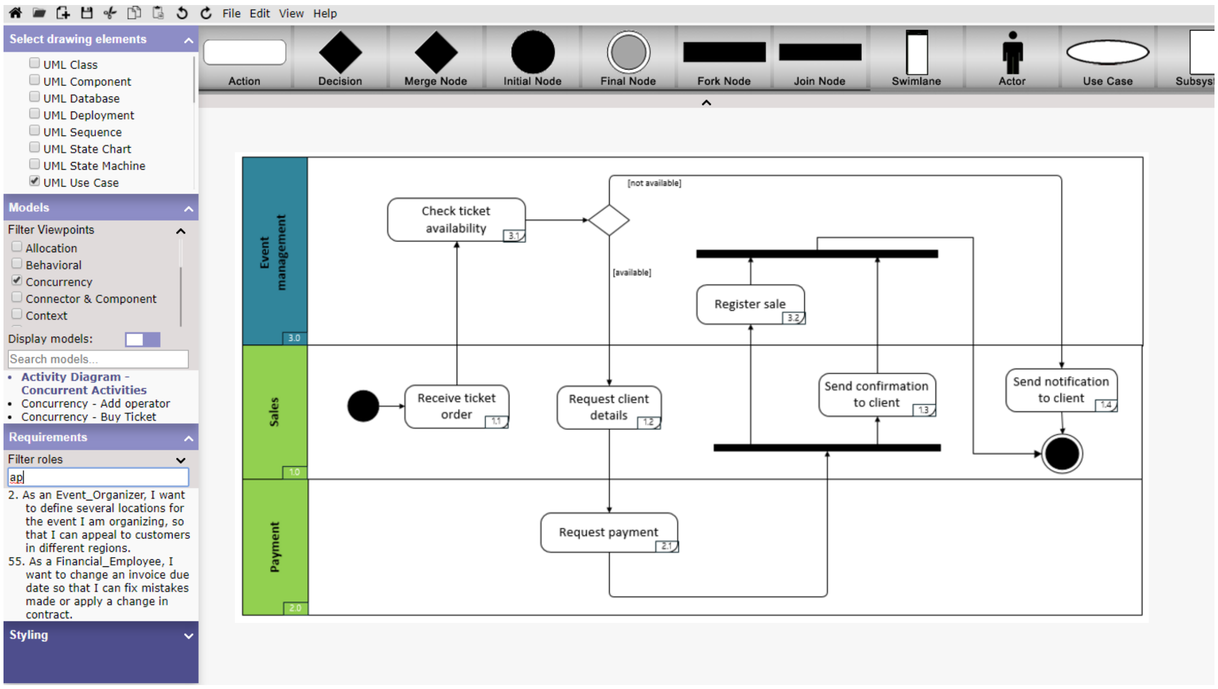
The user is able to drag each element to the canvas to create or modify a model. The ribbon is horizontally scrollable in order to easily select an element further in the ribbon.

4.4 Canvas

The canvas is the area where the user can drop models, drawing elements and User Stories. In here the user will also be able to view information about the elements in the model. This is yet to be implemented.

To empty the canvas, the user drags the model or element to the side menu or the drawing ribbon.

In Figure 9, we see a screenshot of the prototype in action. The drawing elements 'UML Activity' and 'UML Use Case' are active and displayed in the drawing ribbon, the Concurrency models are active, and the thumbnails of the models are disabled, a search request is active for 'ap', and the activity diagram is dragged to the canvas.



- [ ] UML Class
- [ ] UML Component
- [ ] UML Database
- [ ] UML Deployment
- [ ] UML Sequence
- [ ] UML State Chart
- [ ] UML State Machine
- [x] UML Use Case

Models

Filter Viewpoints

- [ ] Allocation
- [ ] Behavioral
- [x] Concurrency
- [ ] Connector & Component
- [ ] Context

Display models:

Search models...

- **Activity Diagram - Concurrent Activities**
- Concurrency - Add operator
- Concurrency - Buy Ticket

Requirements

Filter roles

ap

2. As an Event_Organizer, I want to define several locations for the event I am organizing, so that I can appeal to customers in different regions.
55. As a Financial_Employee, I want to change an invoice due date so that I can fix mistakes made or apply a change in contract.

Figure 9 Layout of the prototype.

In Figure 10 we see a second screenshot of the prototype in action. The drawing ribbon is hidden and on the canvas there are two models present. In Figure 11 we see the prototype in action on the video wall.

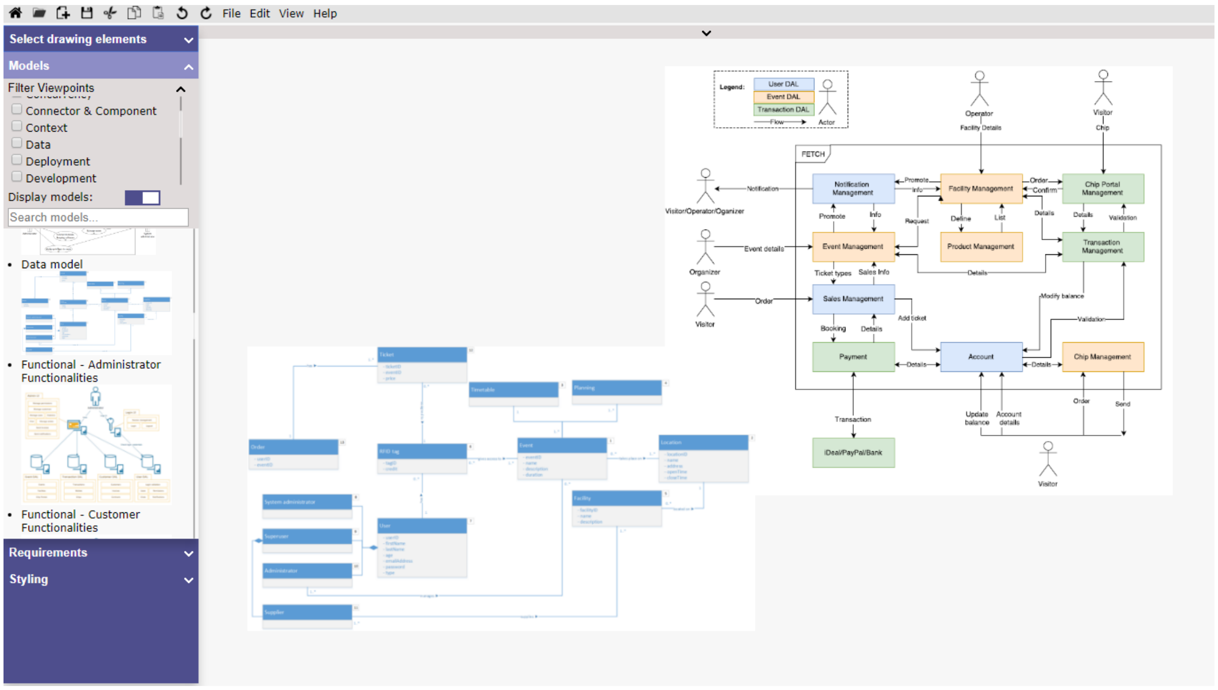


Figure 10 Prototype with two models on the canvas.

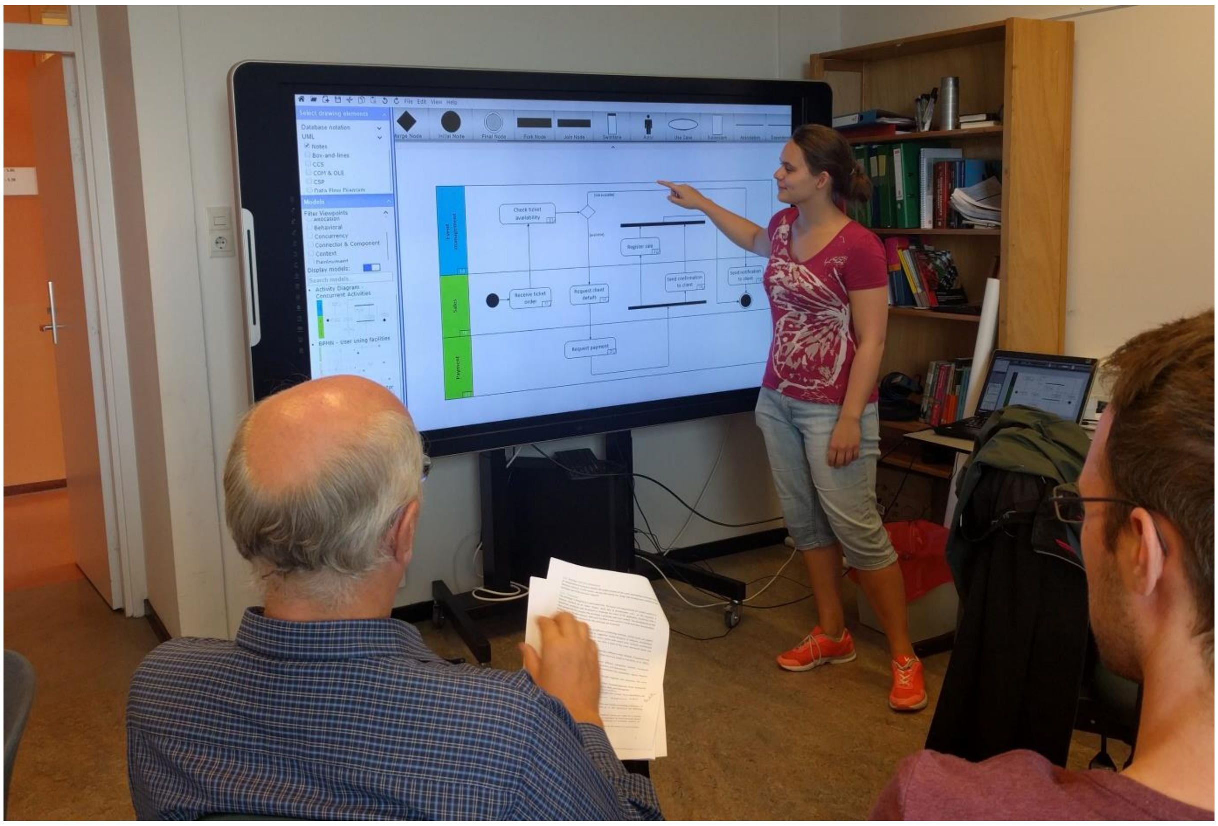


Figure 11 Demo of the prototype on the video wall.

5 Discussion & Outlook

Building a prototype is the first step to actually build an application. It is the easiest (and cheapest) way to implement changes after testing with potential users. For this project a prototype is only build and not tested. The design decisions described in the previous chapter seem logical, but there is always a difference in what looks good on paper and what actually works in a real work environment. Therefore the design as described in this paper, is not the final design and needs to be tested before the next step in the design cycle is taken.

Another point of discussion is that this prototype is built on literature and not a combination of literature and real work experience. Therefore the prototype is missing the link with how real-life software architects work. This is applies to how models are named and grouped together, but also how requirements are written down and stored.

5.1 Outlook

As mentioned in the context, this project is a prototype to showcase what the application can do. In the following paragraphs we discuss the differences that will be made between the prototype and future application and what future functionalities can be incorporated.

5.1.1 Real vs. dummy data

For the application a set of 62 User Stories and 31 models is hardcoded in the prototype, and therefore cannot be changed by the user. In the future application this will be possible by either uploading or importing files with requirements (User Stories) and/or models.

The hardcoded models also means that it is not possible to interact with the models on individual elements level. The model is draggable as one image, instead of individual elements that can be altered in the future application.

5.1.2 Styling vs. no styling

As mentioned before, for this prototype only a placeholder image for styling is implemented. This image showcases what options will be available to the user. In the application the user will be able to style the models.

5.1.3 Future functionalities

Future functionalities that could be incorporated in the near future are:

- The ability to zoom in the model. Either to a sub-model or to enlarge the model.
- Display of element information. By clicking on an element, the user will be able to see some information about that element, for example what User Stories are connected to that element.
- Possibility to change order of drawing ribbon so the user can put frequently used elements in front of the line.
- Automatically opening the styling menu when building a model and changing the available styling function based on what element is selected.
- Presets to highlight certain elements. For example a basic preset where the models have barely to any color, only displaying basic information, and a connecting preset where elements that are connected are highlighted.

- Free shape drawing. The user draws an object. Depending on the shape, the application will:
    - translate the gesture into the actual (pre-programmed) object.
    - translate the drawing to a new non-existent object. The user is able to save this object and place it into the drawing elements ribbon.
- Importing models, requirements (User Stories), drawing elements, and possible other files.
- Possibility to change the layout of the application.
- Changing preferences in either the behavior of the application, like disable the linking of the Viewpoint Matrix, or the layout, like colors.

5.1.4 Future implementation of models

For now only five architecting methods are incorporated into the prototype, but there are many more methods and models to implement. For example Functional Architecture Modeling proposed by Brinkkemper and Parchidi (2010) for the visualization of the functional architecture of a software product.

The application should be extensible in viewpoints and perspectives as well. Constantly, architects and researchers develop new ideas and views on architecture. For example, Jagroep et al. (2017) propose an energy consumption perspective on software architecture.

An important aspect that is not included in this research is the experiences of software architects. An extensive collaboration with the software architects in the industry is necessary to develop an application that is connected to both literature and practice. In such collaboration, additional (non-literature) models can be identified and implemented in the prototype.

5.1.5 Future research topics

Other research topics that can be implemented in the application can be how to avoid and detect Architectural Drift and Erosion (Taylor, Medvidović, & Dashofy, 2010). More research about Continuous Architecting and how to implement this, and how to incorporate the rationale of a design decision in the application, e.g. as proposed by Van der Werf et al. (2017).

6 References

Bass, L., Clements, P., & Kazman, R. (2013). *Software Architecture in Practice* (Third ed.). River, New Jersey: Pearson Education, Inc.

Brinkkemper, S. (2015). *Continuous Architecture a vision [PowerPoint slides].* Retrieved from http://www.cs.uu.nl/docs/vakken/mswa/20152016/lecture_14.pdf

Brinkkemper, S., & Parchidi, S. (2010). Functional Architecture Modeling for the Software Product Indistry. *ECSA 2010* (pp. pp.198-2013). Berlin Heidelberg: Springer-Verlag.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., . . . Stafford, J. (2001). *Documenting Software Architectures: Views and Beyond* (2nd ed.). Boston: Pearson Educatio, Inc.

de Feijter, R. (2017). Scenarios for Continuous Architecting.

Hevner, A., March, S., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly, 28*(1), pp. 75-105.

Hofmeister, C., Kruchten, P., Nord, R., Obbink, H., Ran, A., & America, P. (2007). A general model of software architecture design derived from five industrial approaches. *The Journal of Systems and Software, 80*, pp.106-126.

Ieee std 1471–2000. (2000). Recommended practice for architectural description of software-intensive systems. Technical report, IEEE.

Jagroep, E., Werf, J., Brinkkemper, S., Blom, L., & Vliet, R. (2017, June). Extending software architecture views with an energy consumption perspective. *Computing, 99*(6), pp.553-573.

Kruchten, P. (1995, November). Architectural Blueprints: The “4+1” View. *IEEE Software, 12*(6), pp.42-50.

Maier, M., & Rechtin, E. (2000). *The Art of System Architecting* (2nd ed.). Boca Raton: CRC Press LLC.

May, N. (2005). A Survey of Software Architecture Viewpoint Models. *Proceedings of The Sixth Australasian Workshop on Software and System Architectures (AWSA 2005)*, pp.13-24.

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems, 24*(3), pp.45-77.

Rozanski, N., & Woods, E. (2012). *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (2nd ed.). Addison-Wesley.

Shahin, M., Liang, P., & Babar, M. (2014). A systematic review of software architecture visualization techniques. *The Journal of Systems and Software*(94), pp.161-185.

Taylor, R., Medvidović, N., & Dashofy, E. (2010). *Software Architecture: Foundations, Theory, and Practice.* Hoboken: John Wiley & Sons, Inc. .

van der Werf, J. M., de Feijter, R., Bex, F., & Brinkkemper, S. (2017, April). Facilitating Collaborative Decision Making with the Software Architecture Video Wall. *Software Architecture Workshop (ICSAW) 2017, IEEE International Conference* (pp. 137-140). IEEE.

Appendix A: Overview of Viewpoints

This overview contains all the views and viewpoints of the discussed papers. Per paper certain terms and concepts are highlighted. The underscored concepts highlight the essence of the view. The italic concepts and terms highlight the elements of the view that make up the viewpoint. These highlighted terms and concepts are used to compare the viewpoints. All information is directly quoted from the paper.

Paper	Viewpoint	Description	Additional information	Elements & Models
Bass	Module	Implementation unit that provides a coherent set of *responsibilities*. Explain the <u>system's functionality.</u>	A module might take the form of a *class*, a collection of classes, a *layer*, an *aspect*, or any decomposition of the implementation unit. Every module has a collection of *properties* assigned to it. Unlikely that the documentation of any software architecture can be complete without at least one module view.	Modules (implementation units of software that provide a coherent set of responsibilities), relations (is part of, depends on, is a).
Bass	Component-and-connector	Commonly used to show to developers and other stakeholders <u>how the system works</u> - one can animate/trace through a C&C view, showing an end-to-end *thread of activity.*	Shows element that have some *runtime presence* (components) and *pathways of interaction* (connectors). *Ports* (interface of component) defines a point of potential interaction of a component with its environment. *Roles* (interfaces of connectors) defining the ways in which the connector may be used by components to carry out interaction. *Attachment* indicate which connectors are attached to which components, thereby defying a system as a graph of components and connectors.	Components, connectors, attachments, interface delegation
Bass	Allocation	Describes the mapping of software units to elements of an <u>environment in which the software is *developed*</u> or in which it *executes*.	Can be static or dynamic.	Software element, environmental element, Allocated to
Bass	Quality	Can be tailored for specific stakeholders or to <u>address specific concerns</u>. These are formed by extracting the relevant pieces of structural views and packaging them together.	*Security* view can show all architectural measures taken to provide security. *Communications* view, exception or *error-handling* view, *reliability* view, *performance* view.	-

Paper	Viewpoint	Description	Additional information	Elements & Models
Rozanski	Context	Describes the *relationships*, *dependencies*, and *interactions* between the system and its environment (the people, systems, and *external entities* with which it interacts).	-	Context model, interaction scenario's
Rozanski	Functional	Describes the system's runtime functional elements, their *responsibilities*, *interfaces*, and *primary interactions*.	Cornerstone of most Architecture Description, often first part stakeholders read.	Functional structure model
Rozanski	Information	Describes the way that the system *stores*, *manipulates*, *manages*, and *distributes information*.	As an architect, you can do data modeling only at an architecturally significant level of detail. You use the Information view to answer, at an architectural level, questions about how your system will store, manipulate, manage, and distribute information.	Static information structure, information flow, information lifecycle, information ownership, information quality analysis, metadata models, volumetric model
Rozanski	Concurrency	Describes the concurrency structure of the system and maps functional elements to *concurrently* and how this is *coordinated* and *controlled*.	Extremely relevant to systems with many operations being executed at once.	System-level concurrency models, state models
Rozanski	Development	Describes the *architecture* that *supports* the software development process.	-	Module structure models, common design models, codeline models
Rozanski	Deployment	Describes the *environment* into which the system will be *deployed* and the *dependencies* that the system has on elements of it.	Focuses on aspects of the system that are important after the system has been built and needs to be *validation* tested and transitioned to *live operation*.	Runtime platform models, network models, technology dependency models, inter-model relationships
Rozanski	Operational	Describes how the system will be operated, administered, and supported, when it is *running* in its *production environment*.	Often the one that is least well defined and needs the most refinement and elaboration during the system’s construction, because many details are not fully defined until design and construction are well under way.	Installation, migration, configuration management, administration, support models

Paper	Viewpoint	Description	Additional information	Elements & Models
Taylor	Logical	Capture the logical (often *software*) entities in a system an how they are *connected*.	-	-
Taylor	Physical	Captures the physical (often *hardware*) entities in a system and how they are *interconnected*.	-	-
Taylor	Deployment	Captures how *logical entities* are mapped onto *physical entities*.	-	-
Taylor	Concurrency	Captures how concurrency and *threading* will be *managed* in a system.	-	-
Taylor	Behavioral	Captures the *expected behavior* of (parts of) a system.	-	-
Kruchten	Logical view	Object model of the design (when an object-oriented design method is used).	Supports functional requirements. Decomposed in *key abstractions* (objects, objects classes). They exploit the principles of *abstraction*, *encapsulation*, and *inheritance*.	-
Kruchten	Process view	Captures the concurrency and synchronization aspects of the design.	Takes into account some non-functional requirements. It addresses issues of *concurrency* and *distribution*, of system's *integrity*, of *fault-tolerance*, and how the main abstractions from the logical view fit within the process architecture.	-
Kruchten	Development view	Describes the static organization of the software in its development environment.	Focuses on the actual *software module organization* on the *software development environment*. It serves as the basis for *requirement allocation*, allocation of work to *teams* (or team organization), for *cost* evaluation and *planning*, *monitoring* the *progress* of the project, for reasoning about software *reuse*, *portability* and *security*. It is the basis for establishing a line-of-product.	-

Paper	Viewpoint	Description	Additional information	Elements & Models
Kruchten	Physical view	Describes the *mapping*(s) of the software onto the hardware and reflects its distributed aspect.	Takes into account primarily the non-functional requirements of the system such as *availability*, *reliability* (fault-tolerance), *performance* (throughput), and *scalability*.	-
Kruchten	Scenarios	Instances of more general use cases.	The scenarios are in some sense an abstraction of the most *important requirements*.	-
Maier	Purpose / objective	What the client wants.	To *match* the desirability of the purposes with the practical feasibility of a system to fulfill those purposes	-
Maier	Form	What the system is.	Represent physically identifiable elements of, and *interfaces* to, what will be *constructed* and *integrated* to meet *client objectives*.	Scale models, block diagrams, object diagrams
Maier	Behavioral or functional	What the system does.	Describe *specific patterns* of behavior by the system. These are models of what the system does (how it *behaves*) as opposed to what the system is (which are models of form).	Threads and scenarios (Use Case), Data and event flow networks, Data/control flow, class diagrams, data flow, state charts
Maier	Performance objectives or requirements	How effectively the system does it.	Describes or predicts how effectively an architecture satisfies some function. *Performance* models describe properties like overall *sensitivity*, *accuracy*, *latency*, *adaptation* time, *weight*, *cost*, *reliability*, and many others. *Non-functional* requirements.	Formal methods, data models
Maier	Data	The information retained in the system and its interrelationships.	What *data* does the system retain and what relationships among the data does it develop and maintain?	Entity-Relationship diagram
Maier	Managerial	The process by which the system is constructed and managed.	Describes *the process of building* the physical system. It also tracks construction events as they occur. Most of the models of this view are the familiar tools of *project management*.	-

Appendix B: Overview of Models

Paper	Model/Structure	Purpose	Elements	Notation
Bass	Decomposition	Used as basis for development project's organizations. The units are modules that are related to each other by the is-a-submodule-of relation, showing how modules are decomposed into smaller modules recursively until the modules are small enough to be easily understood.	Module	-
Bass	Uses	Used to engineer systems that can be extended to add functionality, or from which useful functional subsets can extracted. Units are related by the uses relation. A unit of software uses another if the correctness of the first requires the presence of a correctly functioning version (as opposed to a stub) of the second.	Units, modules, classes	-
Bass	Layer	Used to imbue a system with portability, the ability to change the underlying computing platform. Layer is an abstract virtual machine that provides a cohesive set of services through a managed interface.	Layer	-
Bass	Class / Generalization	Allows one to reason about reuse and the incremental addition of functionality. Supports reasoning about collections of similar behavior or capability and parameterized differences.	Class, object	-
Bass	Data model	Describes the static info structure in terms of data entities and their relationship.	Data entities, relationship	-
Bass	Service	Units are services that interoperate with each other by service coordination mechanisms.	Service, ESB, registry, others	-
Bass	Concurrency	Allows architect to determine opportunities for parallelism and the locations where resource contention may occur.	Components, connectors. Processes, threads	-
Bass	Deployment	Shows how software is assigned to hardware processing and communication elements.	Hardware entities, communication pathways, relations	-

Paper	Model/Structure	Purpose	Elements	Notation
Bass	Implementation	Shows how software elements are mapped to the file structure in the system's development, integration, or configuration control environments.	Modules, file structure	-
Bass	Work assignment	Assigns responsibility for implementing and integrating the modules to the teams who will carry it out.	Modules, organizational units	-
Rozanski	Context model	The purpose of the context model is to explain what the system does and does not do, to present an overall picture of the system’s interactions with the outside world, and to summarize the roles and responsibilities of the participants in these interactions. This understanding is essential in order to make sure that all who are involved in the development of the system (and in making any necessary changes outside of it) know what they are responsible for and exactly where the boundaries are. This avoids potential duplication of development effort or, even worse, gaps or inconsistencies in the solution.	System itself, external entities (name, nature, owner, responsibilities), interfaces (between system and external entities, interactions, semantics, exception processing, quality properties)	UML (use case diagram)
Rozanski	Interaction Scenarios	It is often useful to model some of the expected interactions between your system and the external entities in more detail than is provided in a context diagram. This sort of model helps to uncover implicit requirements and constraints (such as ordering, volume, or timing constraints) and helps to provide a further, more detailed level of validation.	Participants, interactions	UML sequence diagrams
Rozanski	Functional structure model	Cornerstone of most ADs. Documents the system’s functional structure—including the key functional elements, their responsibilities, the interfaces they expose, and the interactions between them.	Functional elements, interfaces, connectors, external entities.	UML component diagrams, Yourdon, Jackson System Development, Object Modeling Technique of James Rumbaugh, ADL, Boxed-and-lines diagrams, sketches
Rozanski	Static information structure model	Analyze the static structure of the information: the important data elements and relationships among them.	-	Entity/relationship modeling, (UML) class modeling

Paper	Model/Structure	Purpose	Elements	Notation
Rozanski	Information flow models	Analyze the dynamic movement of information between elements of the system and the outside world. These models identify the main architectural elements and the information flows between them. Most useful for data-intensive systems	Main architectural elements, flow (information interface). Flow: scope, direction, volumetric info, whereby info is exchanged.	Gane and Sarson, SSADM data flow diagrams
Rozanski	Information lifecycle models	Analyze the way information changes over time. Entity life histories: model the transitions that data items undergo in response to external events, from creation through one or more updates to final deletion. State transition models (or state charts in UML terminology): model the overall changes in a system element's state in response to external stimuli.	-	Tree structure (Entity life histories), UML state diagram (State transition model)
Rozanski	Information Ownership models	Define the owner for each data item in the architecture.	Data item (entity (table)), attribute (field), classes of info ownership	Grid (system and data stores vs data items on axis).
Rozanski	System-level concurrency model	The process model shows the planned process, thread, and inter-process communication structure.	Processes, Process groups, threads, inter-process communication, procedural call mechanism, data-sharing mechanisms, execution coordination mechanisms, messaging mechanisms	UML concurrency modeling, LOTOS, Communicating Sequential Processes (CSP), and the Calculus of Communicating Systems (CCS),
Rozanski	State Model	Describes the set of states that runtime elements can be in and the valid transitions between those states. The set of states and transitions for one runtime element is known as a state machine	State, transition, event, action	UML State chart, Petri Nets, SDL, and David Harel's original State charts

Paper	Model/Structure	Purpose	Elements	Notation
Rozanski	Module structure model	Defines the organization of the system's source code, in terms of the modules into which the individual source files are collected and the dependencies among these modules. It is also common to impose some degree of higher-level organization on the modules themselves to avoid having to enumerate many individual dependencies.	Modules, dependencies	UML Component diagram
Rozanski	Common design models	To maximize commonality across element implementations, it is desirable to define a set of design constraints that apply when designing the system's software elements.	1. A definition of the common processing required across elements; 2. definition of standard design approaches that should be used when designing the system's elements; 3. definition of what common software should be used and how it should be used.	Design document with combo of text and UML
Rozanski	Codeline models	The key things to define are the overall structure of the codeline; how the code is controlled; where different types of source code live in that structure; how it should be maintained and extended over time; and the automated tools that will be used to build, test, release, and deploy the software. Defining these aspects of the development environment is an important part of achieving reliable, repeatable build and release processes.	-	UML, but text and tables is better.
Rozanski	Runtime platform models	Defines the set of hardware nodes that are required, which nodes need to be connected to which other nodes via network (or other) interfaces, and which software elements are hosted on which hardware nodes.	Processing nodes, client nodes, runtime containers, online storage hardware, offline storage hardware, network links, other hardware components, runtime element-to-node mapping	UML Deployment diagram, box-and-lines diagram, text and tables

Paper	Model/Structure	Purpose	Elements	Notation
Rozanski	Network models	This model is normally a logical or service-based view of what you require of the network, rather than a physical view that specifies its individual elements.	Processing nodes, network nodes, network connections	UML Deployment diagram, box-and-lines diagram
Rozanski	Technology Dependency Models	Technology dependencies are usually captured on a node-by-node basis in simple tabular form. The software dependencies are typically derived from the Development view. You can also derive hardware dependencies from test or development environments.	-	Text and tables, graphical notations on runtime platform model
Rozanski	Installation model	Discuss installation and/or upgrade as needed for your system. The installation model provides your view of the requirements and constraints the architecture imposes on installation and upgrade.	-	Text and tables, lists
Rozanski	Migration model	As with the installation model, the migration model should focus on the requirements and constraints that the current architecture places on the detailed migration process that will be developed later.	-	Text and tables
Rozanski	Configuration Management model	create a model of the system configuration management approach (rather than identifying lots of individual configuration values).	-	Text and tables
Rozanski	Administration model	define the operational requirements and constraints of your architecture and the facilities it provides for administrative users.	Monitoring and control facilities, required routine procedures, likely error conditions, performance monitoring facilities	Text and tables
Taylor	Natural language	Describing arbitrary concepts with an extensive vocabulary, but in an informal way.	To document architectural design decisions. Often capture non-functional requirements.	-
Taylor	Informal Graphical modeling	Arbitrary diagrams composed of graphical and textural elements, with few restrictions on them.	PowerPoint-style. Good for early prototyping and exploration, capturing ideas on abstract/conceptual level	Geometric shapes, splines, text strings, clip art.

Paper	Model/Structure	Purpose	Elements	Notation
Taylor	UML	Capture design decisions for a software system using up to thirteen different diagram types.	-	Classes, associations, states, activities, composite nodes, constraints etc.
Taylor	Darwin (ADL)	Structures of distributed systems that communicate through well-defined interfaces.	-	Components, interfaces, links, hierarchical composition.
Taylor	Rapide (ADL)	Interactions between components in terms of partially ordered sets of events.	-	Architectures (structures), interfaces (components), actions (messages/events), and operations describing how actions are related to one another.
Taylor	Wright (ADL)	Structures, behaviors, and styles of systems that are composed of communicating components and connectors.	-	Components, connectors, ports and roles (= interfaces), attachments, styles.
Taylor	Koala (ADL)	Capturing the structure, configuration, and interfaces of components in the domain of embedded consumer electronics devices.	-	Components, interfaces, constructs.
Taylor	Weaves (ADL)	Structure and configuration of components in architectures that conform to the Weaves architectural style	-	Components, connectors (queues), directed interconnections
Taylor	AADL	Multilevel models of interconnected hardware and software elements.	-	Networks, buses, ports, processes, threads, etc.
Taylor	Acme (ADL)	Modeling the structural aspects of a software architecture, with the addition of properties to define other aspects.	-	Components, connectors, ports and roles (interfaces), attachments (links), representations (internal structure), properties

Paper	Model/Structure	Purpose	Elements	Notation
Kruchten	Rational/Booch approach	class diagrams shows a set of classes and their logical relationships: association, usage, composition, inheritance, etc. Class templates focus on each individual class; they emphasize the main class operations, and identify key object characteristics. State transition diagrams/state charts, define internal behavior. Class utilities define common mechanisms or services.	Class diagrams, class templates, state transition diagrams/state charts, class utilities.	-
Kruchten	Process	-	Networks, process, tasks, inter-task communication mechanism, flow of messages, process loads	-
Kruchten	Develop	focuses on the actual software module organization on the software development environment.	Module, subsystem, layer, connectors	-
Kruchten	Scenarios	-	-	-
Maier	Scale models	-	-	-
Maier	Block diagrams	-	-	-
Maier	Threads and scenarios	A thread or scenario is a sequence of system operations. It is an ordered list of events and actions which represents an important behavior.	Use Case	-
Maier	Data and event flow networks	Data flow models define the behavior of a system by a network of functions or processes that exchange data objects. The process network is usually defined in a graphical hierarchy.	-	-
Maier	Formal methods	Seek to develop systems that provably produce formally defined functional and nonfunctional properties. Formal methods require explicit determination of allowed and disallowed input/output sequences.	-	-
Maier	Data models	-	-	-