Modeling Functionality in Software Architecture

Wilbert Seele
Department of Computer Science
Utrecht University
Princetonplein 5, 3584CC
Utrecht
wjseele@gmail.com

August 23, 2013

Contents

Chapter 1

Introduction

The field of software architecture has grown since it came into prevalence in the 1990s and many architectural styles, description languages and documentation methods have been researched and produced. Today, many experts such as Bass et al. (2012), Clements et al. (2010) cite its importance for correct software design and documentation purposes. Most software project management schools stress the importance of creating good and comprehensive software architecture documentation before the start of any project. While attention is paid to the required functioning of a product in said documentation, the link between functioning and technical architecture is mostly result based. Clements et al. (2010) states that a Software Architecture Document, or SAD, describes the purpose of the program, and its functioning is described by the software architecture diagrams and documentation. This method of work places very little constraints on the designers in regards to good architecture, especially where the quality attributes mentioned in Software Architecture in Practice by Bass et al. (2012) are concerned. Not creating the software architecture according to these principles can result in problems or issues such as increased cost or limited flexibility. Aside from this, beyond the functional description a SAD remains a very technical document, given its intended audience Clements et al. (2010). Comprehension takes time to understand and requires the user to be familiar with the modeling technique and so the use beyond its intended audience is limited. A functional view of the software architecture would aid in fast understanding of the intended goals and functioning of the software, and by strictly relating the functional architecture with the technical architecture, the quality of the latter could be increased. Adding a common viewpoint for all involved would also increase coordination among both stakeholders and programmers alike, which has been identified by Finkelstein et al. (1992) as an issue as the use of composite systems increases due to the adoption of off-the-shelf software, modular systems and use of technical solutions such as the cloud.

1.1 Problem Statement

Current software architecture design methods focus on the end product of a highly detailed, technically correct architecture. For the architect, this means an intensive and time-consuming design phase and requires extensive knowledge of the used method and knowing all requirements beforehand. Likewise, a stakeholder wishing to make use of a produced architecture in the decision making process or for the production or maintenance of software is required to be familiar with the method and its syntax. Even when this is the case, the focus on the actual working of the program complicates understanding of its purpose, especially where modules with distinct functionality are concerned.

1.2 Key Concepts

The software architecture of a system is defined by Bass et al. (2012) as "the set of structures needed to reason about the system, which comprise software elements, relations among the and properties of both". It focuses on the actual software from a programmers' perspective, offering an abstraction of the technical design using different views. For the purpose of this research, this type of software architecture will be referred to as "technical software architecture", to distinguish it from its functional variety.

Functional architecture is concerned with describing the structure of the product based on the intended function of the various modules of the system. It contrasts with the software architecture by describing what the program and its constituent parts must do, dealing specifically with the functioning and purpose of the system instead of its technical properties. It can be independent of software architecture and is currently not a frequently used step in product design, as seen in Brinkkemper and Pachidi (2010).

Bass et al. (2012) define a software architecture quality attribute as "a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders." The use of these is related to several quality-related properties of the system such as performance, security and modifiability and satisfying these in the design phase of the system results in a technically better software product.

1.3 Thesis Structure

Chapter 1 serves as an introduction and argues the importance of this topic. The key concepts of this thesis are also discussed. This is followed by chapter 2, where the research questions and methods are described. Chapter 3 contains a literature review where related work is discussed in the fields of software architecture, enterprise architecture and method engineering. This is used to determine the requirements and evaluation criteria for the method developed in chapter 4.

Chapter 5 contains two case studies. The method is first tested on the requirements and description of a generic single process browser, followed by another test on a browser with a different set of requirements. Both models are then compared. Chapter 6 then analyses these models and how they translate to the real-world software architectures. Based on this, the method and its results are discussed and evaluated in chapter 7.

The thesis ends with chapter 8, which contains the final conclusions and discusses further research.

Chapter 2

Research Approach

2.1 Research Objectives

This research will bridge the gap between the documented purpose and functioning of software and its technical architecture by creating a means to capture functionality in an architectural model. The purpose of a functional software architectural model is twofold: Primarily it allows for a high-level overview of the functional structure of software without burying the viewer in technical details, allowing for rapid understanding of the total structure and functioning of a program. The secondary goal is to encourage good programming and technical architecture according to the quality attributes mentioned in Software Architecture in Practice Bass et al. (2012). By first establishing distinct functional modules before moving to a technical implementation of those modules several quality attributes are already enforced and the software architecture is already partially done.

Creating a full software architecture document should answer three basic questions about the project: Why, what and how:

- Why as in why is this project here, what is the reason behind it?
- What as in what is the purpose of the software, what must it do?
- How as in how is it going to function?

While the first and last questions can be answered relatively quick in most cases due to the concise nature of the first and the extensive amount of modeling techniques available for the latter, the answer to the second question is often lost in vast lines of requirements, constraints and use cases. This research aims to create a method that allows for the visualization of the core functionality of the project, thus allowing for quick understanding of the intended aims of the various parts of the project.

2.2 Research Questions

This research, as stated in the previous section, aims to bridge the gap between documented requirements and technical software architecture by introducing a separate functional modeling step. Because the eventual aim of the functional architecture is to function as a guideline for the technical architecture, it makes sense to reuse many of the modeling rules of software architecture. This leads to the following main question:

How can functionality be expressed in models of software architecture?

The main research question will be answered by the adaptation of the Functional Architecture Modeling method created by Brinkkemper and Pachidi (2010) that integrates with existing software architecture modeling standards. The method will allow for the creation of functional architectures using the same set of principles that govern existing software architecture. To evaluate the effectiveness of this new method we will use it to answer the following research questions (RQs):

1. To what extent is functionality addressed in current methods for technical software architecture modeling?

The first research question will sketch out the current state of the field in regards to the topic of this research. This allows us to map out the requirements for the newly developed method based on what the current requirements are in software architecture, as well as the needs of stakeholders. Based on this we can establish the value of adding the functional architecture modeling step to the workflow.

2. To what extent are properties and quality attributes of the technical software architecture influenced by using functional software architecture modeling?

Our second research question concerns itself with the usefulness of the created method when added to the software design process. Ideally, use of the method would see certain established quality attributes in technical software architecture ensured.

3. What technical consequences can be predicted on changes in the functional software architecture?

The last research question deals with the strength of the relationship between the functional and technical models of software architecture. If this relationship is strong, the functional modeling step in the software design process is a valuable one, as it can be used as input for the technical software architecture construction and reduces the amount of work the architect has to do.

To keep the scope of this research manageable, we will not discuss any of the preceding steps in the development process, such as requirements engineering, and assume they remain unchanged.

2.3 Research Approach

The research consisted of design science approach followed by a case study. The research approach is a design science approach for the construction and visualization of the method and contains a case study for data collection. Using the described research approach, we will attempt to answer the main research question: "How can functionality be expressed in models of software architecture".

The construction of the Functional Architecture Modelling method can be classified as design research. The design research approach is a major part of the research and uses the data gathered from literature and best-practice examples as input. Design science research creates and evaluates IT artifacts intended to solve identified organizational problems Hevner et al. (2004). In this case the IT artefact is the Functional Architecture Modelling method, solving the problem identified in the introduction.

Following the design cycle as defined by Takeda et al. Takeda et al. (1990), the research consisted of the following phases (also visualized as a Process Deliverable Diagram (PDD) in Fig. 2.1):

- Awareness of problem: The problem has been identified during the exploratory research. This is done using a literature review of previous case studies and research in this area, as well as investigating current architecture languages and practices. This phase is handled as the first sub-activity in the PDD, which results in the answer to the first research question (RQ1).

- Suggestion: Based on the problem found in the previous step, research is performed on literature and documented best practices to identify the requirements of the new method as well as its evaluation criteria. Based on this, a suggestion of a method is developed and serves as input for the development phase. Both phase one and two are characterized by literature, case study and best practices research. This phase also established the requirements that need to be met to answer research questions two and three (RQ2 and RQ3). This phase can be seen as the second sub-activity in de PDD. Phase one and two combined are the first major activity in the PDD.

- Development: During the development phase of the research, the suggested solution is developed. The data and practices collected in the previous step are combined into the adaptation of the Functional Architecture Modelling method. In the PDD this can be found as the activities and results of the "Adapt method" activity.

- Evaluation: The created method is applied to multiple cases to determine is applicability, effectiveness and completeness. Research questions two and three (RQ2 and RQ3) will be answered using the cases. As can be seen in the "Evaluation" activity, the created technical architectures are compared to the official versions if available or discussed with experts, thus

answering the second research question (RQ2). Once these are proven adequate, the created technical architectures will be compared to each other in order to answer the third research question (RQ3). Following this the method itself and its results are evaluated to check if the earlier established requirements are all met.

- Conclusion: After evaluation and finalization of the method the main research question can be answered. A conclusion will be written, as well as lessons learned and areas possibly requiring further research. This is the final deliverable in the PDD.

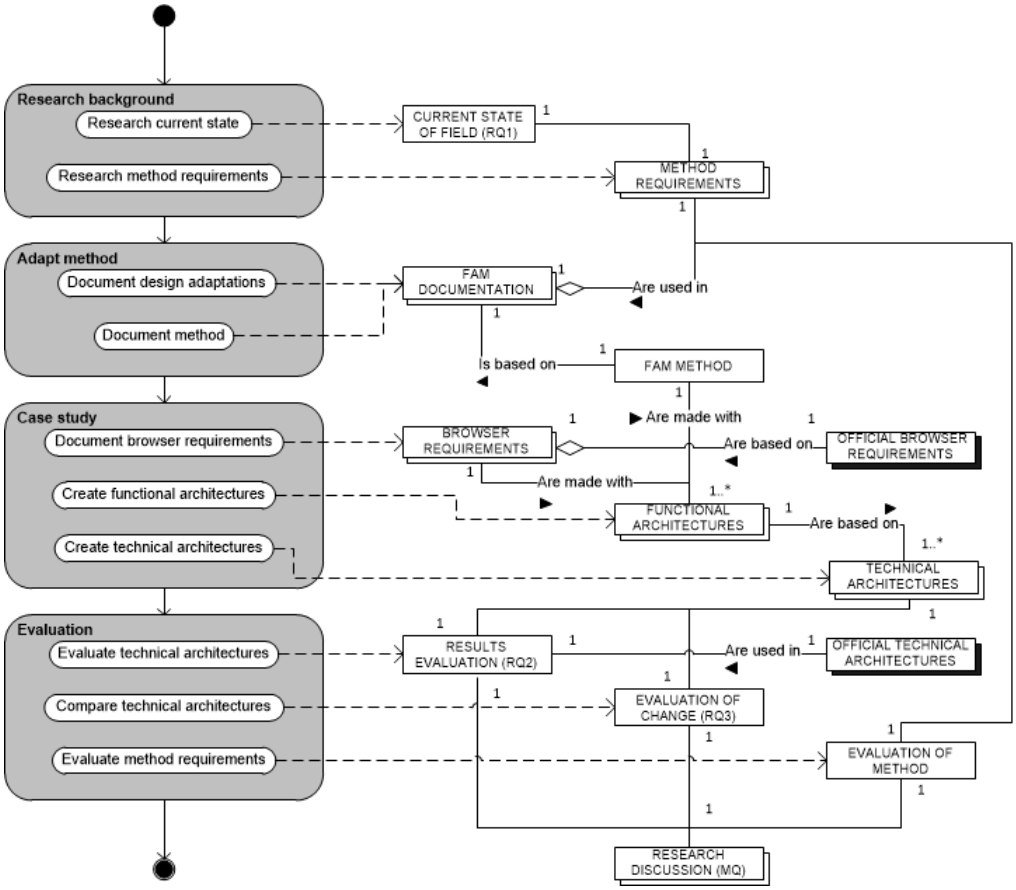


Figure 2.1: Research approach

Activity Table

Activity	Sub activity	Description
Research background	Research current state	In this phase literature is collected to map out the current state of incorporating functional requirements in software architecture. This results in the CURRENT STATE OF FIELD.
	Research method requirements	More literature is collected to find the METHOD REQUIREMENTS to create a usable software architecture method.
Adapt method	Document design adaptations	Based on the METHOD REQUIREMENTS the adaptations made to the original method by Brinkkemper and Pachidi (2010) are made apparent in the new FAM DOCUMENTATION.
	Document method	In the FAM DOCUMENTATION, the new FAM METHOD is explained in both text and a Process Deliverable Diagram.
Case study	Document browser requirements	Using OFFICIAL BROWSER REQUIREMENTS gathered from existing documentation, a set of functional BROWSER REQUIREMENTS is made for the case study.
	Create functional architectures	Using the BROWSER REQUIREMENTS the FUNCTIONAL ARCHITECTURES are created with the FAM METHOD.
	Create technical architectures	Continuing the use of the FAM METHOD, the TECHNICAL ARCHITECTURES are created using the FUNCTIONAL ARCHITECTURES.
Evaluation	Evaluate technical architectures	The TECHNICAL ARCHITECTURES are evaluated by comparing them to the OFFICIAL TECHNICAL ARCHITECTURES and the Quality Attributes identified by Bass et al. (2012). The resulting RESULTS EVALUATION answers the second research question (RQ2).
	Compare technical architectures	By analyzing the differences between the TECHNICAL FUNCTIONAL ARCHITECTURES made by adding BROWSER REQUIREMENTS, and having verified their correctness in the RESULTS EVALUATION, the third research question (RQ3) can be answered using the EVALUATION OF CHANGE.
	Evaluate method requirements	Having used the FAM METHOD in a case study, we check if all the METHOD REQUIREMENTS have been met in the EVALUATION OF METHOD.

Concept Table

Concept	Description
CURRENT STATE OF FIELD	This answers the first research question (RQ1). A literature review to provide a view on what role functional requirements play in the field of software architecture.
METHOD REQUIREMENTS	Again through literature review, we aim to find the requirements for a successful software architecture method.
FAM DOCUMENTATION	The documentation explains the FAM METHOD, what the FAM METHOD does and doesn't do, what it's main METHOD REQUIREMENTS were and how it differs from other software architecture methods.
FAM METHOD	The FAM METHOD itself is explained in both text and a Process Deliverable Diagram, and is a part of the FAM DOCUMENTATION.
OFFICIAL BROWSER REQUIREMENTS	Browser requirements gathered from online browser documentation.
BROWSER REQUIREMENTS	The set of functional browser requirements derived from the OFFICIAL BROWSER REQUIREMENTS to be used in the case study.
FUNCTIONAL ARCHITECTURES	These are created using the FAM METHOD on the BROWSER REQUIREMENTS and aim to visualize and explain the functioning of the software.
TECHNICAL ARCHITECTURES	These follow from the FUNCTIONAL ARCHITECTURES and incorporate the technical requirements.
OFFICIAL TECHNICAL ARCHITECTURES	Browser architectures gathered from online documentation and code analysis.
RESULTS EVALUATION	This answers the second research question (RQ2). The RESULTS EVALUATION is both a review of the produced TECHNICAL ARCHITECTURES using the quality attributes specified by Bass et al. (2012) and by comparing these with the OFFICIAL TECHNICAL ARCHITECTURES.
EVALUATION OF CHANGE	This answers the third research question (RQ3). The EVALUATION OF CHANGE is to see if changes in the architecture by adding new functional requirements result in correctly updated TECHNICAL ARCHITECTURES.
EVALUATION OF METHOD	An evaluation to see if the METHOD REQUIREMENTS were met by the FAM METHOD during the case study.
RESEARCH DISCUSSION	This answers the main research question (MQ). By using functional requirements as the initial input for software architecture, and creating initial FUNCTIONAL ARCHITECTURES based purely on those before expanding those with technical requirements, functionality is expressed in models of software architecture while improving the creation of software architecture as a whole.

Chapter 3

Related Literature

This chapter consists of two major sections. The first section maps out the current state of the field of functional architecture modeling by analyzing current work in functional architecture modelling. The second section reviews a variety of work with the goal of mapping out the requirements for a functional architecture modeling method.

3.1 Functional Architecture Modeling

In the paper Brinkkemper and Pachidi (2010), Brinkkemper and Pachidi argue for the creation and use of functional architecture diagrams to better manage and communicate the function of software products. They state that "some software vendors intuitively tend to design the functional architecture of their products", but that it is limited to certain domains and fairly unstructured. They concur with the statement of Van Vliet (2008) that architecture design should take place between the phases of requirements engineering and technical design, with the documented requirements as the input in this process. This is an interesting contrast with Bass et al. (2012), who state software architecture to be part of the technical design phase. To remedy the issues they see in the current state of affairs they propose the use of a functional architecture modeling technique, which would serve as an efficient method of communicating and discussing the future product. Functional architecture is defined here as "an architectural model which represents at a high level the software product's major functions from a usage perspective", as well as their interactions. The functional architecture should be created by and with input from the architect, product manager and the customer, with the proposed a technique a formalized version of current practices.

The technique itself consists of a set of design and structural principles, which are then distilled into three distinct design structures, namely Modularity, Variability and Interoperability. These are considered from an purely functional view. While Bass et al. (2012) have defined multiple architectural

quality attributes, these are not mentioned as influencing the design structures, despite their functional aspects. Were the functional architecture used as an input for the technical architecture, which is one of the earlier mentioned design principles of Brinkkemper and Pachidi (2010), this would result in discrepancies or unmanaged design when moving into the technical design phase and using the established quality attributes.

Use of the technique by Brinkkemper and Pachidi (2010) is elaborated upon in the paper by Salfischberger et al. (2011), where it is made part of a functional architecture framework. The Functional Architecture Framework (FAF) assists in maintaining an overview of functional and technical requirements, taking into account dependencies and product variability. Decomposing the architecture into functional modules and subsequently functional components allows for development teams to work on separate parts of the product while still sharing a common vision for the end product. To assist in this the graphical functional software modeling method developed by Brinkkemper and Pachidi (2010) is employed and supported by a listing of functional components, structured in a mnemonic hierarchical naming schema. This allows the FAF to support variability in the product and encourages a common language between product managers, architects and developers.

3.2 Architecture Description Languages

In order to map out the requirements for a successful method, as well as determine the incorporation of functional requirements in architecture design in established methods, we analyzed the field of Architecture Description Languages. Clements (1996) defines ADLs as "formal languages that can be used to represent the architecture of a software-intensive system". As the field of Software Architecture emerged around two decades ago, a variety of these languages, that aimed to exhaustively document the software architecture of a product, were designed by academia and enterprises alike. This was a reaction to the then ad-hoc use of informal line and box diagrams to visualize systems in development. Despite all these efforts it was UML that has since been accepted as the de facto standard, despite criticism that it does not fulfill all the requirements of an ADL Pandey (2010). In his paper, Pandey (2010) attempts to review the field of software architecture and determine why it is that ADLs, despite academic backing, never saw significant use. Before going in to this however, let us first list the characteristics Pandey (2010) identifies as necessary for a language to be considered an ADL:

- An ADL must support the tasks of architecture **creation**, **refinement** and **validation**. It must embody rules about what constitutes a complete or consistent architecture.
- An ADL must provide the ability to represent (even if indirectly) most of the common **architectural styles** enumerated in Garlan and Shaw (1993).

- An ADL must have the ability to provide **views** of the system that express architectural information, but at the same time suppress implementation or non-architectural information.
- If the language can express implementation-level information then it must contain capabilities for matching more than one **implementation** to the architecture-level views of the system. That is, it must support specification of families of implementation that all satisfy a common architecture.
- An ADL must support either an **analytical capability**, based on architecture-level information, or a capability of quickly **generating prototype** implementations.

After this, Pandey (2010) attempts to review whether or not UML satisfies these requirements by quoting colleagues and comparing the strengths and weakness of ADLs and UML to each other. A few things are of note here:

- Strengths of ADLs are identified as their unambiguous design, their textual (and thus machine readable) form and, because of this formality, the ability to be analyzed for correctness, completeness and a variety of other quality attributes.
- Weaknesses of ADLs are the earlier mentioned textual form, their domain-specific nature, their lack of graphical elements and the lack of supporting tools. Additionally, on further interviews some additional weaknesses were identified, those being the lack of support for multiple views, the gap between the academic designers and practitioners and their restrictive nature.
- UML's strengths were its graphical focus, support for multiple views, the many tools available and its design as a general purpose language.
- The main weaknesses in UML were the inability to automate any analysis of the design and its lack of formal semantics, leading to ambiguous and inconsistent design.

These are very valid points, but the identification of these strengths and weaknesses is mostly academic, with little focus on the needs of the software architects and stakeholders using these methods. This becomes apparent as we discuss the following paper: In Woods and Hilliard (2005), ADLs are discussed in multiple sessions containing both academics and practitioners in another attempt to explain their lack of adoption. In the discussion after the first session, one of the key problems identified in the adoption of ADLs is a mismatch between the designers and intended users of ADLs. In short, the designers assume a different set of requirements than the actual requirements of practicing architects. Whereas most ADLs are highly formalized to allow for analysis, the main purpose for architecture creation in practice is communication and documentation. The consensus was that for the latter purpose UML was far more

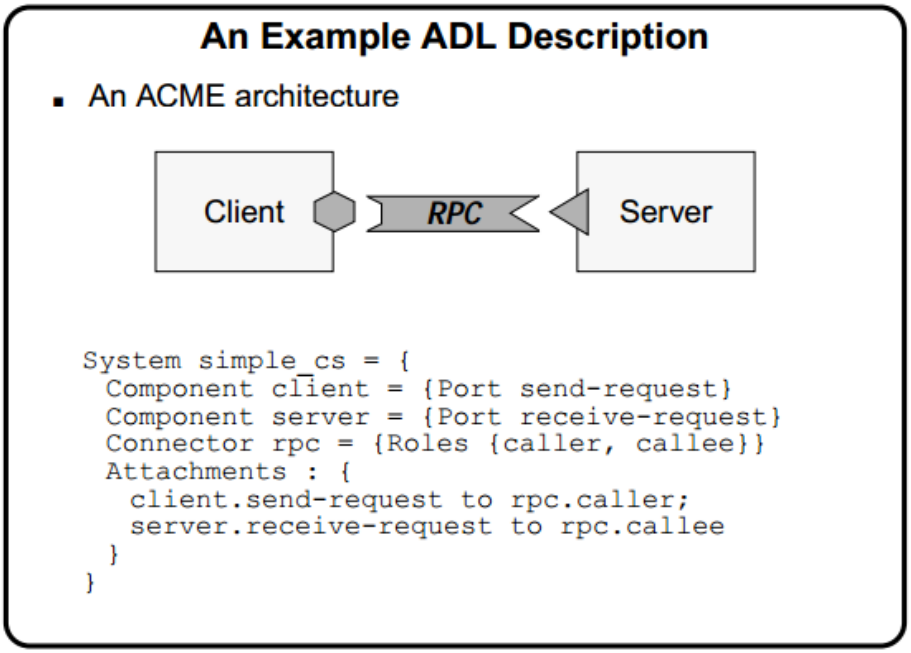


Figure 3.1: Example of an ADL notation and syntax, from Medvidovic (1999)

suited than ADLs. The second session identifies several additions to this list, those being the restrictive nature of ADLs, their support of only a single view (which was also identified by Pandey (2010)), their development within a research context as apposed to a practical one and their lack of supporting tools. It concludes with several consensus points, the most interesting of which is that "...many architects really want a "sketching" tool (rather than a blueprinting tool)". Compare this to an example illustration the notation and syntactic requirements of the ACME ADL in figure 3.1. Take note that ACME was designed to be a "simple, generic" ADL according to its designers at Carnegie Mellon University.

The main lesson for our intended method we can learn from this is that for practical adoption our focus should not be on providing an unambiguous, formal and analyzable method. Instead, success factors appear to be its ability to be used quickly and efficiently ("sketching"), and to keep the design simple for communication purposes and to avoid restricting architects. For this same reason, as well as ubiquitous tool support, our method will not rely on any specific notation, instead aiming for compatibility with any notation scheme, whether formal, semiformal or informal.

3.3 Software Architecture

A large part of this work references the work done by Bass et al. (2012). Upon review, we see that many of the shortfalls of ADLs have been adressed by Bass et al. (2012): they focus on its need to be used for communication and documentation, and explicitly discuss the variety of views it must support. The quality attributes identified in the book are of a sufficiently general nature that they can be adapted to whatever domain the architect works in. On review, we also see that the quality attributes they identify in architecture closely match those identified in software programming in Raymond (2003).

Bass et al. (2012) identify seven core quality attributes in software architecture, those being:

Quality attribute	**Purpose**
Availability	The ability of the systems to handle faults and general problems is such a way that the continued operation of business activities suffers minimal problems.
Interoperability	The extent to which the system is able to cooperate or handle the exchange of information with other systems.
Modifiability	The ability of the system to be adapted to new circumstances or requirements.
Performance	The ability of the system to function in a timely and efficient manner.
Security	A measure of the systems ability to protect its data and functioning from unauthorized access without excessively impeding its users.
Testability	The measure of the difficulty of creating and/or reproducing errors for and through testing.
Usability	The general ability of the systems to be used easily and efficiently by its intended end-user.

In chapter 18, Bass et al. (2012) identify three primary uses for software architecture, those being:

- Architecture documentation serves as a **means of education**.
- Architecture documentation serves as a primary vehicle for **communication among stakeholders**.
- Architecture documentation serves as the basis for **system analysis** and **construction**.

Functional Architecture Modeling would meet the first two of these criteria: It is meant to quickly show what the system does or is supposed to do, allowing for both discussing the system design and functionality, and it allows for new stakeholders or programmers to get up to speed on the aims and goals of the program and its constituent parts. When taking these criteria into account, as well as the lessons learned from ADLs, we can also determine what notation

should be preferred for Functional Architecture Modeling. Bass et al. (2012) list three forms of notation for architecture, those being the informal, which use natural language and whatever diagramming tool is handy; the semiformal, which use a standardized notation and rules of construction, but do not supply any semantic meaning to their elements; and the formal notation, also known as ADLs, which are described elsewhere. Because of the identified need for flexibility, we will develop our method in such a way as to be independent of notation. This allows the architect to chose whatever form of notation best suits the project.

3.4 Requirements in Architecture

In the paper introducing the Twin Peaks Model, Nuseibeh (2001) stresses frequent communication with stakeholders during the design process to manage changing requirements. He notes that requirements management and architecture design should take place concurrently, calling this the Twin Peaks Model. Frequent communication would allow for the a progressively more detailed architecture and requirements document, with the two becoming increasingly dependent on each other until a final design is reached. The major issue however is the more detailed a software architecture (and we assume him to refer to technical software architecture) becomes, the harder it would become to communicate this to what we assume to be non-technically inclined stakeholders.

Functional Architecture Modeling would enhance this model by placing the functional software architecture as one of the Twin Peaks. As it remains a functional view, communication would not suffer due to the increased detail; it would in fact benefit. As the technical software architecture would be based upon both the technical requirements and the functional software architecture, a highly detailed and vetted functional software architecture would result in an equally detailed and vetted technical software architecture. Thus, the goal of the Twin Peaks model is reached more efficiently by upgrading the communication process.

Integrating feature modeling into the design process also becomes an option due to the increased focus on the functional aspect of design. Riebisch (2003) describe feature models as being able to structure requirements by generalizing them by concepts and allow distinguishing between common and variable requirements. The feature model is defined as "a hierarchy of properties of domain concepts", with features being aspects valuable to the user or customer. These features are then grouped into three main categories, those being **functional**, **interface** and **parameter**. When used together with requirements analysis and functional architecture modeling this allows for an overview of both required and optional functional modules and components before commencing into the modeling phase.

3.5 Findings on Literature

In this chapter a literature study was performed for two purposes. To answer the first research question (RQ1), *"To what extent is functionality addressed in current methods for technical software architecture modeling?"*, and to identify the requirements for a software architecture design method incorporating functional requirements. The latter we discuss in the next chapter as they serve as input for our new method.

To answer the former we first looked into the method by Brinkkemper and Pachidi (2010) that explicitly targets functionality in software architecture, finding that it created well-documented and notated functional architectures as an end-product. Technical software architectures are not considered in both its creation or final product, and appear to be regarded as a separate field. This method was expanded upon in the Functional Architecture Framework by Salfischberger et al. (2011), which does incorporate technical requirements but whose main focus lies on modeling variability in the architecture. As it still uses the same design principles and notation scheme as the original method, it remains largely separate from mainstream software architecture. To research methods that did result in a technical software architectures, we looked at Architecture Description Languages (ADLs). What we found is that ADLs were designed to create formal, correct and analyzable architectures, in some cases specific to a domain. The architecture must be unambiguous and machine-readable to allow for analysis or prototype generation. While functional requirements are a part of the ADL design process, the ruleset for ADLs ensures these requirements must be made to fit its mold, because the main focus of each ADL is to create models defined by its creators to be technically correct Pandey (2010).

Thus, to answer our question, we found that the methods that mainly focus on functional requirements are not a part of the technical software architecture domain, whereas those methods that focus on technical software architecture are focused on the technical correctness of their products.

Chapter 4

Functional Architecture Modeling Method

This chapter consists of three sections. In the requirements section the requirements of the new method are discussed and explained. This is followed by a summary of the original method developed by Brinkkemper and Pachidi (2010), where its notation and use are explained. The last section adapts this method, documenting the end product in both text and a product-deliverable diagram.

4.1 Requirements

Because the functional architecture model is to be used as one of the primary inputs of the later technical design, the three possible design structures specified by Brinkkemper and Pachidi (2010) will no longer be used. This means that instead of focusing the design on either Modularity, Variability or Interoperability, the focus is instead on designing the architecture in such a way as to best meet its primary functional purpose. While doing this attention should be paid on meeting the quality attributes identified by Bass et al. (2012). The functional aspect of these should already be met through application of the method, while the technical aspect should be met through specification in the documentation. While the design may favor one quality attribute over the others based on the requirements, they should all be accounted for, if possible, in the final design. If requirements are extensive or very specific feature modeling of the functional requirement according to different quality attributes is an option, thus producing multiple views. Some quality attributes may not be represented if the software is either very basic or very specific. This should not negatively impact the design process.

These quality attributes have a number of related sub-attributes which can also be taken into consideration during the design of both the functional and technical architectures. Additional requirements are that the technical architecture is a logical consequence of the functional architecture. Adapting the UNIX

philosophy of doing one thing and doing it well, a central rule of transforming a functional architecture to a technical one would be "one functional module should equal one technical module, performing only that function". This correlates with Bass et al. (2012)'s quality attribute of modifiability, specifically by increasing semantic coherence. If a module requires a technical aspect to function that does not fit in its feature model, the architect may add an additional module in the technical architecture to meet this requirement. This also includes assumptions on the software context of the product, such as assuming a networking layer or a printer.

This, combined with the knowledge from related work, leads to the following list of requirements:

1. Be quick and easy to use.
2. Allow for communication with stakeholders.
3. Support multiple views.
4. Avoid restricting architects.
5. Integrate with established QA's and principles of Bass et al. (2012).

4.2 Original FAM Method

We will first provide a summary of the original method of Brinkkemper and Pachidi Brinkkemper and Pachidi (2010). This section is divided into an explanation of the used notation and followed by its design method.

4.2.1 Notation

A functional architecture diagram or FAD inherits most of its notation from the field of Enterprise Architecture, resulting in the following conventions:

- Boxes are used to model modules or sub-modules of the product, which represent functions or processes. For the naming substantivized nouns are used (e.g. Planning instead of Plan), which need to start with a capital letter. The choice of names is critical: since the diagram will constitute a fundamental means of communication amongst the stakeholders, precise and determining terms that are well known in the business domain are preferred. Finally, coloring can be used to categorize the modules hierarchically or according to their use.
- Arrows are used to model interactions between modules in the form of information flows. Typical examples of information flows include notifications, requests, feedback to requests, and documents. The names of information flows are all written in lower case.

- A rectangle is used to cover the modules of the product (or the sub-modules of the modules) to indicate the product (or module) scope. The module name should be stated in the lower-right corner of the rectangle.

The FAD is a hierarchic model, using both vertical and horizontal positioning of modules to indicate their role in the hierarchy. The hierarchy is described below and in figure 4.1.

- Strategic modules, mostly related to process management.
- Tactical modules, related to process control.
- Operational modules which focus on the actual process.
- Supportive modules.

From left to right the order is:

- Input functions.
- Processing functions.
- Output functions.
- External functions are positioned outside the scope, such that an external product that provides input for the product are positioned on the left and one that requires output is positioned on the right.

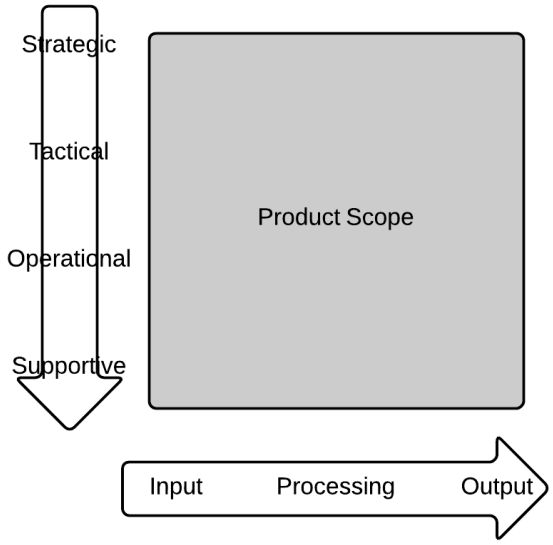


Figure 4.1: Notational hierarchy.

4.2.2 Method

The creation of a functional architecture diagram takes place in five distinct steps, and should incorporate all the functions that the software needs to perform. These steps are as follows:

1: Determine the scope

Start with scoping what functionalities the software product needs to support in the architecture. This is done by identifying the software context within which the product is to be used, as well as what products the software needs to interface with or might need to interface with later. This means the initial assumption is that the newly designed product is not stand alone, and initial requirements will be derived from the products with which it needs to interact.

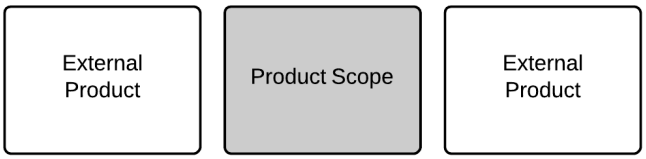


Figure 4.2: Step one: Determine the scope.

2: Define request-feedback flows

After step one, define the functional interactions between the modules of the product and the external products they interact with. Focusing on the interactions required for the primary functionality of the product, determine what requests are made to each module and what they return. The resulting request-feedback flows between modules are known as request-feedback loops.

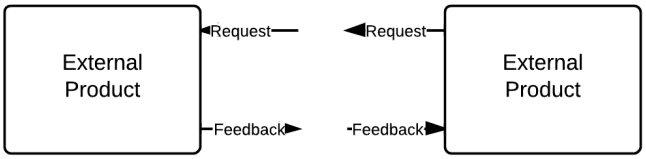


Figure 4.3: Step two: Define request-feedback flows.

3: Model the operational module flow

The next step is determining the flow of the modules that constitute the implementation of the main functionality of the product, which usually will consist of the input, primary process and the output. The different modules that comprise this flow are connected through information flows or waiting queues. Some correctness may be sacrificed for readability, for instance if one modules needs to interact with other modules in such a way as to obscure the main process.

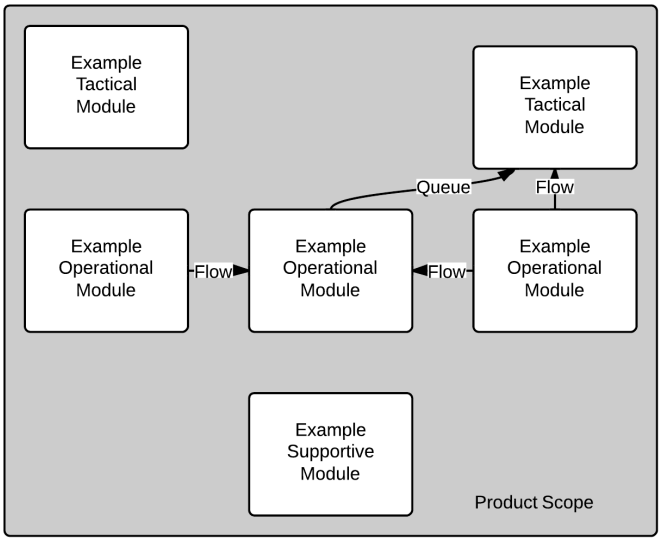


Figure 4.4: Step three: Model the operational module flow.

4: Add control and monitoring modules

In the previous step, most modules will be of the tactical or operational variety. At this point, modules are added in such a way that each operational module is connected with a tactical module, and each tactical module is in turn controlled by a strategic module. These will all be connected through request-feedback loops.

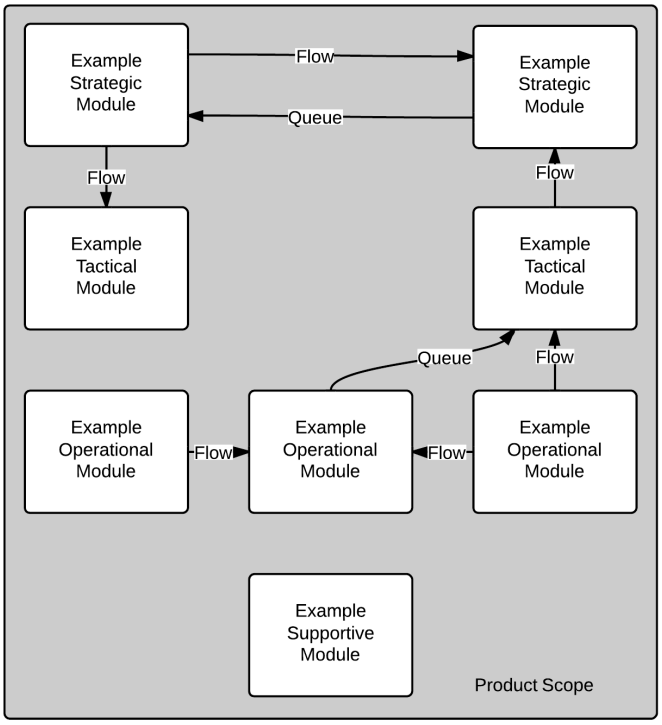


Figure 4.5: Step four: Add control and monitoring modules.

5: Specify external to-from internal interactions

This is a continuation of the second step, where the identified external input/output requirements, or request-feedback flows, are interfaced with the modules of the diagram created in steps three and four. This may result in the discovery or creation of new modules and flows.

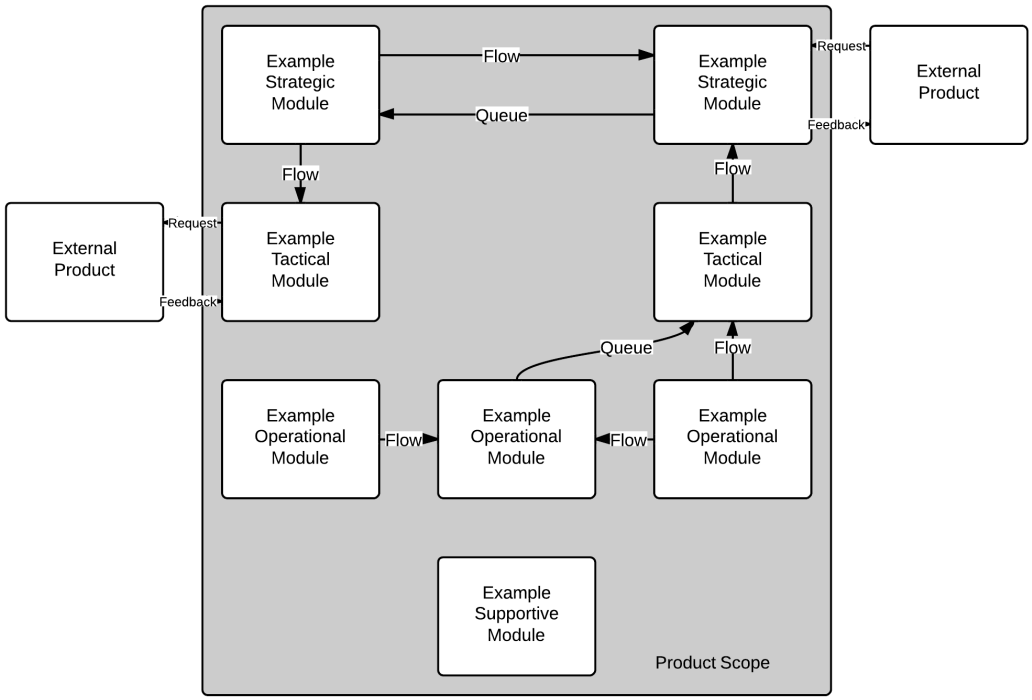


Figure 4.6: Step five: Specify external to-from internal interactions.

4.3 Method Adaptation

Contrary to the designing of the functional architecture after requirements engineering seen in the original method, we would recommend use of the Twin Peaks Model by Nuseibeh (2001) and perform the functional design as part of the requirements engineering process, a move also argued by Salfischberger et al. (2011). We wish to expand upon its design principle of using the functional architecture of the product as a "skeletal system" for later growth and expansion by also utilizing it as an input for the technical architecture, making the functional architecture a part of the overall architecture design process. We believe this to be possible because the quality attributes identified by Bass et al. (2012) consist of both functional and technical aspects. Therefore, their use in the functional design should see its presence in the technical design. For documentation and communication purposes either a non-restrictive approach can be used, but UML or an ADL is also a possibility if the architect is so inclined. The method does not forbid or prevent formalization of the constructed models, but does not require it either.

The majority of the work will actually take place during the requirements engineering and feature modeling processes, as requirements need to be mapped into functional feature models, if necessary according to their governing quality attribute. As the functional architecture is an evolving entity, communicating with stakeholders should be frequent, and architects should not be afraid to add or scrap functional modules and requirements progress.

The functional architecture modeling method is a combination of four activities, three of which occur in a semi-structured order. This is because of the use of the Twin Peaks Model Nuseibeh (2001) and the requirement identified by Woods and Hilliard (2005) that architects want a sketching tool. These are:

1: Requirements Gathering

The requirements analyst or architect collects the functional and technical requirements of the future product, with an initial focus on must-have requirements as these will be the most important in the architecture and determine its initial form.

2: Feature Modeling

The requirements analyst or architect bundles the functional requirements into feature models according to their "theme", or concept feature. These could be user interaction, order processing, data storage, etc.

3: Functional Architecture Design

The architect builds an initial global functional architecture by displaying each identified concept feature as a module and drawing flows between each in such a way that the primary process of the product is established. In this way, the required input and output for the functioning of each module is identified.

This is then expanded upon by decompiling each module in its own diagram, also known as a functional module, using the functional requirements nested under each concept feature as modules within the detailed module view. Input-output flows from the global view can be split to multiple modules if necessary, and additional flows between each module are added where needed, identifying more required information flows.

4: Technical Architecture Design

At this point, the architect will add the technical requirements identified in the first step to the functional architecture. These can either be a part of an existing module (such as a functional requirement of data storage and the technical requirement of a MySQL database) or result in a new module, such as a display backend or networking interface. The architect then adapts the information flows as needed.

The resulting documentation and designs are at this point presented to the stakeholders as the Release 1.0 and a continuous cycle of elaboration begins, also allowing room to incorporate should-have and could-have requirements, while at the same time allowing programming to start on the initial product. This is repeated as long as needed.

Also note that the last three steps happen somewhat simultaneously, in the sense that the architect is aware of technical requirements that may influence the functional design, or may adapt the contents of the feature models as a result of lessons learned during functional architecture creation.

The new method is seen adapting the original method by Brinkkemper and Pachidi (2010) in the following ways:

- Nothing is stated as to notation conventions. The architect should use whatever they are most comfortable with and what produces the required result for them. This could be the notation described in the previous section, UML or an self-defined notation. In our experiment, we used an informal notation which is described in chapter 5.3.

- Our functional architecture design phase is a combination of the third and fourth phase of the original method, namely *model the operational module flow* and *add control and monitoring modules*. These happen simultaneously as we assume all these have been identified as concept features and thus will all be present in the global view. Likewise, we don't have a separate phase where we identify information flows or request-feedback loops, as these will follow logically from connecting the modules to establish the primary process.

- Scope is determined during requirements gathering, in the sense that if the new product needs to communicate with other systems this is simply added as a functional requirement.

Because of the practical need for a sketching tool as opposed to an exhaustively detailed ruleset, our method allows architects a large amount of freedom in how to implement it. This has resulted in a rather bare-bones approach so as not to limit the user, while still accomplishing all its requirements. We especially avoided stating anything specific about the notation, due to the vast variety of stakeholder knowledge in the field. The architect is thus free to adapt his notation to the stakeholder without violating the basic tenets of the method. It is because of this same reason that we can not go in-depth in the use of the method.

A high level overview of the process required to construct a Functional Architecture as part of the entire design process is supplied as a PDD in figure 4.7.

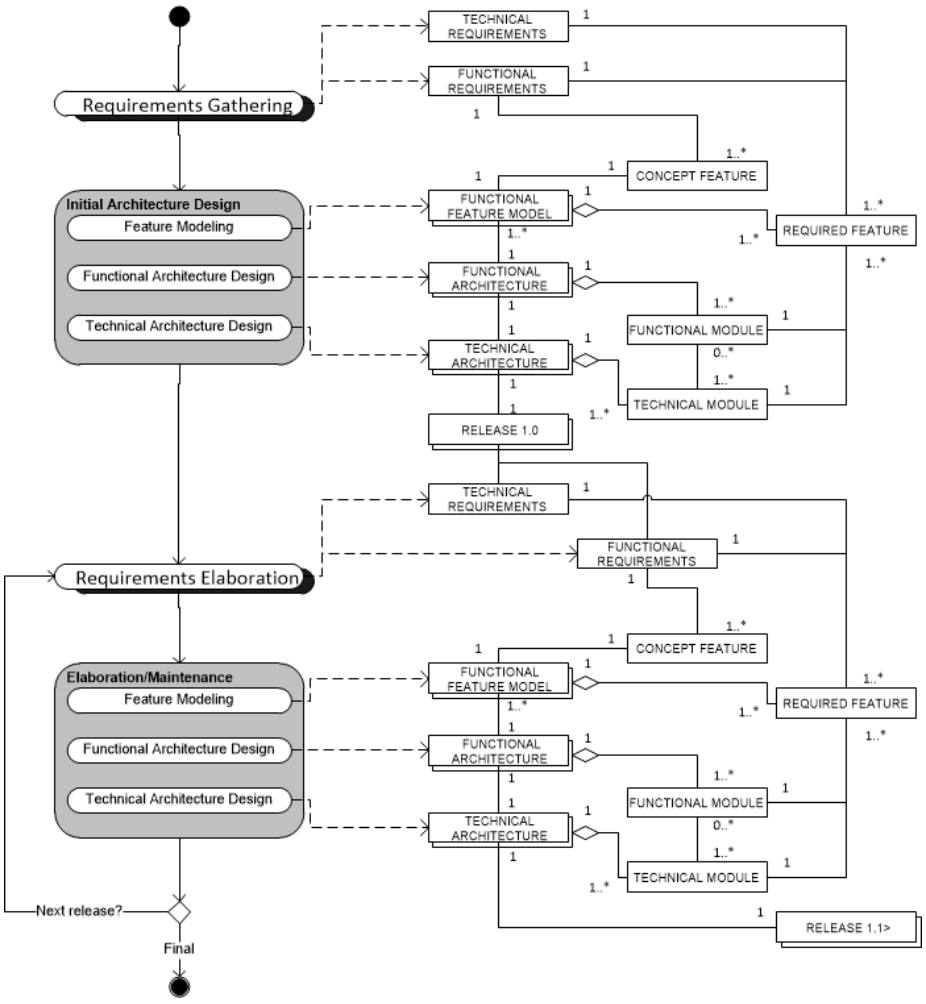


Figure 4.7: Functional Architecture Modeling Method as part of the design process

Activity Table

Activity	Sub activity	Description
Requirements Gathering		In this phase of the design process the requirements analyst (or architect on smaller projects) gathers the functional and technical requirements of the product from various stakeholders. This results in a set of FUNCTIONAL and TECHNICAL REQUIREMENTS
Feature Modeling	Functional Feature Modeling	The architect combines the various FUNCTIONAL REQUIREMENTS into one or more FUNCTIONAL FEATURE MODEL(s), which consist of a CONCEPT FEATURE containing REQUIRED FEATURES.
Functional Architecture Modeling	Main Architecture Design	The architect designs the FUNCTIONAL ARCHITECTURE based on the FUNCTIONAL REQUIREMENTS and FUNCTIONAL FEATURE MODEL(s). The FUNCTIONAL ARCHITECTURE may be elaborated into one or more FUNCTIONAL MODULES.
Technical Architecture Modeling		The architect uses the FUNCTIONAL ARCHITECTURE and TECHNICAL REQUIREMENTS to create the TECHNICAL ARCHITECTURE of the product, as well as the various TECHNICAL MODULES it consists of. Each TECHNICAL MODULE meets at least one REQUIRED FEATURE and should have a corresponding FUNCTIONAL MODULE if possible.
Requirements Elaboration		In this phase the RELEASE documents containing the FUNCTIONAL and TECHNICAL ARCHITECTURE are used to elaborate upon the FUNCTIONAL and TECHNICAL REQUIREMENTS of the software.

Concept Table

Concept	Description
FUNCTIONAL REQUIREMENTS	Describes the functional requirements of the system.
TECHNICAL REQUIREMENTS	Describes the technical requirements of the system.
FUNCTIONAL FEATURE MODEL	A hierarchy of properties of a concept feature Riebisch (2003). In this case focused on functional features.
CONCEPT FEATURE	Represents the general feature domain Riebisch (2003).
REQUIRED FEATURE	Also known as mandatory feature, these are the required parts of the CONCEPT FEATURE Riebisch (2003).
FUNCTIONAL ARCHITECTURE	Functional architecture is concerned with describing the structure of the product based on the intended function of the various modules of the system. It contrasts with the software architecture by describing what the program and its constituent parts must do, dealing specifically with the functioning and purpose of the system instead of its technical properties. Brinkkemper and Pachidi (2010).
FUNCTIONAL MODULE	These are in-depth models of functions in the FUNCTIONAL ARCHITECTURE Salfischberger et al. (2011).
TECHNICAL ARCHITECTURE	This is defined by Bass et al. (2012) as "the set of structures needed to reason about the system, which comprise software elements, relations among the and properties of both". It focuses on the actual software from a programmers' perspective, offering an abstraction of the technical design using different views.
TECHNICAL MODULE	In depth models of modules of the TECHNICAL ARCHITECTURE.
RELEASE	The combined architecture documentation.

Chapter 5

Functional Architecture Modeling In Practice

To test the validity and usability of the new FAM method we chose to test it on two separate but related cases, namely that of a generic browser and a browser with sandboxing, also known as a multi-process browser. This was done because functional browser requirements are manageable and well-known. Another reason is that we could compare our generic browser models with those created from code analysis in Grosskurth and Godfrey (2005). Browser requirements are first described, both of the generic browser and the additional requirement that enables sandboxing. After this the notation we used for the models are briefly explained, followed by the functional models and the technical models based on the functional models.

5.1 Browser Requirements

To keep the size of this section under control only those requirements to make a functionally basic browser will be used, and technical requirements will not be noted, so as to assure a direct transition from functional to technical architecture. Starting at a high level, we can state the following about a browser:

- An interface is needed for the user to interact with the system.
- Content must be retrieved from the internet.
- This retrieved content must be presented in a user-readable format.
- Some form of data storage is needed for user data, such as a browsing history or passwords.
- Some form of process monitoring is required to prevent errors in the program from affecting the users computer.

This is the basic high-level functionality seen in browsers since Mosaic. Combining these allows for the basic function of the browser, namely the displaying of web content. Treating each of these five base functional requirements as concept features, we can assign each a series of sub-requirements which also function as the features in the feature model. This results in the following table:

Main Requirement	**Sub-Requirement**
Interaction	An address bar to input the desired website. An input function to allow for the reloading of content. An input function to stop the loading and rendering of content. An input function to navigate back to earlier-visited areas of the website. An input function to open, close and select a tab containing a website.
Content retrieval	A function to connect with a website. A function to select the correct protocol for accessing the website. A function to download content off the website.
Content rendering	A function to render the text of the website. A function to render pictures of the website. A function to render content requiring plugins. A function to render all elements in a predetermined layout.
Data Storage	A function to store and access the users browsing history. A function to store and access bookmarks. A function to store and access website login data.
Monitoring	A function to manage errors in the program. A function to monitor the programs resource usage. A function to monitor the behavior of website content. A function to report errors to the user.

For the multi-process browser we will several additional major features, namely the ability to separate the tabs we see in the interface into separate processes, the online syncing of browser data and URL blacklisting. Multi-process control was added to allow for better memory management and to increase the stability and security of the browser. The latter is accomplished by sandboxing each process, which prevents errors or malicious content from reaching the main browser, and from there the users' computer. It also means that if a website were to critically fail, only the tab's process would end, and not the browser itself. Online synching of browser data, such as a browsing history or login data, is also a feature seen in many modern browsers. It allows users to share their bookmarked sites and login data across multiple devices, such as desktops, lap-

tops, tablets and smartphones. Finally, URL blacklisting is a security feature that allows the browser to not load certain URLs when these are requested, possibly because they contain malicious programs or unwanted content. Putting these main requirements and sub-requirements together results in the following table:

Main Requirement	**Sub-Requirement**
Multi-process control	A function to allow the separated process to interact with the interface and database. A function to control multiple instances of the browsing functionality.
Online synching	A function to sync browsing history with an online source. A function to sync user login data with an online source.
URL blacklisting	A function to sync blacklisted URLs with an online source. A function to monitor website connections for blacklisted URLs.

These requirements were inspired by Google's Chrome browser, which was the first browser to introduce these features. The advantage with using Chrome as an inspiration is that most design documentation is available online as part of the open source Chromium project. While this documentation is far more detailed than what will be produced in this thesis, we can use high-level generalizations of this to compare to and analyze the models created here.

5.2 Notation

The Functional Architecture Modeling method supports informal, semiformal and formal notations. For this thesis, we chose to use an informal notation scheme to keep the models basic to allow for fast construction and analysis of the core designs. Also note that this notation scheme is not a part of the method. The FAM method was designed for use with whatever notation scheme the user prefers. We chose to create this informal notation scheme to avoid contaminating the results with the extensive notational rules seen in semiformal and formal notation rules.

This notation scheme is not related to the one developed and used by Brinkkemper and Pachidi (2010) which is described in chapter four, section two. Any similarities between the models is because the same modeling application was used when creating the diagrams.

This notation scheme uses the following basic guidelines:

- As the focus is on the functionality of the program, all modules represent something the program does. Each modules name is a verb representing its main activity, or concept feature.

- Each box represents a module. Each module may contain multiple submodules which are modeled separately. Submodules are modeled within a box which represents the main module.
- The size of a box is purely for aesthetic reasons with a focus on readability. Architecture is a communicative device, and functional architecture should support stakeholders of different backgrounds, not all of which are technical.
- The layout of the architecture does not follow a placement scheme. The placement of modules is such that closely cooperating modules are close together with a focus on the main functionality, even if this means not being able to draw arrows to a module of secondary importance.
- Arrows represent communication between modules. Arrows in the global architecture overview require those same arrows to be present in the module view. In the module view, an arrow seen in the global view may be split into multiple arrows if multiple functions are used.
- Each arrow in the global architecture is numbered, its functionality is described in the accompanying documentation. While it is possible to replace the numbers with small descriptions in the picture itself, this was found to negatively impact readability.
- Each submodule is described in detail in the documentation. This was done because whereas the global architecture is a view of connected yet separate functionality, the submodules represent functionalities which are highly related.
- When adapting a model to new or changing requirements, the boxes representing a new module are colored gray and any new or altered arrows between modules are hollow and use dotted lines.

5.3 Case: Generic Single-Process Browser

Combining the concept features as established in the previous section results in the high-level functional architecture seen in figure 5.1. In this model, a standard scenario of accessing a website is as follows:

The numbered arrows represent the flow of activity through the program, and are explained below.

1. The user requests something from the data storage. This could be a previous page or the total browsing history.
2. The data storage returns data to the interface. This could be a list of bookmarked pages, the history or stored passwords.

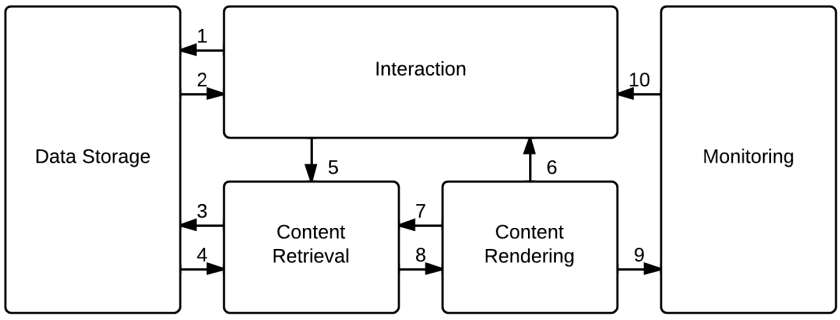


Figure 5.1: High-level functional architecture of a basic browser

3. Content retrieval makes a request or store command to the data storage. This could be to add a new website to the history, or a request for login data.

4. Content retrieval receives data, for instance login data for a website.

5. The user has entered a command. This could be a website through the address bar or back/forward function, or a stop or reload command.

6. The interface receives the requested website.

7. Content rendering may request additional data, such as extra content to be loaded or automatic linkthroughs.

8. Content retrieval sends the received website data to the content rendering module.

9. Content rendering sends what it's rendering to the monitoring module to check it for correct behavior.

10. Monitoring reports errors and security risks to the user.

A view of the process of opening a webpage in this browser design is shown in figure 5.2. This show what modules' functionalities are used in what order. What can not be seen in the functionality of the monitoring module, which was left out as it would connect to each function. As can be seen, the functionalities used in the process view correspond with those seen in the center of figure 5.1.

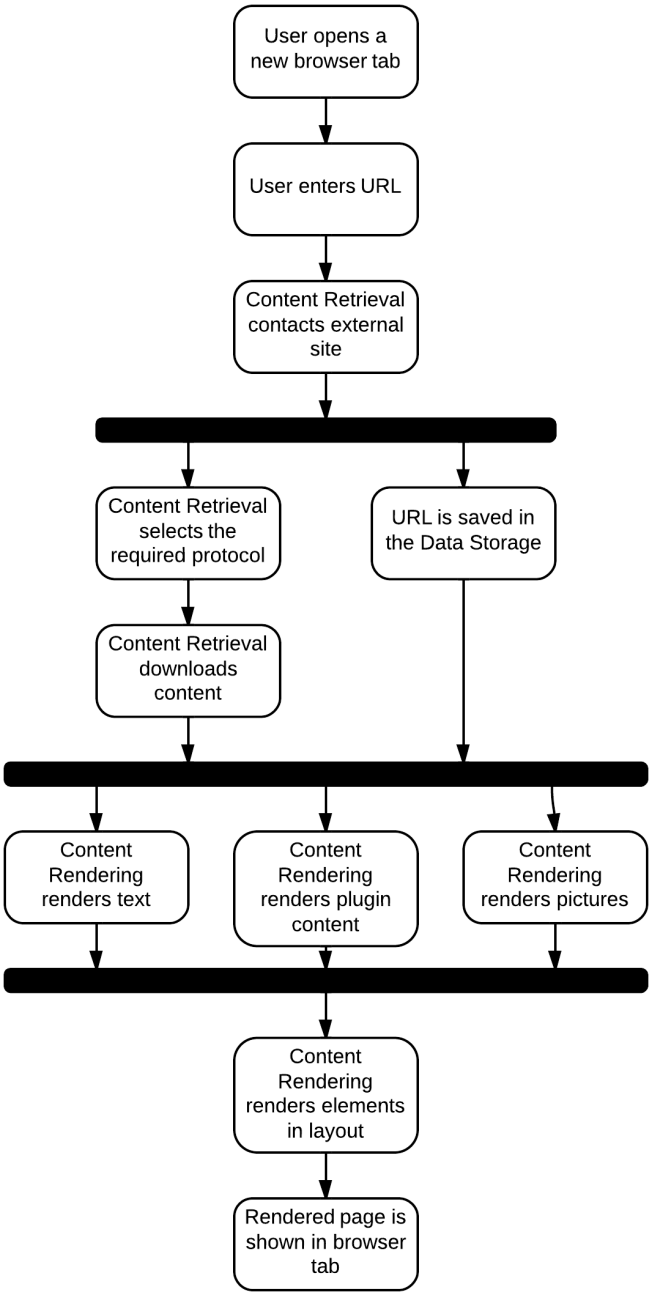


Figure 5.2: Process view of opening a website in the basic browser

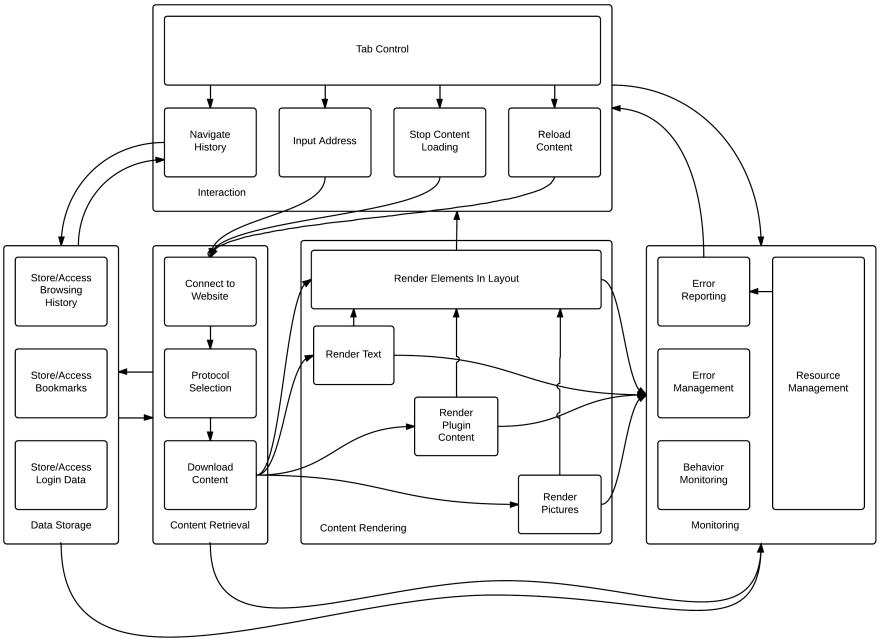


Figure 5.3: Functional architecture including submodules

Further elaborating upon the model, we have created more detailed functional views of each module seen in figure 5.1. These functional modules contain the sub-requirements described earlier in this chapter. Most of these function on an as-needed basis. The exception to this is the monitoring functionality, which is always on during the functioning of the program. The functional modules follow the same flow as the high-level view, but some extra functionality or flow is present and this is explained in each module's documentation. This is because modeling all submodules in one global module results in an overly complex and hard to read model, as can be seen in figure 5.3. For this reason we modeled each module individually and will discuss each in this section before moving on to the technical implementation.

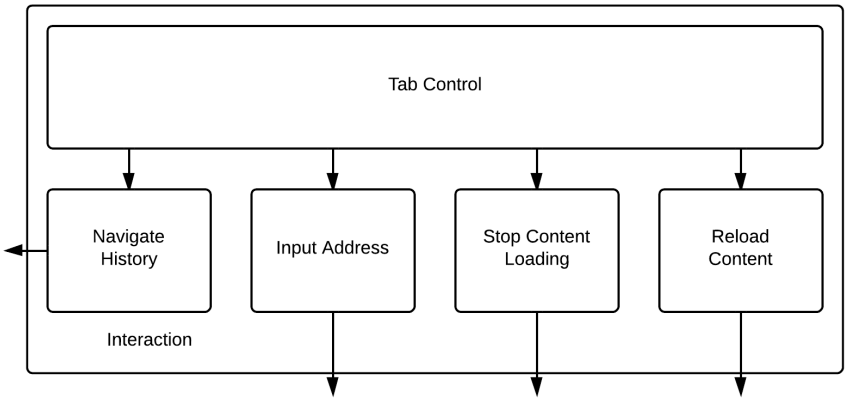
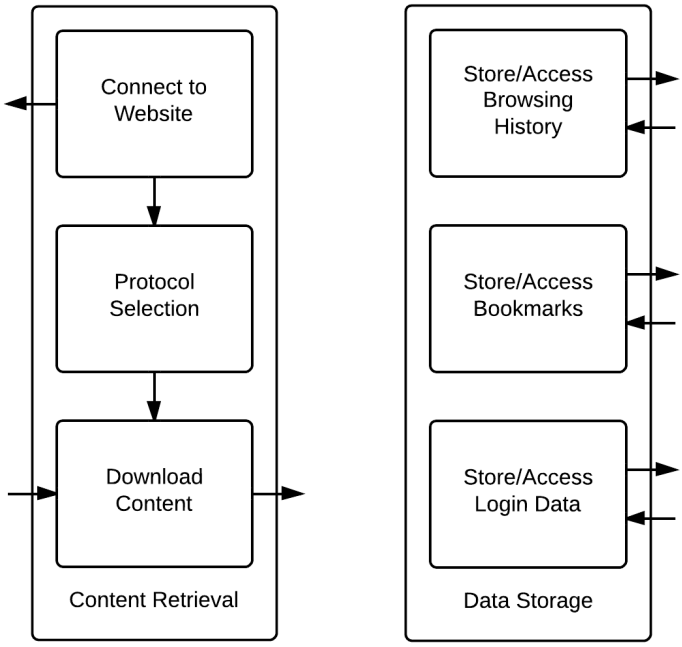


Figure 5.4: Functional module of the browser interaction functionality

Figure 5.4 shows the functional module of the user interaction functionality, or interface. As we assumed our browser to support multiple tabs of content, all interaction with the system starts with the selection of the tab to be used. Website navigation through back and forward functionality means an interaction with the data storage, as it means accessing the browsing history. The other main functions, being the address bar, stop and reload are directly relayed to content retrieval.

Figure 5.5a shows the browsers content retrieval functionality, which processes the requests made by the user in the interface module. Upon receiving the command to access a website (or to reload it or switch to a previous page) a connection is first made to the required webpage, which is also logged in the browsers' history. Upon connection, the correct protocol is used to communicate further, such as http or https for standard website, or ftp for a file transfer. Subsequently the requested content is downloaded by the browser and passed on to the rendering module. In case of websites that require passwords or other previously stored data, this can be supplied bu the content storage module.

In data storage (figure 5.5b) we can see that each function is separately called by either the interface module or the content retrieval module.



(a) The content retrieval function (b) The data storage function

Figure 5.5: The content retrieval and data storage modules

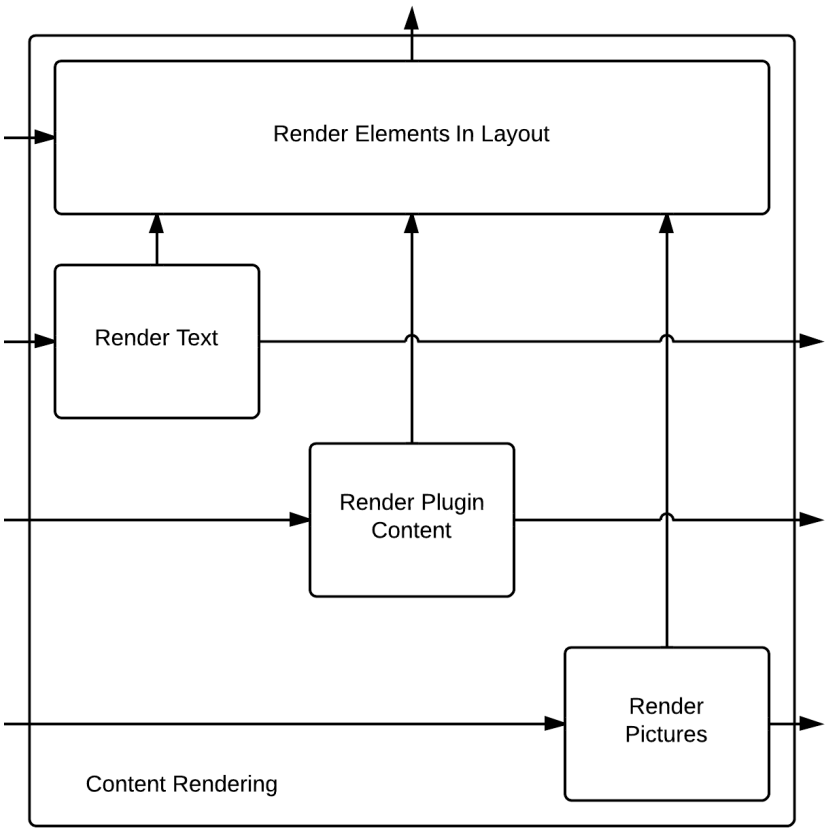


Figure 5.6: Functional module of the browser's content rendering function

The rendering module can be seen in figure 5.6. Contrasting the single arrow seen going towards rendering in figure 5.5a, the module assumes that the data can consist of text, pictures, other media content and layout, and these are delivered to each function of the rendering module. Each function in turn will process this, with the constructed elements being placed in the correct layout before the processed website is delivered to the user. Each function also reports what it's doing to the browsers monitoring module.

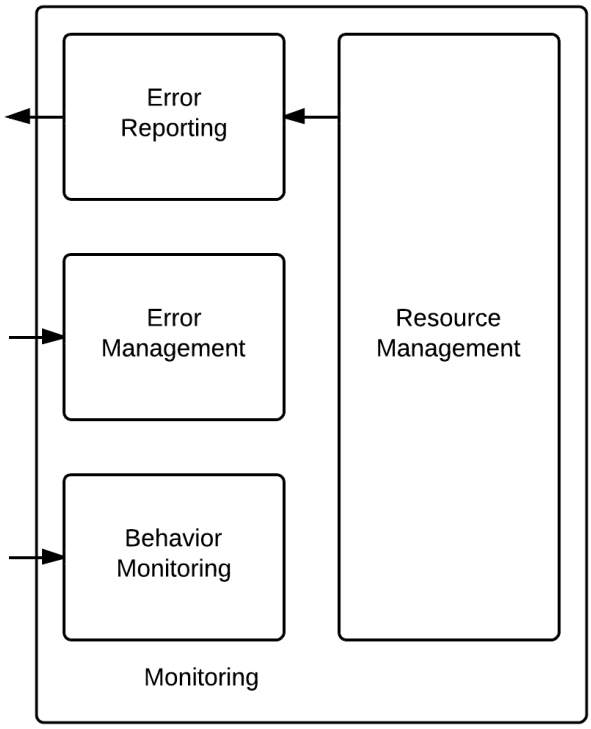


Figure 5.7: Functional module of the browser's monitoring function

The monitoring module, seen in figure 5.7, required some simplification of the resource management module, which governs all other modules. This would have required arrows from all modules and this was not done to enhance the readability of the model. Errors reported to the user from resource management could for instance include the failing of a plugin. Behavior monitoring is done based on data received from the rendering engine to ensure the site contains no malicious code. If this is detected the rendering process should stop. Error management on the other hand deals with the functioning of the rendering module itself, and should allow for the process to elegantly fail if errors are detected. Because the browser in question is single-process with no sandboxing, failing of the rendering process means the exiting of the browser process, hence there is no connection to resource management.

Using these models, we then constructed the high-level technical architecture seen in figure 5.8, where we see that most functionality elegantly translates to it's technical counterpart. We've added three new modules which are required for the program to work on a technical level, and felt confident in adding these because they logically follow from the functional requirements:

1. The display module: Rendered content needs to be displayed to the user. This means that the program must interact with whatever display server the user's computer is running.

2. The networking module: The browser needs to interact with the computers networking capability to access and retrieve content from the internet.

3. The plugin module: For some content, the browser requires plugins such as Flash to function. These are separate programs, but are required as plugin rendering is a core functionality.

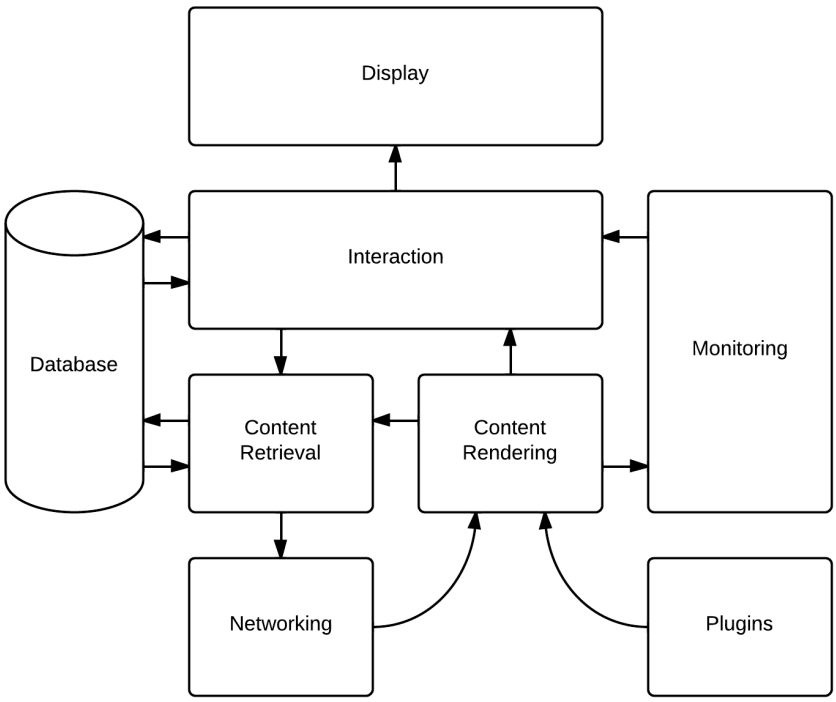


Figure 5.8: High-level technical software architecture of a basic browser

The technical aspects of the browser neatly fit into the functional modules identified at the start of the architecture creating process, thus neatly fitting the quality attribute of cohorence, which falls under modifiability. As all modules encompass differing functionality this means technical interoperability is a given, thus meeting another quality attribute. Finally, the monitoring module explicitly meets the quality attribute of security. While the generated models

will be analyzed more in-depth in a later chapter, it is worthwhile to note that when approaching the technical architecture from a functional architecture some quality attributes are already inherent to the model.

5.4 Case: Generic Multi-Process Browser

We will now adapt the model we constructed in the previous section into a multi-process browser. This requires us to integrate the three additional main functionalities into the existing design. This involved the following adaptations:

- Multi-process control required the addition of a new module to handle the creation and management of new tabs and allow user control to switch between the different instances. An entirely new module was created because this functionality did not integrate with any existing module functionality.
- The online syncing functionality was added to Content Retrieval, due to it being a variation on the general function of retrieving data from the internet. As it updates already present functions of the Data Storage module, nothing needed to be added here to incorporate the new requirements. Online syncing is a feature seen in many modern browsers
- The URL blacklisting feature required two additions to be made to the design. A store of blacklisted URLs in the Data Storage Module and a new function in the Monitoring module to monitor the Content Retrieval module for connections to blacklisted URLs. While it was an option to integrate it directly into the Content Retrieval module, integrating it in the Monitoring module allowed for the user to be warned that they are connecting to a blacklisted URL and giving them the choice to resume loading or cancelling the operation.

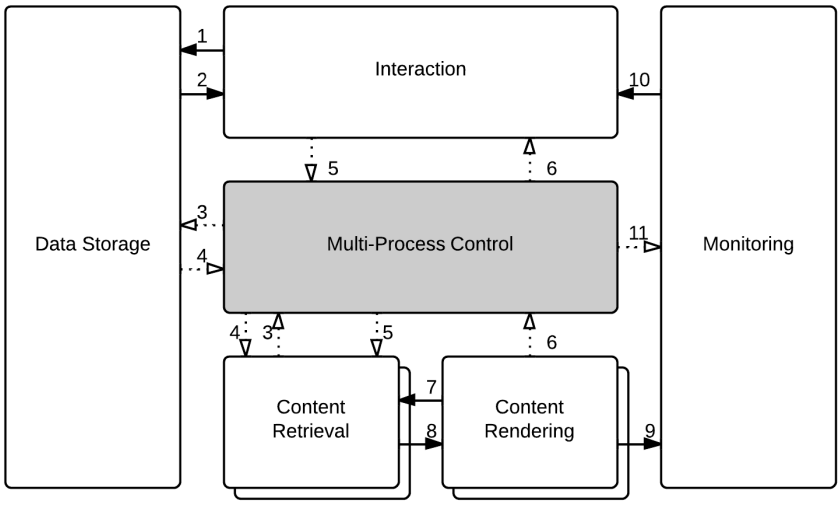


Figure 5.9: High-level functional architecture of a multi-process browser

We start by adapting the high-level functional architecture constructed in the previous section by adding the requirements discussed in the first section of this chapter. This results in the functional architecture seen in figure 5.9. This means adding the required module in between the interface module and those concerned with the retrieval and rendering of content. This results in some changes to the flow of the program, which have also been made visually distinct.

3. Requests to access or store browsing or login data from the data storage module will need to be handled through the multi-process control module, so as to prevent concurrent requests. This also covers arrow 4.

5. As expected, the requesting and receiving of a website will have to run through the multi-process control module, as this is responsible for connecting the right process to its accompanying tab in the interface. This also covers arrow 6.

11. One additional flow arrow was added to the architecture to represent the multi-process controller reporting the failing of a process and its memory usage.

The URL blacklisting functionality of the Monitoring module was not modeled in the global overview because this would negatively impact readability. The online syncing of data was not depicted in the model because it is not a part of the primary process and likewise would complicate the model too much. These functions are modeled in the module view, which allows for more detail.

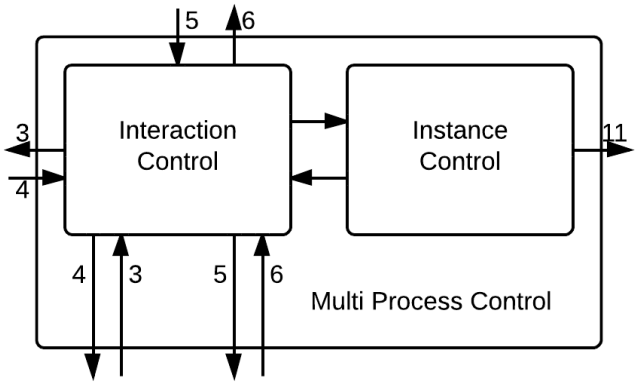
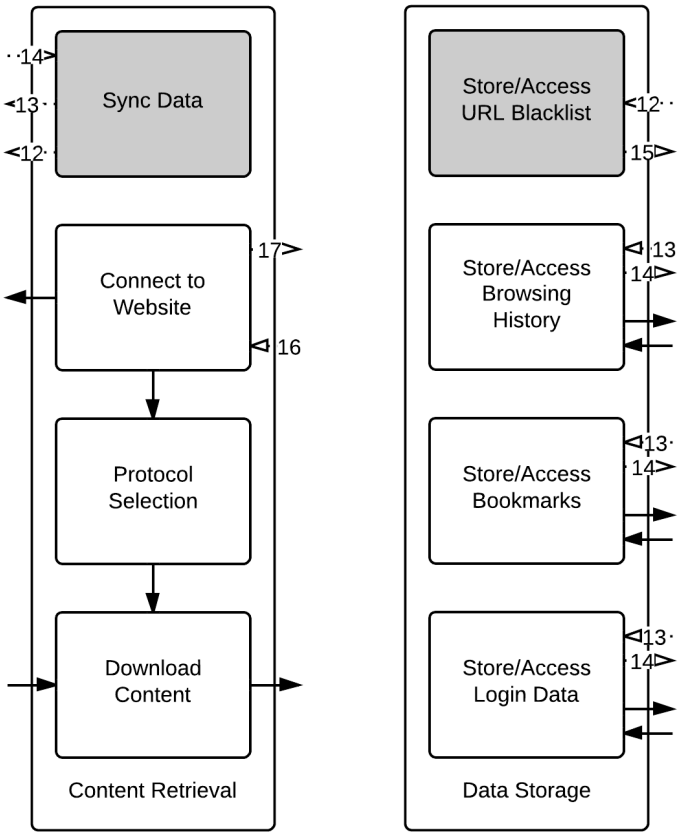


Figure 5.10: Functional module of the browser's multi-process controller

The additional or altered flows created by the new requirements were numbered to separate them from the basic browser design to better illustrate the changes made. The newly added multi-process module (figure 5.10) consists of two main functions, those being interaction control and instance control. Interaction control is what allows the sandboxed processes to interact with the rest of the browsers' modules and, more importantly, deliver content to the user. Instance control is what is responsible for the starting, stopping and controlling of the multiple processes (or instances) of the browsing engine.

The changes made by the URL Blacklisting and Online Syncing requirements become clear in figures 5.11a and 5.11b. In figure 5.11a we see the addition of a function to sync data with an online source, which updates the Data Storage with the URL Blacklist (flow 12) and the user's History, Bookmarks and Login Data (flow 13). The module also backups the local data by updating the online source with the user data, seen in flow 14. The URL Blacklisting functionality in the Monitoring Module seen in figure 5.12 is updated through flow 15. The URL Monitoring function monitors the websites the browser connects to and intervenes when necessary, which is represented by flow 16 and 17.



(a) The content retrieval function (b) The data storage function

Figure 5.11: The content retrieval and data storage modules

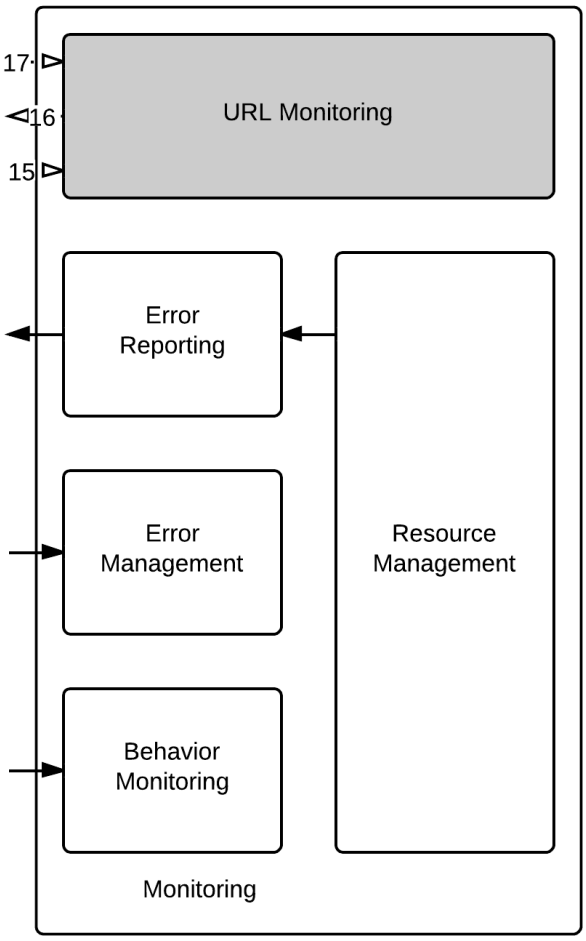


Figure 5.12: Functional module of the browser's monitoring function

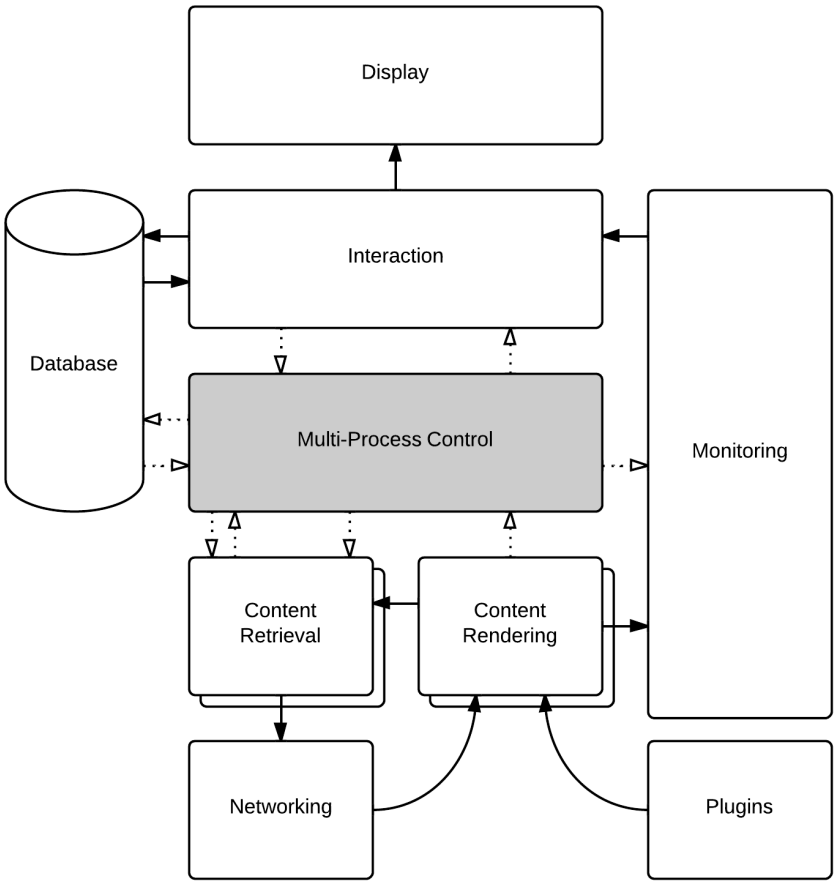


Figure 5.13: High-level technical software architecture of a multi-process browser

Once again using the functional architecture (figure 5.9), the technical architecture seen in figure 5.13 was created. The same purely technical modules seen in the technical architecture of the basic browser (figure 5.8) were added as well, as the reasons for their addition were still valid. We see that the basic structure of the browser remains largely similar to the functional architecture, as well as the technical architecture seen figure 5.8, barring of course the Multi-Process Control module and altered flows. Because the other additions were extensions of already present functional modules, these are not present in the global technical architecture, but are present in a more detailed module view.

5.5 Case Study Findings

In this chapter we performed a case study of a generic web browser based on a set of basic requirements. We then added additional requirements to this generic web browser to create a multi-process browser with common features seen in modern web browsers, such as separating each tab into its own process, online syncing of user data and URL blacklisting. We did this in order to answer our third research question (RQ3): *"What technical consequences can be predicted on changes in the functional software architecture?"*.

To answer this question we first created a generic web browser architecture based on the minimum set of functional requirements. Through the different steps in our method this resulted in a basic technical software architecture. We then incorporated the requirements of the more advanced multi-process browser into this base architecture, in order to find out how this would affect the design of the system. We found that adding a requirement that did not fit within an already established concept feature resulted in the creation of a new feature model and accompanying functional module in the global architecture. This was the case with multi-process control, which had a major impact on the design of the architecture. The other requirements did fit into existing concept features, namely Content Retrieval, Data Storage and Monitoring. For this reason they were added to the already existing modules in the architecture. While the addition of the new Multi-Process Control module resulted in major changes to the flow of the system due to its effect on the primary process, the other modification did not. New flows involving those were thus only modeled in the in-depth module view so as not to negatively impact the readability of the global architecture by excessive elements. The technical architecture created in the final step reflected the changes made in the functionality of the system.

Thus the answer to our second research question is that there is a correlation between functional changes or additions to the system and technical architectural consequences, but the amount to which the design changes is dependent on the type of functionality added. New requirements that fit into or extend already present functionality only change the architecture on the modular level, whereas a completely new functionality results in major changes in the global architecture and the primary process flow.

Chapter 6

Analysis

The analysis in this chapter consists of two sections. In the first, Model Comparison, we compare the models produced in the case study with documented models of functionally similar browsers to gage their correctness. In the second section we review the extent to which the produced models meet the quality attributes, as this was a requirement of the method. The quality attributes are discussed from both a functional and technical viewpoint.

6.1 Model Comparison

To analyze the effectiveness of the modeling method, we will compare the technical architectures created in the previous chapters with technical architectures based on the code of real world browsers. For this we refer to the paper by Grosskurth and Godfrey (2005), who used code analysis tools on a variety of browsers, including Firefox, Epiphany and Konqueror, to create a high-level reference architecture of a regular, basic browser. As this is what we aimed to create in our generic single-process browser case, we would expect these to be largely similar. The difference being that our technical architecture was based purely on functional requirements, whereas the reference architecture created by Grosskurth and Godfrey (2005) came entirely from source code analysis of different browsers. Upon initial inspection the architecture created in the previous chapter, seen in figure 5.8, indeed seems largely similar to the reference architecture seen in figure 6.1. The first section of this chapter will discuss these differences in more detail.

Looking at Grosskurth and Godfrey (2005)'s reference technical architecture (RF) in figure 6.1, we can see the following similarities with the technical architecture (TA) in figure 5.8:

- Interaction (RF) and Interface (TA) are essentially the same. It is described as the "layer between the user and the browser engine" Grosskurth and Godfrey (2005) and allows the manipulating of settings and provides feedback.

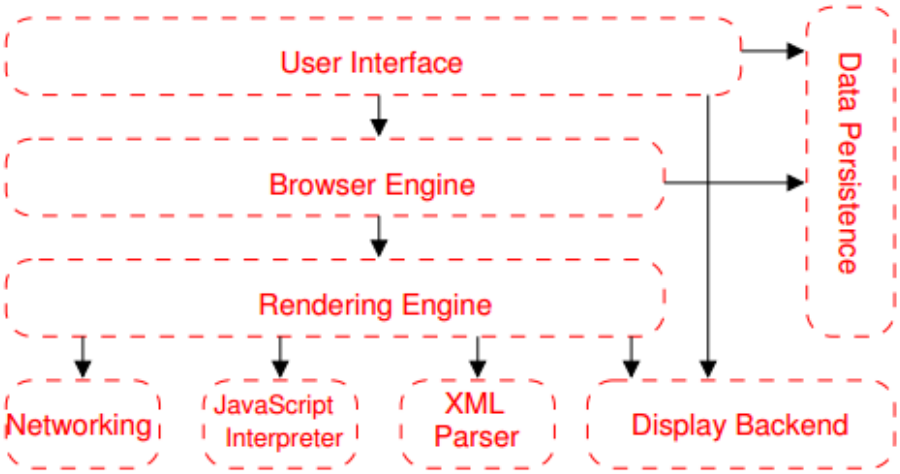


Figure 6.1: Single-process browser reference technical architecture, from Grosskurth and Godfrey (2005)

- Browser Engine (RF) is described as being a high level interface to Rendering Engine (RF), supporting browsing actions such as back, forward and reload and URI loading. When comparing this to our architecture, we see that we placed this functionality in the Content Retrieval module. While the Interface module (TA) contains the commands for back, forward and reload, it is the Content Retrieval module (TA) that actually does this work.
- The Rendering Engine (RF) completely matches the Content Rendering module (TA).
- The JavaScript Interpreter (RF) and XML Parser (RF) are both treated as Plugins (TA) as they both handle content that is not HTML, CSS or pictures.
- Display Backend (RF) provides the drawing function on the screen and thus has the same function as Display (TA).
- The Data Persistence module (RF) stores data such as bookmarks, settings, cache and security certificates. This is the same functionality covered by the Data Storage/Database (TA).
- The Networking module (RF) is described as implementing the various file transfer protocols and general connectivity. While protocol selection is a part of the Content Retrieval module (TA), the actual connectivity is delegated to the Networking module (TA) as described in chapter 5.3.

There seems to be only one major difference between the technical architecture we constructed and the reference architecture: The Monitoring module (TA) is absent in the reference architecture. Grosskurth and Godfrey (2005) mention the Browsing and Rendering Engines as being able to interrupt processes and handle errors, so we can assume that our separate Monitoring functionality is integrated into the other components in the reference architecture.

For the analysis of the multi-process browser we will focus on the modules and submodules we added to incorporate the additional requirements. Because the addition of the multi-process functionality resulted in changes in the global architecture, we will start our analysis with a comparison of the technical architecture seen in figure 5.13 with an overview of the multi-process architecture provided by Google in the Chromium documentation, seen in figure 6.2.

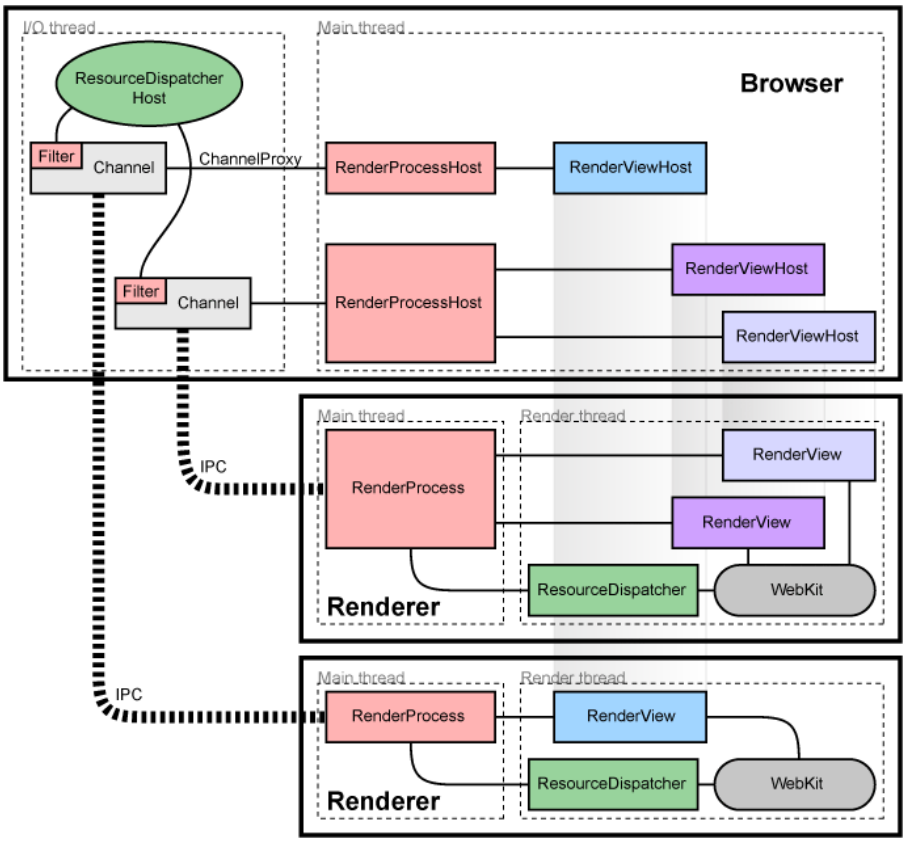


Figure 6.2: Chrome/Chromium technical architecture of the multi-process functionality

This model is a lot more in-depth than the model created from our basic functional requirements so we will go through it step by step. Chrome combines the UI and the tab and plugin processes in what they refer to as the "browser process" or "browser", while the tab-specific processes are referred to as the "render processes" or "renderers". What this means is that what Chrome refers to as the browser combines the functionality of the Multi-Process Control and the Interface modules. The functions described within the Multi-Process Control module are clearly present though, in the form of the Resource Dispatcher Host (Instance Control) and the various Render Hosts (Interaction Control).

Unfortunately, the documentation for Chromium is highly focused on its technical implementation. This means that the information available on the syncing functionality, as well as the URL blacklisting functionality is mainly focused on how this functions in the code, with an architectural overview, such as the one seen for Multi-Process Control in figure 6.2 not being present in the available online resources. What we were able to find is that the syncing code is located in a subdirectory of the browser module (*chrome/browser/sync*), ties into the engine and updates the local data using sqlite. The updates to and from the server are handled using XMPP, an instant messaging protocol, to allow for push based syncing. This means that when a change happens in the user data on one client, for instance a new bookmark, the client can push this update to the server, which in turn can push this to the other connected clients.

Information on the URL blacklisting feature was found as a part of the "Safe Browsing" documentation, which explains how this functionality is handled in the actual technical running of the program. When a new tab process is started by the ResourceDispatcherHost (seen in figure 6.2), the first process loaded therein is a "SafeBrowsingResourceHandler" which has first say on whether a resource (such as a site or site content) is loaded. This is roughly where we expected it to be in our architecture, namely the submodule "Connect to Website" seen in figure 5.11a.

6.2 Software Architecture Quality Attributes

One of the goals of our method was to ensure the produced architectures would adhere to the quality attributes identified by Bass et al. (2012). These are defined as "a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders". We will now analyze if and to what extent the produced software architecture models adhere to the quality attributes. We will look at both the functional and technical aspects of each quality attribute, and will attempt to measure to what extent each has been satisfied in the produced architectures. This will be graded using a five-point scale going from very low to very high, with the center score ("neutral") used if the quality attribute was not applicable in our case study.

6.2.1 Availability

Functional aspect: *High*.

The quality attribute of availability, which deals with the system's ability to handle faults and general problems is partially addressed through the functional requirement for a monitoring function to handle errors and unforeseen behavior. This is expanded upon in the multi-process design by creating a separate process for each tab.

Technical aspect: *Neutral*

Availability is a core part of the design, but due to the absence of technical requirements this was not elaborated on. We would expect these to be added to the accompanying documentation in later stages of the design process.

6.2.2 Interoperability

Functional aspect: *Very High*

Functionally, the system consists of a set of modules with distinct responsibilities working together. The primary process depends on these modules inter-operating, with their functionality being separate and specialized. Simply put, when using the FAM Method the system **must** have high interoperability in order to function.

Technical aspect: *High*

Interoperability is largely a technical quality attribute in practice, so we can suppose that the separation of functionality into distinct modules would encourage designers to create these modules in such a way that they can be programmed independently of each other and communicate through a previously agreed upon set of interface guidelines, thus also ensuring their re-usability in other projects that may share some functional requirements. See for instance the many smartphone apps that function as an interface for websites, and are essentially built upon specific browser modules but output their content into their own user interface instead of a webpage.

6.2.3 Modifiability

Functional aspect: *Very High*

As can be read in the previous chapter, the system is highly modifiable. It can handle the addition of new functionality which alters the flow of the primary process without impacting the rest of the architecture to a major degree. Extending existing functionality with new requirements does not significantly alter the global architecture due to the selfcontained nature of each module.

Technical aspect: *High*

A modular design, with each module functioning independently yet interoperational with the other modules is inherently relatively easy to modify, as each module can be changed without affecting the others. As the FAM Method ensures modular design through its approach to architecture design this quality attribute is ensured in all cases. As for modifiability of the code itself, this, like

availability, should be added into the documentation in a later stage as a hard specification of coding rules.

6.2.4 Performance

Functional aspect: *Very low*

Performance concerns the system's ability to handle events, requests or operations in a timely manner. Performance has historically been one of the most important quality attributes in system development, though with the advent of cheaper hardware and scaling cloud services this is changing. It is, however, a purely technical quality attribute. These is simply no way to approach a performance requirement from a functional standpoint. For this reason the rating is very low.

Technical aspect: *Neutral*

Performance is not addressed in the architecture itself and should be added in the accompanying documentation. This is because this can not be visualized during the functional or technical design stages. Technical requirements may indicate specific performance requirements for each module, as well as the system as a whole. Performance requirements should be added to the documentation in a later stage of the design process as hard specifications and constraints.

6.2.5 Security

Functional aspect: *Low*

Separating the system's functionality into independent modules, and using a common interface that only exposes that information that is required for the system as a whole to function encourages some security through isolation. In the case study, the Monitoring module further increases security by monitoring each module for errors or strange behavior independently of that module. The multi-process browser adds URL blacklisting to warn the user if an attempt is made to connect to a site containing malware, or one associated with phishing. Aside from this though, there is only so much security to be reached through architecture designs.

Technical aspect: *Neutral*

Security remains a largely technical quality attribute. The ability to protect the system from unauthorized access and prevent data tampering is highly dependent on the technical implementation of the system. Security elements such as encryption, sanitized inputs and rule-based monitoring are based in the code and documentation and dependent on the technical requirements of the system, as well as its intended user base and implementation context.

6.2.6 Testability

Functional aspect: *High*

Testability refers to the ease with which faults and errors can be detected in the system. For a system to be properly testable, each module's inputs and outputs should be controllable. The architectures produced by the method are highly modular in nature with distinct functionality. This aids the testing process not only because each modules' input and output is known, but also because errors in the primary process can quickly be assigned to each functionality. Functionally, the Monitoring module also aids in this process as its function is specifically to detect faults and errors in the program.

Technical aspect: *High*

The independent nature of each module allows it to be tested by controlling its input and output, thus enabling developers to start testing the software as early in the design process as possible, even when other modules aren't done yet. Another part of testability identified by Bass et al. (2012) is to limit the complexity of the system through high coherence and loose coupling, both aspects of the modifiability quality attribute. Because the FAM method separates each functionality into its own module, a practice encouraged in system design as the "do one thing and do it well" rule Raymond (2003), this quality attribute synergizes well with the high modifiability of the system.

6.2.7 Usability

Functional aspect: *High*

Usability is concerned with how easy it is for the user to accomplish goals and tasks with the system. It also governs the amount of support the system provides to the user with these tasks. This is handled in the design of the system by only exposing that functionality which is relevant to the user in the User Interaction module, and minimizing the impact of error in the system through use of the Monitoring module. In the multi-process browser this is expanded upon through the URL blacklisting functionality by warning the user if they attempt to connect to a known phishing site, or a site containing malware.

Technical aspect: *Low*

Usability, more than any other of the quality attributes, is highly dependent on the intended end-user of the system. In our case study, most usability requirements were addressed in the functional requirements of the User Interaction module, which handles the interface elements and provides all functionality exposed to the user. Most other usability requirements remain either as hard specifications and constraints in the documentation, or are addressed through interface design.

6.3 Summary of Analysis

The purpose of this chapter was to answer our second research question (RQ2): *"To what extent are properties and quality attributes of the technical software architecture influenced by using functional software architecture modeling?"*.

We investigated this by first comparing the produced architectures to their real world counterparts, in order to prove that the produced models are essentially correct. We found that they were largely similar, and explained the discrepancies were they were not. We then analyzed the extent to which each quality attribute was met by the architectures, from both a functional and technical perspective. We found that the quality attributes interoperability, modifiability and testability are met in both perspectives through use of the method. Availability, security and usability were met to some degree from a functional perspective, but required technical requirements to be fully met. Performance we found not to be influenced by functional software architecture modeling at all, it being a quality attribute without a functional aspect.

What this means in regards to our second research question is that the properties of the technical architecture are highly dependent on its functional architecture, making the influence of the FAM method extensive. Its influence on quality attributes is more varied. Interoperability, modifiability and testability are inherent to the FAM method, whereas the others are more dependent on extensive requirements analysis, especially in regards to their technical perspectives.

Chapter 7

Evaluation & Discussion

This chapter contains the evaluation of our method, as well as a discussion of the research performed in this thesis. In the first section, we discuss the feedback we received from several programmers who we presented and explained our method to. Following that, we evaluate to what extent the initial requirements for the FAM method have been met based on the results from our case study. After this we discuss how complete the models produced were. Finally, the validity of the performed research is discussed.

7.1 Developer Feedback

During the course of this research we communicated with developers associated with various projects related to our research. We took the opportunity to present our method and case study results to each to gain their feedback. The developers were all programmers, and were associated with the Gnome, Chrome/Chromium and CyanogenMod open source projects. Due to the nature of open source development the function of each member of a team is often more than "just" programming, oftentimes including the function of project manager, architect, requirements analyst and whatever else is needed.

Feedback was unanimous in that the method provides an easy way to communicate their design ideas to other members of the project and in documentation. Due to the decentralized nature of open source development efficient communication is highly important, yet this is also one of the harder tasks to do given that most of this communicating is done asynchronously, such as through email and wiki's. Given that this communication is done between skilled programmers, contact about the programs is often highly technical and uses code-snippets, which results in a barrier for new entrants. The simplicity of the models combined with the fast method of generation was found to be highly advantageous, as it is a relatively simple step to move from simply sketching an architecture from memory to using the method to generate one based on available information.

As to the produced models, the opinions were more varied. Given that most of the Chrome/Chromium development is done by very skilled and paid programmers in a somewhat central setting, their focus is far more on technical depth in documentation. This is because they can rightfully assume that all project members are intimately familiar with the program or will bring themselves up to speed quickly because it is a part of their job. Volunteer-run open source projects though are not able to place such demands upon their prospective members and thus require an introduction to the software to be as smooth as possible, something that the models produced by the FAM method excel at. A major advantage in their opinion was that it clearly communicated what each module of the software was supposed to do without having to read design documents or code.

Aside from that, we found that software architecture did not play a very large role in each of these projects. For Chrome/Chromium, this was because most architecture visualization was done after the initial code was written and most of the design is done on the purely technical level due to the expertise of the associated developers. There is simply no need for a visualization of the functionality of the product, as there are no stakeholders to benefit from it. With Gnome, the focus of the projects are on the user experience, so most development time before and during coding is set aside for interface design and usability. As most of the work is done on an already existing codebase there is little incentive to design something again from the ground up and with the project being as large as it is, modeling the existing code is a daunting task. With CyanogenMod we see something similar, as this builds upon the Android source code. Though in the case of new subprojects that are not based on existing code, a design process is used with one or more iterations of requirements analysis. This is also where adoption of the FAM method has been considered, as it would allow coding to start sooner while still incorporating new requirements and feedback on the design.

7.2 Requirements Evaluation

In this section the results and experiences from the case study are compared against the five requirements identified for the method creation. We will discuss each requirement separately after a short recap of the process by which the models were created.

Once the requirements were gathered the initial (or version 1.0) of the architecture was created within a day. Requirements were easily grouped within their respective concept features, and these naturally translated into the functional architecture. The functional architecture in turn logically evolved into a technical one, with extra technical modules added where they were deemed to be technically required and followed from the functional requirements.

Looking at each requirement individually we come to the following conclusions:

1: Be quick and easy to use

As stated previously, once the requirements were known it was a relatively quick process to group them into feature models along their respective concept features. These were then transposed upon the Functional Architecture diagram and its modules, and accompanied by documentation. Information flow between the modules followed logically from the requirements. Upon transitioning to a technical architecture additional modules were added where necessary, based on common and well-known dependencies, such as a display and networking layer. The use of an informal notation allowed for a quick design stage without being hampered by unfamiliar rulesets.

2: Allow for communication with stakeholders

The produced diagrams, and use of modules to obscure specific functional elements from the global view resulted in diagrams with few elements and flows, making them easy to understand for stakeholders. Each functional module's decompositional view was aided with documentation explaining its general functioning. Because of our notation, no prior knowledge of ADLs or UML, or even the software architecture field in general, would have been required to communicate with stakeholders, while the technical models themselves provided enough depth to give programmers a good starting point to add requirements and specifications. This was confirmed by the similarity of the technical architecture models with their code-derived and documented architectures, as seen in chapter 6.1.

3: Support multiple views

This was tested by providing a process view of the activity of opening a website, which is one of the main functions of a browser. This was done by mapping out the sequence in which the earlier identified functions would be utilized in the process according to the information flow. Furthermore, the method advocates using a global architecture view to model the primary process and using a decomposition of each module to show its detailed functions. This is done to prevent information overload for the stakeholders.

4: Avoid restricting architects

The method was flexible enough to allow the use of a informally specified notation scheme without compromising on the essential correctness of the produced models, as can be seen in the comparison with the official architectures in chapter 6.1. The FAM Method would also allow for a more formal notation if the architect or stakeholder would require this, as this does not interfere with the basic steps the method is comprised of.

5: Integrate with established QA's and principles of Bass et al. (2012)

As can be seen in chapter 6.2, most of the quality attributes identified by Bass et al. (2012) are inherently encouraged or ensured by following the steps of the FAM Method. Some however are not as easily visualized, or cannot be visualized, leaving their inclusion a matter of proper documentation of the requirements of the system. In general though, it can be stated that use of the FAM Method will result in a technical software architecture that is both interoperable, modifiable and testable, with some security and availability through the explicit separation of functional responsibilities.

7.3 Completeness

During the quality attributes evaluation, we found that some functional and technical requirements can not be visualized in an architecture diagram or approached through our architecture design method without gathering more technical requirements. Every diagram is accompanied by documentation explaining its functionality and information flows, also allowing the designer to model flows not present in the global architecture because of readability issues. The accompanying documentation to each architecture would also include technical requirements such as performance or availability in later stages of the design process.

What we found remarkable is that even without taking any technical requirements into account, and only using a very basic set of functional requirements, the produced technical software architectures are so similar to the real world browser we see in the reference browser architecture derived from code analysis in the paper of Grosskurth and Godfrey (2005) and official Google Chrome documentation. This supports the thought that when using a full set of functional and technical requirements in the FAM Method the produced architectures would be of comparable or higher quality than the ones currently used in the field of web browsers.

7.4 Validity

To confirm the quality of the research performed in this thesis, we will now discuss several factors regarding validity that were taken into account during the writing of this thesis:

- Construct validity: Construct validity is concerned with identifying the correct concepts and measuring them correctly. The construct validity was guaranteed by deriving the requirements of our method from established research in the field of software architecture and expanding upon a proven method. Furthermore, we measured the results of our method, the

resulting architectures, to both their existing documented counterparts and to established quality attributes of software architecture.

- Internal validity: The internal validity is to determine if the results really follow from the gathered data. Internal validity was assured by documenting our method design in chapter three and four and the design process of our architectures in chapter five.

  In chapter three we answered our first research question by researching literature in the field of software architecture with a focus on functional modeling, followed by an analysis of established architecture modeling methods, also known as Architecture Description Languages. We then did further research on these methods as well as other requirements in software architecture use with the intent to produce a method that allowed for the creation of software architecture that is of a high quality and usable by its intended audience. In chapter four we used the identified method requirements to adapt an existing method, and integrated this in a general software architecture design process.

  In chapter five the architectures were constructed using only the functional requirements stated at the beginning of the chapter. Each step of the method was explained and documented, along with the produced models. We then used these architectures and the analysis thereof to answer our second and third research question.

- External validity: External validity was tested in chapter six, where we compared the models we produced with a reference architecture established from code analysis in the case of the basic browser. In the case of the multi-process browser, we compared our results to the available Chromium documentation. The Chrome/Chromium browser served as an inspiration for the requirements we added for the multi-process browser, so we compared the official documentation on how these features were implemented with how they were implemented in our architecture.

- Reliability: Reliability indicates whether repeating the case study will provide the same results. By carefully documenting the research approach and the protocol, as well as the requirements and criteria of the developed method, the repeatability of the method design was ensured. For our case study, we documented the requirements we used in the method and the requirements we used to adapt our initial architecture, as well as the decisions and inferences made during the design process.

Chapter 8

Conclusion

In this final chapter we discuss the overall findings of this thesis and future research options. In the first section we provide a summary of the research performed and state how each research question was answered, as well as the overall worth of this product in the field of software architecture. This is followed by a discussion on future research based on this research.

8.1 Overall Findings

In this thesis we set out to answer the main question "*How can functionality be expressed in models of software architecture?*". This question was resolved by the creation of a Functional Architecture Modeling Method that utilized functionality as the main design input in the creation of a Software Architecture. We created this method by researching both the extent to which functionality is addressed in the current method used in technical software architecture modeling and what requirements a software architecture modeling method should meet to be used by software architects.

We found that functional architecture modeling was separate from the field of technical software architecture modeling, with only Salfischberger et al. (2011) making an attempt to unite the fields using the Functional Architecture Framework. Methods explicitly designed for software architecture modeling, the Architecture Description Languages, almost unanimously focused on highly formal technical software architectures and turned out not to meet the requirements of most software architects in practice. This provided us with the answer to our first research question (RQ1), namely that functionality was not addressed in the current methods of technical software architecture modeling.

Using the criticisms leveled against ADLs in the paper by Woods and Hilliard (2005), we derived a set of requirements for our method with a focus on practical usability while maintaining essential correctness and adherence to existing standards. We achieved this by creating a method flexible enough to allow the use of strict ADLs, while simultaneously also allowing the use of informal "sketching"

notations. We achieved this by integrating a functional modeling step into the software architecture design process.

We then tested our method in a case study on two browsers, one being as basic as they come and another with the newer feature of process separation. We focused on the absolute basic set of requirements for a browser and did not take any technical constraints into account so as to compare our purely functionally derived technical software architectures with the most closely matching real-world versions, which are often the result of an evolving code with little focus on modeling. Upon comparing the technical software architectures we produced in the case study with the official architectures, we found that they largely matched. Evaluating our models and documentation to the quality attributes identified by Bass et al. (2012), we found that these were also mostly satisfied, with the degree to which they weren't highly dependent on our use of an absolute minimum of functional requirements, and no technical ones. This answered our second research question (RQ2), namely that the properties and quality attributes of the technical software architectures turned out to be highly dependent on the functional architecture. Its use in fact almost seems to guarantee the satisfying of certain quality attributes.

The case study on two browsers, with one having additional functional requirements was done to answer our third research question (RQ3), which concerned itself with what technical consequences could be predicted by changes in the functional software architecture. We tested this by adding several requirements to our basic browser design, thus creating a new functional software architecture which resulted in an updated technical software architecture. Upon comparing the resulting technical software architecture with the official documentation we found that the addition of a specific functionality did result in the presence of a new module in the technical software architecture addressing that functionality, with corresponding changes in the information flow within the system. Extending the functionality of existing modules did not affect the global architecture. This confirmed that the addition of specific functionality did result in technical consequences if the functionality is distinct enough from the already met feature models.

This research expands the field of software architecture by first proving that existing methods which focused on exhaustively modeling the entire system did not meet its users' needs. We then identified these needs in a literature research and combined them into a set of requirements for a user-centric architecture modeling method. We then adapted a stakeholder-focused method according to these identified requirements and integrated it into the general software architecture design process. Through a case study we then proved the correct functioning of this method, proving that a user-centric method can still produce high quality architectures while meeting the needs of its intended audience, as opposed to the previous system-centric methods which sacrificed usability to focus on technical correctness.

8.2 Future Research

To maintain a reasonable size, our case study consisted of the minimum set of functional requirements to achieve a functioning browser. In a future research, we would use the method on a fully documented and developed system. This would allow for the final technical software architecture to be compared to one derived from a traditional technical software architecture modeling method, while also allowing for a comparison of the time used and ease of use of our method compared to the original method. A completely documented architecture would also allow for a full evaluation of the quality attributes present in the resulting design, as our set of requirements did not result in any performance related requirements. Finally, for completeness sake, we would also use a formal notation in the design phase to confirm that the method indeed allows for their use without issue.

Bibliography

Bass, L., Clements, P., and Kazman, R. (2012). *Software architecture in practice*. Addison-Wesley Professional.

Brinkkemper, S. and Pachidi, S. (2010). Functional architecture modeling for the software product industry. In *Software Architecture*, pages 198–213. Springer.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. (2010). *Documenting software architectures: views and beyond*. Addison-Wesley Professional.

Clements, P. C. (1996). A survey of architecture description languages. In *Proceedings of the 8th international workshop on software specification and design*, page 16. IEEE Computer Society.

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M. (1992). Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(01):31–57.

Garlan, D. and Shaw, M. (1993). An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1:1–40.

Grosskurth, A. and Godfrey, M. W. (2005). A reference architecture for web browsers. pages 661–664.

Hevner, A., March, S., Park, J., and Ram, S. (2004). Design science in information systems research. *Mis Quarterly*, 28(1):75–105.

Medvidovic, N. (1999). Adl examples. http://sunset.usc.edu/ neno/teaching/s99/February25.pdf. Accessed: 2013-08-20.

Nuseibeh, B. (2001). Weaving together requirements and architectures. *Computer*, 34(3):115–119.

Pandey, R. K. (2010). Architectural description languages (adls) vs uml: a review. *SIGSOFT Softw. Eng. Notes*, 35(3):1–5.

Raymond, E. S. (2003). *The art of Unix programming*. Addison-Wesley Professional.

Riebisch, M. (2003). Towards a more precise definition of feature models. *Modelling Variability for Object-Oriented Product Lines*, pages 64–76.

Salfischberger, T., van de Weerd, I., and Brinkkemper, S. (2011). The functional architecture framework for organizing high volume requirements management. In *Software Product Management (IWSPM), 2011 Fifth International Workshop on*, pages 17–25. IEEE.

Takeda, H., Veerkamp, P., and Yoshikawa, H. (1990). Modeling design process. *AI magazine*, 11(4):37.

Van Vliet, H. (2008). *Software engineering : principles and practice*. John Wiley Sons, Chichester, England Hoboken, NJ.

Woods, E. and Hilliard, R. (2005). Architecture description languages in practice session report. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 243–246. IEEE.