# SOFTWARE ARCHITECTURE QUALITY EVALUATION

## APPROACHES IN AN INDUSTRIAL CONTEXT

Frans Mårtensson

Blekinge Institute of Technology
Licentiate Dissertation Series No. 2006:03

School of Engineering

# Software Architecture Quality Evaluation - Approaches in an Industrial Context

Frans Mårtensson

Blekinge Institute of Technology Licentiate Dissertation Series
No 2006:03
ISSN 1650-2140
ISBN 91-7295-082-X

# Software Architecture Quality Evaluation - Approaches in an Industrial Context

**Frans Mårtensson**


Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
SWEDEN

Department of Systems and Software Engineering
School of Engineering
Publisher: Blekinge Institute of Technology
Printed by Kaserntryckeriet, Karlskrona, Sweden 2006
ISBN 91-7295-082-X

# Abstract

Software architecture has been identified as an increasingly important part of software development. The software architecture helps the developer of a software system to define the internal structure of the system. Several methods for evaluating software architectures have been proposed in order to assist the developer in creating a software architecture that will have a potential to fulfil the requirements on the system. Many of the evaluation methods focus on evaluation of a single quality attribute. However, in an industrial system there are normally requirements on several quality aspects of the system. Therefore, an architecture evaluation method that addresses multiple quality attributes, e.g., performance, maintainability, testability, and portability, would be more beneficial.

This thesis presents research towards a method for evaluation of multiple quality attributes using one software architecture evaluation method. A prototype-based evaluation method is proposed that enables evaluation of multiple quality attributes using components of a system and an approximation of its intended runtime environment. The method is applied in an industrial case study where communication components in a distributed real-time system are evaluated. The evaluation addresses performance, maintainability, and portability for three alternative components using a single set of software architecture models and a prototype framework. The prototype framework enables the evaluation of different components and component configurations in the software architecture while collecting data in an objective way. Finally, this thesis presents initial work towards incorporating evaluation of testability into the method. This is done through an investigation of how testability is interpreted by different organizational roles in a software developing organization and which measures of source code that they consider affecting testability.

# Acknowledgments

I would first like to thank my advisors Dr. Michael Mattsson and Dr. Håkan Grahn for all your help and support during the years leading up to the completion this thesis. Your patience, guidance, and good advice are what made this possible.

I am also thankful to Professor Claes Wohlin for comments and guidance during the completion of this thesis. And to my colleagues in the SERL research group and the BESQ research project. You provide a very creative and fun work environment.

I would like to thank the people at Danaher Motion Särö AB. You have provided an industrial point of view and a professional organization where our studies could be performed. In particular I would like to thank Henrik Eriksson and Jonas Rahm.

My friends in Lunchmobben shall of course be mentioned. Over the years you have provided many relevant comments and questions hidden in a roar of completely irrelevant comments and bad puns.

Finally, I would like to thank my parents, Håkan and Kerstin, for inspiring and encouraging me to find my own path. My sister Tina that always seems to be so far away, but always is close to my heart. Finally my love Kim, you are always there and you are my dearest friend. You all support me when things are hard and cheer me on when things go well.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project “Blekinge - Engineering Software Qualities (BESQ)” http://www.bth.se/besq.

# Table of Contents

CHAPTER

# 1

# Introduction

Software plays an increasingly large role in our society. Many services that we take for granted would have been impossible to create without the use of software. Software systems are in turn becoming more and more complex. The number and complexity of the functions that software systems provide continuously increase. Development of new technologies and reuse of existing technologies further complicate the development of new systems. Systems have to take in consideration not only existing systems that they interact with, but also possible future changes. These considerations are present and have to be managed throughout the lifecycle of software systems.

Software architecture has been introduced to help manage complexity and risk during development and maintenance of software systems. It guides and documents the structure of the system as well as the responsibilities of the system's components within this structure. The software architecture is not only based on requirements for functions but also requirements for different qualities of the software system. The quality requirements can be used as a basis for evaluation of one, or comparison of two software architectures. Bengtsson [17] describes four main situations

where software architecture evaluations can be used. The first is when two different software architectures have to be compared (A versus B), and we want to find out which architecture that is the best. The second situation is when we want to evaluate changes that have been introduced in an architecture (A versus A'). Will the changes of the architecture improve the system? The third situation is when we want to see how well an architecture fulfils a quality attribute. Finally, in the fourth situation, an architecture can be compared to an alternative theoretical architecture. By evaluating architectural alternatives it becomes possible to select the most promising software architecture for continued development. The exploration of alternatives increases the developers' trust in that the final architecture is the one best suited for fulfilling the systems requirements. This also decreases the risk that the development project will fail.

The main contribution of this thesis is the work towards a software architecture evaluation method for addressing multiple quality attributes. We present a survey of evaluation methods that can be used to address four important quality attributes: performance, maintainability, testability, and portability. We present an evaluation of continuous simulation for performance evaluation of software architectures and continue with a method for performing prototype-based software architecture evaluations. We then apply the prototype-based method, in cooperation with an industrial partner, to evaluate how it can be used to address performance, maintainability, and portability. In preparation for continued research on evaluation of testability we investigate how testability is perceived by different roles in a software developing organization.

In this chapter we give an introduction to the field of software architecture, we introduce the research questions that this thesis focuses on, and the research methodologies applied to examine the questions. We describe the papers that are in the remaining chapters and their contributions towards answering the research questions. Finally, we discuss future research directions based on our results.

## 1.1 Software Development Process

Software engineering is an engineering discipline that concerns the development of software. As an engineering discipline it is focused on making things work, applying theories, methods, and tools when and where it is appropriate.

In order to better understand the concept of software architecture and enable us to see where it fits in the software development process, we give a short introduction to software development. In this context we then describe the use of software architecture.

Software development is usually performed in a development project. The project identifies requirements from a customer or intended market for the system. Then it analyses, designs, and implement a software system that will fulfil these requirements. Several tasks are performed by the people in the development project resulting in the creation of a number of artefacts, e.g., requirements specifications, prototypes, and an implementation. In order to make the tasks easier and more structured, the people in the development project usually follow a software development process.

A software development process describes activities that have to be done and when, during the development, they should be done. A process can also describe which roles that should exist in the development organization. Examples of roles are requirements engineer, programmer, software architect, tester, and manager. The process defines each role's tasks and responsibilities. Depending on how far the development has progressed, the number of people in each role may vary. In essence, the process helps to structure the developers' work and make the development process less unpredictable.

Most software development processes can be described using a general set of activities (or collections of activities) for software development. These have been described by Sommerville [84] as:

1. Software specification.
2. Software development.
3. Software validation.
4. Software evolution.

**Software specification.** This activity concerns the identification of requirements from the intended users of the system. The requirements are verified and prioritized so that the most important requirements can be identified. Having a clear set of requirements is important for the success of the project. If the wrong functionality is implemented it does not matter how well it is implemented, it is still wrong.

**Software development.** The requirements are used as input for the creation or modification of a software architecture for the system. The architecture describes the high level components of the system, their responsibilities, and interactions. The architecture is then further refined into a more detailed design for the system. Finally the implementation is done according to the design.

**Software validation.** The system has to be validated to make sure that it fulfils the requirements gathered during the specification phase. Defects that are identified should be corrected so that the software system that is delivered to the customer is as correct as possible. Many methods for testing a system exist and which method that should be used depends on the requirement that is being tested.

**Software evolution.** The development of a software system does not end once it has been delivered to the customer. A system might be in use during ten to twenty years and the likelihood that changes have to be made to it during this time is high. New functionality can be added and changes to existing functionality can occur. Evolution can be seen as a form of maintenance, which is the term that we use in the remainder of this thesis.

These basic activities are present in most software development processes, for example waterfall [73] and iterative development methods [53] such as Extreme Programming [16]. In the waterfall development method, the development progresses from one activity to the following, putting the focus on one activity at a time. In iterative development, the activities of the development process are repeated over a number of smaller iterations. Some requirements may be changed for each iteration so that the software system grows incrementally and adapts to changing requirements.

In the model described by Sommerville we find software architecture in the second step, i.e., software development. The soft-

ware architecture is used as input to the continued design of the software system. Sommerville continues to describe a set of design activities within the software development activity:

1. Architectural design
2. Abstract specification
3. Interface design
4. Component design
5. Data structure design
6. Algorithm design

**Architectural design.** The activity of identifying the sub-systems of the software system. The sub-systems' relations are identified and documented.

**Abstract specification.** Each sub-system that was identified is assigned responsibilities and constraints that it has to fulfil. The necessary services are identified and documented.

**Interface design.** The interfaces that enable the interaction between the sub-systems are designed.

**Component design.** The responsibility for the services previously identified is assigned to components. The interfaces for the components are designed.

**Data structure design.** The data structures that will be used in the services and for communication between services are specified and documented.

**Algorithm design.** The algorithms that will be used in the services are selected or designed.

## 1.2 Functional and Quality Requirements

A requirement is defined in the encyclopedia of software engineering [61] as:

> *"A requirement is an externally observable characteristic of a desired system. When trying to determine whether a candidate requirement is really a requirement, two criteria from this definition must be met: it must be externally observable, and it must be desired."*

An externally observable characteristic is something that can be perceived by the user of the system. That the characteristic should be desired means that it has been identified as important to the intended user. Following these two rules helps secure that no unwanted characteristics are introduced in the system. The remaining problem is then to identify a representative set of users for the system and identifying important requirements. The user that will use the system from day to day is an important source of requirements, but a developer can also be a user of the system since the developers might have to maintain the system over a long period of time. This leads to the identification of a number of stakeholders in the system. Each stakeholder has a set of requirements that they want the completed system to fulfil. Stakeholders during development can be different types of users and developers, such as end-user, programmer, and tester.

Requirements are collected from stakeholders during the initial phase of software development. They are commonly divided into two main groups: functional and non-functional. Non-functional and quality requirements are often grouped together [19], but there exist requirements that are non-functional and unrelated to quality, e.g., the choice of programming language. The quality requirements that a software system has to fulfil can in turn be divided into two groups based on the quality they are requesting, i.e., development and operational qualities. A development quality requirement will benefit the developers work, e.g., maintainability, understandability, and flexibility. Operational quality requirements, on the other hand, focus on making the system better from the users point of view, e.g., performance and usability. Depending on the domain and priorities of the users and developers, quality requirements can become both development and operational, such as performance in a real-time system.

A quality attribute is a property of a software system. Compared to a quality requirement that is placed on a software system by a stakeholder; a quality attribute is what the system actually presents once it has been implemented. During the development of an architecture it is therefore important to validate that the architecture has the required quality attributes. This can be done by one or more software architecture evaluations. A deeper discussion of architecture evaluation can be found in Section 1.3.3.

In this thesis we will focus on the following quality attributes: maintainability, performance, testability, and portability. Quality attributes can be hard to define in an unambiguous way,. Standardization bodies such as IEEE [41] and ISO [43] have therefore created standardized definitions for many quality attributes. IEEE standard 610.12-1990 [42] defines the previously mentioned quality attributes as:

**Maintainability.** This is defined as:

> *"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment."*

Maintainability is a multifaceted quality requirement. It incorporates aspects such as readability and understandability of the source code. It is also concerned with testability to some extent as the system has to be re-validated during the maintenance.

**Performance.** Performance is defined as:

> *"The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage."*

There are many aspects of performance, e.g., latency, throughput, and capacity.

**Testability.** Testability is defined as:

> *"The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met."*

We interpret this as the effort needed to validate the system against the requirements. A system with high testability can be validated quickly.

**Portability.** Portability is defined as:

> *"The ease with which a system or component can be transferred from one hardware or software environment to another."*

We interpret this as portability not only between different hardware platforms and operating systems, but also between different virtual machines and versions of frameworks.

## 1.3 Software Architecture

We have previously discussed where in the development process that software architecture has its place. We have also discussed the main influencing factors (requirements) for the software architecture and how they are identified. Now we discuss the definition of software architecture and methods that exist for designing and evaluating architecture alternatives.

### 1.3.1 Definition of Software Architecture

The concept of software architecture has been discussed by for example Shaw and Garland in [81]. The concept has since then evolved. Today there exists a number of definitions with minor differences depending on domain and peoples' experience. However, most definitions share common characteristics that can be exemplified by looking at the definition by Bass et al. [12]:

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."*

This means that the architecture describes which high level components a software system consists of as well as which responsibilities that the components have towards other components in the system. The architecture also describes how these components are organized, both on a conceptual level as well as on a decomposed detailed level since there can be an architectural

structure inside components as well. Finally, the architecture defines which interfaces the components present to other components and which interfaces and components that they use.

### 1.3.2 Designing Software Architectures

Several methods for designing software architecture have been proposed. Examples of such methods are the Siemens' four views method by Hofmeister et al. [40] and the quality attribute-oriented software architecture design method (QASAR) by Bosch [19]. The methods differ on where they put their focus during the design of the architecture.

Which architecture design method that should be used during the development of a software system depends on several factors. It depends on the organization and people working in it and it depends in the requirements on the system. Large development organizations put requirements on well documented and easily understandable architecture documentation so that designers and programmers can understand the architecture [23].

Over the years of software and architecture design, certain problems have been identified as recurring. As developers gained experience in designing systems they found that certain solutions could be applied to the recurring problems and thus simplify the design of the system. This codification of applicable solutions was named patterns and initially focused on solutions to programming problems [30]. When the concept of software architecture was introduced the same thing happened again. Certain ways of organizing architectural elements appeared over and over. The patterns that were identified were described as architectural styles that help the architect to identify and organize components of the architecture [21]. Architectural styles are often associated with quality attributes [51]. By using a certain style the architect gives the system the potential to be better at a desired quality attribute and through that fulfilling its quality requirements. It is however still possible to affect the quality attributes of the system during the design and implementation of the system [37].

Even though a software architecture has been created to fulfil the requirements on the system, it is difficult to guarantee that it actually will fulfil them. Some quality attributes such as performance can be specified using a set of test cases that the system can pass or fail, but several quality attributes, e.g., maintainability and

usability are difficult to validate. The qualitative nature of many quality attributes make them difficult to quantify and therefore it becomes difficult to say if a quality requirement has been fulfilled or not. It is easier to make a qualitative statement that the architecture fulfils the quality requirements to some extent.

### 1.3.3 Evaluating Software Architectures

Architecture evaluations can be performed in one or several stages of a software development process. They can be used to compare and identify strengths and weaknesses in architecture alternatives during the early design stages. They can also be used for evaluation of existing systems in preparation for maintenance or continued development. The evaluations help software developers make sure that a software architecture will be able to fulfil the quality requirements and several approaches for evaluating software architectures have been proposed. The approaches can be divided into four main groups, i.e., experience-based, simulation-based, mathematical modelling, and scenario-based [19].

**Experience-based** evaluations are based on the previous experience and domain knowledge of developers or consultants [4]. People who have encountered the requirements and domain of the software system before can, based on the previous experience, say if a software architecture will be good enough [19].

**Simulation-based** evaluations rely on a high level implementation of some or all of the components in the software architecture and its environment. The simulation can then be used to evaluate quality requirements such as performance and correctness of the architecture. Simulation can also be combined with prototyping, thus prototypes of an architecture can be executed in the intended context of the completed system. Examples of methods in this group are Layered Queuing Networks (LQN) [3] based approaches and event-based methods such as RAPIDE [59, 60]

**Mathematical modelling** uses mathematical proofs and methods for evaluating mainly operational quality requirements such as performance and reliability [75] of the components in the architecture. Mathematical modelling can be combined with simulation to more accurately estimate performance of components in a system.

**Scenario-based** architecture evaluation tries to evaluate a particular quality attribute by creating a scenario profile that forces a very concrete description of the quality requirement. The scenarios from the profile are then used to step through the software architecture and the consequences of the scenario are documented. Several scenario-based evaluation methods have been developed, e.g., the Software Architecture Analysis Method (SAAM) [47], the Architecture Tradeoff Analysis Method (ATAM) [49], and the Architecture Level Modifiability Analysis (ALMA) [17].

Depending on which evaluation method that is used and how the evaluation is performed, the nature of the results vary between qualitative and quantitative. Results of a qualitative nature give an answer to the question “Is architecture A better than architecture B?”, while quantitative results also can address the question “How much better is architecture A than architecture B?”. The use of methods that give quantitative results makes it possible to compare architectures and trace how quality attributes develop over time.

As we have discussed, there exists a number of methods for evaluating different quality attributes of software architectures, e.g., [3, 17, 47, 75]. However, few methods can address several quality attributes. ATAM [49] is probably the best known method that address multiple quality attributes, but the focus of the method is on trade-off between qualities, and not on the quantification of each quality in the trade-off.

Simulation-based and mathematical modelling-based methods usually also focus on one quality attribute such as performance or reliability [3]. Examples of prototypes can also be found but they focus on evaluation of functionality [15] or a single quality attribute [11] and not on the method for creating the prototype.

We think it is important to have a method that can evaluate more than one quality attribute and that the results from the evaluation are quantitative. We try to achieve this by using a simulation-based approach in combination with experience and scenarios to show how it is possible to address more than one quality attribute (performance, maintainability, testability, and portability) during an evaluation.

## 1.4 Research Questions

In this section we introduce and motivate the research questions that have been addressed in this thesis.

The initial questions focus on the need to make comparisons between architecture alternatives based on quantification of a selected set of quality attributes (questions one to four). Questions five and six relate to the testability quality attribute and are relevant for continued work with architecture evaluation, focusing on testing and validation of software systems.

Most of the research questions have been identified in cooperation with our industrial partner. This has helped us make sure that the research has been interesting not only from an academic perspective, but also relevant from an industrial perspective.

***Research Question 1***

*"Which architecture evaluation methods exist that can address one or more of the quality attributes performance, maintainability, testability, and portability?"*

Many methods for evaluation of software architectures have been proposed [47, 49, 54, 56]. Most of these methods only address one quality attribute, which means that several methods have to be used if we want to evaluate more than one quality attributes of a software architecture. Each evaluation method requires different input in the form of different views of the architecture, different documentation, etc. Therefore it becomes relevant to extract as much information as possible from an evaluation. If several quality attributes can be evaluated based on one set of inputs, the process of evaluating a software architecture becomes more efficient.

***Research Question 2***

*"Is it possible to use continuous simulation as a supporting aid for software architecture performance evaluations?"*

Discrete simulation has been shown to be useful in simulation of software architectures [34]. But, if the discrete events in the simulation are abstracted and seen as flows of data between modules

in an architecture, then it might be possible to use continuous simulation for simulating these flows.

***Research Question 3***

*"How can we use prototypes for evaluation of the performance of software architectures?"*

Previous work on prototype-based evaluation that we found focused on the prototypes themselves rather than the method behind their design and implementation [11, 15]. We wanted to explore how accurate the prototypes could be made compared to the completed system. Focus was on the performance quality attribute during the evaluation. Describing the method for creating and analysing the performance of a software architecture prototype opens the way for exploring which other quality attributes that can be evaluated using this approach.

***Research Question 4***

*"Can the prototype-based evaluation approach be used to assist selection of communication components for a real-time system based on comparison of multiple quality attributes?"*

We apply the prototype-based software architecture evaluation method to quantitatively evaluate a set of components for use in a software system. The addition of quality attributes such as maintainability and portability to the evaluation makes the evaluation more exhaustive than evaluating only performance. We think that this is important since the more information that can be collected during an evaluation the more informed the outcome of the evaluation would be.

***Research Question 5***

*"How do different roles in a software development organization perceive the testability quality attribute?"*

This was a question that was put forward by our industry partner. They felt that there were different interpretations of testability in their development organization. The ideal situation would be that all roles shared the same interpretation or at least were aware of the different interpretations that each role used. We wanted to see which differences that existed and how large they were.

***Research Question 6***

*"Which source code metrics do the programmers think are representative for testable code?"*

Making sure that the metrics that are used to evaluate source code are relevant for the developers is important for the credibility of the metric. We asked a number of developers which source code metrics that they thought had a positive or negative impact on the testability of a software system.

In the following sections we discuss the research methods that we have applied to address each question.

## 1.5 Research Methods

A number of methods can be used for data collection in software engineering research. A number of such methods have been identified by Zelkowitz and Wallace [93]. They divide the methods into three main groups: observational, historical, and controlled. Depending on the type of data that is sought and the conditions of the research project, different methods are applicable for the data collection.

**Observational.** Here, the researcher collects data from an ongoing project. The researcher does not have any control or influence over the development process and is strictly an observer. Observational methods are:

- **Project monitoring,** which is a way of passively collecting data during the span of a project.
- **Case studies,** which actively collect relevant data, the collection is a goal of the project.
- **Assertion**, which is used to make comparisons of technologies.
- **Field studies,** which are similar to case studies but less intrusive and covers more than one project simultaneously.

**Historical.** The researcher collects data from already completed projects. This means that the researcher has to rely on the data that can be identified from artefacts that were created during the project. Examples of such artefacts are documents and source code. Historical methods that can be used are:

- **Literature search,** which focus on analysis of papers and documents.
- **Study of legacy data,** which means search for patterns in previously collected data.
- **Study of lessons-learned** searching through lessons-learned documents from completed projects.
- **Static analysis** focuses on analysis of the products' structure, syntax, comments, etc.

**Controlled.** This family of methods makes it easier to replicate an experiment, making it possible to statistically analyse observations. This makes it easier to ensure the validity of experiments based on methods from this group. Controlled methods are:

- **Replicated experiment** runs several instances of an experiment, differences are introduced and outcomes can be studied statistically.
- **Synthetic environment experiment** means that one isolated task is performed over a short period of time by the experiments participants.
- **Dynamic analysis** executes the product, gathering data during runtime.
- **Simulation** uses a model for simulating the product or environment and uses this simulation for the data collection.

A number of these approaches has been used to answer the research questions in this thesis. Specifically, we have used the following methods:

- Literature search
- Simulation
- Dynamic analysis
- Static analysis
- Case study

For the case study we used a questionnaire and telephone interviews for the data collection. Robson [77] describes the two methods for gathering information:

**Self-completion questionnaires** are answered by the respondents independently of the researcher. The questionnaires can be distributed to respondents by for example e-mail, which makes it

easy to distribute questionnaires to respondents distributed over a large geographical area.

**Telephone interviews** are conducted by the researcher when face-to-face interviews are impractical. The drawback is that it is difficult to make sure that the respondent is completely focused on the interview, and that the interaction between the researcher and respondent is more limited than when face-to-face interviews are used.

We now go through the research questions and discuss which research methods that we used for each research question.

> ***Research Question 1***
>
> *"Which architecture evaluation methods exist that can address one or more of the quality attributes performance, maintainability, testability, and portability?"*

To address this question we used a historical method, i.e., literature search. A literature search gathers data from available documentation or literature. This method is inexpensive and non invasive towards an organization.

Conducting a literature search is in a way part of every research project. The literature search is used to gather related work from already published papers. To answer this research question we used it as our primary source of information. We searched through the Inspec and Compendex research databases [27] identifying 240 primary papers which were then reviewed. Resulting in a selection of 25 relevant papers.

> ***Research Question 2***
>
> *"Is it possible to use continuous simulation as a supporting aid for software architecture performance evaluations?"*

For this question we used the simulation method from the group of controlled methods. Simulation can be used if no executable part of a system is available. It is based around the creation of a model of the system or its environment which is then used to gather the data that is needed.

The simulation model was created using a tool for continuous simulation. The model was based on an existing software system and we modelled its architecture using the facilities that the continuous simulation tool provided. The simulation was initialized using data collected from logs of the completed system and several simulation runs were executed.

***Research Question 3***

*"How can we use prototypes for evaluation of the performance of software architectures?"*

For this question we used dynamic analysis from the controlled group of research methods. Dynamic analysis can be used when a system is partially or fully completed, as long as it can be compiled into an executable form. Dynamic analysis focuses on characteristics that can be collected from a system (or part of a system) by instrumenting and executing it.

To explore how the process of developing prototypes should be implemented and where the difficulties are, we started by creating a prototype framework. The framework was used to make the modelling of an architecture faster by minimizing the amount of code that would have to be replicated. The prototype framework handled the communication between the components of the architecture. It also gathered data on events such as when messages were sent or received. The data was then analysed using an analysis program, resulting in a quantitative estimation of how a system based on the architecture would perform.

***Research Question 4***

*"Can the prototype-based evaluation approach be used to assist selection of communication components for a real-time system based on comparison of multiple quality attributes?"*

For this question we used two different methods for collecting our data: static and dynamic analysis. Static analysis is a historical method that gathers data from a partially or fully completed system by parsing source code and counting the occurrence of some feature in the source code. It is for example common to count the number of lines and comments in a system's source code.

We reused the prototype framework created for question 3 and added a tool for static analysis. The static analysis tool parsed through the source code of the components that we wanted to evaluate. The data from the static analysis was used to compute a maintainability index [69] for the components. Performance was evaluated through the dynamic analysis of the time behaviour of the components using the tools previously created. The portability aspects were qualitatively evaluated by comparing the time it took to move prototypes between platforms.

***Research Questions 5 and 6***

*"How do different roles in a software development organization perceive the testability quality attribute?"*

*"Which source code metrics do the programmers think are representative for testable code?"*

For these questions we decided to use a questionnaire for collecting the data. We used self-completion questionnaires to limit our impact on the organization that we were going to study. The use of the questionnaires lets the respondent fill out the form whenever he or she has time. After the initial deadline for responding to the questionnaire ran out, we extended the deadline for a few days and sent out a reminder to the respondents that had not yet responded. After the final deadline we analysed the responses. During the analysis we found a few respondents that deviated significantly from the other respondents in their group. Rather than sending them a new questionnaire we called them and conducted telephone interviews to clear up any questions that we had regarding their original responses.

## 1.6 Contributions of this Thesis

The work presented in this thesis focuses on the development and application of methods for evaluation of software architectures. As previously described, we have focused on methods for evaluating four important quality attributes, e.g., performance, maintainability, testability, and portability. Our main contribution is a method for performing prototype-based software architecture evaluations. We apply this method, in cooperation with an industrial partner, to evaluate how it can be used to address performance, maintainability, and portability. In order to prepare for the addition of testability evaluation we also investigate how testabil-

ity is perceived by different roles in a software developing organization and find differences between how programmers, testers, and managers define testability.

More specifically, the following contributions can be identified:

1. A survey of existing architecture evaluation of the methods that can be used to evaluate one or more of the quality attributes performance, maintainability, testability, and portability. We find that few methods exist that are capable of addressing all of these four quality attributes. This contribution relates to research question one.
2. Continuous simulation is not useful for software architecture simulation and evaluation. The lack of discrete events in the simulation made it difficult to model the components of the architecture and to describe their behaviour. A combined simulation approach where it is possible to use discrete events in the model would be a better approach. The experiment resulted in a number of arguments against the continuous simulation approach. This contribution answers research question two.
3. A method for performing prototype-based evaluation of components in a software architecture. In response to research questions three and four, it is possible to evaluate both components in a software architecture as well as the structure and distribution of the components in the architecture. The evaluation is focused on the performance, maintainability, and portability of communication components. The use of a prototype framework introduces a clean interface between the architecture models and the components that are tested. This separation makes reuse of both the architecture models or the components easy.
4. An industrial case study capturing the view on testability in a software developing organization. The aim was to investigate how testability was defined by different roles in the organization. We found that programmers and testers share similar definitions and have a large understanding for each other's definitions of testability. The managers on the other hand differ both between themselves and with the programmers and testers. This contribution comes from the study of research questions five and six.

## 1.7 Thesis Papers

The remaining chapters in this thesis contain the articles that have been produced during the course of the research work. In this section we list the abstracts and publication information for the papers.

### 1.7.1 Chapter 2

#### A Survey on Software Architecture Evaluation Methods with Focus on Performance, Maintainability, Testability, and Portability

Mattsson, M., Mårtensson, F., and Grahn, H. To be submitted.

"The software architecture has been identified as an important part of a software system. The architecture describes the different components in a system and their interaction. Further, the software architecture impacts the quality attributes of a system, e.g., performance and maintainability. Therefore, methods for evaluating the quality attributes of software architectures are becoming increasingly important. Several evaluation methods have been proposed in order to assist the developer in creating a software architecture that will have a potential to fulfil the requirements on the system. Most evaluation methods focus on evaluation of a single quality attribute but in industrial practice it is common to have require¬ments on several quality aspects of a system.

In this paper, we present a survey of software architecture evaluation methods. We focus on methods for evaluating one or several of the quality attributes performance, maintainability, testability, and portability. Based on a literature search and review of 240 articles, we present and compare ten evaluation methods. We have found that most evaluation methods only address one quality attribute, and very few can evaluate several quality attributes simultaneously in the same framework or method. Further, only one of the methods includes trade-off analysis. Therefore, our results suggest an increased research focus on software architecture evaluation methods than can address several quality attributes and the possible trade-offs between different quality attributes."

### 1.7.2 Chapter 3

#### A Case Against Continuous Simulation for Software Architecture Evaluation

Mårtensson, F., Jönsson, P., Bengtsson, PO., Grahn, H., and Mattsson M., *Proc. Applied Simulation and Modelling*, pp. 97-105, ISBN: 0-88986-354-9, September 2003, Marbella, Spain.

"A software architecture is one of the first steps towards a software system. The design of the architecture is important in order to create a good foundation for the system. The design process is performed by evaluating architecture alternatives against each other. A desirable property of a good evaluation method is high efficiency at low cost. In this paper, we investigate the use of continuous simulation as a tool for software architecture performance evaluation. We create a model of the software architecture of an existing software system using a tool for continuous simulation, and then simulate the model. Based on the case study, we conclude that continuous simulation is not feasible for software architecture performance evaluation, e.g., we identified the need of discrete functionality to correctly simulate the system, and that it is very time consuming to develop a model for performance evaluation purposes. However, the modelling process is valuable for increasing knowledge and understanding about an architecture."

### 1.7.3 Chapter 4

#### An Approach for Performance Evaluation of Software Architectures using Prototyping

Mårtensson, F., Grahn, H., and Mattsson, M., *Proc. Software Engineering and Applications*, pp. 605-612, ISBN: 0-88986-394-63-5, November 2003, Los Angeles, USA.

"The fundamental structure of a software system is referred to as the software architecture. Researchers have identified that the quality attributes of a software system, e.g., performance and maintainability, often are restricted by the architecture. Therefore, it is important to evaluate the quality properties of a system already during architectural design. In this paper we propose an approach for evaluating the performance of a software architecture using architectural prototyping. As a part of the approach we have developed an evaluation support framework. We also show

the applicability of the approach and evaluate it using a case study of a distributed software system for automated guided vehicles."

### 1.7.4 Chapter 5

#### Evaluating Software Quality Attributes of Communication Components in an Automated Guided Vehicle System

Mårtensson, F., Grahn, H., and Mattsson, M., *Proc. 10th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 550-558, ISBN: 0-7695-2284-X, June 2005, Shanghai, China.

"The architecture of a large complex software system, i.e., the division of the system into components and modules, is crucial since it often affects and limits the quality attributes of the system, e.g., performance and maintainability. In this paper we evaluate three software components for intra- and inter-process communication in a distributed real-time system, i.e., an automated guided vehicle system. We evaluate three quality attributes: performance, maintainability, and portability. The performance and maintainability are evaluated quantitatively using prototype-based evaluation, while the portability is evaluated qualitatively. Our findings indicate that it might be possible to use one third-party component for both intra- and inter-process communication, thus replacing two in-house developed components."

### 1.7.5 Chapter 6

#### Forming Consensus on Testability in Software Developing Organizations

Mårtensson, F., Grahn, H., and Mattsson, M., *Proc. Fifth Conference on Software Engineering Research and Practice in Sweden*, pp. 31-38, ISBN: 91-88834-99-9, October 2005, Västerås, Sweden.

"Testing is an important activity in all software development projects and organizations. Therefore, it is important that all parts of the organization have the same view on testing and testability of software components and systems. In this paper we study the

view on testability by software engineers, software testers, and managers, using a questionnaire followed by interviews. The questionnaire also contained a set of software metrics that the respondents grade based on their expected impact on testability. We find, in general, that there is a high consensus within each group of people on their view on testability. Further, we have identified that the software engineers and the testers mostly have the same view, but that their respective views differ on how much the coupling between modules and the number of parameters to a module impact the testability. Base on the grading of the software metrics we conclude that size and complexity metrics could be complemented with more specific metrics related to memory management operations."

## 1.8 Validity of Results

All research has to be critically evaluated for validity. Results that lack in validity are less useful as it is difficult to know if they are correct or not. Identifying and reporting validity as well as threats to the validity in the research show that the researcher is aware of the problems with the study and might show how they can or should be addressed in future studies.

Validity is the trustworthiness of the research results. If the researcher has done what is possible to limit factors with negative impact on the result of a study, then the validity of the study becomes higher. Robson [77] defines three aspects of validity: construct validity, internal validity, and external validity.

**Construct validity.** The first type of validity concerns the choice of measures that are used in the study. Can it be shown that the measure really measures what it says it does. Identifying useful metrics is difficult, and several points for and against the use of a specific metric can usually be found.

The study in Chapter 2 has no problems with construct validity. The nature of the literature study makes it easy to quantify the presence of a software architecture evaluation method that is able to evaluate several quality attributes.

In Chapter 3 we measure the ability of our simulation model to mimic the performance of an existing system.

In Chapter 4 we focus on creating a method for evaluating software architectures using prototypes. The performance of the architecture is measured by counting the milliseconds that it takes to deliver messages as well as timing larger sets of events in the architecture.

Chapter 5 uses the same performance measure as in Chapter 4. In addition, a metric of maintainability called Maintainability Index [69] is also used. There has been papers published on the drawbacks of the components of this metric, e.g., [14] but we consider the metric to be useful as an indication of the maintainability of the source code.

Finally, in Chapter 6 we try to gather how roles in a software developing organization define testability. We gather our data by distributing a questionnaire that lets the respondents rate how much they agree or disagree with a number of statements.

**Internal validity.** This concerns the setup of the study, i.e., if there are any factors such as perturbation, bias in selection of participants, etc.

In Chapter 2 we gather candidate papers from Compendex and Inspec [27], we then performed a first screening by going through abstracts and keywords to identify papers that were relevant to our survey. After the first screening we read the remaining papers to identify and summarize the papers that would be included in the study. It is possible that papers that were relevant were excluded in the first screening and that we thus missed cases where an evaluation method had been used in part or full. We addressed this by initially doing a very wide search and intentionally included papers that might be borderline. The screening was then performed by three people to make sure that no relevant papers were excluded.

Chapter 3 has a weakness in the development of the model for the simulation. The model of the architecture has to be driven using inputs that reflect the real environment of the system in order to give useful results. The model elements that were needed to simulate this environment quickly outgrew the model of the architectural structure. This made it difficult to focus on the simulation and evaluation of the software architecture model.

The study in Chapter 4 has to deal with the possibility of perturbation of the system that it is trying to simulate. The collection of data for the performance estimation adds a slight overhead to the interactions between the components of the architecture. Initial problems with hard disk activity when the prototype had executed for a period of time was circumvented and the added overhead was eventually kept to a minimum.

Chapter 5 continued to build on the prototype-based evaluation and therefore had the same problem with perturbation as in Chapter 4. The introduction of the static analysis tools does not affect the dynamic-analysis aspects of the study.

Finally, in Chapter 6 the largest problem was the limited number of replies that we got from the respondents. Even after an extension of the deadline we had too few replies (14 replies to 25 distributed questionnaires) to be able to use statistical tests during the analysis of the results. In addition, 9 replies were from programmers, 3 from managers, and 2 from testers, resulting in a potential bias towards the programmers point of view.

**Generalizability.** Generalizability (or external validity) concerns the possibility to generalize the results to other cases. If we for example conduct a study on an organization and then replicate the study on another organization and get the same result, then the generalizability of the results can be seen as high. In gauging the generalizability of results one should be very careful. Especially when working with organizational issues such as the definitions of testability, many factors such as experience and background of the respondents affect the outcome of the study and the results may not be true for another organization working under different conditions.

In Chapter 2 we perform a literature survey where we collect our data from a number of research databases. The generalizability of the survey is limited to the set of methods that can address one or more of the quality attributes: performance, maintainability, testability, and portability. The results might change if the search is repeated since additional publications are continuously added to the research databases.

In Chapter 3 the conclusions show that the use of continuous simulation lacks some mechanisms that are necessary to correctly simulate a software architecture. This result is generalizeable to

the application of continuous simulation of software architectures.

The results from the study in Chapter 4 are generalizable to the evaluation of software architectures with respect to performance. The evaluation requires the presence of a runtime environment that is similar to or the same as the target platform in order to be effective.

In Chapter 5 we extend on the study from Chapter 4, the addition of a software metric for maintainability can have a negative effect on the generalizability. It is not sure that the maintainability index can be applied to another evaluation because the tools that were used to gather the data for the analysis are language specific. The maintainability index is applicable to an organization as long as it is calibrated to the systems and the programmers that work in the organization.

Finally, in Chapter 6, the generalizability is threatened by the low number of respondents to the survey. The fact that only four people responded in two of the roles makes it very hard to generalize from the results. Even if the responses had been more evenly distributed, it would be difficult to generalize the results to another organization. However, the result show that there are at least one organization where opinions on how a quality attribute is defined differ between roles. It would however be possible to replicate the study on several other companies.

## 1.9 Future Work

During the later parts of the work presented in this thesis, discussions have tended towards validation and testing of software architectures. We have identified two main paths forward based on the work that has been presented here.

The first possibility is to explore automated testing and validation of software architectures. To integrate the validation of the software architecture into a daily build process would enable the developers to follow how the quality attributes of the system are developing. It would then be possible to give early feedback to developers and also to see when the implementation is starting to deviate from the intended architecture of the system.

The second possibility is to investigate visualization of how of a systems' quality attributes are changing during development or maintenance. This requires quantification of quality attributes in a way that can be automated and repeated. Possible the prototypes developed during an architecture evaluation could be used as a benchmark that could be reused throughout the development.

## 1.10 Summary

In this chapter we have introduced and discussed the software development process. During the development of software systems, software architecture plays an important role. The architecture provides a blueprint for the continued work with design and implementation of a software system. Because of the influential role of the architecture, it is important that it will be able to fulfil the requirements on the system. Evaluation techniques for software architectures are therefore important, since they are used to evaluate one or more aspects of the software architecture.

We have also discussed the research questions that are addressed in the remaining chapters of this thesis. We have focused on the evaluation of four important quality attributes (performance, maintainability, testability, and portability). The majority of our research questions have focused on the evaluation of one or more of these quality attributes.

There has also been an introduction of the research methods that have been used in the search for answers to the research questions. The choice of methods brings different concerns over the validity of the data that has been gathered. We have therefore discussed the validity of each study from the perspectives of construct validity, internal validity, and generalizability. The research methods that have been used are: literature search, simulation, dynamic analysis, static analysis, and case study. Finally, we have presented and discussed the main contributions of this thesis.

CHAPTER

# 2

# A Survey of Software Architecture Evaluation Methods with Focus on Performance, Maintainability, Testability, and Portability

*Michael Mattsson, Frans Mårtensson, and Håkan Grahn*

## 2.1 Introduction

The software engineering discipline is becoming more widespread in industry and organisations due to the increased presence of software and software-related products and services. This demands for new concepts and innovations in the development of the software. During the last decades, the notion of software architecture has evolved and today, a software architecture is a key asset for any organization that builds complex software-intensive systems [12, 19, 82]. A software architecture is created early in the development and gives the developers a means to create a high level design for the system, making sure that all requirements that has to be fulfilled will be possible to implement in the system.

There exists a number of definitions of software architecture with minor differences depending on domain and people's experience. However, most definitions share common characteristics that can be exemplified by looking at the definition by Bass et al. [12]:

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." [12]*

This means that the architecture describes which high level components a software system consists of as well as which responsibilities that the components have towards other components in the system. It also describes how these components are organized, both on a conceptual level as well as a decomposed detailed level since there can be an architectural structure inside components as well. Finally the architecture defines which interfaces the components present to other components and which interfaces and components that they use.

The architecture is created based on a set of requirements that it has to fulfil. These requirements are collected from the stakeholders of the system, i.e., users and developers. The functional requirements describe what the system should do, e.g., the functions that the system should provide to the users. Quality requirements describe a set of qualities that the stakeholders want the systems to have, e.g., how long time it may take to complete a certain operation, how easy it is to maintain the system. Other examples of quality attributes are availability, testability, and flexibility.

In order to help software developers make sure that a software architecture will be able to fulfil the quality requirements, several approaches for evaluating software architectures has been proposed. The approaches can be divided into four basic categories, i.e., experience-based, simulation-based, mathematical modelling-based, and scenario-based [19] and are described in more detail later.

In this chapter we present a survey of software architecture evaluation methods, we focus on methods that address one or more of the quality attributes performance, maintainability, testability, and portability. We think that this selection of quality attributes is relevant for development of software systems that will be used and maintained over a long period of time. The methods are described and compared based on a set of criterias.

## 2.2 Software Architecture Evaluation

Architecture evaluations can be performed in one or more stages of a software development process. They can be used to compare and identify strengths and weaknesses in different architecture alternatives during the early design stages. They can also be used for evaluation of existing systems before future maintenance or enhancement of the system as well as for identifying architectural drift and erosion. Software architecture evaluation methods can be divided into four main categories [19]. Methods in the categories can be used independently but also be combined to evaluate different aspects of a software architecture, if needed.

**Experience-based** evaluations are based on the previous experience and domain knowledge of developers or consultants. People who have encountered the requirements and domain of the software system before can based on the previous experience say if a software architecture will be good enough [19].

**Simulation-based** evaluations rely on a high level implementation of some or all of the components in the software architecture [3, 59]. The simulation can then be used to evaluate quality requirements such as performance and correctness of the architecture. Simulation can also be combined with prototyping, thus prototypes of an architecture can be executed in the intended context of the completed system.

**Mathematical modelling** uses mathematical proofs and methods for evaluating mainly operational quality requirements such as performance and behaviour of the components in the architecture [75]. Mathematical modelling is similar to simulation and can be combined with simulation to more accurately estimate performance of components in a system.

**Scenario-based** architecture evaluation tries to evaluate a particular quality attribute by creating a scenario profile which forces a very concrete description of the quality requirement. The scenarios from the profile are then used to go through the software architecture and the consequences are documented [47, 49, 17].

## 2.3 Quality Attributes

Software quality is defined as the degree to which software possesses a desired combination of attributes [42]. According to [19] the quality requirements that a software architecture has to fulfil is commonly divided in two main groups based on the quality they are requesting i.e. development and operational qualities. A development quality requirement is a requirement that is of importance for the developers work, e.g. maintainability, understandability, and flexibility. Operational quality requirements are requirements that make the system better from the users point of view, e.g. performance and usability. Depending on the domain and priorities of the users and developers, quality requirements can become both development and operational, such as performance in a real-time system.

A quality attribute can be defined as a property of a software system [12]. A quality requirement is a requirement that is placed on a software system by a stakeholder; a quality attribute is what the system actually presents once it has been implemented. During the development of the architecture it is therefore important to validate that the architecture has the required quality attributes, this is usually done using one or more architecture evaluations.

This survey focus on software architecture evaluation methods that address one or more of the following quality attributes: performance, maintainability, testability, and portability. The IEEE standard 610.12-1990 [42] defines the four quality attributes as:

**Maintainability.** This is defined as:

> *"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment."*

Maintainability is a multifaceted quality requirement, it incorporates aspects such as readability and understandability of the source code. Maintainability is also concerned with testability to some extent as the system has to be re-validated during the maintenance.

**Performance.** Performance is defined as:

> *"The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage."*

This is also an attribute with several aspects, performance can be measured in, e.g., latency, throughput, and capacity. Which aspect of performance that is desired depends in the type of system.

**Testability.** Testability is defined as:

> *"The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met."*

We interpret this as the effort needed to validate the system against the requirements. A system with high testability simplifies the testers work and can be validated quickly.

**Portability.** Portability is defined as:

> *"The ease with which a system or component can be transferred from one hardware or software environment to another."*

We interpret this as portability not only between different hardware platforms and operating systems, but also between different virtual machines and versions of frameworks.

These four quality attributes are selected, not only for their importance for software developing organizations in general, but also for their relevance for organizations developing software in the real-time system domain in a cost effective way, e.g., by using a product-line approach. Performance is important since a system must fulfil the performance requirements, if not, the system will be of limited use, or not used. The long-term focus forces the system to be maintainable and testable, it also makes portability important since the technical development on computer hardware technology moves quickly and it is not always the case that the initial hardware is available after a number of years.

## 2.4 Related Work

Surveying software architecture evaluation methods has, as far as we know, been done in four previous studies. In two of the cases, Dobrica and Niemelä [25] and Babar et al. [5], the software architecture evaluation methods are compared with each other in a comparison framework, specific for each study. The survey by Etxeberria and Sagardui [28] compares architecture evaluation methods with respect to the context of architectures in software product lines. The last survey, by Kazman et al. [48], does not address a large number of architecture evaluation methods but uses two evaluation methods as examples for illustrating how the methods fulfil a number of criteria the authors argue are highly needed for an architecture evaluation method to be usable.

The Dobrica and Niemelä survey [25], the earliest one, presents and compares eight of the "most representative", according to themselves, architecture evaluation methods. The discussion of the evaluation methods focus on 1) discovering differences and similarities and 2) making classifications, comparisons and appropriateness studies. The comparison and characterization framework in the survey comprises the following elements; the methods goal, which evaluation techniques are included in the method, quality attributes (what quality attributes and what number of quality attributes is considered), the software architecture description (what views are the foci and in which development phase), stakeholders' involvement, the activities of the method, support for a reusable knowledge base and the validation aspect of the evaluation method.

The objective of the Babar et al. survey [5] is to provide a classification and comparison framework by discovering commonalities and differences among eight existing scenario-based architecture evaluation methods. To a large extent, the framework comprises features that are either supported by most of the existing methods or reported as desirable by software architecture researchers and practitioners. The framework comprises the following elements; the method's maturity stage, what definition of software architecture is required, process support, the method's activities, goals of the method, quality attributes, applicable project stage, architectural description, evaluation approaches (i.e., what types of evaluation approaches are included in the method?), stakeholders involvement, support for

non-technical issue, the method's validation, tool support, experience repository, and resources required.

The survey by Etxeberria and Sagardui [28] addresses an evaluation framework for software architecture evaluation methods addressing software product-line architectures. Since the life span of a product-line architecture is longer than for ordinary software architectures evolution becomes prioritized quality attribute that deserves extra attention in an evaluation. There exist other quality attributes as well, e.g. variability. The context of software product lines imposes new requirements on architecture evaluation methods and this is discussed by Etxeberria and Sagardui and reflects their classification framework. The framework comprises the following elements; The goal of the method, attribute types (what domain engineering and application engineering quality attributes are addressed), evaluation phase (in the product-line context the evaluation can take place on different phases in application engineering and domain engineering, respectively, as well as in a synchronization phase between the two), evaluation techniques, process description, the method's validation and relation to other evaluation methods.

The purpose of the last survey, by Kazman et al. [48], is primary to provide criteria that are important for an evaluation method to address, and not to compare existing evaluation methods. The authors argue for criteria addressing what it means to be an effective method, one that produces results of real benefit to the stakeholders in a predictable repeatable way, and a usable method one that can be understood and executed by its participants, learned reasonably quickly, and performed cost effectively. Thus, the survey ends up with the following four criteria: 1) Context and goal identification, 2) Focus and properties under examination, 3) Analysis Support, and 4) Determining analysis outcomes.

The survey by Dobrica and Niemelä [25] provides an early, initial overview of the software architecture evaluation methods. This was followed up by the survey by Babar et al. [5] that presents a more detailed break-down (including requirements on detailed method activities etc.) and a more holistic perspective, e.g., process support, tool support. The survey by Kazman et al. [48] presents additional requirements on what a software architecture method should support. The software product-line context survey by Etxeberria and Sagarduia [28] addresses evaluation methods from a prescribed way of developing software. This per-

spective opened up some additional phases where an evaluation can take place and put product-line important quality attributes more in focus, e.g. variability, maintainability.

Our survey takes the perspective from a set of quality attributes that are of general importance for software developing organizations. This means that we are taking a more solution-oriented approach, i.e., we are focusing on finding knowledge about what existing evaluation methods can provide with respect to the identified quality attributes. We are not aiming at obtaining knowledge about general software architecture evaluation methods or pose additional requirements on the methods due to some completeness criteria or specific way of developing the software, as in the four performed surveys. We may add additional requirements on the evaluation method, but if that is the case, the requirements will have its origin from the four quality attributes addressed, performance, testability, maintainability and portability.

## 2.5 Overview of the Architecture Evaluation Methods

In this survey each of the software architecture evaluation methods will be described according to a pre-defined template. The template structures the description of the architecture according to the following elements: Name and abbreviation (if any), Category of method, Reference(s) where the method are described in detail, Short description of the method, Evaluation goal of the method, How many quality attributes the method addresses, (one, many or many where trade-off approaches exist), What specific quality attributes the method address (or if it is a more general evaluation method) and finally, the usage of the method. Table 1 summarizes the template with indication of potential values for each element.

The initial selection of research papers was made by searching through the Compendex, Inspec, and IEEE Xplore research databases. We searched for papers that mentioned (software architecture) and (evaluation or analysis or assessment) and (performance or maintainability or testability or portability) in their title or abstract. The search in Compendex and Inspec resulted in 194 papers, and the search in IEEE Xplore produced

**Table 2.1 Method Description Template**

| Item | Potential values |
|---|---|
| Name and abbreviation | The method's name and abbreviation (if any) |
| Category of method | Experience-based, Simulation-based, Scenario-based, Mathematical modelling or a mix of categories |
| Reference(s) | One or more literature source(s) |
| Short description of the method | Text summary of the method |
| Evaluation goal | Text description of goal |
| Number of quality attributes addressed | One, many or many (with trade-off approach) |
| Specific quality attributes addressed | Any of Maintainability, Performance, Testability, Portability, General and any additional ones |
| Method usage | Has the method been used by the method developer(s) only or by some other? |

an additional 46 papers. In total, we had 240 papers from the database searches. We then eliminated duplicate papers and started to read abstracts in order to eliminate papers that were obviously off topic. We then read through the remaining papers and removed those that did not provide or mention a method for how they had performed an evaluation. After the elimination we had about 25 papers that contained architecture evaluation methods and experience reports from their use that addressed one or more of our four quality attributes. From these papers we have identified 10 methods and approaches that can be applied for architecture-level evaluation of performance, maintainability, testability, or portability.

### 2.5.1 SAAM — Software Architecture Analysis Method

Software Architecture Analysis Method (SAAM) [47] is a scenario-based software architecture evaluation method, targeted for evaluating a single architecture or making several architectures comparable using metrics such as coupling between architecture components.

SAAM was originally focused on comparing modifiability of different software architectures in an organization's domain. It has since then evolved to a structured method for scenario-based software architecture evaluation. Several quality attributes can be addressed, depending on the type of scenarios that are created during the evaluation process. Case studies where maintainability and usability are evaluated have been reported in [44], and modifiability, performance, reliability, and security are explicitly stated in [49].

The method consists of five steps. It starts with the documentation of the architecture in a way that all participants of the evaluation can understand. Scenarios are then developed that describe the intended use of the system. The scenarios should represent all stakeholders that will use the system. The scenarios are then evaluated and a set of scenarios that represents the aspect that we want to evaluate is selected. Interacting scenarios are then identified as a measure of the modularity of the architecture. The scenarios are then ordered according to priority, and their expected impact on the architecture. SAAM has been used and validated in several studies [22, 26, 44, 47, 55].

### 2.5.2 ATAM — Architecture Trade-off Analysis Method

Architecture Trade-off Analysis Method (ATAM) [49] is a scenario-based software architecture evaluation method. The goals of the method are to evaluate architecture-level designs that considers multiple quality attributes and to gain insight as to whether the implementation of the architecture will meet its requirements. ATAM builds on SAAM and extends it to handle trade-offs between several quality attributes.

The architecture evaluation is performed in six steps. The first one is to collect scenarios that operationalize the requirements for the system (both functional and quality requirements). The second step is to gather information regarding the constraints and environment of the system. This information is used to validate that the scenarios are relevant for the system. The third step is to describe the architecture using views that are relevant for the quality attributes that were identified in step one. Step four is to analyse the architecture with respect to the quality attributes. The quality attributes are evaluated one at a time. Step five is to identify sensitive points in the architecture, i.e., identifying those points that are affected by variations of the quality attributes. The

sixth and final step is to identify and evaluate trade-off points, i.e., variation points that are common to two or more quality attributes. ATAM has been used and validated in several studies [49, 67].

### 2.5.3 ALMA — Architecture-Level Modifiability Analysis

Architecture-Level Modifiability Analysis (ALMA) [17, 18] is a scenario-based software architecture evaluation method with the following characteristics: focus on modifiability, distinguish multiple analysis goals, make important assumptions explicit, and provide repeatable techniques for performing the steps. The goal of ALMA is to provide a structured approach for evaluating three aspects of the maintainability of software architectures, i.e., maintenance prediction, risk assessment, and software architecture comparison.

ALMA is an evaluation method that follows SAAM in its organization. The method specifies five steps: 1. determine the goal of the evaluation, 2. describe the software architecture, 3. elicit a relevant set of scenarios, 4. evaluate the scenarios, and 5. interpretation of the results and draw conclusions from them. The method provides more detailed descriptions of the steps involved in the process than SAAM does, and tries to make it easier to repeat evaluations and compare different architectures. It makes use of structural metrics and base the evaluation of the scenarios on quantification of the architecture. The method has been used and validated by the authors in several studies [17, 18, 54].

### 2.5.4 RARE/ARCADE

RARE and ARCADE are part of a toolset called SEPA (Software Engineering Process Activities) [6]. RARE (Reference Architecture Representation Environment) is used to specify the software architecture and ARCADE is used for simulation-based evaluation of it. The goal is to enable automatic simulation and interpretation of a software architecture that has been specified using the RARE environment.

An architecture description is created using the RARE environment. The architecture description together with descriptions of usage scenarios are used as input to the ARCADE tool. ARCADE then interprets the description and generates a simulation model. The simulation is driven by the usage scenarios.

RARE is able to perform static analysis of the architecture, e.g., coupling. ARCADE makes it possible to evaluate dynamic attributes such as performance and reliability of the architecture. The RARE and ARCADE tools are tightly integrated to simplify an iterative refinement of the software architecture. The method has, as far as we know, only been used by the authors.

### 2.5.5 Argus-I

Argus-I [87] is a specification-based evaluation method. Argus-I makes it possible to evaluate a number of aspects of an architecture design. It is able to perform structural analysis, static behavioural analysis, and dynamic behavioural analysis, of components. It is also possible to perform dependence analysis, interface mismatch, model checking, and simulation of an architecture.

Argus-I uses a formal description of a software architecture and its components together with statecharts that describe the behaviour of each component. The described architecture can then be evaluated with respect to performance, dependence, and correctness. There is no explicit process defined that the evaluation should follow, but some guidance is provided. The evaluation results in a quantification of the qualities of the architecture. The performance of the architecture is estimated based on the number of times that components are invoked. The simulation can be visualized using logs collected during the simulation. The method has, as far as we know, only been used by the authors.

### 2.5.6 LQN — Layered Queuing Networks

Layered queuing network models are very general and can be used to evaluate many types of systems. Several authors have proposed the use of queuing network models for software performance evaluation [29, 35, 50, 65]. Further, there also exist many tools and toolkits for developing and evaluating queuing network models, e.g., [29, 35]. A queuing network model can be solved analytically, but is usually solved using simulation.

The method relies on the transformation of the architecture into a layered queuing network model. The model describes the interactions between components in the architecture and the processing times required for each interaction. The creation of the models

require detailed knowledge of the interaction of the components, together with behavioural information, e.g., execution times or resource requirements. The execution times can either be identified by, e.g., measurements, or estimated. The more detailed the model is the more accurate the simulation result will be.

The goal when using a queuing network model is often to evaluate the performance of a software architecture or a software system. Important measures are usually response times, throughput, resource utilization, and bottleneck identification. In addition, some tools not only produce measures, but also have the ability to visualize the system behaviour.

### 2.5.7 SAM

SAM [89] is a formal systematic methodology for software architecture specification and analysis. SAM is mainly targeted for analysing the correctness and performance of a system.

SAM has two major goals. The first goal is the ability to precisely define software architectures and their properties, and then perform formal analysis of them using formal methods. Further, SAM also support an executable software architecture specification using time Petri nets and temporal logic. The second goal is to facilitate scalable software architecture specification and analysis, using hierarchical architectural decomposition. The method has, as far as we know, only been used by the authors.

### 2.5.8 Empirically-Based Architecture Evaluation

Lindvall et al. describe in [56] a case study of a redesign/reimplementation of a software system developed more or less in-house. The main goal was to evaluate the maintainability of the new system as compared to the previous version of the system. The paper outlines a process for empirically-based software architecture evaluation. The paper defines and uses a number of architectural metrics that are used to evaluate and compare the architectures.

The basic steps in the process are: select a perspective for the evaluation, define/select metrics, collect metrics, and evaluate/compare the architectures. In this study the evaluation perspective was to evaluate the maintainability, and the metrics were structure, size, and coupling. The evaluations were done in a late

development stage, i.e., when the systems already were implemented. The software architecture was reverse engineered using source code metrics.

### 2.5.9 ABAS — Attribute-Based Architectural Styles

Attribute-Based Architectural Styles (ABASs) [51] build on the concept of architectural styles [21, 82], and extend it by associating a reasoning framework with an architectural style. The method can be used to evaluate various quality attributes, e.g., performance or maintainability, and is thus not targeted at a specific set of quality attribute.

The reasoning framework for an architectural style can be qualitative or quantitative, and are based on models for specific quality attributes. Thus, ABASs enable analysis of different quality aspects of software architectures based on ABASs. The method is general and several quality attributes can be analysed concurrently, given that quality models are provided for the relevant quality attributes. One strength of ABASs is that they can be used also for architectural design. Further, ABASs have been used as part of evaluations using ATAM [49].

### 2.5.10 SPE — Software Performance Engineering

Software performance engineering (SPE) [83, 90] is a general method for building performance into software system. A key concept is that the performance shall be taken into consideration during the whole development process, not only evaluated or optimized when the system already is developed.

SPE relies on two different models of the software system, i.e., a software execution model and a system execution model. The software execution model models the software components, their interaction, and the execution flow. In addition, key resource requirements for each component can also be included, e.g., execution time, memory requirements, and I/O operations. The software execution model predicts the performance without taken contention of hardware resources into account.

The system execution model is a model of the underlaying hardware. Examples of hardware resources that can be modelled are processors, I/O devices, and memory. Further, the waiting time

and competition for resources are also modelled. The software execution model generates input parameters to the system execution model. The system execution model can be solved by using either mathematical methods or simulations.

The method can be used to evaluate various performance measures, e.g., response times, throughput, resource utilization, and bottleneck identification. The methods is primarily targeted for performance evaluation. However, the authors argue that their method can be used to evaluate other quality attributes in a qualitative way as well [90]. The method has been used in several studies by the authors, but do not seem to have been used by others.

### 2.5.11 Summary of Architecture Evaluation Methods

Table 2.2 summarizes the most important characteristics (see Table 2.1) of our survey of software architecture evaluation methods. As we can see, half of the methods address only one quality attribute of those that we consider in this survey, and the most common attribute to address is performance. Surprisingly, no method was found that specifically address portability or testability. Further, we can observe that only one method exists that support trade-off analysis of software architectures. Finally, we also observe that only two methods seem to have been used by others than the method inventor.

## 2.6 Discussion

Despite the promising number of primary studies found, i.e., 240, it turned out that only 10 software architecture evaluation methods were possible to identify that addressed one or more of the performance, maintainability, testability, or portability quality attributes. There exist several reasons for this large reduction of the number of articles. First, there were some duplicate entries of the same article since we searched several databases. Second, a large portion of the papers evaluated one or several quality attributes in a rather ad hoc fashion. As a result, we excluded those papers from our survey since they did not document a repeatable evaluation method or process. Third, several papers addressed both hardware and software evaluations, thus they did not qualify either in our survey with its focus on methods for software architecture evaluation.

**Table 2.2 Summary of evaluation method characteristics.**

| Name | Category | Quality attributes | Method usage |
|---|---|---|---|
| SAAM [47] | Scenario-based | General | creator [47, 44], other [22, 26, 55] |
| ATAM [49] | Scenario-based | General, trade-off | creator [49], other [67] |
| ALMA [18] | Scenario-based | Modifiabity | creator [17, 18, 54] |
| RARE/ARCADE [6] | Simulation-based | Performance, other | creator [6] |
| ARGUS-I [87] | Simulation-based | Performance, other | creator [87] |
| LQN [29, 65, 72] | Simulation-based, mathematical modelling | Performance | creator [72, 3] |
| SAM [89] | Simulation-based | Performance, other | creator [89] |
| EBAE [56] | Experience-based, metrics | Maintainability | creator [56] |
| ABAS [51] | Experience-based | General | creator [51] |
| SPE [83, 90] | Simulation-based, mathematical modelling | Performance | creator [83, 90] |

Continuing with the 10 remaining articles, we found that five of the methods addressed only one of the quality attributes that we were interested in. Only one (ATAM) of the remaining methods addressing multiple attributes provide support for trade-off between the quality attributes. No specific methods evaluated testability or portability explicitly. These quality attributes could be addressed by any of the three evaluation methods that are more general in their nature, i.e., that could address more arbitrary selected quality attributes, ATAM [49], SAAM [47], or EBAE [56].

Many of the methods have been used several times of the authors. Multiple use of the method indicates an increase in validity of the method. However, only two methods have been used by others than the original authors of the method. We believe that external use of a method is an indication of the maturity of the method. These two methods are SAAM and ATAM.

However, experience papers that use a method in whole or part are difficult to identify, since the evaluation method that has been used is not always clearly stated.

## 2.7 Conclusions

The architecture of a software system has been identified as an important aspect in software development, since the software architecture impacts the quality attributes of a system, e.g., performance and maintainability. A good software architecture increases the probability that the system will fulfil its quality requirements. Therefore, methods for evaluating the quality attributes of software architectures are important.

In this chapter, we present a survey of evaluation methods for software architecture quality attribute evaluation. We focus on methods for evaluating one or several of the quality attributes performance, maintainability, testability, and portability. Methods that evaluate several quality attributes and/or trade-off analysis are especially interesting. Based on a broad literature search in major scientific publication databases, e.g., Inspec, and reviewing of 240 articles, we present and compare ten evaluation methods.

We focus our survey around four important quality attributes (performance, maintainability, testability, and portability) and have found that many evaluation methods only address one of the quality attributes, and very few can evaluate several quality attributes simultaneously in the same framework or method. Specifically, only one of the methods includes trade-off analysis. Further, we have identified that many methods are only used and validated by the method inventors themselves.

In summary, our results suggest

- an increased research focus on software architecture evaluation methods than can address several quality attributes simultaneously,
- an increased research focus on software architecture evaluation methods that can address the possible trade-offs between different quality attributes, and
- an increased focus on validation of software architecture evaluation methods by people other than the method inventors.

CHAPTER

3

# A Case Against Continuous Simulation for Software Architecture Evaluation

*Frans Mårtensson, Per Jönsson, PerOlof Bengtsson, Håkan Grahn, and Michael Mattsson*

## 3.1 Introduction

The software architecture is fundamental for a software system [7, 33, 81], as it often restricts the overall performance of the final system. Before committing to a particular software architecture, it is important to make sure that it handles all the requirements that are put upon it, and that it does this reasonably well. The consequences of committing to a badly designed architecture could be disastrous for a project and could easily make it much more expensive than originally planned. Bad architecture design decisions can result in a system with undesired characteristics such as low performance, low maintainability, low scalability etc.

When designing the architecture for a system, the architect often has the possibility to choose among a number of different solutions to a given problem. Depending on which solution is chosen, the architecture evolves in different ways. To be able to make a proper decision, the architect needs to identify quantifiable advantages and disadvantages for each one. This can be done by

using, e.g., prototyping or scenario-based evaluation [19]. A desirable property of a good evaluation method is high efficiency at low cost.

In this chapter we evaluate the use of continuous simulation for system architecture performance evaluation purposes. The main idea is that tools for continuous simulation can be used to quickly create models of different architecture alternatives. These models can then be used, through simulation, to evaluate and compare different architectures to each other. The work presented has been conducted in co-operation with Danaher Motion Särö (referred to as "DMS"). We have co-operated with DMS to use the software architecture of their Automated Guided Vehicle (AGV) system (hereafter referred to as the "DMS system") as a case for the research.

Unfortunately, we found that continuous simulation does not work very well for software architecture performance evaluation. First, when a continuous simulation model is used only average flow values can be used to parameterize the model. This makes the model less dynamic and may have the consequence that the simulation model can be replaced with a static mathematical model. Second, it is impossible to address unique entities when using continuous simulation. This is not always necessary when simulating flows of information, but if the flows depend on factors that are discrete in their nature, for example vehicles in an AGV system, then continuous simulation is a bad choice.

We do however believe that an architecture modeling tool that incorporates some simulation functionality could be helpful when designing and evaluating software architectures. It could, e.g., provide functions for studying data flow rates between entities in an architecture. Such a tool would preferably be based on combined simulation techniques, because of the need to model discrete events.

The rest of the chapter is structured as follows: We begin with an introduction to software architectures in Section 3.2. In Section 3.3, we discuss some simulation approaches, and in Section 3.4, we describe two software tools for continuous simulation. Next, Section 3.5 introduces the AGV system domain. Section 3.6 describes our attempts to model and simulate the architecture of the DMS system. In Section 3.10, we have a discussion of our results, and finally, in Section 3.11, we present our conclusions.

## 3.2 Software Architecture

There are many different definitions of what a software architecture is, and a typical definition is as follows [33]:

A critical issue in the design and construction of any complex software system is its architecture: that is, its gross organization as a collection of interacting components.

In other words, through the creation of a software architecture we define which parts a system is made up of and how these parts are related to each other. The software architecture of a system is created early in the design phase, since it is the foundation for the entire system.

The components in an architecture represent the main computational elements and data storage elements. The architecture is created on a high level of abstraction which makes it possible to represent an entire subsystem with a single component. The communication between the components can be abstracted so that only the flow of information is considered, rather than technical details such as communication protocol etc. Individual classes and function calls are normally not modelled in the architecture.

With the creation of the software architecture, designers get a complete view of the system and its subsystems. This is achieved by looking at the system on a high level of abstraction. The abstracted view of the system makes it intellectually tractable by the people working on it and it gives them something to reason around [7].

The architecture helps to expose the top level design decisions and at the same time it hides all the details of the system that could otherwise be a distraction to the designers. It allows the designers to make the division of functionality between the different design elements in the architecture and it also allows them to make evaluations of how well the system is going to fulfil the requirements that are put upon it. The requirements on system can be either functional or non-functional. The functional requirements specify what function the system shall have. The non-functional requirements include, e.g., performance, maintainability, flexibility, and reusability.

Software architectures are often described in an ad hoc way that varies from developer to developer. The most common approach is to draw elements as boxes and connections simply as connecting lines. A more formal way of defining architectures is to use an architecture description language (ADL) that is used to describe the entities and how they connect to each other. Examples of existing ADL:s are ACME [32] and RAPIDE [59], which are still mainly used for research. ADL:s have been successfully used to describe new architectures and to document and evaluate existing architectures [31].

A software architecture is, among other things, created to make sure that the system will be able to fulfil the requirements that are put upon it. The architecture usually focuses more on non-functional requirements than on functional ones. The non-functional requirements are for example those that dictate how many users a system should be able to handle and which response times the system should have. These requirements does not impact which functionality the system should provide or how this functionality should be designed. They do however affect how the system should be constructed.

It is important to evaluate different software architecture alternatives and architectural styles against each other in order to find the most appropriate ones for the system. There exists a number of evaluation methods, e.g., mathematical model-, experience-, and scenario-based methods [19]. The process of evaluating the software architectures is mainly based on reasoning around the architecture and the chosen scenarios. How successful this evaluation is depends currently heavily on the level of experience of the people performing it. More experienced people are more likely to identify problems and come up with solutions.

It is during this evaluation phase that we believe that it would be useful to use continuous simulation for evaluating the performance of software architectures. It could be used as a way of quickly conducting objective comparisons and evaluations of different architectures or scenarios. The advantage over other evaluation methods, for example experience-based evaluation, is that simulation gives objective feedback on the architecture performance.

## 3.3 Model and Simulation

A *model* is a representation of an actual system [10], or a "selective abstraction" [76], which implies that a model does not represent the system being modelled in its whole. A similar definition is that a model should be similar to but simpler than the system it models, yet capture the prominent features of the system [62]. To establish the correctness of a model, there is a need for model *validation* and *verification*. Validation is the task of making sure that the right model has been built [7, 62]. Model verification is about building the model right [7, 78].

A *simulation* is an imitation of the operation of a real-world process or system over time [10, 62, 86]. A simulation can be reset and rerun, possibly with different input parameters, which makes it easy to experiment with a simulation. Another important property of simulation is that time may be accelerated, which makes simulation experiments very efficient [82].

The *state* of a simulation is a collection of variables that contain all the information necessary to describe the system at any point in time [9]. The input parameters to a simulation are said to be the initial state of the simulation. The state is important for pausing, saving, and restoring an ongoing simulation, or for taking a snapshot of it. A simulation model must balance the level of detail and number of state variables carefully in order to be useful. The goal is to find a trade-off between simplicity and realism [62].

*Continuous simulation* is a model in which the system changes continuously over time [86]. A continuous simulation model is characterized by its state variables, which typically can be described as functions of time. The model is defined by equations for a set of state variables [10], e.g, $dy/dt = f(x, t)$. This simulation technique allows for smooth system simulation, since time is advanced continuously, i.e. changes occur over some period of time.

When using the *discrete simulation technique*, time is advanced in steps based on the occurrence of discrete events, i.e., the system state changes instantaneously in response to discrete events [62, 86]. The times when these events occur are referred to as event times [9, 80]. In an event-driven discrete simulation, events are popped from a sorted stack. The effect of the topmost event

on the system state is calculated, and time is advanced to the execution time of the event. Dependent events are scheduled and placed in the stack, and a new event is popped from the top of the stack [52]. With the discrete simulation technique, the ability to capture changes over time is lost. Instead, it offers a simplicity that allows for simulation of systems too complex to simulate using continuous simulation [86].

*Combined continuous-discrete simulation* is a mix of the continuous and discrete simulation techniques. The distinguishing feature of combined simulation models is the existence of continuous state variables that interact in complex or unpredictable ways with discrete events. There are mainly three fundamental types of such interactions [7, 52]: (i) a discrete event causes a change in the value of a continuous variable; (ii) a discrete event causes a change in the relation governing the evolution of a continuous variable; and (iii) a continuous variable causes a discrete event to occur by achieving a threshold value.

A software system can be modeled and simulated using either discrete or continuous simulation techniques. When looking at the software architecture of a system, communication between the components can be viewed as flows of information, disregarding discrete events. By leaving out the discrete aspects of the system, continuous simulation can be used to study information flows. It is our assumption that it is simpler to model a software architecture for continuous than for discrete simulation, because low-level details can be ignored.

A good example of a low-level detail is a function call, which is discrete since it happens at one point in time. By looking at the number of function calls during some amount of time, and the amount of data sent for each function call, the data transferred between the caller and the callee can be seen as an information flow with a certain flow rate. Some reasons that make this advantageous are:

- It is valid to consider an average call frequency and to disregard variations in call interval etc.
- Multiple function calls between two components can be regarded as one single information flow.
- Accumulated amounts and average values are often interesting from a measurement perspective.

## 3.4 Software Tools

Once a model has been constructed, we want to run it to see what results it produces. If it is a simple model then it might be possible to simulate it using pen and paper or perhaps a spreadsheet. But if the model is too complex for "manual" execution then it becomes necessary to use some kind of computer aid. Since we in this chapter focus on the possibilities of using continuous simulation in architecture evaluation, we look at GUI-based general-purpose simulation tools that require little knowledge about the underlying mathematical theories.

The first tool that we evaluated was the STELLA 7.0.2 Research simulation tool, which is a modeling and simulation software for continuous simulation that is created and marketed by High Performance Systems Inc. The second tool was Powersim Studio Express 2001, created by Powersim. This is a similar tool that offers more functionality than STELLA as it is based on combined simulation, and has some more advanced features. Both tools are based on the concept of *Systems Thinking* [76] for the creation of models, and both programs are capable of performing the simulation directly in the program and also to perform some basic analysis.

In both STELLA and Powersim, models are constructed from a number of basic building blocks which are combined in order to build a model. The model is simulated by the use of entities that are sent through the model. The flow of entities can then be measured and analyzed. We use snapshots of the STELLA tool, but they look very similar in Powersim and they work in a similar fashion. In Figure 3.1 we show the five basic building blocks Stock, Flow, Converter, Connector, and Decision Process.

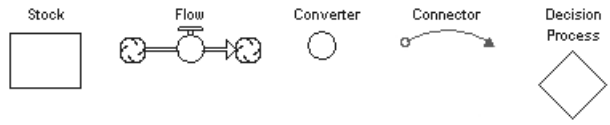


**Figure 3.1 Examples of basic building blocks.**

*Stocks* are used to represent accumulation of entities various ways. *Flows* are used to connect stocks and to enable and control

the flow of entities between them. *Converters* are often used in order to modify the rate of flows, and to introduce constants in a model. *Connectors* are used to connect, e.g., stocks and flows so they can exchange information. To make a model less complex it is possible to hide parts of it by using *decision processes*.

Once the model is completed it is possible to run it. Both Powersim and STELLA are capable of accelerating the simulation time. The tools can visualize simulation outputs and results as the simulation runs, e.g., time-graphs, time-tables and value labels. Powersim also has the possibility to export results to a standard file format.

We used Powersim for the following three reasons: (i) Powersim has the ability to check the consistency of the model via the use of units on every flow; (ii) Powersim offers the possibility to create discrete flows and visually distinguish them in a model; and (iii) STELLA crashed repeatedly when we tried to use the decision process functions, and also sometimes even during normal work. The unreliability of the tool made us hesitant to use STELLA.

## 3.5 AGV Systems

An AGV (Automated Guided Vehicle) system is an automatic system that usually is used for materials handling in manufacturing environments, e.g., car factories and metal works. They are however not restricted to these environments and can be used in very different environments such as hospitals and amusement parks.

An AGV is usually a driverless battery-powered truck or cart that follows a predefined path [24]. A path is divided into a number of *segments* of different lengths and curvatures. There can be only one vehicle on a segment at any given time. The amount of computational power in a vehicle may vary depending on how advanced the behavior of the vehicle is. With more computational power, it is possible to let the vehicle be autonomous. However, computational power costs money, and with many vehicles, a computationally strong solution can be expensive.

The management and control of the AGV system is usually handled by a central computer that keeps track of all the vehicles and

their orders. This computer maintains a database of the layout of the paths that the vehicles can use to get to their destinations [24]. With this information it acts as a planner and controller for all the vehicles in the system, routing traffic and resolving deadlocks. The central server gets orders from, e.g., production machines that are integrated with the AGV system.

In order for the AGV system to work it must be possible to find the position of the vehicles with good precision. This is achieved by the use of one or more positioning and guidance systems, e.g., electrical track, optical guidance, and magnetic spots. With electrical track guidance, the vehicle path is defined by installing a guidance wire into the floor of the premises. Optical guidance is achieved for example by the use of a laser positioning system which uses reflectors placed on the walls of the premises in order to calculate an accurate position of the AGV as it moves. Magnetic guidance works by the use of magnetic spots, which are placed on the track. The vehicles have magnetic sensors that react on the presence of the spots.

In an AGV system it is desirable to minimize the communication between the server and the clients. The choice of communication strategy affects the amount of information that is communicated in the system.

An early communication strategy was to let the vehicles communicate with the server only at certain designated places. As a result, the vehicles can only be redirected at certain points, since the server has no control of a vehicle between communication spots. A more advanced way of communicating is via the use of radio modems. The early modems however had very low bandwidth, which imposed limitations on the amount of information that could be transferred. This limitation has diminished as advancements made in radio communication technology have increased the amount of available bandwidth. The next step in communication is to make use of cheaper off-the-shelf hardware such as wireless LAN, e.g., IEEE 802.11b. An advantage with using such a strategy is that an existing infrastructure can be used.

We mention here two alternative ways to design an AGV system, and they are interesting because they represent the extremes of designing a system architecture. The goal of a *centralized approach* is to put as much logic in the server as possible. Since

the vehicles cannot be totally free of logic (they have to have driving logic at least), the centralized approach is in practise distributed. However, we may choose different degrees of centralization by transferring modules from the vehicle logic to the server logic. In an entirely *distributed approach* there is no centralized server, thus making the system less vulnerable to failure. This requires all information in the system to be shared among, and available to, all vehicles, which can be realized, e.g., by using a distributed database solution.

## 3.6 The Experiment

The architecture studied is a client-server architecture for a system that controls AGVs. The server is responsible for such tasks as order management, carrier management, and traffic management. It creates “flight plans” and directs vehicles to load stations. The vehicles are “dumb” in the sense that they contain no logic for planning their own driving. They fully rely on the server system. A more in-depth explanation of the system can be found in [85].

The communication between server and clients is handled by a wireless network with limited capacity, set by the radio modems involved. Topics of interest are for example:

- Has the network capacity to handle communication in highly stressed situations with many vehicles?
- Can the system architecture be altered so less traffic is generated?
- Can the system share an already present in-use wireless LAN?

With this in mind, we decided to simulate the architecture with respect to the amount of generated network traffic. The intention is to provide a means for measuring how communication-intense a certain architecture is.

## 3.7 The System Behavior

The purpose of the studied system is to control a number of AGVs. The AGVs must follow a pre-defined track which consists of segments. A fundamental property of a segment is that it

can only be “allocated” to one AGV at a time. Sometimes, several segments can be allocated an AGV to prevent collisions. The primary controlling unit for the system is an *order*. An order usually contains a loading station and an unloading station. Once an order has been created, the server tries to assign a vehicle to the order and instructs the vehicle to carry it out. During the execution of an order, the vehicle is continuously fed segments to drive.

In certain situations, deadlock conflicts can arise. A deadlock occurs, e.g., when two vehicles are about to drive on the same segment. A traffic manager tries to resolve the deadlock, according to a set of deadlock avoidance rules. As the number of vehicles involved in the deadlock increases, it becomes harder and harder for the traffic manager to resolve the situation.

Each vehicle, i.e., each client, contains components for parsing drive segments fed from the server, controlling engines and steering, locating itself on the map etc. The vehicle is highly dependent on the drive commands sent from the server; if the segment-to-drive list is empty, it will stop at the end of the current segment. If the vehicle gets lost and can’t rediscover its location, it will also stop.

The communication between server and clients is message-based. The server sends vehicle command messages to control the vehicles, and the vehicles respond to these with vehicle command status messages. There are also status messages, which are used to report vehicle status.

## 3.8 The Model

We will look at the flow of information over the network in the system architecture. Thus, the communication network plays a central role in the model, and the purpose of all other entities is to generate input traffic to the network. The network traffic in a client-server system has two components; server generated traffic and client generated traffic. However, when measuring the network utilization, the sum is interesting. In our model, the network traffic is modelled as a whole. Further, the network is more or less a “black hole”, since the output is discarded.

Prior to constructing the model we had basic knowledge of the behavior of both the server architecture and the client architecture, e.g., which components that communicate over the network. However, we had only vague understanding of what caused communication peaks and which communication that could be considered “background noise”. Therefore, we studied a version of the current client-server system. The server system can handle both real and simulated AGVs, which allowed us to run a simulation of the real system in action, but with simulated vehicles instead of real ones (30 vehicles were simulated).

An example of logged network traffic can be seen as the solid line in Figure 3.2 (the y-axis has no unit, because the purpose is only to show the shape of the traffic curve). In the left part of the diagram, all AGVs are running normally, but in the right part they are all standing still in deadlock. Except for the apparent downswing in network traffic during deadlock, no obvious visual pattern can be found. When analyzing the traffic, we found that normal status messages are responsible for roughly 90% of the traffic, and that the number of status messages and the number of vehicle command messages fluctuate over time. However, the number of vehicle command status messages seems to be rather stable regardless of system state (e.g. normal operation vs. deadlock).

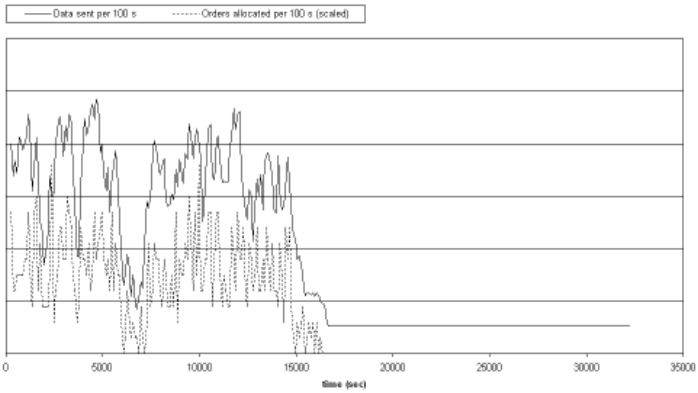


**Figure 3.2 Example of network traffic during a system run, with order allocations superimposed on the traffic diagram.**

We sought reasons for the traffic fluctuations, and examined the log files generated. We found a connection between order alloca-

tions and network traffic. An order allocation takes place when a vehicle is assigned to a new order, and this causes an increase in traffic. In Figure 3.2, a correlation between the order allocations (dotted line) and the network traffic (solid line) is observed. In particular, during the deadlock there are no order allocations at all. Mathematically, the correlation is only *0.6*, which is not very strong but enough for us to let order allocations play the largest role in the model. The reason for an upswing in traffic when an order allocation takes place is simple; it changes the state of a vehicle from "available" to "moving", and in the latter state the traffic per vehicle is higher.

The network is modelled as a buffer with limited storage capacity. It holds its contents for one second before it is released. Immediately before the network buffer is a transmission buffer to hold the data that cannot enter the network. If the network capacity is set too low, this buffer will be filled. In a real system, each transmitting component would have a buffer of its own, but in the model the buffer acts as transmission buffer for all components.

To model network congestion, the network buffer outlet is described by a function that depends on the current network utilization, e.g., it releases all network data up to a certain utilization limit, and thereafter gradually releases less data as the utilization increases. The data that remains in the network buffer represents data that in a real system would be re-sent. A visual representation of the modeled network is seen to the left in Figure 3.3. The entity "Network indata" in Figure 3.3 is at every time the sum of all traffic generated in the model at that time

The primary controlling unit for the system is an order, and order allocations generate network traffic. The amount of traffic generated also depends on how many vehicles that are available, processing orders, and in deadlock. Therefore, we need constructs for the following:

- Order generation
- Order allocation
- Available vs. non-available vehicles
- Deadlock

Orders can be put into the system automatically or manually by an operator. We have chosen to let orders be generated randomly

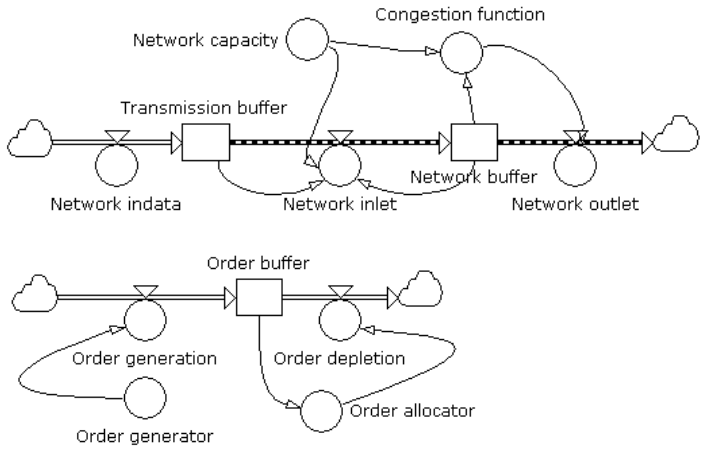


**Figure 3.3 Network component (top), and Order component and order allocator (bottom).**

over time, but with a certain frequency. Each time an order is generated, it is put in an order buffer. As orders are allocated to vehicles, the number of orders in the buffer decreases. The order component and the order allocator is shown to the right in Figure 3.3.

For an order allocation to take place, there must be at least one available order, and at least one available vehicle. Then, the first order in queue is consumed and the first available vehicle is moved to the busy-queue. The busy queue contains several buffers to delay the vehicles' way back to the buffer for available vehicles and a mechanism for placing vehicles in deadlock. In the deadlock mechanism each vehicle runs the risk of being put in a deadlock buffer. The risk of deadlock increases as more and more vehicles are put into the deadlock buffer. Once in deadlock, each vehicle runs the chance of being let out of the deadlock again. The chance for this to happen is inversely proportional to the number of vehicles in deadlock. Figure 3.4 shows the construct describing the vehicle queues and the deadlock mechanism.

The remaining parts of the model fill the purpose of "gluing" it together. They are simple constructs that, given the current state of vehicles, generate the proper amounts of traffic to the network.

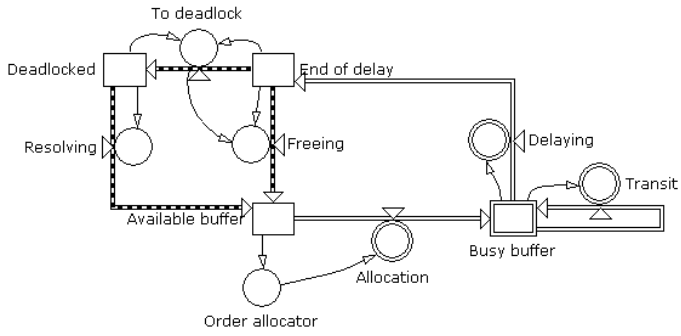


**Figure 3.4 Vehicle queues and deadlock mechanism.**

### 3.8.1 Simulation Parameters and Results

In the current system, each vehicle is equipped with a modem capable of transmitting 19 200 bps, while both the network and the server system have higher capacity. We therefore chose to set the network speed to 2 400 byte/s (19 200 bps) in the simulation, since the first step was to build a model that approximates the real system, rather than to study the impact of different network speeds.

In our simulation, we let the order creation probability be high enough to ensure that there is always at least one order in queue when a vehicle becomes available. The average order processing time is set to *230 seconds*. This is based on the average order processing time in the real system when run with 1 vehicle.

The probability for a vehicle to enter deadlock is set to $P_{enter} = 1 - 0{,}99^{x+1}$ where $x$ is the number of vehicles currently in deadlock, i.e., the probability increases as more vehicles enter deadlock. The probability for a vehicle to leave deadlock is set to $P_{leave} = 0{,}2^{y}$ where $y$ is the number of vehicles currently in deadlock, i.e., the more vehicles involved in a deadlock, the harder it is to resolve.

Table 3.1 contains data points measured in the real system in standby state, i.e., when no vehicles were moving. As seen in Figure 3.5, the traffic is linearly related to the number of vehicles.

The situation when all vehicles are standing still is assumed to be similar to a deadlock situation, at least traffic-wise.

**Table 3.1 Traffic generated in standby state (avg. bytes/s)**

| No. of vehicles | Status msg. | Command msg. | Command status msg. |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1.2 | 0 | 0.5 |
| 5 | 6.0 | 0 | 2.5 |
| 10 | 12.0 | 0 | 5.0 |
| 20 | 24.0 | 0 | 10.0 |

Table 3.2 contains data points measured in moving state. In Figure 3.5, we see that this traffic does not appear to be linearly related to the number of vehicles. We suspect that this has to do primarily with deadlock situations when the number of vehicles is high. Otherwise it would mean that for some certain number of vehicles (more than 30), there would be no increase in traffic as more vehicles are added to the system. To sum up, the traffic generated in different situations is as follows, deduced from the data in Tables 3.1 and 3.2:

- Available vehicles and vehicles in deadlock generate on average *1.2 bytes/s* of status messages per vehicle.
- Vehicles processing orders generate on average *17 bytes/s* of status messages per vehicle.
- The server sends *3.2 bytes/s* of command messages per running vehicle.
- Each vehicle sends *0.5 bytes/s* of command response status messages in standby state and *0.33 bytes/s* in moving state..

We ran the simulation for different periods of time, varying between 10 minutes and 10 hours. The behavior of the model is rather predictable, as Figure 3.6 depicts. With the limited set of controllable parameters in the model, patterns in the simulation output are more apparent and repetitive than in output from the real system. An important reason for this is that we cannot take segment lengths, vehicle position and varying order processing time into account in the model. Furthermore, there may also be factors affecting the network traffic that we have not found.

**Table 3.2 Traffic generated in moving state (avg. bytes/s)**

| No. of vehicles | Status msg. | Command msg. | Command status msg. |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 19.0 | 3.4 | 0.2 |
| 3 | 54.8 | 9.6 | 0.7 |
| 5 | 92.5 | 16.8 | 1.2 |
| 8 | 138.2 | 27.2 | 2.2 |
| 10 | 187.8 | 33.6 | 3.3 |
| 13 | 211.8 | 39.8 | 4.2 |
| 20 | 298.9 | 52.7 | 7.5 |
| 30 | 335.0 | 59.5 | 10.1 |

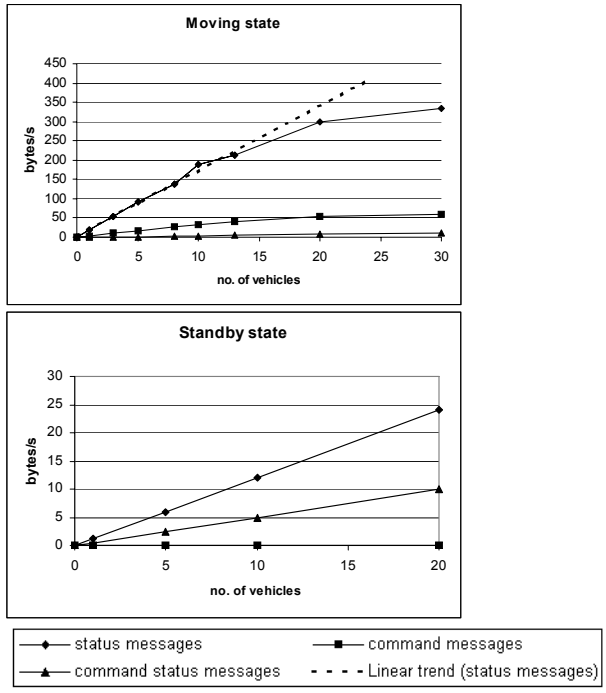


**Figure 3.5 Relation between number of vehicles and the generated traffic**

One reason that we cannot say much about the simulation results, is that its inputs do not match the inputs to the real system. In

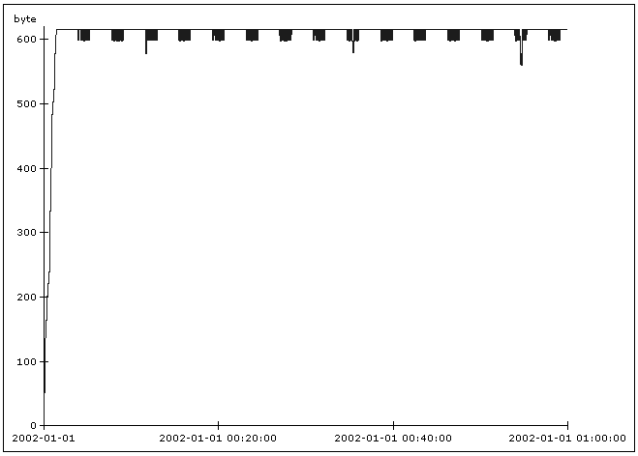


**Figure 3.6 Output from a simulation of 30 vehicles.**

other words, we cannot validate the model using the simulation results. Even if we could extract all external inputs to the real system, they would not all apply directly to the model because of the approximations made.

In the simulation output in Figure 3.6, we see that the average network utilization in the simulation is higher than in the real system (Figure 3.5). The reason is that the model keeps vehicles busy as the number of vehicles increase, while the real system is not because of the fact that deadlocks and conflicts occur more often there. The lack of variation in the traffic diagram in Figure 3.6 is an effect of the fixed order processing time, and the fact that vehicles do not enter deadlock until the end of the busy loop.

## 3.9 Problems With the Approach

One fundamental problem with the simulation technique we have focused on, is that it is not possible to distinguish between single “entities” that make up flows in the simulation. An example is the balance of available vehicles and vehicles in use. The time it takes for a vehicle to process an order has to be set to some average time, because the tool does not allow us to associate a random process time with each vehicle. This has to do with the fact that, in continuous simulation, entities are not atomic.

A possible solution in our case would be to let each vehicle be a part of the model instead of being an entity that flows through the model. In such a model, however, the complexity would increase with the number of vehicles. In particular, to change the number of vehicles in the model, one would have to modify the model itself, rather than just one of its parameters.

One of the simulation parameters is the total number of vehicles in the system. The number of vehicles must be chosen carefully, as it has great impact on the efficiency of the system as a whole. In addition, it is not possible to add an arbitrary number of vehicles without taking into consideration the size and complexity of the segment map.

As mentioned, the processing time for an order is set to a fixed value due to limitations in the tool (and simulation technique). In the real system, the processing time depends on a number of factors, e.g., the vehicle's location, the size of the map, and where the loading stations are

Parameters that have to do with the segment map, such as number of segments and segment lengths, are not included in the model at all. For the same reason as the processing time for an order is fixed, it had not been possible to include other than average values.

In an architecture, the primary entities are components that act together as a whole system. Connections between components can be of the same importance as components, but can also be assumed to simply exist when needed. A reason for this may be that connections can be realized by standardized protocols, e.g., CORBA. In a simulation model like the one we have created, the connections control how data are moved, and components are often merely data generators or data containers, e.g., see the network component in Figure 3.3. It represents a connection, but is not modeled as a simple connector. Instead, it is a complex unit to show the characteristics it is supposed to have. Thus, the components of the model do not map the components of the architecture very well.

## 3.10 Discussion

We have found that the part of the simulation process that was most rewarding was to develop the model. When creating the model, you are forced to reflect over the choices that has to be made in the architecture, resulting in a deepened understanding of the system that helps to identify potential points of concern.

When creating a model of a system, lots of decisions are taken to simplify it in order to speed up the modeling process. A simplification of some system behavior may be valid to make, but if it is erroneous it may as well render the model useless. Therefore, each step in the modeling has to be carefully thought through, something that slows down the entire modeling process.

A model easily becomes colored by the opinions and conceptions of the person that creates the model. Two persons may model the same system differently from each other, which indicates that it is uncertain whether or not a model is correct. Model verification and validation are the apparent tools to use here, but it is still inefficient to risk that a model is not objectively constructed. Therefore, we recommend that modeling always should be performed in groups.

While experimenting with the simulation tool, we have found that the ability to simulate a system is a good way to provide feedback to the modeler. It is possible to get a feeling for how the system is going to behave, which is a good way to find out if something has been overlooked in the architecture model. We believe this is independent of the method of simulation that is being used.

While building our experiment model we found that a library of model building blocks would have been of great help. The availability of a standardized way of modeling basic entities such as processes, networks, etc. would both speed up the modeling process and allow modelers to focus on the architecture instead of the modeling.

When simulating a software architecture, the focus can be put on different aspects, e.g., network or CPU utilization. The choice of aspect dictates what in the model that has to be modeled in detail. In our experiment, we chose to look at network utilization, and therefore it is the communication ways in the architecture that

have to be specifically detailed. This is noticeable in that communication channels in the model are complex structures rather than simple lines as in an architecture diagram.

## 3.11 Conclusions

In this chapter we have evaluated the applicability of continuous simulation as a support tool during evaluation of software architectures. Unfortunately, we conclude that continuous simulation does not fit for evaluation of software architectures. There are three reasons that make us come to this conclusion.

First, if continuous simulation is to be used, then we have to use average flow values when we parameterize the model. This makes the model become less dynamic and may have the consequence that the simulation model can be replaced with a static mathematical model.

Second, it is impossible to address unique entities when using continuous simulation. This is not always necessary when simulating flows of information, but if the flows depend on factors that are discrete in their nature, for example vehicles in an AGV system, then continuous simulation is a bad choice.

Third, the process of creating a model for simulation takes considerable time. Since an architecture evaluation generally has to be completed within a limited time, modeling becomes an impractical and uneconomical activity to perform during an evaluation.

We do, however, still believe that an architecture modeling tool that incorporates some simulation functionality could be helpful when designing software architectures. It could for example provide functionality for studying data flow rates between entities in an architecture. Such a tool would preferably be based on combined simulation techniques, because of the need to model discrete factors.

## Acknowledgments

We would like to thank Mikael Svahnberg, Blekinge Insitute of Technology, and Henrik Eriksson and Lars Ericsson, Danaher

Motion Särö, for valuable discussions, comments, and information about their AGV system. Finally, thanks to Åke Arvidsson at Ericsson AB for valuable simulation advises.

CHAPTER

4

# An Approach for Performance Evaluation of Software Architectures using Prototyping

*Frans Mårtensson, Håkan Grahn, and Michael Mattsson*

## 4.1 Introduction

The size and complexity of software systems are constantly increasing. During recent years, software engineering research has identified that the quality properties of software systems, e.g., performance and maintenance, often are constrained by their architecture [12]. Before committing to a particular software architecture, it is important to make sure that it handles all the requirements that are put upon it, and that it does this reasonably well. Bad architecture design decisions can result in a system with undesired characteristics, e.g., low performance and/or low maintainability.

When designing an architecture, there exists many different solutions to a given problem. Therefore, the design of the architecture should be supported by a well-defined, explicit method and relia-

ble data predicting the effects of design decisions, preferably in a quantifiable way. Examples of architecture evaluation methods are prototyping and scenario-based evaluation [19]. Each method has its own advantages and drawbacks, and there is no general consensus that a certain method is the best. Which method to use depends on time constraints and which quality attributes that are to be evaluated.

One important quality attribute to evaluate during architectural design is performance. Many times performance problems are not detected until system integration test, and thus are very costly to correct [83]. Some even argue that a design change is at least ten times more expensive after the code has been written than during architectural design. Therefore, it is important to evaluate the performance of a system as early as possible in the system development process, i.e., during the architectural design phase.

In this chapter, we present an approach that assess the performance characteristics of a software architecture, or a part of it. We apply the approach in a case study, an automated guided vehicle (AGV) system, where an early version of a communication component of the architecture is evaluated in order to identify its performance characteristics. The protypical method is based on an adaptation of the simulation based evaluation method as described by Bosch in [19]. We extend that approach by building an executable prototype of the software architecture, and thus evaluate the performance at the architectural level. The extensions to Bosch's method include, among others, the introduction of an evaluation support framework for gathering data in a consistent way during several subsequent evaluations as well as evaluation of candidate implementations or technologies.

We will with some background about software architecture in Section 4.2. In Section 4.3, we describe the simulation-based evaluation method, how we adapted it to prototype-based evaluation, and finally describe the resulting evaluation approach. Then, in Section 4.5, we illustrate the prototype based evaluation approach using a case study where an evaluation is conducted on an AGV system architecture. In Section 4.6 and Section 4.7 we discuss the results of the case study and how the evaluation approach worked, respectively. Finally, we conclude our study in Section 4.9.

## 4.2 Software Architecture

Software systems are constructed with a requirement specification as a base. The requirements in the requirement specification can be categorized into *functional* requirements and *non-functional* requirements, also called *quality requirements*. The design of software systems has traditionally been centred around the functional requirements. Although software engineering practice was forced to incorporate the quality requirements as well, software engineering research focused on the system functionality.

During recent years, the domain of software architecture [12, 71, 81] has emerged as an important area of research in software engineering. This is in response to the recognition that the architecture of a software system often constrains the quality attributes. Thus, architectures have theoretical and practical limits for quality attributes that may cause the quality requirements not to be fulfilled. If no analysis is done during architectural design, the design may be implemented with the intention to measure the quality attributes and optimize the system at a later state. However, the architecture of a software system is fundamental to its structure and cannot be changed without affecting virtually all components and, consequently, considerable effort.

Software architecture can be divided into three problem areas, i.e., designing, describing, and evaluating a software architecture. In this chapter we focus on evaluating software architectures, and in particular evaluating their performance. Four approaches to architecture evaluation can be identified, i.e., scenarios, simulation, mathematical modelling, and experience-based reasoning. Smith [83] discusses an approach to modelling system performance mathematically, although one may require simulation in certain cases. Our approach relies on the construction of an executable prototype of the architecture.

## 4.3 The Prototype-based Evaluation Approach

In the core of the prototype-based evaluation approach is the architecture prototype that approximates the behavior of the completed software system. When we were asked to perform the evaluation of the AGV system (see Section 4.5), we were unable to find a description of the steps involved in creating an architecture prototype. As a result we decided to take the basic workflow

from simulation based evaluation as described in [19] and adapt it to our needs. We will in the following section give a short introduction to the steps involved in performing a simulation-based architecture evaluation. We will then describe the changes that we made to that approach, and finally describe the resulting prototype-based evaluation approach.

### 4.3.1 Simulation-based Architecture Evaluation

A simulation-based evaluation is performed in five steps [19]:

1. Define and implement context.
2. Implement architectural components.
3. Implement profile.
4. Simulate system and initiate profile.
5. Predict quality attribute.

**Define and implement context.** In this first step two things are done. First, the environment that the simulated architecture is going to interact with is defined. Second, the abstraction level that the simulation environment is to be implemented at is defined (high abstraction gives less detailed data, low abstraction gives accurate data but increases model complexity).

**Implement architectural components.** In this step the components that make up the architecture are implemented. The component definitions and how the components interact with each other can be taken directly from the architecture documentation. The level of detail and effort that is spent on implementing the architecture components depends on both which quality attribute that we are trying to evaluate and the abstraction level that we have chosen to conduct the simulation at. If we are going to evaluate several quality attributes, then we will most likely have to implement more functionality than if we focus on only one.

**Implement profile.** A profile is a collection of scenarios that are designed to test a specific quality attribute. The scenarios are similar to use-cases in that they describe a typical sequence of events. These sequences are implemented using the architectural components that are going to be evaluated. This results in a model of the system components and their behavior. How a profile is implemented depends on which quality attribute we are

trying to assess as well as the abstraction level that is necessary for getting relevant data.

**Simulate system and initiate profile.** The simulation model is executed. During the execution data is gathered and stored for analysis. The type of data that is gathered depends on which quality attribute that we want to evaluate.

**Predict quality attribute.** The final step is to analyse the collected data and try to predict how well the architecture fulfils the quality requirements that we are trying to evaluate. This step is preferably automated since a simulation run usually results in a large amount of raw data.

### 4.3.2 Adaptations to the Evaluation Method

The workflow from the simulation-based evaluation had to be adapted to incorporate steps that we wanted to perform in our prototype-based evaluation. The main changes that we made were to introduce an evaluation support framework and put more emphasis on iteration in the evaluation process. We also did minor changes in the existing steps. These changes are described in more detail when we present our case study.

### 4.3.3 Evaluation Support Framework

We added the step of creating an evaluation support framework for use during the evaluation. A layered view of where the support framework is placed is shown in Figure 4.1. We choose to create the evaluation support framework for two reasons.

First, it makes us less dependent on the architecture component that we want to evaluate. The framework decouples the architecture component that we are evaluating from the architecture model that is used to generate input to the component. This increases the reusability of the architecture model as it only depends on the API provided by the framework and not directly on the architecture component.

Second, all logging can be performed by the framework, resulting in that neither the architecture model nor the architecture component that are evaluated need to care about the logging. This leads to both that the logging is done in a consistent way

independent of the underlying architecture component, and that no change has to be made to the architecture component when it is fitted to the framework. All that is needed is a wrapper class that translates between the calls from the framework and the architecture component. A more thorough discussion on how we constructed our framework can be found in section 4.5.2

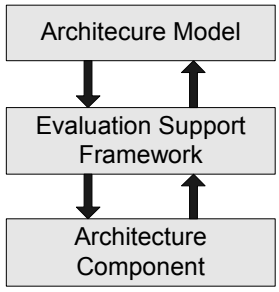


**Figure 4.1 A layered view of the prototype design.**

### 4.3.4 Iteration

During the development and execution of the prototype we found that it became necessary to perform the development of both the architecture model and the evaluation support framework in an iterative way. We needed to reiterate steps two to five in order to make adjustments to the way data was logged, and also to the behavior of the architecture model that was used. The need to make these changes was identified first after an initial execution of the simulation and analysis of the generated data. The positive thing with adding an iteration is that the initial results can be reviewed by experts (if such are available) that can determine if the results are sensible or not, and if changes to the model are necessary. We also got a confirmation that the log analysis tools were working correctly.

## 4.4 Prototype-based Architecture Evaluation

In order to perform a prototype based evaluation there are some conditions that has to be fulfilled.

- First, there has to be at least one architecture defined, if the goal of the evaluation is to compare alternative architectures to each other then we will of course need more.
- Second, if we want to evaluate the performance of one or more candidates for a part of the software architecture then these components has to be available. This is usually no problem with COTS components but might pose a problem if the components are to be developed in house.

In addition, it is a preferable, but not necessary condition, that the target platform (or equivalent) of the architecture is available. If it is possible to run the prototype on the correct hardware, it will give more accurate results.

After integrating our adaptations in the evaluation method we ended up with the following method for prototype based architecture evaluation.

1. Define evaluation goal.
2. Implement an evaluation support framework.
3. Integrate architectural components.
4. Implement architecture model.
5. Execute prototype.
6. Analyse logs.
7. Predict quality attribute.
8. If necessary, reiterate.

**Define evaluation goal.** Define what it is that should be evaluated, are we looking at more one or more architecture candidates or architecture components, and which quality attributes are we interested in evaluating.

**Implement an evaluation support framework.** The evaluation support framework's main task is to gather data that is relevant for fulfilling the evaluation goal that has been defined. Depending on the goal of the evaluation, the framework has to be designed accordingly, but the main task of the support framework

is always to gather data. The support framework can also be used to provide common functions such as utility classes for the architecture model.

**Integrate architectural components.** The component of the architecture that we want to evaluate has to be adapted so that the evaluation support framework can interact with it.

**Implement architecture model.** Implement a model of the architecture with the help of the evaluation support framework. The model together with the evaluation support framework and the component that is evaluated becomes an executable prototype.

**Execute prototype.** Execute the prototype and gather the data for analysis in the next step. Make sure that the execution environment matches the target environment as close as possible.

**Analyse logs.** Analyse the gathered logs and extract information regarding the quality attributes that are under evaluation. The analysis is with advantage automated as much as possible since the amount of data easily becomes overwhelming.

**Predict quality attribute.** Predict the quality attributes that are to be evaluated based on the information from the analysed logs.

**If necessary, reiterate.** This goes for all the steps in the evaluation approach. As the different steps are completed it is easy to see things that were overlooked during the previous step or steps. Once all the steps has been completed and results from the analysis are available, you could let an expert review them and use the feedback for deciding if adjustments have to be done to the prototype. These adjustments can be necessary in both the architecture model and the evaluation support framework. Another advantage is that it is possible to make a test run to validate that the analysis tools are working correctly and that the data that is gathered really is useful for addressing the goals of the evaluation.

## 4.5 A Case Study of an AGV System

The prototype based evaluation approach was specified in order to perform an evaluation for Danaher Motion Särö AB that is developing a new version of a control system for Automated Guided Vehicles (AGV:s). The system consists of a central server that controls a number of vehicles through a wireless network. Each vehicle has a client that controls it and communicates with the server. The client regularly position the vehicle through, e.g., laser measurements. The position is then sent back to the server which, based on the positioning information and information stored in a map database, decides what the client is to do next. Typical commands for the client is to drive a certain sequence of path segments, or load and unload cargo.

The client in the new system has a number of quality requirements that has to be accommodated, for example portability between different operating systems, scalability in functionality, and cost efficiency. The cost efficiency of the client is largely influenced by the price of the hardware that is needed to provide the necessary processing power to complete its computational tasks within a given timeperiod. This brings us to the performance of the client, since an efficient client will be able to work on slower hardware than a less efficient version, i.e., a more efficient client will be more cost efficient. The target platform for the new client is a Intel Pentium CPU at 133 MHz with an embedded version of the Linux operating system.

The prototype based evaluation method is applied to the architecture of the client and focus on how the internal communication in the client is handled. The clients consist of a number components that exchange information with each other. The components are realised as a number of threads within a process. In order to decrease the coupling between the components it was decided to introduce a component that provided a level of indirection between the other components by managing all communication in the client. This communication component is very crucial for the overall performance of the client as all communication between the other components in the client goes through this component. The communication component handles asynchronous communication only, the components communicate with each other by publishing telegrams (messages) of different types. Each component that is interested in some type of information has to register as a subscriber for that telegram type.

A first version of the communication component was already in use for the development of the other components. There were however some concerns regarding how well the communication component would perform on the target hardware for the new system. In order to verify that the new client would be able to fulfil the performance requirement it was decided that a performance evaluation should be done before to much time was spent on further development.

We will now go through the steps in the prototype-based evaluation method and describe what we did in each step. This will hopefully give the reader a more concrete feeling for the tasks that have to be done.

### 4.5.1 Define the Evaluation Goal

We defined the goal of the evaluation to be the performance of the communication component of the new AGV client. The component is critical as it handles the dispatching of messages between all the components in the client.

Because of safety and requirements regarding the positioning accuracy of the vehicles, the client has to complete a navigation loop within 50 ms. During this time a number of messages has to be passed between the components of the client. Together with the communication a certain amount of computation has to be performed in order to decide where the vehicle is and necessary course corrections. The amount of computation that has to be done varies only slightly from one loop to the next, so what remains that can affect the time it takes to complete the loop is the time it takes for the components to communicate with each other. In order to determine how good the communication component was we decided to gather the following three datapoints:

- The time it takes for a component to send a message.
- The time it takes for the communication component to deliver the message (message transit time).
- The time it takes to complete a navigation loop in our architecture model.
- Aside from the pure performance questions there were two additional questions, i.e., questions that we did not intend to focus the prototype on but that we would keep an eye out for

in order to get a feel for how the communication component handled them.

- Is there a large difference in performance between Linux and Windows 2000? This was a concern raised by some of the engineers developing the system.
- How easy is it to port the communication component from Windows 2000 to Linux?

So now we have defined the goal of our evaluation and we have defined the data that we will need in order to perform the evaluation.

### 4.5.2 Implement an Evaluation Support Framework

Based on the defined goal of the evaluation we created an evaluation support framework that would handle the gathering of data as well as separate the communication component from the architecture model. The conceptual model for the evaluation support framework that we constructed consisted of four main concepts: worker, log, framework, and communication component.

- A worker is someone that performs work such as calculations based on incoming data. A worker can both produce and consume messages. Instances of the worker are used to create the architecture model.
- The log is used to store information regarding the time it takes to perform tasks in the system. The framework uses the log to store information regarding sent and received messages.
- The framework provides a small API for the workers. It is mainly responsible for sending relevant events to the log but it also provides methods for generating unique message id:s as well as sending and receiving messages etc.
- The communication component is some kind of communication method that we are interested in evaluating. This is the exchangeable part of the framework

The four concepts were realised as a small number of classes and interfaces, as shown in Figure 4.2. The evaluation support framework provided an abstract worker class that contained basic functionality for sending and receiving messages. This class also generated log entries for each event that happened (such as the sending or receiving of a message). The class was also responsi-

ble for shutting down the testing once a predetermined time had elapsed, during the shutdown the log was flushed to disk. When creating a worker the user extended the abstract class and through that got all the messaging an logging functionality ready to use. Only some initialization was left to be done.

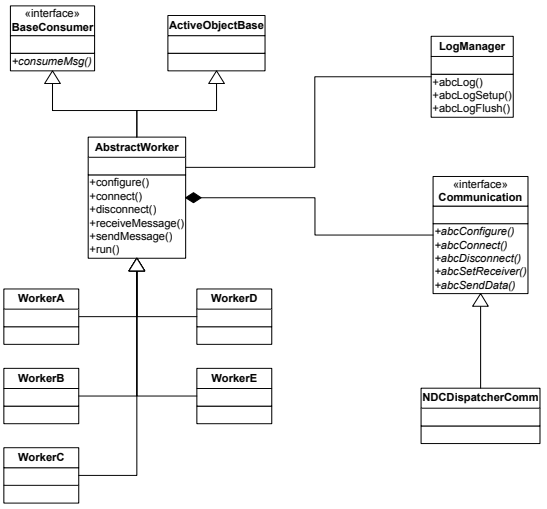


**Figure 4.2 A class diagram of the simulation framework.**

The log was realised in a LogManager class that stored log entries together with a timestamp for each entry. All log entries were stored in memory during the execution of the model and written to disk first after the execution had ended. This construction was chosen as it ensured that there, during execution, was no disk activity other than what was initiated by the OS, workers, or communication component.

The communication is represented by an interface in the framework. The interface only provided basic methods for starting up, configuring, shutting down, sending, receiving, connecting and disconnecting. If a new communication component is to be tested, a wrapper class is written that implements the communication interface and is able to translate the calls from the framework to the communication component.

### 4.5.3 Integrate Architectural Components

The communication component that we wanted to evaluate was integrated with the evaluation support framework. The component provided asynchronous communication based on publisher-subscriber, meaning that all components that are interested in some type of message subscribes to it using the communication component. When a message is sent to the communication component it is published to all the components that have subscribed to that type of message.

The communication interface for the framework was implemented by a wrapper class that passed on messages to be sent and received. It also handled the translation of telegram types from the support framework to the communication component.

### 4.5.4 Implement Architecture Model

With the framework in place and ready to use we went on to create the architecture model. It was built based on the architecture of the navigation system and focused on modeling the navigation loop. During the loop, several components interact with each other and perform computations as a response to different messages, as shown in Figure 4.3.

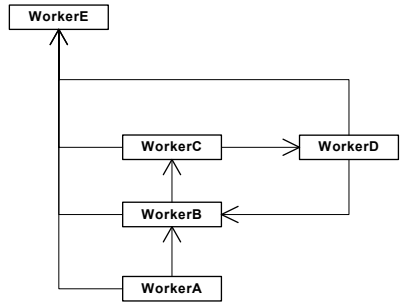


**Figure 4.3 The model of the different workers that interact during the simulation.**

Workers B, C, and D are the most critical components of the navigation loop. WorkerB initiates the loop when it receives a message from WorkerA which sends a message periodically every 50 ms. When workerB receives a message it works for 1 ms and then sends a message with its results. WorkerC then receives the

message from workerB and it proceeds to work for 5 ms before sending its message. This message is in turn received by workerD that also works for 5 ms before sending a new message. The message from workerD is received by workerB which notes that it has been received but does nothing more. WorkerE has subscribed to all the message types that exist in the model and thus receives all messages that are sent between the other workers.

In order to keep the model simple and easy to understand we simplified the interactions between the components so that each component only published one type of message.

Since we wanted to evaluate the performance of the communication component without affecting the system too much during execution we logged only time stamps together with a small identifier in runtime. This kept the time it took to create a log entry to a minimum. Log entries were created every time a worker entered or returned from the communication component and also when a message was actually sent. When sending a message we logged a time stamp together with the whole message.

### 4.5.5 Execute Prototype

The prototype was executed on three different hardware platforms, these were all Intel Pentium based platforms at the speeds of 133 MHz, 700 MHz, and 1200 MHz. The operating systems that were used were Linux, Windows 2000, and Windows XP. The operating system that the client is targeted for is Linux but all development is performed on Windows 2000 and test runs of the system are performed on the development platform. Therefore we wanted to run our prototype on that operating system as well. It also helped us to determine how portable the communication component was.

The architecture prototype was executed during 20 seconds on each platform and the logs from all runs were gathered and stored for further analysis.

Each execution of the prototype resulted in roughly half a megabyte of logged data. The data was stored in five separate log files (one for each worker) and the individual size of the log files varied between 50 and 150 KB. Each log entry was between 50 and 70 bytes and we gathered about 7000 log entries.

Based on the timestamps and identifiers from the logs we were able to extract information regarding two things.

- The time it took from that a worker called the send message function until the method returned. This measure is important as it is the penalty that the worker has to pay for sending a message.
- How long time any given message spent "in transit," i.e., how long time it took from that a worker sent a message until it was received by the recipient or recipients.

Measurements were also made in order to see how much overhead that was added to the communication by the evaluation support framework. We found that the framework added a delay of between 6 to 15 percent to the time it took to send a message. This results in 0.1 to 0.3 ms on the average send of a message on the Pentium 133Mhz based machine with Linux as OS (Table 4.1).

### 4.5.6 Analyse Logs

We implemented a small program that parsed the generated logs and extracted the information that we wanted from the data files. Based on the time stamps and identifiers from the logs we were able to extract information that we had defined as necessary for evaluating the communication component, namely:

- The time it takes for a component to send a message (Table 4.1).
- The time it takes for the communication component to deliver the message (Table 4.2).
- The time it takes to complete a navigation loop in our architecture model (Table 4.3).

In all the tables, L stands for Linux, W2K stands for Windows 2000, WXP stands for Windows XP. The number is the speed of the Pentium processor, N stands for NFS mounted system and F

stands for a system with a flash memory disk. Windows 2000 and Windows XP test used hard drives.

**Table 4.1 This table shows the time it took for a client to execute the method for sending a message. Values are in microseconds.**

| OS/ HW | L133N | L133F | L700N | L700F | W2K 700 | WXP 1200 |
|---|---|---|---|---|---|---|
| **Min** | 1562 | 1571 | 88 | 90 | 88 | 38 |
| **Med** | 1708 | 1716 | 97 | 97 | 97 | 67 |
| **Max** | 2093 | 2601 | 645 | 280 | 645 | 163 |

**Table 4.2 The table shows how long time a message spent in transit from sender to receiver. Values are in microseconds.**

| OS/ HW | L133N | L133F | L700N | L700F | W2K 700 | WXP 1200 |
|---|---|---|---|---|---|---|
| **Min** | 921 | 922 | 55 | 55 | 55 | 18 |
| **Med** | 3095 | 3094 | 1174 | 1173 | 1174 | 1157 |
| **Max** | 9241 | 9228 | 5076 | 5061 | 5076 | 5063 |

**Table 4.3 The table shows the time it took for the prototype to complete a navigation loop. Values are in microseconds.**

| OS/ HW | L133N | L133F | L700N | L700F | W2K 700 | WXP 1200 |
|---|---|---|---|---|---|---|
| **Min** | 22765 | 22712 | 12158 | 12158 | 12096 | 12067 |
| **Med** | 22840 | 22834 | 12161 | 12159 | 12130 | 12100 |
| **Max** | 23128 | 23128 | 12165 | 12163 | 12245 | 12190 |

In Table 4.3 we can see that the average time that it takes to complete a navigation loop on the L133 platforms is 22,8 ms. This figure can be broken down into two parts: work time and message transit time. During the loop workerB and workerC has to perform 10 ms of work and besides this workerE has to perform 3 times 1 ms of work resulting in 13 ms total work time. The average message transit time of about 3,1 ms per message adds up to an average of 9,3 ms for communication during the loop. The figures add up to 22,5 ms for a navigation loop where only three

messages are delivered. The fact that we spend about 40% of the time on communicating is taken as an indication that the communication component is unsuitable for the Pentium 133 MHz based system.

### 4.5.7 Predict Quality Attributes

Based on the information that were extracted from the logs, we concluded that the 133 Mhz Pentium based platform probably would be unable to fulfil the performance requirements for the systems. The time it took to dispatch a message was far to great to be able to handle the amounts of messages that the real system would generate.

Regarding the additional questions about the difference between operating systems and portability, we were able to draw the following conclusions:

- We found that the expected difference in performance between Windows 2000 and Linux didn't exist. The two operating systems performed equally well in the evaluation with less than 1% performance difference between the two.
- We showed that it would be easy to port the component from one platform to another, and from one compiler to another. All that was necessary for the port to build on Linux was that the necessary makefiles were written. The high portability of the system was attributed to the use of the ACE framework together with following the ANSI C++ standard.

### 4.5.8 Reiterate if Necessary

After a first iteration of all the steps in the evaluation method, some questions were raised regarding how the evaluation support framework handled the logging of data. The first version of the support framework flushed the communication logs to disk every 100 log entry. This could cause the OS to preempt the architecture prototype in order to complete the write operation. This in turn lead to spikes in the time it took for a message to be sent and received. In order to remove this problem the framework was changed so that the logs were stored in memory during the execution of the prototype and flushed to disk just before the prototype exited.

## 4.6 Results From the Case Study

The evaluation we performed in the case study resulted in that some fears regarding the performance of the new client for the AGV system were confirmed and that the developers took steps to investigate possible solutions to the problem. The evaluation also successfully answered the additional questions that were posed in the beginning of the evaluation.

When the implementation of the new system had become stable we made measurements on the new system in order to validate the results from the prototype. The data was gathered at the same points in the code as the evaluation support framework did and the measurements showed that the prototype produced the same message delivery times as the real system.

## 4.7 Analysis of the Evaluation Method

We feel confident that the prototype based evaluation approach is useful for assessing the performance characteristics of an architecture component and also for evaluating the performance of architecture models derived from proposed software architectures. New architecture models are easily implemented using the evaluation support framework and the amount of reuse, both in code and analysis tools makes the creation of a support framework and analysis tools worth the effort.

The support framework separates the component that we are evaluating from the architecture model, making it possible to compare alternative components in a consistent way as the data for the evaluation is gathered in the same way independently of the component.

A concern that can be raised against the use of an evaluation support framework is that since a message has to go through the framework classes before it reaches the component that we are evaluating there is an added delay. In our case study we found that the delay between that the worker sent the message and that the message was actually sent by the interaction component was quite small. The framework added between 0,1 to 0,3 ms to the time it took to send a message.

## 4.8 Future Work

We plan to continue to use the test framework and the test applications and try to evaluate other quality attributes such as scalability and maintainability.

The test-framework and applications will be used to perform follow-up evaluations at Danaher Motion Särö AB in order to see how the communication component develops, and ultimately to compare how well the original estimations match the performance of the final system.

## 4.9 Conclusions

In this chapter we have described the prototype based architecture evaluation approach and the steps that it consists of. The approach is based on the simulation based evaluation approach but adds mainly the construction of an evaluation support framework and a clearer focus on iteration during the evaluation. The use of the evaluation support framework simplifies the implementation of alternative architecture models, makes consistent gathering of data simpler, and makes it possible to evaluate alternative implementations of an architecture component.

The evaluation approach has been used to evaluate the performance characteristics of a communication component in an AGV system architecture. The evaluation resulted in that a performance problem was identified and that two additional questions were evaluated as well (Portability and performance differences between Windows 2000 and Linux). Results from the case study were also used to validate the evaluation approach once the new system was stable, and showed that it produced accurate results.

The case study illustrates the steps within the evaluation process and can be seen as a guide for performing a prototype based evaluation.

## Acknowledgments

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.ipd.bth.se/

besq). We would also like to thank Danaher Motion Särö AB for providing us with a case for our case study and many interesting discussions and ideas.

CHAPTER

5

# Evaluating Software Quality Attributes of Communication Components in an Automated Guided Vehicle System

*Frans Mårtensson, Håkan Grahn, and Michael Mattsson*

## 5.1 Introduction

The size and complexity of software systems are constantly increasing. It has been identified that the quality properties of software systems, e.g., performance and maintenance, often are constrained by their software architecture [12]. The software architecture is a way to manage the complexity of a software system and describes the different parts of the software system, i.e., the components, their responsibilities, and how they interact with each other. The software architecture is created early in the development of a software system and has to be kept alive throughout the system life cycle. One part of the process of creat-

ing a software architecture is the decision of possible use of existing software components in the system.

The system we study in this chapter is an Automated Guided Vehicle system (AGV system) [24], which is a complex distributed real-time system. AGV systems are used in industry mainly for supply and materials handling, e.g., moving raw materials, and finished products to and from production machines. Important aspects to handle in such systems are, e.g., the ability to automatically drive a vehicle along a predefined path, keeping track of the vehicles' positions, routing and guiding the vehicles, and collision avoidance. The software in an AGV system has to be adaptable to quite different operating environments, e.g., iron works, pharmacy factories, and amusement parks. More importantly, the system may under no circumstances inflict harm on a person or object. The safety and flexibility requirements together with other quality- and functional requirements of the system make it a complex software system to create and maintain. In the system in our case study, the software in the vehicle can be divided into three main parts that continuously interact in order to control the vehicle. These parts communicate both within processes as well as between processes located on different computers.

In this chapter we evaluate two communication components used in an existing AGV system and compare them to an alternative COTS (commercial-off-the-shelf) component for communication [79] that is considered for a new version of the AGV system. We evaluate three quality attributes for each of the components: performance, maintainability, and portability. We use three prototypes built using a prototype framework to measure the performance of each component. Both intra-process as well as inter-processes communication are evaluated. The communicating processes reside both on the same computer and on two different computers connected by a network. We measure the maintainability of the three components using the Maintainability Index metric [69]. We also discuss qualitative data for the portability aspects of the components.

The evaluation is performed in an industrial context in cooperation with Danaher Motion Särö. The usage scenarios and architecture description that are used during the evaluations have been developed in cooperation with them.

Our results indicate that the performance of the COTS component is approximately half the performance of the in-house developed communication components. On the other hand, using a third party COTS component significantly reduce the maintenance effort as well as increase the functionality. Finally, all three components turned out to be portable from Windows XP to Linux with very little effort.

The rest of the chapter is organized as follows. Section 5.2 presents some background to software architecture, architecture evaluation, and automated guided vehicle systems. In Section 5.3 we introduce the components and the quality attributes that we evaluate. We present our evaluation results in Section 5.4. In Section 5.5 we discuss related work and, finally, in Section 5.6 we conclude our study.

## 5.2 Background

In this section, we give some background about software architectures, how to evaluate them, different quality attributes, and the application domain, i.e., automated guided vehicle systems.

### 5.2.1 Software Architecture

Software systems are developed based on a requirement specification. The requirements can be categorized into *functional* requirements and *non-functional* requirements, also called *quality requirements*. Functional requirements are often easiest to test (the software either has the required functionality or not) but the non-functional requirements are harder to test (quality is hard to define and quantify).

In the recent years, the domain of software architecture [12, 19, 71, 81] has emerged as an important area of research in software engineering. This is in response to the recognition that the architecture of a software system often constrains the quality attributes. Software architecture is defined in [12] as follows:

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.”

Software architectures have theoretical and practical limits for quality attributes that may cause the quality requirements not to be fulfilled. If no analysis is done during architectural design, the design may be implemented with the intention to measure the quality attributes and optimize the system. However, the architecture of a software system is fundamental to its structure and cannot easily be changed without affecting virtually all components and, consequently, considerable effort. It has also been shown that several quality attributes can be in conflict with each other, e.g., maintainability and performance [38]. Therefore, it is important to evaluate all (or at least the most) relevant quality attributes at the software architecture level.

### 5.2.2 Evaluation Methodology

In order to make sure that a software architecture fulfils its quality requirements, it has to be evaluated. Four main approaches to architecture evaluation can be identified, i.e., scenarios, simulation, mathematical modelling, and experience-based reasoning [19]. In this chapter we use a prototype-based architecture evaluation method which is part of the simulation-based approach and relies on the construction of an executable prototype of the architecture [19, 63, 83]. Prototype-based evaluation enables us to evaluate software components in an execution environment. It also lets the developer compare all components in a fair way, since all components get the same input from a simplified architecture model. An overview of the parts that go into a prototype is shown in Figure 5.1. A strength of this evaluation approach is that it is possible to make accurate measurements on the intended target platform for the system early on in the development cycle.

The prototype-based evaluation is performed in seven steps plus reiteration. We will describe the steps shortly in the following paragraphs.

**Define the evaluation goal.** In this first step two things are done. First, the environment that the simulated architecture is going to interact with is defined. Second, the abstraction level that the simulation environment is to be implemented at is defined (high abstraction gives less detailed data, low abstraction gives accurate data but increases model complexity).

**Implement an evaluation support framework.** The evaluation support framework's main task is to gather data that is relevant to

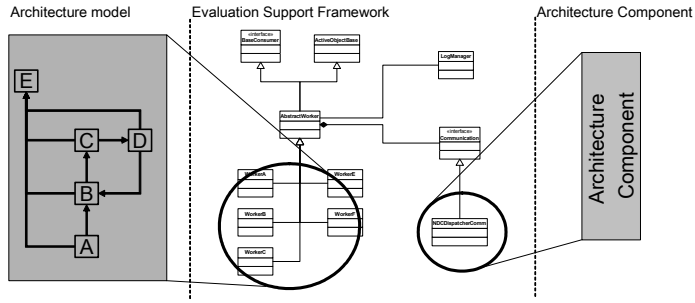


**Figure 5.1 The prototype consists of three main parts: the architecture model, the evaluation support framework, and the architecture components.**

fulfilling the evaluation goal. Depending on the goal of the evaluation, the support framework has to be designed accordingly, but the main task of the support framework is to simplify the gathering of data. The support framework can also be used to provide common functions such as base and utility classes for the architecture models.

**Integrate architectural components.** The component of the architecture that we want to evaluate has to be adapted so that the evaluation support framework can interact with it. The easiest way of achieving this is to create a proxy object that translates calls between the framework and the component.

**Implement architecture model.** Implement a model of the architecture with the help of the evaluation support framework. The model should approximate the behavior of the completed system as far as necessary. The model together with the evaluation framework and the component that is evaluated is compiled to an executable prototype.

**Execute prototype.** Execute the prototype and gather the data for analysis in the next step. Try to make sure that the execution environment matches the target environment as close as possible.

**Analyse logs.** Analyse the gathered logs and extract information regarding the quality attributes that are under evaluation. Automated analysis support is preferable since the amount of data easily becomes overwhelming.

**Predict quality attribute.** Predict the quality attributes that are to be evaluated based on the information from the analysed logs.

**Reiteration.** This goes for all the steps in the evaluation approach. As the different steps are completed it is easy to see things that were overlooked during the previous step or steps. Once all the steps has been completed and results from the analysis are available, you should review them and use the feedback for deciding if adjustments have to be done to the prototype. These adjustments can be necessary in both the architecture model and the evaluation support framework. It is also possible to make a test run to validate that the analysis tools are working correctly and that the data that is gathered really is useful for addressing the goals of the evaluation.

### 5.2.3 Automated Guided Vehicle Systems

As an industrial case we use an Automated Guided Vehicle system (AGV system) [24]. AGV systems are used in industry mainly for supply and materials handling, e.g., moving raw materials, and finished products to and from production machines.

Central to an AGV system is the ability to automatically drive a vehicle along a predefined path, the path is typically stored in a path database in a central server and distributed to the vehicles in the system when they are started. The central server is responsible for many things in the system, it keeps track of the vehicles positions and uses the information for routing and guiding the vehicles from one point in the map to another. It also manages collision avoidance so that vehicles do not run into each other by accident and it detects and resolves deadlocks when several vehicles want to pass the same part of the path at the same time. The central server is also responsible for the handling of orders from operators. When an order is submitted to the system, e.g., “go to location A and load cargo”, the server selects the closest free vehicle and begins to guide it towards the pickup point.

In order for the central server to be able to perform its functions, it has to know the exact location of all vehicles under its control on the premise. Therefore every vehicle sends its location to the server several times every second. The vehicles can use one or several methods to keep track of its location. The three most common methods are induction wires, magnetic spots, and laser range finders.

The first method, and also the simplest, is to use induction wires that are placed in the floor of the premises. The vehicles are then able to follow the electric field that the wire emits and from the modulation of the field determine where it is. A second navigation method is to place small magnetic spots at known locations along the track that the vehicle is to follow. The vehicle can then predict where it is based on a combination of dead reckoning and anticipation of coming magnetic spots. A third method is to use a laser located on the vehicle, that measures distances and angles from the vehicle to a set of reflectors that has been placed at known locations throughout the premises. The control system in the vehicle is then able to calculate its position in a room based on the data returned from the laser.

Regardless of the way that a vehicle acquires the information of where it is, it must be able to communicate its location to the central control computer. Depending on the available infrastructure and environment in the premises of the system, it can for example use radio modems or a wireless LAN.

The software in the vehicle can be roughly divided into three main components that continuously interact in order to control the vehicle. These components require communication both within processes and between processes located on different computers. We will perform an evaluation of the communication components used in an existing AGV system and compare them to an alternative COTS component for communication that is considered for a new version of the AGV system.

## 5.3 Component Quality Attribute Evaluation

In this section we describe the components that we evaluate, as well as the evaluation methods used. The goal is to assess three quality attributes, i.e., performance, portability and maintainability for each component. The prototypes simulate the software that is controlling the vehicles in the AGV system. The central server is not part of the simulation.

### 5.3.1 Evaluated Communication Components

The components we evaluate are all communication components. They all distribute events or messages between threads within a process and/or between different processes over a network con-

nection. Two of the components are developed by the company we are working with. The third component is an open source implementation of the CORBA standard [68].

**NDC Dispatcher.** The first component is an implementation of the dispatcher pattern which provides publisher-subscriber functionality and adds a layer of indirection between the senders and receivers of messages. It is used for communication between threads within one process and can not pass messages between processes. The NDC Dispatcher is implemented with active objects using one thread for dispatching messages and managing subscriptions. It is able to handle distribution of messages from one sender to many receivers. The implementation uses the ACE framework for portability between operating systems. This component is developed by the company and is implemented in C++.

**Network Communication Channel.** Network Communication Channel (NCC) is a component is developed by the company as well. It is designed to provide point to point communication between processes over a network. It only provides one to one communication and has no facilities for managing subscriptions to events or message types. NCC can provide communication with legacy protocols from previous versions of the control system and can also provide communication over serial ports. This component is developed by the company and is implemented in C.

**TAO Real-time Event Channel.** The third component, The ACE Orb Real-time Event Channel (TAO RTEC) [79], can be used for communication between both threads within a process, and between processes both on the same computer and over a network. It provides communication from one to many through the publisher-subscriber pattern. The event channel is part of the TAO CORBA implementation and is open source. This component can be seen as a commercial of-the-shelf (COTS) component to the system. We use TAO Real-time Event Channel to distribute messages in the same way that the NDC Dispatcher does.

**Software Quality Attributes to Evaluate.** In our study we are interested in several quality attributes. The first is performance because we are interested in comparing how fast messages can be delivered by the three components. We assess the performance at

the system level and look at the performance of the communication subsystem as a whole.

The second attribute is maintainability which was selected since the system will continue to be developed and maintained under a long period. The selected communication component will be an integral part of the system, and must therefore be easy to maintain.

The third attribute is portability, i.e., how much effort is needed in order to move a component from one environment/platform to another. This attribute is interesting as the system is developed and to some extent tested on computers running Windows, but the target platform is based on Linux.

**Performance.** We define performance as the time it takes for a communication component to transfer a message from one thread or process to another. In order to measure this we created one prototype for each communication component. The prototypes were constructed using a framework that separates the communication components from the model of the interaction, i.e., the architecture. As a result, we can use the same interaction model for the prototypes and minimize the risk of treating the communication components unequally in the test scenarios. The framework from a previous study [63] was reused, and functionality was added for measuring the difference in time for computers that were connected via a network. This information was used to adjust the timestamps in the logs when prototypes were running on separate computers, and to synchronize the start time for when the prototypes should start their models.

We created two different models of the interaction: one for communication between threads in a process and one for communication between processes on separate computers that communicate via a network. The NDC Dispatcher was tested with the model for communication between threads in a process and NCC was tested with the model for communication over the network. TAO RTEC was tested with both models since it can handle both cases.

An example of a model can be seen in Figure 5.2 which shows the interaction between threads in a process for the NDC Dispatcher prototype. Here, thread A sends a message to thread B every 50 ms. Thread B sends the message on to thread C. Thread C sends it to thread D which in turn send it back to thread B. Each message is marked with a timestamp and stored to a logfile for offline analysis.

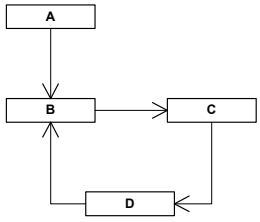


**Figure 5.2 Interaction within a prototype.**

The prototypes were executed three times on a test platform similar to the target environment and we calculated the average response time of the three runs. The test environment consisted of two computers with a 233Mhz Pentium 2 processor and 128 MB RAM each. Both computers were running the Linux 2.4 kernel and they were connected with a dedicated 10Mbps network.

**Maintainability.** We use a tool called CCCC [57, 58] to collect a number of measures (e.g., number of modules, lines of code, and cyclomatic complexity) on the source code of the components. The objective is to use these measures to calculate a maintainability index metric [69] for the components. The maintainability index (MI) is a combination of the average halstead volume per module (aveVol), the average cyclomatic complexity per module (aveV(g')), average lines of code per module (aveLoc), and average percentage of lines of comments per module (aveCM), as shown in Figure 5.3. The maintainability index calculation results in a value that should be as high as possible. Values above 85 are considered to indicate good maintainability, between 85 and 65 is medium maintainability, and finally, values below 65 are indicating low maintainability [69]. Based on the maintainability index together with our qualitative experiences from developing the prototypes, we evaluate and compare the

maintainability of the components. We do not see the maintainability index as a definite judgement of the maintainability of the components but more as a tool to indicate the properties of the components and to make them comparable.

$$MI = 171 - 5.2 \times \ln(aveVol) - 0.23 \times aveV(g') - 16.2 \times \ln(aveLoc) + 50 \times \sin(\sqrt{2.46 \times aveCM})$$

**Figure 5.3 Formula for calculating the maintainability index (MI) [69].**

**Portability.** We define portability as the effort needed to move the prototypes and communication components from a Windows XP based platform to a Linux 2.4 based platform. This is a simple way of assessing the attribute but it verifies that the prototypes actually works on the different platforms and it gives us some experience from making the port. Based on this experience we can make a qualitative comparison of the three components.

## 5.4 Evaluation Results

During the evaluation, the largest effort was devoted to implementing the three prototypes and running the performance benchmarks. The data from the performance benchmarks gave us quantitative performance figures which together with the experience from the implementations were used to assess the maintainability and portability of the components.

### 5.4.1 Performance Results

After implementing the prototypes and performing the test runs, the gathered logs were processed by an analysis tool that merged the log entries, compensated for the differences in time on the different machines and calculated the time it took to transfer each message.

### 5.4.2 Intra Process Communication

The intra process communication results in Table 5.1 show that the average time it takes for the NDC Dispatcher to deliver a message is 0,3 milliseconds. The same value for TAO RTEC is 0,6 milliseconds. The extra time that it takes for TAO RTEC is mainly due to the differences in size between TAO RTEC and the

NDC Dispatcher. TAO RTEC makes use of a CORBA ORB for dispatching the events between the threads in the prototype. This makes TAO RTEC very flexible but it impacts its performance when both publisher and subscriber are threads within the same process; the overhead in a longer code path for each message becomes a limiting factor. The NDC Dispatcher on the other hand is considerably smaller in its implementation than TAO RTEC, resulting in a shorter code path and faster message delivery.

**Table 5.1 Intra process communication times.**

| | NDC Dispatcher | TAO RTEC |
|---|---|---|
| **Intra process** | 0,3 ms | 0,6 ms |

During the test runs of the NDC Dispatcher and the TAO RTEC based prototypes we saw that the time it took to deliver a message was not the same for all messages. Figure 5.4 and Figure 5.5 show a moving average of the measured delivery times in order to illustrate the difference in behavior between the components. In both the NDC Dispatcher and the TAO RTEC prototypes this time depends on how many subscribers there are to the message, and the order the subscribers subscribed to a particular message. We also saw that there is a large variation in delivery time from message to message when using TAO RTEC. It is not possible to guarantee that the time it takes to deliver a message will be constant when using neither the NDC Dispatcher nor TAO RTEC, but the NDC Dispatcher's behavior is more predictable.

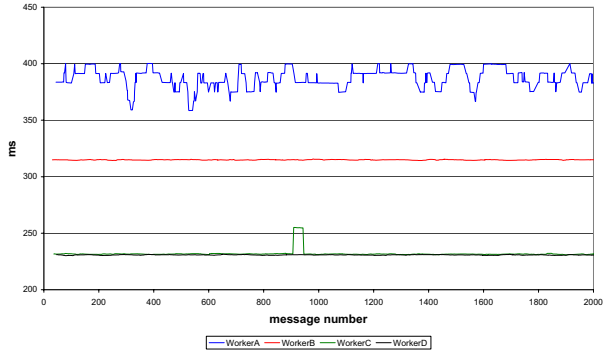


**Figure 5.4 NDC Dispatcher message delivery time (moving average).**

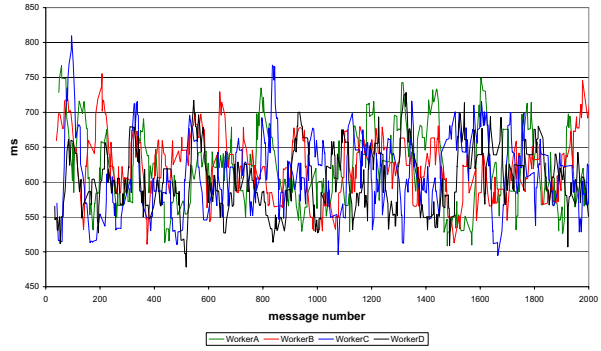


**Figure 5.5 TAO RTEC message delivery time (moving average).**

### 5.4.3 Inter Process Communication

The inter process communication is evaluated in two different environments; when the communicating processes are on the same computer and when they reside on two different computer connected by a network.

In Table 5.2 we present the message delivery times when the processes reside on the same computer. We find that TAO RTEC takes 2,0 milliseconds on average to deliver a message, while NCC only takes 0,8 milliseconds to deliver a message. Much of the difference comes from the fact that TAO RTEC offers much more flexibility, e.g., communication one-to-many, while NCC only provides one-to-one communication. Another contributing factor is that TAO RTEC runs the event channel in a separate process from the prototypes. This results in an added delay as messages are sent to the event channel process before they are delivered to the recipient process. NCC on the other hand, delivers the messages directly to the recipient process.

The inter process communication in Table 5.3 shows that TAO RTEC takes on average 2 milliseconds to deliver a message from one computer to another in our test environment. NCC takes on average 1 millisecond. The extra time needed for TAO RTEC to deliver a message is, as discussed earlier, a result of the longer code path involved due to the use of CORBA, and the need of an intermediary process for distributing the messages to the subscribers. The gain of using this component is added flexibility in how messages can be distributed between subscribers on differ-

ent computers. In comparison, the NCC component is only able to pass messages from one point to another, making it less complex in its implementation.

**Table 5.2 Communication times between processes running on the same computer.**

| | TAO RTEC | NCC |
|---|---|---|
| **Inter process** | 2,0 ms | 0,8 ms |

**Table 5.3 Communication times between processes running on different computers.**

| | TAO RTEC | NCC |
|---|---|---|
| **Inter process over network** | 2,0 ms | 1,0 ms |

In Table 5.4 we present the amount of data that is transmitted (and in how many TCP/IP packages) over the network by prototypes using TAO RTEC and NCC, respectively. In the architecture model, both prototypes perform the same work and send the same number of messages over the network. In the table we see that both components send about the same number of TCP/IP packages (TAO RTEC sends 37 more than NCC). The difference is located to the initialization of the prototypes where a number of packages are sent during ORB initialization, name resolution, and subscriptions to the event channel etc. When we look at the amount of data sent in the packages we see that TAO RTEC sends about 55% more data than NCC does. This indicates that NCC has less overhead per message than TAO RTEC has. Both components do however add considerably to the amount of data that is generated by the model, which generated 6 kb of data in 300 messages.

**Table 5.4 Network traffic generated by TAO RTEC and NCC.**

| | TAO RTEC | NCC |
|---|---|---|
| **TCP/IP packages** | 800 packages | 763 packages |
| **Data over network** | 137 kb | 88 kb |

In summary, we find that TAO RTEC has half the performance of both the NDC Dispatcher and NCC for both intra- and inter-process communication. However, TAO RTEC has the advantage that it can handle both intra- and inter-process communication using the same communication component, while the NDC Dispatcher and NCC can handle only one type of communication (either intra-process or inter-process).

### 5.4.4 Maintainability Results

The measures that we gathered using CCCC are listed in Table 5.5, and the metrics are defined as follows. Modules (MOD) is the number of classes and modules with identified member functions. Lines of code (LOC) and Lines of comments (COM) are measures for the source code, and a combination of them gives an indication of how well documented a program is (LOC/COM and LOC/MOD). The COM measure can also be combined with the cyclomatic complexity (CYC) to give an indication of how well documented the code is in relation to the code complexity. The cyclomatic complexity is also used in combination with the module count in order to give an indication of the program complexity per module (CYC/MOD). When analyzing the results we put the most weight on compound measures such as the maintainability index, cyclomatic complexity per comment, and cyclomatic complexity per module.

**Table 5.5 Metrics from the CCCC tool [58].**

| Metric | NDC Dispatcher | NCC | TAO RTEC |
|---|---|---|---|
| **Modules (MOD)** | 23 | 57 | 3098 |
| **Lines of code (LOC)** | 533 | 23982 | 312043 |
| **Lines of comments (COM)** | 128 | 19827 | 78968 |
| **LOC/COM** | 4,164 | 1,210 | 3,952 |
| **LOC/MOD** | 23,174 | 420,737 | 100,724 |
| **Cyclomatic complexity (CYC)** | 69 | 3653 | 34927 |
| **CYC/COM** | 0,539 | 0,184 | 0,442 |
| **CYC/MOD** | 3,0 | 64,088 | 11,274 |
| **Maintainability Index** | 128,67 | 50,88 | 78,91 |

The NDC Dispatcher is the smallest of the three components with 533 lines of code in 23 modules (see Table 5.5). The complexity per module is the lowest but the complexity per comment is the highest of all the components. While working with this component we found it easy to use and easy to get an overview of. The component also has the highest maintainability index (128,67) of the three components, indicating a high maintainability.

NCC is 23982 lines of code in 57 modules. It is also the most commented component of the three, which is shown in the low cyclomatic complexity per comment value. However, there are indications in the LOC/MOD and CYC/MOD measures that the component has very large modules. This can make NCC difficult to overview, thus lowering its maintainability. The maintainability index supports this assessment, since NCC is the component with the lowest maintainability index (50,88) indicating poor maintainability.

TAO RTEC is 312043 lines of code in 3098 modules. This is by far the largest component of the three. Although the parts that are used for the real-time communication channel are smaller (we gathered metrics for all the parts of TAO) it is still difficult to get an overview of the source code. The maintainability index for TAO RTEC (78,91) puts it in the medium maintainability category. We do, however, think that the size of the component makes it difficult to maintain within the company. The question of maintainability is relevant only if one version of TAO is selected for continued use in the company. If newer versions of TAO are used as they are released then the maintenance is continuously done by the developer community around TAO. On the other hand, there is a risk that API:s in TAO are changed during development, breaking applications. But since the application developers are with the company, this problem is probably easier to deal with than defects in TAO itself.

### 5.4.5 Portability Results

Based on our experiences from building the prototypes we found that moving the prototypes from the Windows-based to the Linux-based platform was generally not a problem and did not take very long time (less than a day per prototype). Most of the time was spent on writing new makefiles and not on changing the code for the prototypes.

Both the NDC Dispatcher and TAO RTEC are developed on top of the ADAPTIVE Communications Environment (ACE) [79]. ACE provides a programming API that has been designed to be portable to many platforms. Once ACE was built on the Linux platform it was easy to build the prototypes that used it.

NCC was originally written for the Win32 API and uses a number of portability libraries built to emulate the necessary Win32 API:s on platforms other than windows. Building the prototype using NCC was not more complicated than those using the NDC Dispatcher or TAO RTEC.

## 5.5 Related Work

Prototypes are commonly used in interface design, where different alternatives to graphical user interfaces can be constructed and tested by users and developers [15]. The use of prototypes for architecture simulation and evaluation has been described and discussed in [11, 63]. The goal is to evaluate architectural alternatives before the detailed design documents have been developed, making it possible to obtain performance characteristics for architecture alternatives and hardware platform working together. Other commonly used performance models are queueing networks, stochastic petri nets, stochastic process algebra, and simulation models [8]. Software Performance Engineering based and architectural-pattern based approaches both use information obtained from UML design documents (Use Case, System Sequence, and Class diagrams) for the evaluation of the software architecture. This makes it possible to make performance evaluations as soon as the design of the system begins to take shape. A weakness of these performance evaluation models is that it is difficult to capture the dynamic properties of the executing code when it interacts with the operating system.

Several methods for evaluating one aspect of a components quality attributes have been described [91]. Most of the methods focus on the evaluation of different performance aspects of components. However, when selecting components it is likely that more than the performance attribute is of interest for the developers, this result in a need to perform evaluations for several quality attributes for the components. Qualities such as maintainability

can be quantified and compared using for example the maintainability index [69]. Using tools for static analysis of the source code of the components makes it possible to extract complexity and maintainability metrics for components.

Methods for assessing several quality attributes during an evaluation exist in several architecture level evaluation methods. Methods such as the scenario-based Software Architecture Analysis Method (SAAM) [47] and Architecture Tradeoff Analysis Method (ATAM) [49], as well as the attribute-based ABAS [51] method can be used to assess a number of quality attributes using scenario-based evaluation. Especially ATAM tries to handle several quality attributes and their impact on the software architecture simultaneously. The evaluation methods that we used in this chapter can be used to supply input for both SAAM and ATAM. In addition, the method that we have used in this chapter can also complement the results from SAAM and ABAS, i.e., they focus around qualitative reasoning while our method provides quantitative data. Together, the methods can address a broader spectrum of quality attributes.

## 5.6 Conclusions

In this chapter we have used a prototype-based evaluation methods for assessing three quality attributes of three different communication components. We have shown that it is possible to compare the three evaluated components in a fair way using a common framework for building the prototypes and analyzing the resulting data. The components were one COTS component, i.e., The ACE Orb Real-Time Event Channel (TAO RTEC), and two inhouse developed components, the NDC Dispatcher and NCC. For each of the components we have evaluate three quality attributes: performance, maintainability, and portability. The performance and maintainability are evaluated quantitatively, while portability is evaluated qualitatively.

The performance measurements show that TAO RTEC has half the performance of the NDC Dispatcher in communication between threads within a process. Our results also show that TAO RTEC has approximately half the performance of NCC in com-

munication between processes. On the other hand, TAO RTEC provides functionality for both intra- and inter-process communication, while the NDC Dispatcher and NCC only support one type of communication.

As for the maintainability, the NDC Dispatcher has the highest maintainability index of the three components (it indicated a high maintainability for the component). NCC turned out to have the lowest maintainability index (so low that it indicated a low maintainability for the component). However, even though NCC has the lowest maintainability index of all the components, we think that it is rather easy for the company to maintain since it has been developed within the company and is well documented. TAO RTEC is the largest of the three components, with a medium high maintainability index, and the knowledge of how it is constructed is not within the company. Therefore, we think that TAO RTEC is less maintainable for the company. On the other hand, the company can take advantage of future development of TAO RTEC with little effort as long as the API:s remain the same.

Finally, considering the portability aspects. All three evaluated components fulfill the portability requirement in this study. We had no problems moving the prototypes from a Windows-based to a Linux-based platform.

## Acknowledgments

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project “Blekinge - Engineering Software Qualities (BESQ)” http://www.bth.se/besq. We would like to thank Danaher Motion Särö AB for providing us with a case for our case study and many interesting discussions and ideas.

CHAPTER

6

# Forming Consensus on Testability in Software Developing Organizations

*Frans Mårtensson, Håkan Grahn, and Michael Mattsson*

## 6.1 Introduction

In software developing organizations there exist a number of roles. These roles range from, e.g., project managers through software and hardware engineers to test engineers. As an organization grows and evolves, new people are introduced to the different roles. Each person brings their own knowledge and experience to the organization based on their background and education. Their background thus influences how they practice their role. As a result, an organization that from the beginning had a shared set of definitions and understandings between people in different roles, can after some time end up in a state where this is no longer the case. Roles can start to have different mean-

ings of the same concept. But, when people in different roles no longer understand what a concept means to another role in the organization, it can become a source of misunderstandings, and also generate additional costs.

For example, an organization decides that the software system that they are developing needs to be improved, and a set of desirable quality attributes and requirements is selected. The changes made to a software system can be driven by many sources, ranging from business goals, e.g., new functionality requested by the customers, to pure maintenance changes, e.g., error corrections. In addition, the developers designing and implementing the system also have an impact on how the system changes. Therefore, it is important that all roles in the organization have an understanding of what the quality attributes mean to the other roles. One important quality attribute for a software system is testability [61, 39, 42], having high testability simplifies the task of validating the system both during and development and maintenance [88].

In this chapter we examine a software developing organization. We look for different definitions of and views on testability between different roles in the organization. We devise a questionnaire and use it to gather the opinions and views on testability of the people in the organization. The questionnaire is then followed up with some additional questions raised during the analysis of the responses. The follow-up questions were posed during telephone interviews. Finally we analyze and discuss the results of the examination. We plan a workshop where we will try to identify the reasons for the different opinions of the respondents, other than the ones that we have identified from the questionnaire.

The rest of the chapter is organized as follows. In the next section we introduce software testing and testability. Then, in Section 6.3, we define the basis for our case study, e.g., the goals and participants in the study. In Section 6.4 and Section 6.5, we present the results from our questionnaire along with an analysis of them. Then, in Section 6.6 we discuss the validity of our findings. Finally, we conclude our study in Section 6.7.

## 6.2 Software Testing and Testability

Software testing is the activity of verifying the correctness of a software system. The goal is to identify defects that are present in the software so that they can be corrected. Several classes of tests exist and the tests can be performed on a number of levels in the system. Typical tests used in software development are unit tests, regression tests, integration test, and system tests.

Different types of tests can be introduced at different points in the software life cycle. Unit tests are often introduced during the implementation phase, and focus on testing a specific method or class in the software system. Unit tests can be constructed by the developer as he writes the program code and used as a verification that the code that has been produced meets the requirements (both functional and non-functional) posed by the requirements specification.

Test types used later in the development cycle are, e.g., integration tests and system function tests. The integration tests test that the software modules that have been developed independently of each other still work as specified when they are integrated into the final system configuration. Integration testing becomes particularly important when the development organization is geographically dispersed, or when parts of the system have been developed with only little interaction between different development teams. System tests verify that all the integrated modules provide correct functionality, i.e., correct according to the requirements specification. System tests view the system from the user's point of view and look at the complete system as a black box which the user interacts with using some sort of user interface.

Catching defects early in the development process is an important goal for the development organization. The sooner a defect is identified the sooner it can be corrected. Software defects can be at the code level, in algorithms, but also at the design or architecture level. Defects at the design and architecture levels become more expensive to correct the later in the development cycle that they are identified, since larger parts of the design have been implemented. Hence the earlier that the testing can begin, the more likely it will be that the completed software will be correct. Several development methods exist that put emphasis on testing, e.g., agile methods.

During software development different roles focus on different types of tests, depending on the organization. It is common that the programmers create and implement the unit tests while they implement their part of the system. Later tests, such as integration and particularly system tests, are usually done by a dedicated test team that has software testing as their main task.

One way of simplifying the repetition of tests is to automate the process. This is useful for, e.g., regression, unit, and performance tests. Automated testing can be implemented as a part of a daily build system. Test cases that have been automated can be executed once a build cycle is completed. Report generation and comparison to previous test results can be created as feedback to the developers. An example of such a system is Tinderbox by the Mozilla Foundation [66]. During the daily build it is also possible to collect data regarding source code metrics, e.g., the Maintainability Index [69] can be computed.

A software system that makes the testing activity easy to perform is described as having a high testability. Testability is a quality attribute [19] of a software system. It describes how much effort that is required to verify the functionality or correctness of a system, or a part of a system. One aspect of testability is to give the software testers useful feedback when something goes wrong during execution. Testability exists at a number of levels ranging from methods/classes through subsystems and components up to the system function level. Several definitions of testability exist, e.g., [61, 39, 42].

> *"Attributes of software that bear on the effort needed to validate the software product." [61]*
>
> *"Testability is a measure of how easily a piece of hardware can be tested to insure it performs its intended function." [39]*
>
> *"The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met." [42]*

An important observation is stated in [61]:

> *"Testability is predicted on the definition employed, and there are numerous definitions. All definitions are legitimate; however, it is necessary to agree on terminology up front to avoid misunderstandings at test time."*

The statement above leads us to the first focus in this study, i.e., how to identify different definitions of testability between roles in an organization. The second focus in our study is to identify those metrics the software developers and testers believe have an impact on testability. We focus our investigation on static analysis of a software system. Dynamic analysis requires an executable system, and that is out of scope of this study. Static analysis typically collects metrics by analyzing the complexity of the source code. Looking for and counting the number of statements in and the structure of the code. There are metrics that can be collected at the method/algorithm level but there are also a number of object oriented metrics that can be collected at the design level, e.g., inheritance depth, number of methods in a class, and coupling.

## 6.3 Objectives and Methodology

Our case study was performed at a company that develops embedded control and guidance systems for automated guided vehicles. The company has worked with software development for more than 15 years, and has a development organization consisting of about 20 software engineers, 5 project managers, 3 product managers, and 2 software test engineers.

The company expressed concerns that there were misunderstandings between organizational roles regarding software testability and software testing. As part of the BESQ project we devised this study to evaluate how the people in different roles in the organization define testability. We look for disagreement between the roles as well as within the roles.

Three roles (groups) are included in this case study: software engineers, test engineers, and managers. The group of software engineers includes the programmers and designers that develop the software for the control system. The group of test engineers

includes people that perform system function testing of the completed software system. Finally, the third group consists of the project managers and product managers. Project managers manage the groups of software and test engineers, and make sure that the software development is proceeding according to schedule. The product managers represent the customers' interest in the development process, and are responsible for forwarding the requirements from the customers during the development and maintenance of the software.

Together with representatives from the company we determined two short-term objectives as well as one long-term objective. The two short-term objectives were:

1. To see how testability is defined by different roles in the organization.
2. To identify a set of software metrics that the developers indicate as useful when determining the testability of source code.
3. The third, long-term, objective will only be discussed briefly in this case study. We will not try to implement any changes in the organization yet.
4. Based on the results from objective 1, try to create a unified view of the testability concept within the organization.

To fulfil the first objective we need to determine the current state of the organization. We use a questionnaire to gather the required information, which also enables us to analyze the results relatively easy. The questionnaire contains a set of statements regarding testability, and the respondents indicate to what extent they agree or disagree with each statement.

The second objective require a number of software engineers to grade a set of source code metrics. The idea is to identify which source code metrics that the respondents believe have an impact on the testability of a system. This information is also collected through the questionnaire. For each metric, the respondent indicates whether he/she believes the metric has a positive or negative impact on testability, and how large the impact would be. This part of the questionnaire is divided into two sub parts; one that focuses on metrics that can be collected from C/C++ code, and one that focuses on metrics that only are relevant for C++ code, i.e., object oriented structural metrics [13, 20].

The third objective will be addressed through a workshop where the results of the survey will be presented to participants from the different roles. The intention is to discuss with them the different testability definitions that exist as well as the other roles' views on testability and expectations on their own role.

The questionnaire addressing the first two objectives contains 65 questions, which are divided into three categories. Category one contains questions related to the role of the respondent in the organization as well as background and previous experience. The second category contains questions regarding how the respondent view testability. This is formulated as a number of statements that the respondent agree or disagree with. The statements are graded using a five point Likert scale [77]. The scale ranges from 1 to 5, where 1 is that the respondent does not agree at all, 3 indicates that the respondent is neutral to the statement, and 5 indicates that the respondent strongly agrees with the statement. The third category contains a number of source code metrics that is used to assess the testability of C/C++ source code. These metrics are also graded using a five point Likert scale. For each question we also ask how sure the respondent is on the answer. These questions are also graded using a Likert scale.

The questionnaire was distributed as an Excel file via e-mail and the respondents sent their replies back via e-mail. Together with the Excel file we sent a short dictionary with descriptions of concepts and definitions that were used in the questionnaire. This was done in order to minimize the amount of questions that the respondents might have and to make sure that all respondents used similar definitions of basic concepts. We sent the questionnaire to all people with the roles that we wanted to examine.

## 6.4 Results and Analysis of Testability Statements

We distributed 25 questionnaires and got 14 responses, resulting in a 56% response rate. The number of responses for each of the groups is found in Table 6.1. The number of responses was to few to apply statistical methods. Instead, we rely on quantitative and qualitative reasoning based on the answers. The first part of the questionnaire (general testability statements) was answered by all

14 respondents, and the second part (C/C++ metrics) was answered only by the software engineers.

**Table 6.1 The number of replies divided per role.**

| Role | Nr of replies |
|---|---|
| **Software Engineer** | 9 |
| **Test Engineer** | 2 |
| **Project/Product Manager** | 3 |
| **Total number of replies** | 14 |

The statements in the first part of the questionnaire are listed in Table 6.2, and are all related to definitions of testability. The statements can be grouped into three groups: S1, S3, S5, S6 and S12 are related to the size and design of a software module, S2, S4, and S7 are related to the size of a software module only, and S8, S9, S10, and S11 are related to the internal state of a module.

The analysis is done in several steps. First, we compare the responses by the people within each group (role) in order to evaluate the agreement within the group. Second, we compare the answers between the groups in order to evaluate agreement or disagreement between the groups. Third, we compare the expectations the software engineers and test engineers have on each other regarding testability.

### 6.4.1 Consistencies Within the Roles

The first analysis of the responses is to look for inconsistencies within the roles. We want to see if all respondents in a role give similar answers, and if they differ, where the inconsistencies are. The answers are summarized so the frequency of each response is identified, and we also evaluate the distribution of responses. The responses for each question is grouped into four categories for further analysis. The categories are defined as:

1. **Consensus**. There is a clear consensus between the respondents.
2. **Consensus tendency**. There is disagreement but with a tendency towards one side or the other.
3. **Polarization**. There are two distinct groupings in the responses.

**Table 6.2 Statements regarding testability.**

| ID | Statement |
|---|---|
| **S1** | Modules with low coupling have high testability. |
| **S2** | Functions with few parameters have high testability. |
| **S3** | Modules with high coupling have low testability. |
| **S4** | Modules with few lines of code have high testability. |
| **S5** | Modules with few public methods in their interface have high testability. |
| **S6** | Modules with many public methods in their interface have high testability. |
| **S7** | Functions with many parameters have high testability. |
| **S8** | If it is easy to select input to a module so that all execution paths in it are executed, then it has high testability. |
| **S9** | If it is likely that defects in the code will be detected during testing then the module has high testability. |
| **S10** | If it is easy to set the internal state of a module during testing then it has high testability. Modules that can be set to a specific state makes it easier to retest situations where faults have occurred. |
| **S11** | If it is easy to see the internal state of a module then it has high testability. Modules that for example log information that is useful to the testers/developers have high testability. |
| **S12** | If a module has high cohesion, then it has high testability. |

4. **Disagreement**. There is a clear disagreement between the respondents.

Statements that end up in categories Consensus and Consensus tendency are ok, i.e., the respondents agree with each other and a tendency can be seen even though the responses might spread somewhat. The Polarization category is less good and indicates that there exists two different views on the statement among the respondents. The Disagreement category is also less good since there are several views of the statement and no real consensus among the respondents exists.

### 6.4.2 Software Engineers

The software engineers' responses can be found in Table 6.3. The responses to statements S1, S3, S5, and S10 belong to the Con-

sensus category since there are a clear consensus in the answers. Statements S2, S6, S7, S9, and S11 we put in the Consensus tendency category where we have a tendency towards agreement or disagreement with the statement. In the Polarization category we put statements S8 and S12 since there are two groupings in the answers. Finally, in the Disagreement category we put statement S4.

**Table 6.3 Summary of replies from the software engineers.**

| Answer: | 1 | 2 | 3 | 4 | 5 | Result Category |
|---|---|---|---|---|---|---|
| **S1** | 0 | 0 | 0 | 2 | 6 | Consensus |
| **S2** | 0 | 1 | 2 | 2 | 3 | Consensus tendency |
| **S3** | 0 | 0 | 1 | 3 | 4 | Consensus |
| **S4** | 0 | 2 | 2 | 1 | 3 | Disagreement |
| **S5** | 0 | 1 | 4 | 2 | 1 | Consensus |
| **S6** | 2 | 2 | 3 | 1 | 0 | Consensus tendency |
| **S7** | 3 | 2 | 2 | 1 | 0 | Consensus tendency |
| **S8** | 0 | 0 | 3 | 0 | 5 | Polarization |
| **S9** | 0 | 0 | 3 | 2 | 3 | Consensus tendency |
| **S10** | 0 | 0 | 1 | 4 | 3 | Consensus |
| **S11** | 0 | 1 | 2 | 2 | 3 | Consensus tendency |
| **S12** | 0 | 0 | 3 | 1 | 4 | Polarization |

The answers show that there are good consensus among the software engineers regarding coupling of modules, the number of public methods, the number of function parameters, and their relation to testability. There is also a high degree of consensus that modules have high testability if defects will be detected during testing, and if it easy to set and view the internal state of the module.

The software engineers have different opinions about high cohesion and the easiness of selecting input data for testing all execution paths. These issues are subjects for further investigation and discussion in the organization. Finally, there is a large disagreement if a module with few lines of code has high testability or not.

It is good that so many statements are agreed upon by the respondents. There are only two statements where the respondents form two distinct groupings, and only one statement where no clear consensus can be identified. This leads us to believe that the software engineers, as a group, have a rather coherent view on what testability is, although some disagreements exist.

### 6.4.3 Test Engineers

The responses from the test engineers can be found in Table 6.4. The number of respondents in this role is only two. Therefore, it is more difficult to place the statements in the four categories. In category Consensus we put the statements S1, S3, S5, S6, S8, S9, S10, and S12 since the respondents give similar responses to the statements. Statements S2, S4, S7, and S11 are placed in category Disagreement because the responses are further apart. The responses could not be placed in the Polarization category since they are spread between both agree and not agree to the statements. The issues where there exist disagreements are functions with few and many parameters, few lines of codes for a module, and viewing the internal state.

**Table 6.4 Summary of replies from the test engineers.**

| Answer: | 1 | 2 | 3 | 4 | 5 | Result Category |
|---|---|---|---|---|---|---|
| **S1** | 0 | 1 | 1 | 0 | 0 | Consensus |
| **S2** | 1 | 0 | 1 | 0 | 0 | Disagreement |
| **S3** | 0 | 2 | 0 | 0 | 0 | Consensus |
| **S4** | 0 | 1 | 0 | 0 | 1 | Disagreement |
| **S5** | 0 | 1 | 1 | 0 | 0 | Consensus |
| **S6** | 0 | 1 | 1 | 0 | 0 | Consensus |
| **S7** | 0 | 1 | 0 | 1 | 0 | Disagreement |
| **S8** | 0 | 0 | 0 | 1 | 1 | Consensus |
| **S9** | 0 | 0 | 0 | 2 | 0 | Consensus |
| **S10** | 0 | 0 | 0 | 1 | 1 | Consensus |
| **S11** | 0 | 1 | 0 | 0 | 1 | Disagreement |
| **S12** | 0 | 0 | 1 | 1 | 0 | Consensus |

### 6.4.4 Managers

The responses from the managers can be found in Table 6.5. They suffer from the same problem as the testers, i.e., we only got three responses to the questionnaire from this role which makes the categorization of the responses difficult. Statements S5 and S6 are placed in category Consensus, and statements S4 and S9 in the category Consensus tendency. The rest of the statements are placed in the Disagreement category since the respondents disagree on the grading of the statements. When one of the respondents agree then two disagree and vice versa. This group of the respondents is the one that has the most disagreement in their responses.

**Table 6.5 Summary of replies from the managers.**

| Answer: | 1 | 2 | 3 | 4 | 5 | Result Category |
|---|---|---|---|---|---|---|
| **S1** | 0 | 1 | 0 | 1 | 1 | Disagreement |
| **S2** | 0 | 2 | 0 | 1 | 0 | Disagreement |
| **S3** | 0 | 1 | 0 | 2 | 0 | Disagreement |
| **S4** | 0 | 1 | 1 | 1 | 0 | Consensus tendency |
| **S5** | 0 | 1 | 2 | 0 | 0 | Consensus |
| **S6** | 0 | 0 | 1 | 2 | 0 | Consensus |
| **S7** | 0 | 2 | 0 | 1 | 0 | Disagreement |
| **S8** | 0 | 1 | 0 | 1 | 1 | Disagreement |
| **S9** | 0 | 1 | 1 | 1 | 0 | Consensus tendency |
| **S10** | 0 | 1 | 0 | 2 | 0 | Disagreement |
| **S11** | 0 | 1 | 0 | 2 | 0 | Disagreement |
| **S12** | 0 | 1 | 0 | 2 | 0 | Disagreement |

From the managers viewpoint we see that the amount of public methods has no impact if a model has high testability or not (S5 and S6). They also believe that defects detected in code and modules with few lines of code indicate high testability. For all other statements there are differences in opinions, and this has to be addressed.

### 6.4.5 Differences Between Roles

The next analysis step is to compare the view on testability between the different groups (roles). In order to make the groups easier to compare, we aggregate the responses for the groups. The results of the aggregation can be found in Table 6.6. The numbers are translated into their literal meaning, i.e., 5 - Strongly agree, 4 - Agree, 3 - Neutral, 2 - Do not agree, and 1 - Strongly disagree.

**Table 6.6 Aggregation of replies for the groups.**

| | Software engineers | Test engineers | Managers |
|---|---|---|---|
| **S1** | Strongly agree | Do not agree | Neutral |
| **S2** | Strongly agree | Do not agree | Neutral |
| **S3** | Strongly agree | Do not agree | Neutral |
| **S4** | Agree | Neutral | Neutral |
| **S5** | Neutral | Neutral | Neutral |
| **S6** | Neutral | Neutral | Neutral |
| **S7** | Do not agree | Do not agree | Neutral |
| **S8** | Agree | Strongly agree | Neutral |
| **S9** | Agree | Agree | Neutral |
| **S10** | Agree | Strongly agree | Neutral |
| **S11** | Agree | Agree | Neutral |
| **S12** | Strongly agree | Agree | Neutral |

From the aggregation we find that the software engineers and the test engineers do not agree on three main statements (S1, S2, and S3). These are statements that are related to the design and size of software, i.e., coupling-cohesion vs. high-low testability and few function parameters. For the remainder of the statements there is mainly an agreement between the two roles.

Most of the managers' answers are neutral in the aggregation. The reason is that the responses from the managers had a large spread. One respondent answered similarly to the software engineers and another answered almost the direct opposite, and both were very sure on their answers. The differences in opinion can maybe be attributed to the backgrounds of the managers. One of them had never worked with testing, while the other one had long

test experience. The differences also make it hard to make statements about the expectations on the other roles from the managers since they inside their group have different opinions, making the relations to the other groups hard to interpret. As mentioned earlier, this difference in opinions must be unified.

### 6.4.6 Understanding and Expectations Between Software Engineers and Test Engineers

We made follow-up interviews which focused on how the respondents perceive the expectations of the other roles that participated. The additional questions regarding this aspect complement and further focus the results of the study. The follow-up interviews were done over telephone. We did not include the managers since they as a group had a to scattered view on the statements. Hence, we discuss the expectations of the software engineers on the test engineers in Table 6.7, and vice versa in Table 6.8. For each statement the respondent answers what he/she thinks that the people in other role would answer. This give an indication of how much the roles are aware of each others opinion of testability.

**Table 6.7 Software engineers' expected answers from the test engineers.**

| | Expected answers | Actual answer |
|---|---|---|
| **S1** | Strongly agree | Do not agree |
| **S2** | Agree | Do not agree |
| **S3** | Strongly agree | Do not agree |
| **S4** | Neutral | Neutral |
| **S5** | Neutral | Neutral |
| **S6** | Neutral | Neutral |
| **S7** | Do not agree | Do not agree |
| **S8** | Strongly agree | Strongly agree |
| **S9** | Strongly agree | Strongly agree |
| **S10** | Strongly agree | Strongly agree |
| **S11** | Strongly agree | Agree |
| **S12** | Agree | Agree |

Overall we conclude that the software engineers and the test engineers seem to have a good understanding of each others interpretation of testability. The answers only differ on three statements, S1, S2, and S3, where both roles predicted different answers from the other role than they actually gave. From the interviews we think that the difference can be attributed to different interpretations of the concepts of high and low coupling in object oriented design.

**Table 6.8 Test engineers' expected answers from the software engineers.**

| | Expected answers | Actual answer |
|---|---|---|
| **S1** | Strongly agree | Agree |
| **S2** | Neutral | Agree |
| **S3** | Do not agree | Strongly agree |
| **S4** | Strongly agree | Agree |
| **S5** | Neutral | Neutral |
| **S6** | Do not agree | Do not agree |
| **S7** | Do not agree | Do not agree |
| **S8** | Strongly agree | Agree |
| **S9** | Strongly agree | Agree |
| **S10** | Strongly agree | Agree |
| **S11** | Strongly agree | Agree |
| **S12** | Strongly agree | Strongly agree |

## 6.5 Selection of Testability Metrics

The second objective of our study is to identify and select a number of software metrics that can be collected and used to assess the testability of a system under development. The data for this selection are collected through the second part of the questionnaire. The statements and questions from this part of the questionnaire is presented in Table 6.9. Statements M1 to M9 are general questions related to code size. Statements M10 to M23 are questions related to C and C++, since those are the major programming languages used at the company. Finally, statements M24 to M38 are related only to C++. The C++ related metrics cover aspects such as class structures and inheritance while the C/C++ metrics focus on metrics related to code structure, meth-

ods, and statements. The participants in the study answered both in what way a statement impacts testability (improve or deteriorate) as well as how much it impacts testability relative to the other metrics.

We got 5 responses where all questions were answered, and 4 responses where only questions M1 to M23 were answered. The reason given by the respondents for not answering all questions was usually that they felt unsure about the C++ statements since they usually worked with the C programming language.

The results are analyzed in a similar way as in the previous sections. We aggregate the results and translate them from numbers to their literal meaning in order to make the results easier to compare as well as more readable. The mapping for the impact is done as follows: 1 - Very small, 2 - Small, 3 - Average, 4 - Large, 5 - Very large, and for the direction: 1 - Very negative, 2 - Negative, 3 - No impact, 4 - Positive, 5 - Very positive. We focus our analysis on the metrics that the respondents identify as the ones with large positive or negative impact on the testability of a system. The results of the aggregation is presented in Table 6.10 (large negative impact) and Table 6.11 (large positive impact).

Of the metrics with negative impact, see Table 6.10, we find that the developers focus on memory management aspects of the source code as well as the structural aspects. This is interesting as there is usually little focus on what the source code actually does and more on how understandable code is through its structure and complexity [69]. This indicates that structural measures need to be complemented with some measures of the occurrence of memory related operations in the source code. Traditional size measures such as the lines of code in files and methods are also present in M1, M2, and M5. Source code complexity measures such as cyclomatic complexity [64] can also be seen in M17. Finally, we find class structure and inheritance measures related to object oriented metrics [13, 20] in M26 and M27.

The metrics that are graded as having a large positive impact on the testability of a system is presented in Table 6.11. These metrics can also be divided into two groups: the first is related to object oriented structure (M24, M28, M30, M31, and M32) and the second is related to documentation, e.g., lines of comments (M3) and descriptive function names (M6).

**Table 6.9 Statements regarding how different code metrics impact testability.**

| ID | Metric Statement |
|---|---|
| **M1** | How many lines of code the module contains. |
| **M2** | How many lines of code that each function contains. |
| **M3** | How many lines of comments the module contains. |
| **M4** | How many lines of comments that each function contains. |
| **M5** | How many parameters that are passed to a function. |
| **M6** | The length of the function name. |
| **M7** | The length of the class name. |
| **M8** | The number of lines of comments that are present directly before a function declaration. |
| **M9** | The number of lines of comments present inside a function. |
| **M10** | The number of variables present in a struct. |
| **M11** | The size of macros used in a module. |
| **M12** | The number of macros used in a module. |
| **M13** | The number of functions in a module. |
| **M14** | The number of parameters of a function. |
| **M15** | The number of macros called from a module. |
| **M16** | The number of times that a struct is used in a module. |
| **M17** | The number of control statements (case, if, then, etc.) in the module. |
| **M18** | The number of method calls that is generated by a method. |
| **M19** | The number of assignments that are made in a module. |
| **M20** | The number of arithmetic operations in a module. |
| **M21** | The presence of pointer arithmetic. |
| **M22** | The number of dynamic allocations on the heap. |
| **M23** | The number of dynamic allocations on the stack. |
| **M24** | The total number of classes in the system. |
| **M25** | The number of classes in a module (name space?). |
| **M26** | The number of interfaces that a class implements. |
| **M27** | The number of classes that a class inherits. |
| **M28** | The number of classes that a class uses. |
| **M29** | The number of methods present in a class. |
| **M30** | The number of private methods present in a class. |
| **M31** | The number of public methods present in a class. |
| **M32** | The number of private variables in a class. |
| **M33** | The number of public variables in a class. |
| **M34** | The number of overloaded methods in a class. |
| **M35** | The number of calls to methods inherited from a superclass. |
| **M36** | The size of templates used in a module. |
| **M37** | The size of templates used in a module. |
| **M38** | The number of different template instantiations. |

**Table 6.10 Metrics with large negative impact on testability.**

| ID | Metric | Impact | Direction |
|---|---|---|---|
| **M1** | How many lines of code the module contains. | Large | Negative |
| **M2** | How many lines of code that each function contains. | Large | Negative |
| **M5** | How many parameters that are passed to a function. | Large | Negative |
| **M17** | The number of control statements (case, if, then, etc.) in the module. | Large | Negative |
| **M21** | The presence of pointer arithmetic. | Large | Negative |
| **M22** | The number of dynamic allocations on the heap. | Large | Negative |
| **M23** | The number of dynamic allocations on the stack. | Large | Negative |
| **M26** | The number of interfaces that a class implements. | Large | Negative |
| **M27** | The number of classes that a class inherits. | Large | Negative |

**Table 6.11 Metrics with large positive impact on testability.**

| ID | Metric | Impact | Direction |
|---|---|---|---|
| **M3** | How many lines of comments the module contains. | Large | Positive |
| **M6** | The length of the function name. | Large | Positive |
| **M24** | The total number of classes in the system. | Large | Positive |
| **M28** | The number of classes that a class uses. | Large | Positive |
| **M30** | The number of private methods present in a class. | Large | Positive |
| **M31** | The number of public methods present in a class. | Large | Positive |
| **M32** | The number of private variables in a class. | Large | Positive |

Finally, we find three metrics that are graded as having a large impact but rated as neither positive nor negative, see Table 6.12. We interpret these results as an indecisiveness from the respondents, that the metric should have an impact on testability but that there is no feeling for towards which direction the impact is.

**Table 6.12 Metrics that have an unclear impact on testability.**

| ID | Metric | Impact | Direction |
|---|---|---|---|
| M13 | The number of functions in a module. | Large | No impact |
| M25 | The number of classes in a module. | Large | No impact |
| M33 | The number of public variables in a class. | Large | No impact |

From the tables we can see that the common size and structure metrics are identified as properties that impact the testability of a system. But we also find properties related to memory management and addressing (M22, M23, and M21). It is common to focus the testability discussion on the understandability of the source code and not so much on what the source code actually does [36, 64, 69]. Our results indicate that memory management metrics should be taken into account when assessing the testability of software developed by this organization. We believe that this is data that relatively easy can be gathered from the source code.

Because of the developers' interest in memory related operations, we think that it would be interesting to complement existing testability measures, such as the testability index [36], with these metrics to see if the accuracy of the measure is improved. When creating such a measure it would also be possible to tune it to the developing organization and the systems they develop.

## 6.6 Discussion

In this section we shortly discuss some validity aspects in our study. The first issue concerns the design of the questionnaire. It is always difficult to know whether the right questions have been posed. We believe that we at least have identified some important issues considering the overall view on testability by different

roles in the company. The questionnaire was discussed with company representatives before it was issued to the software developers. In future studies, the questionnaire can be further refined in order to discern finer differences in peoples' opinion on testability. More statements would perhaps give a better result.

The second issue concerns the number of replies from the questionnaire, i.e., we only got 14 replies in total. The low number of replies prohibit us from using statistical methods for analysis. Instead, we have relied on qualitative reasoning and interviews to strengthen the confidence in the results. Further, the low number of respondents makes it difficult to generalize from the results. An interesting continuation would be to do a replicated study in another company.

The third validity issue is also related to the number of replies. There is a clear imbalance in the distribution of replies, most of the respondents were from one role, i.e., the software engineers. This results in that there is higher confidence in the results from one group than from the other groups. However, the difference in the number of responses per group reflects the distribution of people working in the different roles at the company.

## 6.7 Conclusions

Software testing is a major activity during software development, constituting a significant portion of both the development time and project budget. Therefore, it is important that different people in the software development organization, e.g., software engineers and software testers, share similar definitions of concepts such as testability in order to avoid misunderstandings.

In this chapter we present a case study of the view on testability in a software development company. We base our study on a questionnaire distributed to and answered by three groups of people: software engineers, test engineers, and managers. The questionnaire was then followed up by interviews.

Our results indicate that there is, in general, a large consensus on what testability means within each group. Comparing the view on testability by different groups, we find that the software engineers and the test engineers mostly have the same view. However, their respective views differ on how much the coupling

between modules and the number of parameters to a module impact the testability.

We also evaluate the expected impact of different code metrics on testability. Our findings indicate that the developers think that traditional metrics such as lines of code and lines of comments as well as object oriented structure and source code complexity are important. But we also found that the developers think that the presence of memory operations, e.g., memory allocation, has a negative impact on the testability of software modules. We think that common weighted metrics such as testability index can be complemented by the collection and integration of this information, depending on the type of domain that the organization is operating in.

Future work include organizing a workshop at the company where both software developers and software testers participate. The goal of the workshop will be to enhance the awareness of the different views on testability, based on the results presented in this chapter, and also to reach some agreement on testability within the organization.

## Acknowledgments

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project “Blekinge - Engineering Software Qualities (BESQ)” http://www.bth.se/besq. We would like to thank Danaher Motion Särö AB for their time answering our questions, as well as many interesting discussions and ideas.

# References


[1] Alan, A., Pritsker, B.: *"Principles of Simulation Modeling", in Handbook of Simulation*. ISBN: 0-471-13403-1, Wiley, 1998.

[2] Allen, R., Garlan, D.: "A Case Study in Architectural Modelling: The AEGIS System," In *proc. 8th International Conference on Software Specification and Design*, pp. 6-15, 1996.

[3] Aquilani, F., Balsamo, S., and Inverardi, P.: "Performance Analysis at the Software Architectural Design Level," *Performance Evaluation*, vol. 45, pp. 147-178, 2001.

[4] Avritzer, A. and Weyuker E. J.: "Metrics to Assess the Likelihood of Project Success Based on Architecture Reviews," *Empirical Software Engineering*, vol. 4(3), pp. 199-215, 1999.

[5] Babar, M. A. , Zhu, L., Jeffery, R.: "A Framework for Classifying and Comparing Software Architecture Evaluation Methods," In *proc. Australian Software Engineering Conference*, pp. 309-318, 2004.

[6] Barber, K. S., Graser, T., and Holt, J.: "Enabling iterative software architecture derivation using early non-functional property evaluation," In *proc. 17th IEEE International Conference on Automated Software Engineering*, pp. 23-27, 2002.

[7] Balci, O.: "Principles and Techniques of Simulation Validation, Verification, and Testing," In *Proc. 1995 Winter Simulation Conference*, pp. 147-154, 1995.

[8] Balsamo, S., Di Marco, A., Inverardi, P., and Simeoni, M: "Model-based Performance Prediction in Software Development: A Survey," *IEEE Transactions on Software Engineering*, vol. 30(5), pp. 295-310, 2004.

[9] Banks, J., Carson II J.S.: "Introduction to Discrete-Event Simulation," *Proc. 1986 Winter Simulation Conference*, pp. 17-23, , 1986.

[10] Banks, J.: Principles of Simulation, In *Handbook of Simulation*. ISBN: 0-471-13403-1, Wiley, 1998.

[11] Bardram, J. E., Christensen, H. B., and Hansen, K. M.: "Architectural Prototyping: An Approach for Grounding Architectural Design and Learning," In *Proc. 4th Working IEEE/IFIP Conference on Software Architecture*, pp. 15-24, 2004.

[12] Bass, L., Clements, P. and Kazman, R.: *Software Architecture in Practice*. ISBN: 0-631-21304-X, Addison-Wesley, 2003.

[13] Baudry, B., Le Traon, Y., and Sunyé, G.: "Testability Analysis of a UML Class Diagram," In *proc. of the 8th IEEE Symposium on Software Metrics*, pp. 56-63, 2002.

[14] Bennett, B.; Satterthwaite, C.P.: "A Maintainability Measure of Embedded Software," In *proc. of the IEEE 1993 National Aerospace and Electronics Conference*, pp. 560-565, 1993.

[15] Baumer, D., Bischofberger, W., Lichter, H., and Zullighoven, H.: "User Interface Prototyping-Concepts, Tools, and Experience," In *Proc. 18th International Conference on Software Engineering*, pp. 532-541, 1996.

[16] Beck, K.: *Extreme Programming Explained*. ISBN: 0-201-61641-6, Addison-Wesley, 2000.

[17] Bengtsson, PO.: *Architecture-Level Modifiability Analysis*. ISBN: 91-7295-007-2, Blekinge Institute of Technology, Dissertation Series No 2002-2, 2002.

[18] Bengtsson, PO., Lassing, N., Bosch, J.: "Architecture Level Modifyability Aalysis (ALMA)," *Journal of Systems and Software*, vol. 69, pp. 129-147, 2004.

[19] Bosch, J.: *Design & Use of Software Architectures – Adopting and evolving a product-line approach*. ISBN: 0-201-67494-7, Pearson Education, 2000.

[20] Bruntink, M. and can Deursen, A.: "Predicting Class Testability using Object-Oriented Metrics," In *proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 136-145, 2004.

[21] Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., and Stal, M.: *Pattern-Oriented Software Architecture - A System of Patterns*. ISBN: 0-471-95869-7, Wiley, 1996.

[22] Castaldi, M., Inverardi, P., and Afsharian, S.: "A Case Study in Performance, Modifiability and Extensibility Analysis of a Telecommunication System Software Architecture," In *proc. 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pp. 281-290, 2002.

[23] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J. and Little, R.: *Documenting Software Architectures: Views and Beyond*. ISBN: 0201703726, Addison-Wesley, 2002.

[24] Davis, D. A.: "Modeling AGV Systems," In *Proc. 1986 Winter Simulation Conference*, pp. 568-573, 1986.

[25] Dobrica, L. and Niemela, E.: "A Survey On Architecture Analysis Methods," *IEEE Transactions on Software Engineering*, vol. 28(7), pp. 638-653, 2002.

[26] Eikelmann, N. S. and Richardson, D. J.: "An Evaluation of Software Test Environment Architectures," In *proc. 18th International Conference on Software Engineering*, pp. 353-364, 1996.

[27] Engineering Village 2 (n.d.). *Compendex & Inspec*. Retrieved February 1st, 2006.
**Web site**: www.engineeringvillage2.org

[28] Etxeberria, L., Sagardui, G.: Product-Line Architecture: New Issues for Evaluation, In *Lecture Notes in Computer Science, Volume 3714*. ISBN: 3-540-28936-4, Springer-Verlag GmbH, 2005.

[29] Franks, G., Hubbard, A., Majumdar, S., Petriu, D., Rolia, J., and Woodside C.M.: "A Toolset for Performance Engineering and Software Design of Client-Server Systems," In *Performance Evaluation*, pp. 117-135, 1995.

[30] Gamma, E., Helm, R., Johnson, R., and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*. ISBN: 0-201-63361-2, Addison-Wesley, 1994.

[31] Gannod, G., Lutz, R.: "An Approach to Architectural Analysis of Productlines," In *Proc. 22nd International Conf. on Software Engineering*, pp. 548-557, 2000.

[32] Garlan, D., Monroe, R., Wile, D.: "Acme: An Architecture Description Interchange Language," In *Proc. the 7th annual IBM Center for Advanced Studies Conference*, pp. 169-183, 1997.

[33] Garlan, D.: "Software Architecture: A Roadmap," In *proc. Conference on The Future of Software Engineering*, pp. 91-101, 2000.

[34] Grahn, H., Bosch, J.: "Some Initial Performance Characteristics of Three Architectural Styles," *proc. First International Workshop on Software and Performance*, pp. 197-198, Santa Fe, New Mexico, 1998.

[35] Gunther, N.: *The Practical Performance Analyst*. ISBN: 0-07-912946-3, McGraw-Hill, 1998.

[36] Gupta, V., Aggarwal, K. K., and Singh, Y.: "A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability," *Journal of Computer Science*, vol. 1, pp. 276-282, 2005.

[37] Häggander, D., Bengtsson, PO, Bosch, J., Lundberg, L.: "Maintainability myth causes performance problems in parallel applications," *proc. 3rd Annual IASTED International Conference Software Engineering and Applications*, pp. 288-294, Scottsdale, USA, 1999.

[38] Häggander, D., Lundberg, L., and Matton, J.: "Quality Attribute Conflicts - Experiences from a Large Telecommunication Application," In *proc. 7th IEEE International Conference of Engineering of Complex Computer Systems*, pp. 96-105, 2001.

[39] Hare, M. and Sicola, S.: "Testability and Test Architectures," In *IEEE Region 5 Conference, 1988: 'Spanning the Peaks of Electrotechnology'*, pp. 161-166, 1988.

[40] Hofmeister, C., Nord, R. and Soni, D.: *Applied Software Architecture*. ISBN: 0-201-32571-3, Addison-Wesley, 2000.

[41] IEEE (n.d.). *Institute of the Electrical and Electronics Engineers*. Retrieved January 18, 2005.
**Web site**: www.ieee.org

[42] IEEE std 610.12-1990 (n.d.). *IEEE Standard Glossary of Software Engineering Terminology*. Retrieved January 19, 1990.
**Web site**: http://ieeexplore.ieee.org/

[43] ISO (n.d.). *International Organization for Standardization*. Retrieved January 18, 2005.
**Web site**: www.iso.org

[44] Kazman, R., Abowd, G., Bass, L., and Clements, P: "Scenario-based Analysis of Software Architecture," *IEEE Software*, vol. 13, pp. 47-55, 1996.

[45] Kazman, R., Barbacci, M., Klein, M., Carriere, S. J., and Woods, S. G.: "Experience with Performing Architecture Tradeoff Analysis," In *proc. of the 1999 International Conference on Software Engineering*, pp. 54-63, 1999.

[46] Kazman, R. and Bass, L.: "Making Architecture Reviews Work in the Real World," *IEEE Software*, vol. 19, pp. 67-73, 2002.

[47] Kazman, R., Bass, L., Abowd, G., and Webb, M.: "SAAM: A Method for Analyzing the Properties of Software Architectures," In *proc. 16th International Conference of Software Engineering*, pp. 81-90, 1994.

[48] Kazman, R., Bass, L., Klein, M., Lattanze, T., Northrop, L.: "A Basis for Analyzing Software Architecture Analysis Methods," *Software Quality Journal*, vol. 13(4), pp. 329-355, 2005.

[49] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, S. J.: "The Architecture Tradeoff Analysis Method," In *proc. 4th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 68-78, 1998.

[50] King, P.: *Computer and Communication Systems Performance Modelling*. ISBN: 0-13-163065-2, Prentice Hall, 1990.

[51] Klein, M., Kazman, R.: *Attribute-Based Architectural Styles*, CMU/SEI-99-TR-22, Software Engineering Institute, Carnegie Mellon University, 1999.

[52] Klingener, J. F.: "Programming Combined Discrete-Continuous Simulation Models for Performance," In *proc. 1996 Winter Simulation Conference*, pp. 883-839, 1996.

[53] Larman, C., Basili, V. R.: "Iterative and Incremental Developments. A Brief History," *Computer*, vol. 36, pp. 47-56, June 2003.

[54] Lassing, N., Bengtsson, P., Van Vliet, H., and Bosch, J.: "Experiences With ALMA: Architecture-Level Modifyabilithy Analysis," *Journal of Systems and Software*, vol. 61, pp. 47-57, 2002.

[55] Lassing, N., Rijsenbrij, D., and van Vliet, H.: "Towards a Broader View on Software Architecture Analysis of Flexibility," In *Proc. Sixth Asia Pacific Software Engineering Conference*, pp. -, 1999.

[56] Lindvall, M., Tvedt, R. T., and Costa, P.: "An empirically-based process for software architecture evaluation," *Empirical Software Engineering*, vol. 8(1), pp. 83-108, 2003.

[57] Littlefair, T.: "An Investigation Into the Use of Software Code Metrics in the Industrial Software Development Environment," In *Ph.D. thesis*, pp. , 2001.

[58] Littlefair, T. (n.d.). *CCCC*. Retrieved , 2004.
**Web site**: http://cccc.sourceforge.net/

[59] Luckham, D. C.: "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," *DIMACS Partial Order Methods Workshop IV*, pp. , Princeton University, 1996.

[60] Luckham, D., John, K., Augustin, L., Vera, J., Bryan, D., and Mann, W.: "Specification and Analysis of Sytem Architecture

using RAPIDE," *IEEE Transactions on Software Engineering*, vol. 21(4), pp. 336-335, 1995.

[61] Marciniak, J: *Encyclopedia of Software Engineering 2:nd ed.*. ISBN: 0471210080, John Wiley & Sons, 2002.

[62] Maria, A.: "Introduction to Modeling and Simulation," In *proc. 1997 Winter Simulation Conference*, pp. 7-13, 1997.

[63] Mårtensson, F., Grahn, H., and Mattsson, M.: "An Approach for Performance Evaluation of Software Architectures using Prototyping," In *proc. International Conference on Software Engineering and Applications*, pp. 605-612, 2003.

[64] McCabe, T. J.: "A Complexity Measure," *IEEE Transactions on Software Engineering SE-2*, vol. , pp. 308-320, 1976.

[65] Menascé, D., Almeida, V., and Dowdy, L.: *Capacity Planning and Performance Modelling*. ISBN: 0-13-035494-5, Prentice Hall, 1994.

[66] Mozilla Foundation (n.d.). *Tinderbox*. Retrieved , 2006.
**Web site**: http://www.mozilla.org/projects/tinderbox/

[67] Mukkamalla R., Britton M., Sundaram P.: "Scenario-Based Specification and Evaluation of Architectures for Health Monitoring of Aerospace Structures," In *proc. 21st Digital Avionics Systems Conference*, pp. -, 2002.

[68] Object Management Group (n.d.). *CORBA™/IIOP™ Specification, 3.0*. Retrieved , 2004.
**Web site**: www.omg.org

[69] Pearse, T. and Oman, P.: "Maintainability Measurements on Industrial Source Code Maintenance Activities," In *Proc. International Conference on Software Maintenance*, pp. 295-303, 1995.

[70] Pérez, M., Grimán, A., and Losavio, F.: "Simulation-based Architectural Evaluation for Collaborative Systems," In *proc. 12the International Conference of the Chilean Computer Science Society*, pp. 204-213, 2000.

[71] Perry, D. E. and Wolf, A. L.: "Foundations for the Study of Software Architecture," *Software Engineering Notes*, vol. 17(4), pp. 40-52, 1992.

[72] Petriu, D., Shousha, C., and Jalnapurkar, A.: "Architecture-Based Performance Analysis Applied to a Telecommunication System," *IEEE Transactions on Software Engineering*, vol. 26(11), pp. 1049-1065, 2000.

[73] Pfleeger, S. L.: *Software Engineering: Theory and Practise, intl. ed*. ISBN: 0-13-081272-2, Prentice Hall, 1998.

[74] Ramani, S., Gokhale, S. S., and Trivedi, K. S.: "SREPT: Software Reliability Estimation and Prediction Tool," *Performance Evaluation*, vol. 39, pp. 37-60, 2000.

[75] Reusner, R., Schmidt, H.W., Poernomo, I. H.: "Reliability prediction for component-based software architectures," *Journal of Systems and Software*, vol. 66(3), pp. 241-252, 2003.

[76] Richmond, B,: *An Introduction to Systems Thinking*. ISBN: 0-9704921-1-1, High Performance Systems Inc., 2001.

[77] Robson, C.: *Real World Research, 2nd ed*. ISBN: 0-631-21304-X, Blackwell Publishing, 2002.

[78] Sargent, R.: "Validation and Verification of Simulation Models," In *proc. 1999 Winter Simulation Conference*, pp. 39-48, 1999.

[79] Schmidt, D. et al. (n.d.). *The ACE ORB*. Retrieved , 2004. **Web site**: http://www.cs.wustl.edu/~schmidt/TAO.htm

[80] Schriber, T. J., Brunner, D. T.: How Discrete-Event Simulation Software Works, In *Banks, J.*. ISBN: 0-471-13403-1, Wiley, 1998.

[81] Shannon, R. E.: "Introduction to the Art and Science of Simulation," In *proc. 1998 Winter Simulation Conference*, pp. 389-393, 1998.

[82] Shaw, M., and Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. ISBN: 0-13-182957-2, Prentice-Hall, 1996.

[83] Smith, C. and Williams, L.: *Performance Solutions*. ISBN: 0-201-72229-1, Addison-Wesley, 2002.

[84] Sommerville, I.: *Software Engineering, 6th ed*. ISBN: 0-631-21304-X, Addison-Wesley, 2001.

[85] Svahnberg, M., Mattsson, M.: "Conditions and Restrictions for Product Line Generation Migration," In *proc. 4th International Workshop on Product Family Engineering*, pp. , 2002.

[86] Thesen, A., Travis, L. E.: "Introduction to Simulation," In *proc. 1991 Winter Simulation Conference*, pp. 5-14, 1991.

[87] Vieira, M. E. R., Dias, M. S., and Richardson, D. J.: "Analyzing Software Architectures with Argus-I," In *proc. the 2000 International Conference on Software Engineering*, pp. 758-761, 2000.

[88] Voas, J. M., Miller, K.W.: "Software Testability: The New Verification," *IEEE Software*, vol. 12(3), pp. 17-28, 1995.

[89] Wang, J., He, X., and Deng, Y.: "Introducing Software Architecture Apecification and Analysis in SAM Through an Example," *Information and Software Technology*, vol. 41, pp. 451-467, 1999.

[90] Williams, L. G. and Smith, C. U.: "Performance Evaluation of Software Architectures," In *proc. 1st International Workshop on Software and Performance*, pp. 164-177, 1998.

[91] Yacoub, Y.: "Performance Analysis of Component-based Applications," In *proc. Second International Conference on Software Product Lines*, pp. 299-315, 2002.

[92] Yacoub, S. M., Ammar, H. H., and Robinson, T.: "A Methodology for Architectural-level Risk Assessment Using Dynamic Metrics," In *proc. International Symposium on Software Reliability Engineering*, pp. , 2000.

[93] Zelkowitz, M. V. and Wallace, D. R.: "Experimental Models for Validating Technology," *Computer*, vol. 31(5), pp. 23-31, 1998.

## ABSTRACT

Software architecture has been identified as an increasingly important part of software development. The software architecture helps the developer of a software system to define the internal structure of the system. Several methods for evaluating software architectures have been proposed in order to assist the developer in creating a software architecture that will have a potential to fulfil the requirements on the system. Many of the evaluation methods focus on evaluation of a single quality attribute. However, in a real system there are normally requirements on several quality aspects of the system. Therefore, an architecture evaluation method that addresses multiple quality attributes, e.g., performance, maintainability, testability, and portability, would be more beneficial. This thesis presents research towards a method for evaluation of multiple quality attributes using one software architecture evaluation method. A prototype-based evaluation method is proposed that enables evaluation of multiple quality attributes using components of a system and an approximation of its intended run-time environment. The method is applied in an industrial case study where communication components in a distributed real-time system are evaluated. The evaluation addresses performance, maintainability, and portability for three alternative components using a single set of software architecture models and a prototype framework. The prototype framework enables the evaluation of different components and component configurations in the software architecture while collecting data in an objective way. Finally, this thesis presents initial work towards incorporating evaluation of testability into the method. This is done through an investigation of how testability is interpreted by different organizational roles in a software developing organization and which measures of source code that they consider have an affect on testability.




2006:03

ISSN 1650-2140
ISBN 91-7295-082-X