NTNU
Norwegian University of
Science and Technology

An Architectural Process for Achieving Robustness

Tor-Erik Hagen

Master of Science in Informatics
Submission date: December 2007
Supervisor: Tor Stålhane, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Chapter 1

Problem description

How does the current software development industry define the software robustness problem, and how can one achieve robustness through architecture and process in a software centric solution.

Abstract

As our reliance on software has increased, robustness has become an important subject. Software that is not robust enough may lead to frustration, or loss of time or value. Software architecture forms the main structures of applications. Having focus on the quality of an applications architecture may increase the robustness of the application. This thesis tries to find a suitable architectural process for achieving robustness.

I report on the findings from ten interviews with software architects from the software industry, around the theme robustness. Interview results are used in order to form a definition of robustness which is wider than the definitions I found in literature.

The thesis's main contribution is a proposed process for designing and analyzing robust software architectures that make use of elements from existing methods. The proposed process is grounded on results from interviews, personal experience, and evaluation of existing methods in literature.

Chapter 2

Preface

Writing a master thesis on robustness from the view of an architect ended up being quite hard. You need to be abstract, but at the same time make sure the thoughts and ideas are useable. Joel Spolsky sums up the problem with getting too abstract quite nicely in his blog:

> **Architecture Astronauts**: When you go too far up, abstraction-wise, you run out of oxygen. Sometimes smart thinkers just don't know when to stop, and they create these absurd, all-encompassing, high-level pictures of the universe that are all good and fine, but don't actually mean anything at all. [69]

I want to thank my supervisor, Dr. Tor Stålhane for his suggestions and help along the way. Also I want to thank all the architects who let me interview them on robustness and its application to architecture and processes, you all gave valuable input and ideas.

Contents

List of Figures

List of Tables

Chapter 3

Introduction

3.1 Introduction

Our society relies more and more on software. If you look through a regular home you find that quite a few appliances rely on software. Your personal computer, your washing machine, your TV, your watch, your electric shaver, your phone, and your portable music player to name a few. Outside our homes we find even more things we rely on, like trains, planes, the Internet, phone networks, ticket machines, and cars. We basically rely on software for large parts of our day.

When software fails we often have no alternatives. There are no ways you can continue to drive your car if the software crashes because of some unexpected input. But you could always walk, or catch a bus. So there is some redundancy in transport. You will not be able to be as flexible by taking the bus, but at least you are able to move around. We can say that you still have a transportation service, but it is degraded.

If we look at the business world where the use of software solutions is quite extensive, the consequence of a failing service provided by a software solution might be severe. If a cash register is no longer able to look up price information from a central price service, one must revert to other ways of determining the pricing. It can be performed using paper based lists or calling someone that knows the price. Imagine this happening at a large shopping mall. Similar problems apply to quite a lot of scenarios; a web based travel agent or web shop is not able to function without a working software solution.

The consequences could be quite catastrophic. If for instance a web shop that has its only income through the web does not have a working web front-end; there is no income. Imagine that the system is unavailable for a week or two.

It is not difficult to see that we are highly dependent on software, and that it is an important area to focus on.

In this thesis I will first investigate how the software development industry defines robustness, and which process and architectural measures they believe may influence robustness. The background for this work is interviews with ten people working as software architects. The definition of robustness expressed by the interviewees will be compared to existing definitions in literature. I will

then propose a new definition of robustness.

Having established how the current software development industry defines robustness, I will present a selection of methods for design and analysis relevant for architecture. These methods will then be evaluated against a set of evaluation criteria based on information from interviews and personal experience. Some research directions that have relevance to robustness research will then be presented to illustrate some existing research efforts in the field.

The method evaluation indicated that none of the methods presented fulfils all of the criteria, and a suggested process that combines parts of the methods presented earlier is described and exemplified.

A literature review looking for architectural solutions or patterns that can be employed to influence robustness was started. I, however, had to stop this review due to lack of time, but some partial results from this review will be presented as a part of future work.

3.2 Research method and objectives

3.2.1 Background and project development

The background for this project is a personal interest in software architecture, after working as a software architect during the last years. Further, my supervisor Dr. Tor Stålhane has robustness as a field of research. Combining architecture and robustness seemed like a challenging but interesting area to investigate. My field of experience is in software based business applications, and I chose to investigate what relation architecture has to robustness in this kind of applications.

Based on my suggestions, Stålhane proposed the following initial project description:

- Definition of robustness, clarification of the concept and separation from reliability
- How can robustness be influenced through architecture?
- Is ATAM or similar methods useful, and if so, when?
- Patterns, start by looking at patterns for safety and robustness
- Proposal for solution, process to achieve robustness
- Testing and evaluation, own development, experiments etc.

I started by conducting a review of literature looking for a definition of robustness, and soon discovered that the term was quite often used but defined in several ways. I could not find a de facto definition, so I wanted to get information about robustness from a different source. Collecting information from someone that faces issues related to robustness was a possible source of more information, and I decided to collect information about robustness from the software development industry. Stålhane had at this point talked to three companies about their perception of robustness and suggestions on how robustness could be achieved. I chose to perform interviews after considering various options for how to collect information.

The next section will discuss options and the reason for selecting interviews as the method of choice for collecting info.

From now on research followed two parallel tracks. One track conducting interviews, while the other track continued the literature review with the following goals:

- Investigate methods for architectural design and evaluation.
- Collect definitions of robustness used in literature.
- Study research directions related to robustness.
- Find patterns or solutions that are useful for robustness from an architectural view.

This formed the final problem description as presented in Chapter 1.

As the project progressed, it became apparent that the interview process itself would provide valuable information about robustness in the software development industry. As a result, more interviews than initially planned were conducted. In order to perform the method evaluations and suggest a process proposal, information from the interviews was needed. The result of this dependency on the interview results and the fact that performing interviews and analyzing interview results required quite a lot of calendar time, was that there was little time left to conduct an experiment based on the proposed method. Available time made it necessary to leave the experiment part to further work.

As I started reviewing some pattern collections looking for relevant patterns, it became apparent that they contained little information about the pattern's relation to robustness. Information was scarce on both to what extent the patterns themselves were robust, and to what extent they could contribute to the robustness of a system. This in addition to the amount of time used on the interviews and evaluation of architectural design and analysis methods, left little time to analyze patterns. As a result of this I also had to stop the part of the literature review looking for patterns, and leave it to further work.

3.2.2 Research method

Information gathering

Cornford and Smithson [16] list the following main methods for conducting empirical research:

- Interviews
- Surveys
- Literature reviews
- Laboratory experiments
- Case studies
- Action research

Each of the methods has strengths and weaknesses, and is suitable for different types of research. A brief description of the methods based on Cornford and Smithson [16] follows:

Interviews is based on an interviewer that asks questions to an interviewee. Interviews allows interaction between the interviewer and interviewee, and can range from the totally unstructured interview that is just a conversation around a topic, to the structured interview that follows a strict set of questions. Interviews give the ability to explore topics in depth, and explore areas where little is known. Interviews can also be used as a supplement to surveys or vice versa.

Surveys are normally a fixed set of questions that is answered by a group of people. It can either be performed by participants answering a questionnaire, or as structured interviews. If interviews are used, one has to make sure that the same questions are used. This is because the power of surveys is that all respondents answer the same questions. Surveys conducted through the use of questionnaires can be an effective way of collecting information from a large group. With good sampling, results can be quite representative for a target population. For surveys to be successful, well designed questions have to be used. Surveys are best suited for simple questions, not requiring long answers.

Literature reviews looks at work that has already been performed. Sources of information are published books and articles. Most research projects conduct a literature review. The review can be used to relate the research project to existing research within the field. Some projects are completely based on literature review and contribute by providing a refined understanding of existing work within a field.

Laboratory experiments are experiments conducted under controlled conditions. Variables are modified, and results observed. A typical example of a laboratory experiment is a comparative study. In a comparative study two or more groups perform the same task in similar conditions except that one or more variables is modified for some of the groups. The purpose of the study is to investigate whether the changed variable(s) has the expected effect or not. You should be careful about generalizing results, as the experiments are often simpler than the real-world equivalent.

Case studies explores a single situation in detail. One example of a case study could be to study how a company uses a particular method. Compared to laboratory experiments the conditions are harder to control. Further, findings from a single case study are not directly generalizable. One solution could be to perform multiple case studies and compare the results.

Action research is similar to a case study, except that the researcher takes an active role in the situation under study. An example could be a researcher that participates in a development project. The researcher can use his knowledge to influence activities, and use experience with the activities to gain more knowledge. On the positive side, action research leads to deep understanding of the activities they participate in. This deep understanding could however lead to a too narrow focus and the wider view could be missing.

As already mentioned, the initial idea was to use a literature review to find an established definition for robustness. The original plan was also to investigate which existing methods that can be used to influence robustness.

Although not being able to use literature reviews for establishing the definition for robustness, literature review was still used for parts of this text. To investigate how the software development industry defined robustness and which thoughts the industry has about influencing it, several alternatives were available. Literature review was already ruled out, and conducting a laboratory experiment is not suitable. The rest of the methods however could be used. I could have conducted a case study to study a project or a company or two in detail. Doing action research by participating in a project to see how the company handled robustness in their daily development process could also be an option. Both alternatives suffer from one major drawback: Time would only allow the study of one or two companies. It is no real way to know if it is possible to generalize the results to be valid for the whole industry[16].

This left me with three options. I could perform a survey, do a set of interviews or a combination. Asking open questions in a survey is not optimal[16]. This means that quite specific questions have to be formed. At this stage, I had some definitions of robustness from literature, and information from three companies. Results from the three companies were hard to interpret. The information indicated that there is a possibility that a definition for robustness exists, but that different fields within the industry had different additions.

With little information available and indications that the definition of robustness might vary between business sectors, it would be difficult to define a set of closed questions that would provide valuable information about robustness. Conducting interviews, which are suitable for asking open questions, appeared as a better option.

Interviews can range from the completely unstructured interview that merely is a free discussion on a topic, to the structured interview following a strict interview guide[16]. My choice was to create a set of open questions that could be used to guide discussion, but at the same time letting the interviewee talk freely. I did not want to impose a set of important aspects, or process steps on the interviewee. More information on the interview process can be found in Chapter 5.

Data analysis

When collecting data it is important to determine how the data should be analyzed so it is possible to analyse the gathered data. Cornford and Smithson [16] describe two main methods for data analysis; quantitative analysis, and qualitative analysis.

Quantitative analysis requires data to be transformed into a form that is appropriate for numerical analysis. In this case a series of in depth interviews was planned, as the interview subjects were to be allowed to talk freely around the theme. Data collected from this kind of interview is not suited for quantitative analysis. Analysis of the number of interviewees mentioning different ideas could be an option. It would, however, be no way to determine if an idea was not mentioned because it was not relevant or simply because the interviewee forgot to mention it.

Qualitative analysis was a better choice based on the interview form planned. Qualitative analysis is difficult, as it is difficult to determine if the information is representative. Discovering errors in the data is also difficult [16]. Testing the data is one important aspect. Testing can be performed in various ways, by comparing with similar studies, redoing the interviews on a different population, or comparing answers from a small and a large company. Testing using a survey is also an option.

The selected method for analyzing the interview results was to transcribe all the interviews, and form a list of aspects brought up by the individual interviewees grouped by main topic. This was done at an early stage using the results from the initial three interviews conducted by Stålhane. After having conducted each individual interview, the list was updated and extended with new items. The list was also used during the following interviews to ask about items in the list that was not mentioned by the interviewee. As a result of this, each interview tested the list of items. Some interviewees had conflicting opinions, which will be discussed in Chapter 5. In the end, the list based on the initial three interviews fit the rest of the interview results well, only additions was needed to include new ideas and opinions.

It would, however, be beneficial to conduct a survey to check the proposed definition in Chapter 6 on a larger population, but this study did not allow this due to time constraint this is suggested for future research.

Chapter 4

Background

4.1 Terminology

4.1.1 Failure, error and fault

As the definition of error, fault and failure varies in literature, it is important to define them as they will be used throughout this paper.

Definition 4.1.1. Failure The inability of a system or component to perform its required functions within specified performance requirements. [10, p. 32]

Definition 4.1.2. Error The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or definition. [10, p. 32]

Definition 4.1.3. Fault An incorrect step, process or data definition in a computer program. [10, p. 32]

So how do these definitions work together? It is best illustrated by an example:

Example 4.1.1 (Sample of faulty implementation of a ComputeSquare function)**.**

```
function ComputeSquare(int a){

    int result = a * 2;

    return result;

}
```

This function should return the square of the number provided as input through the parameter `a`. The function does however contain a bug so it really returns the number provided multiplied by two. Error is the wrong value being computed, while the fault is the cause of the error. In this case the cause is that the developer has written `a * 2` instead of `a * a`. Failure is that the function ends up returning the incorrect result; it does not comply with its specification. Another way to distinguish error from failure is that a failure is when the internal error is exposed to the caller of the function.

In this paper I will distinguish between fault, error and failure where necessary. I will, however, often use the term error if it is not explicitly the cause of an error, or a failure of a system or component being described.

4.1.2 Exceptions

Programming languages like C#, Java, and C++ have support for exceptions. This is a mechanism that makes it possible to throw an error such that the error propagates up the call chain until a catch block for the exception type is found. The benefit of the method is that the execution point is automatically moved to the catch clause, aborting the executing statement where the exception occurred. If no matching catch clause is found, the exception propagates up to the top of the call stack. At this level some generic mechanism might catch the exception and report the error. As an example, Microsoft's common language runtime that executes compiled C# applications shows a dialog with the exception including the full call stack if an exception is not caught by the application code. The concept is illustrated by the following example:

Example 4.1.2 (Sample of exception handling)**.**

```
function foo(){
    try{
        bar();
    }catch(ValidationException ex){
        // error handling logic
    }
}
function bar(){
    // Some code
    MethodThatCouldThrowError();
    // Some more code
}
```

Here the method `foo` calls `bar` that in turn calls a method `MethodThatCouldThrowError` that might throw an exception. If an exception of type `ValidationException` is thrown, execution is moved to the catch clause in `foo`, or else the exception is thrown to the method that calls `foo`.

An alternative approach to exception handling is the use of *function return values*. A problem with this approach is that the return value might not be checked. This could result in failures not being discovered. Take for instance the following function:

Example 4.1.3 (Example misuse of return codes)**.**

```
public void WriteToLog(string textToWrite){
    int fp = OpenFile(GetAppPath() + "/log.txt");
    Puts(fp, textToWrite);
    CloseFile(fp);
}
```

This method would not return any error if there was a problem writing to the log file. OpenFile might not return a valid file handle, but an error code instead. This error code is not checked. If exceptions had been used instead, the exception would have been raised to the caller of `WriteToLog`.

4.1.3 Architecture, components, and patterns

Architecture

There are several definitions for software architecture. Bass et al. [7] has the following definition:

Definition 4.1.4. Software Architecture. 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. [7, p. 21]

Thus, architecture can exist at several levels of abstraction and architectures can be contained in other architectures (e.g. componentization). This does also apply to how this thesis handles architecture in relation to robustness. It is also important to mention that not everything is architecture. More precisely, the nearer the code level you get, the more details and less architecture you have. This thesis will define architecture as everything from the class level and up to the system deployment model.

Component

Another related term is *component*. Bass et al. [7] mention that in the first revision of their book, the term component were used instead of element. The reason for changing it was the fact that the component based software engineering movement had gotten the word associated with the runtime flavour of the element.

Definition 4.1.5. Component. An encapsulated part of a software system. A component has an interface that provides access to its services. Components serve as building blocks for the structure of a system. On a programming language level components may be represented as modules, classes, objects or a set of related functions. A component that does not implement all of the elements of its interface is called an abstract component. [13, p. 434]

In other words, components are pieces of software that have an interface and provide some sort of service. In this text I will use the term component where I talk about a piece of a system. This means that classes and groups of classes are also considered to be components. Basically, I will mostly use component in the same way as the definition for software architecture uses software element.

Pattern

Christopher Alexander is often quoted when the term *pattern* is to be described. One such quote from Alexander et al. [4] can be found in Gamma et al. [34] and Fowler et al. [32]:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Alexander was talking about physical architecture, but the same ideas apply to the software world as well. Consider the following statements about software design patterns:

> Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain specific designs for an entire application or subsystem. The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. [34]

> A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate. [13]

A pattern is a description of a solution to a problem in a specific context. The solution is not a copy-and-paste solution; it has to be adapted to the place it is to be applied. The solution is well-proven, but at the same time a pattern does not always solve a problem [31].

The problem and solution can be at several levels of abstraction, from a quite concrete observer pattern [34] to more generic solutions like recovery blocks [47, 52, 59]. The term pattern will be used for both high-level and low-level solutions.

A related term to pattern is *idiom*, this is defined as:

Definition 4.1.6. Idiom. Idioms are low-level patterns specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them with the features of the given language.[13]

One example of such an idiom could be using the IDisposable interface together with the `using` keyword in the C# programming language[25] for resource management.

At the other end of the scale there is a special category of patterns called *architectural styles*. Architectural styles covers larger chunks of the architecture than a typical pattern, and are inspired by the early works of Shaw and Garlan [65]. Examples of architectural styles are pipes and filters and blackboard. Multiple styles can be applied in the same solution, and can also overlap each other. In the remainder of this paper I will usually not distinguish between patterns and architectural styles, as there is no clear line that separates them.

4.2 Why is robusness important?

Applications that often fail are seldom popular among users. Work and time is lost, and users end up getting frustrated because the application does not

work as expected. The reason for failure varies. Failures may for instance be caused by incorrect use of the application, due to of inadequate specifications. End users are not the only group that get frustrated by applications not meeting their expectations. Developers may experience the same frustration for libraries, components or other systems they use in development. Consultants can also get frustrated when applications does not behave during installation or upgrade.

Lack of robust behaviour in applications is one source of failures. An application that crashes and corrupts the database because the user presses a set of buttons in the wrong order, or reports some strange error to the end user before suddenly exiting, does not exhibit robust behaviour. How should such problems be handled?

Non-formal *specifications* contains functional descriptions that describe what should happen when pressing different buttons or calling methods, how screens or interfaces should be designed and so on. The specification might also contain information about validation rules for data, and invalid operations in the user interface. Information provided by the specification will however not be complete. Describing all possible sequences of actions that should not be possible would require a huge specification. Furthermore, some interpretation will be left to the reader of the specification. For instance, a specification could state that a field should be used for entering an amount, but will most likely not explicitly state that this field should be validated as numeric. This is left to the reader to understand.

It is hard to argue why the application behaves in an undefined manner when used in a way which is not explicitly covered by the specification. A customer will most likely not agree with a contractor that argues that the application crashes when pressing the wrong button because the specification does not explicitly state that it should not crash in that scenario. Obviously, the customer wants an application built in such a way that it responds in a reasonable manner when such a situation occurs.

The input and usage of an application, component, library or class is one area where robustness is important, but being robust to the response of components, libraries and classes used is also important. Validating input to a function for its own use does not help much if the function uses a component that might return an exception the function does not handle. In order to create a robust application, the behaviour of all interaction within an application has to be taken into account. Interaction that occurs without being aware of possible errors, failures and misuse might lead to an application that is not predictable in its behaviour.

Today's society depends largely on software to accomplish several tasks. Many use computers in their daily work, and many companies cannot function without functioning software applications. Software that does not work could have major implications. When bank systems fail, customers of the bank might not be able to make deposits, pay merchandise with their credit cards or check their current balance. Another user might lose text written in a document if the word processor suddenly crashes. Even worse the word processor might not terminate, just have a silent failure that results in that the document stored to disk, when saving is corrupt. The user would believe that the document was stored, but when opening the document all its content might be lost.

Creating a robust application is not simple, and the degree of robustness

```
using System;

namespace SimpleMath{
    class Program{
        static void Main(string[] args){
            double res = 0;
            if (args[0] == "help"){
                System.Console.WriteLine("Console math, help");
                System.Console.WriteLine("help - displays this help");
                System.Console.WriteLine("div a b - divides a by b");
                System.Console.WriteLine("mult a b - multiplies a by b");
                System.Console.WriteLine("mod a b - a modulus b");
                return;
            }else if (args[0] == "div"){
                res = double.Parse(args[1]) / double.Parse(args[2]);
            }else if (args[0] == "mult"){
                res = double.Parse(args[1]) * double.Parse(args[2]);
            }else{
                res = double.Parse(args[1]) % double.Parse(args[2]);
            }
            Console.WriteLine("Result: " + res.ToString());
        }
    }
}
```

Figure 4.1: Example non-robust C# program that does basic math.

has to be weighed against other important properties. The next section will illustrate the difficulty of robustness through a small sample application.

4.2.1 A simple robust application

Getting a clear view of the difficulties involved in creating a fully robust application is hard. Often the application has a high complexity and a large number of features. To illustrate the difficulty of robustness, I will create a small console application in C# that can do some basic math. The application will support three basic math operations; division, multiplication and modulus. Input to the application is to be provided through command line parameters. The first argument should be the mathematical operation to perform, and the rest should be input to the operation. In addition, a help function should be supported to guide the user in how the application should be used. A sample usage of the program to multiply 2 by 4 would be `math mult 2 4`. The implemented program can be found in Figure 4.1.

This program is by no means robust. I.e. if you use it correctly it is robust, but as soon as you do not use it as it was intended, it crashes. Simply running the program without any arguments results in a crash. Omitting numeric values or sending non-numeric values will also result in crashes. Further, if typing the name of the mathematical operation wrong it will be computed as a modulus operation. A user can easily run `math multiply 2 4` instead of `math mult`

`2 4`. The program will accept the input and present a result, but the number is not calculated using the operation the user intended. Sending in too high numbers in a multiplication or a high number to be divided by a very small number in division would normally result in an overflow. The same would normally be the case if the user tried to divide by zero. The double data type in C# comes to the rescue in situations that would normally cause a larger value than the data type can hold. Infinite is a special value of the data type, so $1/0$ would give the value infinite instead of an overflow exception. If I used a different data type like integer or decimal these operations would result in crashes due to overflows or division by zero. To demonstrate how complex such a simple program becomes when taking robustness into consideration I have made a robust version that can be found in Appendix A.1. The improved version is more robust, but it probably still contains robustness issues. A potential issue is that it should warn the user if it was not able to accurately represent the input or result numbers. The size of the code has increased from 24 to 94 lines (219%). This increase is in no way representative for transforming non-robust applications, but illustrates that making an application robust is by no means a trivial operation.

The new version of math is less maintainable. Introducing support for a new mathematical operation requires more effort in the robust version than in the non-robust version. Effort will depend on the number of arguments the mathematical operation should support. Adding support for factorial (taking only one argument) would require more effort and add more complexity than adding plus or minus operations. In this example, other quality attributes like performance, security and availability did not change noteworthy, but due to the introduction of error messages the usability increased. This decrease of maintainability with respect to adding support for new operations is a direct consequence of the chosen design and not the increased robustness by itself. It would be possible to get a better maintainability by choosing a different design.

Although far from being a critical business application, this simple example illustrates the issues involved in creating a robust application. Such challenges will also occur in a full blown business application.

4.3 Quality

Everything around us holds some kind of quality. It can be low quality, high quality or somewhere between. It is often easier to classify the quality of a physical item. Does it break easily, does its size fall within the defined margins of error, does the paint cover the whole item, and is the colour right? General quality is seldom of much help. We often have to formulate which demands we have to quality. Based on these requirements, we are able to evaluate the quality of an item.

A measurable definition of quality is needed in order to be able to compare the quality of two different items. Thus, not only do we have to define what we mean by quality, we also have to make it comparable and testable to some extent.

The following quote from an article by Cavano and McCall [14] illustrates the challenges:

> Consider two application programs, A and B, which were given the same problem requirement, written in the same language, and implemented on the same computer. Program A runs 10 percent faster, has 5 percent fewer errors under identical testing conditions, and costs 20 percent less than program B as is similar in maintainability and documentation aspects. Which program has the higher quality?

As Cavano and McCall state, the immediate answer is program A, but can one be sure that the testing was really equal? The memory footprint might be completely different, or one of them might be designed to be run in a distributed manner. *Quality* on its own does not say much. It has to be evaluated in context.

Pressman and Ince [58] describe two types of quality based on an item's measurable characteristics:

Quality of design: Refers to the characteristics that the designers specify for an item. They include factors like tolerance levels, material quality and performance specifications.

Quality of conformance: This is the extent to which the manufacturing follows the design specification. The greater the conformance, the greater the quality.

If these definitions are applied to software, the requirements, specifications and the design of the system make up the quality of design, while the implementation is the main contributing factor to the quality of conformance.

Multiple factors contribute to different forms of quality. Thus design, implementation and deployment are all important with regard to quality. A later section will focus on how architecture is important to quality.

Pressman's two types of quality were described earlier, but quality can be viewed in different ways leading to different views and definitions. For instance, a user will look at how often the system fails or does not behave as the user would expect. A developer may look at the number of defects that occurs in production normalized by the size of the system. This difference in the perceived quality comes from different views of quality. It can however be argued that in the end, it is the user's perception of the system during use that is the real measure of quality. It does not help if the system has excellent internal characteristics, if it does not function as expected by the user of the system.

Kitchenham and Pfleeger [44] describe different views of quality from a paper by Davin Garvin [35]. According to Garvin there are five different views:

- The *transcendental view* — quality is something that only can be recognized, but not described.
- The *user view* — quality is the fitness for purpose.
- The *manufacturing view* — quality is the conformance to specification.
- The *product view* — quality is tied to inherent characteristics of the product.

- The *value based view* — quality is dependent on how much a customer is willing to pay for it.

As time has passed by, software development has, according to Côté et al. [17], shifted from focusing on functionality to focus more on the user experience. The sum of improvements in ease-of-use, security, reliability and stability has resulted in a better overall user experience. This shift in focus has made quality a more important subject, and has created a need for defining more explicit quality requirements. In order for these requirements to be defined precisely, quality needs to be defined precisely as well. Quality models form the framework upon which such a definition can be created.

In the following sections, I will describe several quality models. Afterwards, I will look at how the models incorporate robustness.

4.3.1 McCall's quality model

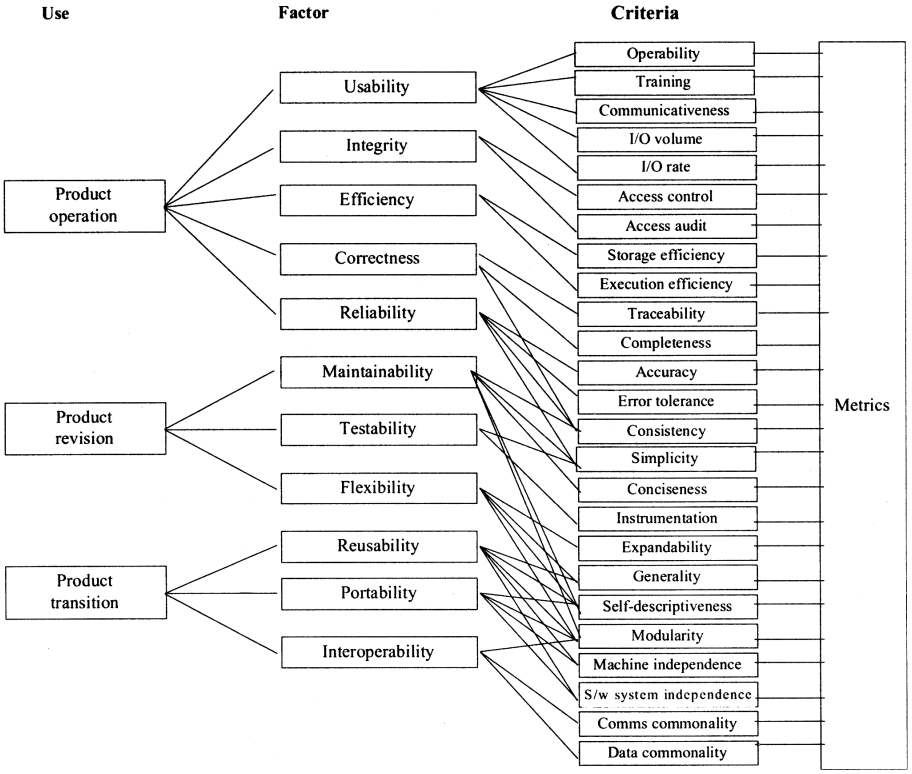

Figure 4.2: McCall's quality model [29]

McCall's quality model [14, 17, 29, 44, 58] is according to Kitchenham and Pfleeger [44] one of the earliest quality models. McCall defines software product qualities as a hierarchy of factors, criteria and metrics. The model defines quality through decomposition. The structure of the model is shown in Figure 4.2. The high-level quality factors in the model are *Correctness*, *Reliability*,

Efficiency, *Integrity*, *Usability*, *Maintainability*, *Flexibility*, *Testability*, *Portability*, *Reusability* and, *Interoperability*.

These 11 factors contribute to a complete view of quality[44], but none of them are directly measurable. To solve this, the McCall model splits each factor into a set of criteria that can be measured. If each of these criteria is fulfilled in a software solution, the factor is also present. To evaluate each criterion, metrics should be used. These metrics can vary from a checklist that can be run on a document to grade it, to the number of individual code-paths through a module.

Cavano and McCall [14] stresses the fact that the measurements are meant to be used during the development process, not as a test at the end: "Their purpose is to provide an indication of the progression toward a desired level of quality." This means that the measurements and evaluation of quality can be used to predict and control the quality of the end product.

Kitchenham and Pfleeger [44], and Côté et al. [17] note an issue with the model. As many of the metrics can only be evaluated subjectively, it is difficult to use this model to set precise and specific quality requirements. Côté et al. [17] also note that the model emphasises on the product view of quality, thus the model is less suited for other views of quality.

4.3.2 Boehm's quality model

Boehm's model [11, 17, 29] is based upon McCall's model, but does introduce some new higher level abstractions (see Figure 4.3). At the top level one finds *General Utility*. The second level consists of [11]:

- *As-is Utility* - how well can I use the software package as is?
- *Portability* - can I still use it if I change environment?
- *Maintainability* - how easy is it to maintain?

At the third level we find *Reliability*, *Efficiency*, *Human engineering*, *Testability*, *Understandability* and, *Modifiability*. Below these there is a list of primitive constructs. These make it simpler to define quantitative measures to measure progress on both the primitive and higher-level constructs. As in McCall's model, a list of metrics is used to evaluate each of the primitive constructs.

This model is built on the assumption that a system has to be useful in order to be considered a quality system [17]. The model can be viewed as having a user perspective at the top of the hierarchy and a technical perspective at the bottom. According to Côté et al. [17] this does not fit really well when looking at the definition of the characteristics. Apart from General Utility and As-is Utility, all definitions begin with "code possesses the characteristic [...]". These definitions focus on technical personnel, and the general top level abstractions are too vague to be helpful in specifying requirements.

One important aspect with the model is that the General Utility view of quality is not necessarily complete for all kinds of applications. Boehm et al. [11] mention that an application with e.g. security requirements needs additional characteristics.

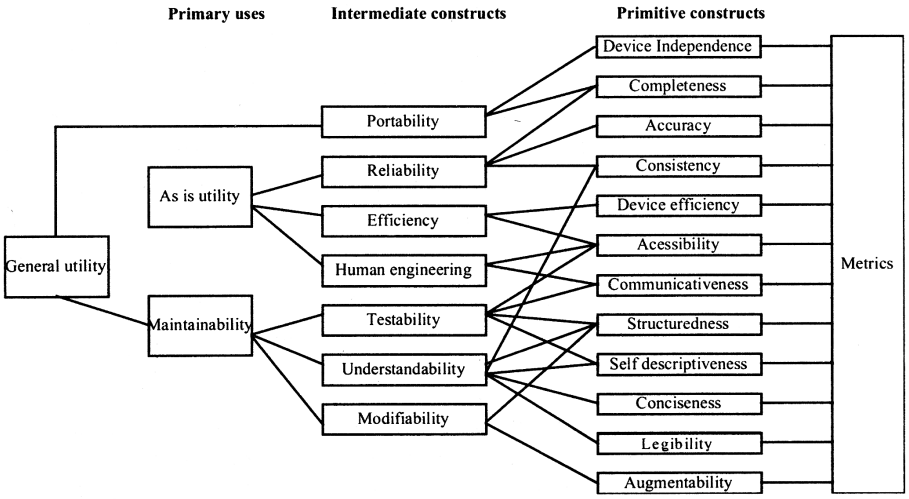

Figure 4.3: Boehm's quality model [29]

4.3.3 The ISO 9126(-*) model

The ISO 9126 [39] model came in its first version in 1999 and was according to Fenton and Pfleeger [29] based on McCall's model. It consists of the following six high-level quality factors:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

These were, according to Fenton and Pfleeger [29], said to be comprehensive. Any aspect of software quality should be possible to describe using a mixture of these six factors. The standard suggested that these six could be decomposed into multiple levels of subcharacteristics until a level that could be measured. In an annex to the standard, decomposition to a second level is described, but this is not a part of the standard. In contrast to the models McCall and Boehm, this model is strictly hierarchical, that is; each sub-characteristic refers to only one characteristic.

Côté et al. [17] refers to a set of issues with the first version of the standard described by Pfleeger [57]:

- There were no guidelines on how to provide an overall assessment of quality.
- There were no indications on how to perform the measurements of the quality characteristics.

- The model focuses on the developer view instead of the user view.

The new standard ISO 14598 focusing on software product evaluation is supposed to solve the first issue. In 2001 and 2003 a revision of the standard were published in four different parts. This revised standard tries to solve the two other issues. The four parts of the standard is now:

- ISO 9126-1 - Provides an updated quality model
- ISO 9126-2 - Provides a set of external measures
- ISO 9126-3 - Provides a set of internal measures
- ISO 9126-4 - Provides a set of quality in use measures

The standard now has three views of quality. The 9126-2 has an external quality view where metrics are suggested to evaluate the software while running. An internal quality view is described through metrics in 9126-3. Quality in use measures are provided in 9126-4 and focus on the user view of quality of the software product.

The internal and external view are inspired by McCall and Boehm and have the same division on the top level as the first version of the standard. A schematic view is shown in Figure 4.4. The views are modelled as a three level hierarchical model with quality-characteristics, quality sub-characteristics and quality measures much like McCall's model.

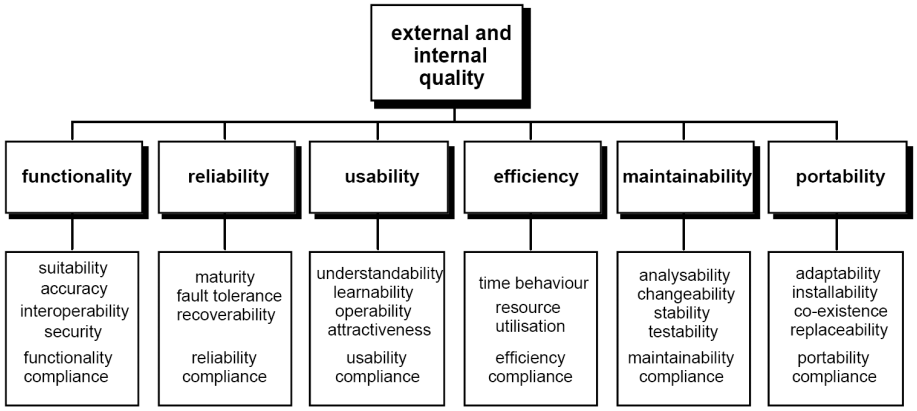

Figure 4.4: ISO 9162 view of internal and external quality [39]

The quality in use is a bit different and can be seen in Figure 4.5. It is a two level hierarchical model with quality characteristics and quality measures.

The standard also provides a theoretical relationship between the three views of quality. This relationship is illustrated in Figure 4.6.

Quality in use requirements should be established first. They should be used to specify external quality requirements that in turn should then be used to specify internal quality requirements. Based on the defined relationships between the views of quality shown in Figure 4.6, the internal quality could theoretically be used to predict external quality after development has started. However, as Côté et al. [17] note, this relationship is only theoretical and should be used with care for prediction, and needs to be empirically verified.

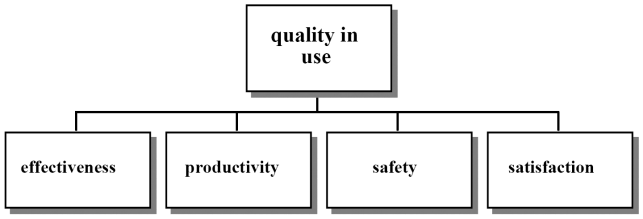

Figure 4.5: ISO 9162 view of quality in use [39]

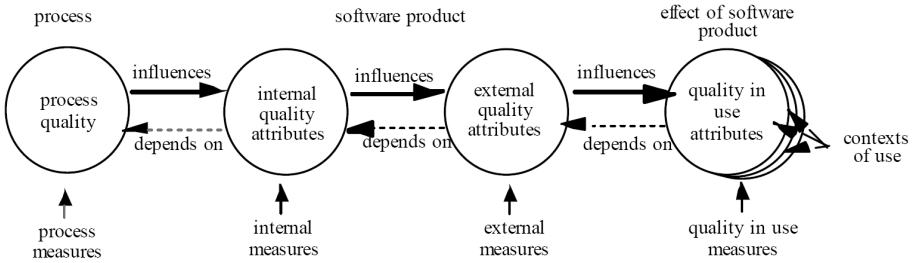

Figure 4.6: ISO 9162 relationship between the three views of quality [39]

4.3.4 Software Architecture in Practice

The SEI institute [1] frequently applies a quality model in their work, which is presented in Software Architecture in Practice [7]. Quality is here divided into the following groups:

- Availability
- Modifiability
- Performance
- Security
- Testability
- Usability

These are high level groups, but Bass et al. [7] also define quite general scenarios for each individual group so that for instance scalability fits nicely into performance, and portability fits into modifiability.

The approach used here differs from the other models mentioned. The SEI approach is to describe the different forms of quality as quality attribute scenarios. The reason behind this approach is that different communities have formed around different quality attributes, resulting in incompatible terminologies. At the same time the main groups of quality attributes are not operational, e.g. — it gives little meaning to say that a system is modifiable. All systems are modifiable for a set of modifiability scenarios, but might not be considered modifiable based on a different set of scenarios. Another issue is

[1] `http://www.sei.cmu.edu/`

that there is a large discussion going on regarding what main quality attribute a particular aspect belongs to. For instance, a system failure can be a part of availability, security or usability.

A scenario consists of: a stimulus, an environment and a response. A general availability scenario can be seen in Figure 4.7. There is a distinction between general scenarios that are applicable to any system and specific ones that are specific to a specific system. We will be looking more closely at scenarios in Section 7.3.1.

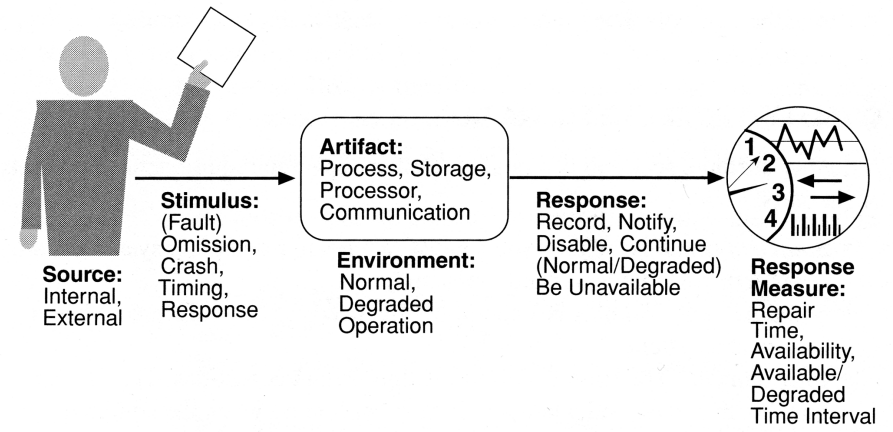

Figure 4.7: General availability scenarios [7]

Quality attribute characterizations are presented as general scenarios, which are meant to be adapted to the system in question in order to translate the quality attribute into system requirements. One example of such a requirement described as a scenario can be found in Figure 4.8. For each main type of quality attribute the general scenario gives a list of possible sources of stimuli, types of stimuli and so on that helps designing relevant testable quality requirements.

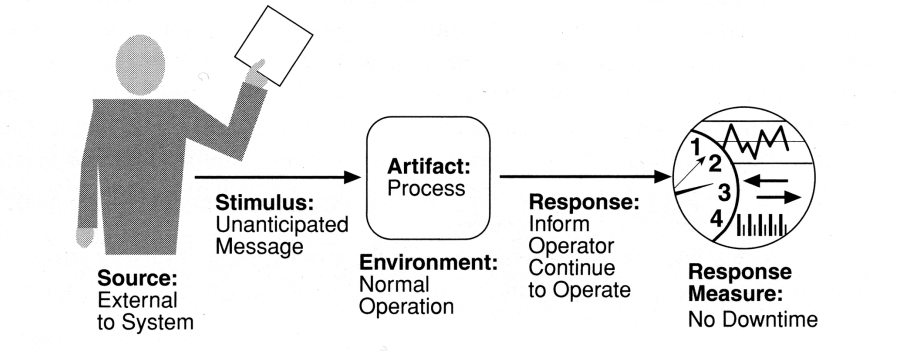

Figure 4.8: Sample availability scenario [7]

The list of quality attributes is not meant to be complete. For instance, if you have a system that is required to be highly integrateable to other systems, you should consider creating a general scenario for interoperability. Basically, the model is adaptable so that you can focus on the aspects that are important for your applications.

4.3.5 Dromey's model for product quality

Geoff Dromey [23, 24] takes a slightly different approach to building quality models. He focuses on the product characteristics of software as he sees characteristics as the key to creating a good quality model.

> What must be recognized in any attempt to build a quality model is that software does not directly manifest quality attributes. Instead it exhibits product characteristics that imply or contribute to quality attributes and other characteristics (product defects) that detract from the quality attributes of a product. Most models of software quality fail to deal with the product characteristics side of the problem adequately and they also fail to make direct links between quality attributes and the corresponding product characteristics. [23]

He is critical to the top-down approach employed in models like McCall and ISO 9126. In his opinion there is no good way of decomposing high-level quality attributes to measurable attributes through multiple levels of decomposition. He also claims that McCall and other similar models focus to much on the manufacturing view, and that more focus on the product view is needed.

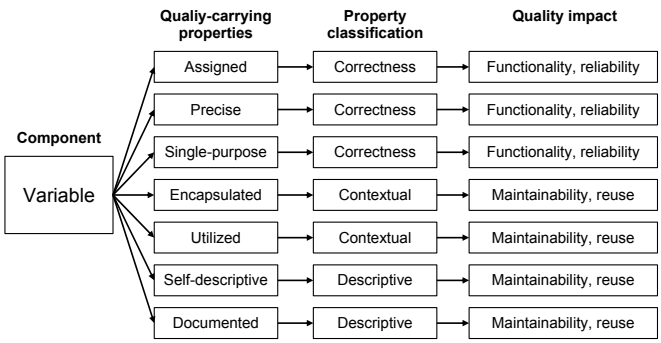

Figure 4.9: Product properties of a variable component and their effect on quality [24]

Droomey suggests building the quality models by using a single layer of linkage between a set of tangible quality carrying properties and the high-level quality attributes. To find the quality carrying properties, he views the product as a set of components, some of which are composite. The quality is mostly determined by the choice of components, the tangible properties of the components and the tangible properties associated with component composition. He suggests classifying the tangible properties into four categories: correctness, internal, contextual and descriptive. Then the link between these four

categories can be established and each individual property associated to one of these categories. An example of product properties for a variable component, and their effect on quality can be seen in Figure 4.9

By using this bottom-up approach, the idea is to be able to build a model for software quality by going from the tangible to the intangible. This approach is demonstrated by Dromey [24] where he builds parts of several quality models for implementation, design and requirements.

The model can be summed up to that composing a product of high-quality components in a high-quality manner results in a high quality product.

4.3.6 Other models

The list of quality models presented is by no means complete. A large amount of models are available, some more complete and more widely used than others. There is no de facto standard today, although some are more popular than others. According to Côté et al. [17], it seems like the ISO 9126 series is gaining some momentum in the industry, but is still not a de facto standard. The models presented give some overview of different ways of characterizing and dividing quality into attributes. There is an ongoing effort that will produce a new series of ISO standards to replace the ISO 9126 and ISO 14598 series. It is known as SQuaRE and the standards will all be in the 25000 series [70]. More information on the ISO 25000 series can be found in an article by Suryn et al. [70].

A important point when it comes to quality models is that you do not need to use a fixed model [29, p. 340]. You can choose to build your own model that either adapts an existing model to your needs, or that combines elements from various other models. One important thing to remember if you end up creating your own model, is that you will not be able to make use of empirical evidence that exists for predefined models. You have to consider whether you need your model to be compatible with one of the existing standard models, or not.

4.3.7 Quality models and robustness

None of the models described in the previous sections include robustness explicitly in the model. Looking at an early presentation of Boehm's model in the article "Quantitative evaluation of software quality" by Boehm et al. [11], robustness/integrity was included as a primitive construct at the fourth level in the model. It was linked to the intermediate construct's reliability and human engineering. In the book "Characteristics of software quality" by Boehm et al. [12] published two years later, robustness/integrity along with accountability and self-containedness has been removed from the model. I have not found any explanation to why this change was made. One possibility is that Boehm and others considered these aspects to be covered by the sum of the remaining primitive constructs.

Even though robustness is not explicitly mentioned in these quality models, it is most likely covered by the sum of the other elements included in the model. Further, the models are just theoretical representations of quality. In the end, it is the definition of quality itself that counts. ISO 9126's definition of quality in use is formulated as:

> Quality in use is the user's view of the quality of the software product when it is used in a specific environment and a specific context of use. It measures the extent to which users can achieve their goals in a particular environment, rather than measuring the properties of the software itself.
>
> NOTE 'Users' refers to any type of intended users, including both operators and maintainers, and their requirements can be different.[39]

I will get back to the definition of robustness in Chapter 6, but as illustrated in Section 4.2, robust behaviour of an application means that the application should behave in an acceptable way when an error occurs. This means that it should not crash uncontrollably, corrupt data, or fail because a user presses the wrong button, or due to some other fault. If we look at the definition for quality in use in ISO 9126, issues related to robustness would be covered. An application that is not robust, like the math application presented in Section 4.2.1 does not make it easy for the users to achieve their goals. This would be true for any application that crashes uncontrollably, or in another way does not handle errors properly. When looking at the hierarchical model for quality in use, robustness can be said to be covered by the combination of productivity, safety and satisfaction. An application that does not behave properly when errors occur, hinders productivity. It might cause harm because of e.g. lost productivity, unavailability, or corrupt data.

4.4 Quality attributes and architecture

Architecture helps design and defines the main structures of a system. Looking at buildings and other structures like bridges, it is important how the main internal structures are built in order to be able to evaluate how well they cope with wear and tear and the forces of nature. When looking at software, a solution only consisting of code without any well defined main structures could be working well, but apart from a possible lack of maintainability it is difficult to predict the other quality attributes of the solution. This is where architecture plays an important role.

> Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture. [7, p. 30]

The extent to which the system is able to meet its quality attribute requirements is not determined by the architecture alone, but the architecture represents the foundation upon which the rest of the system is built. This means that the architectural design, implementation and deployment all influence the quality attributes of the system. This also means that if one or more of them are ignored when designing a system, the level of quality might be affected. The goal is to get both the big architectural picture and the implementation details right.[7]

The architectural design is among the first activities in a development project, and in the architecture the first choices are made to how the system should be

realized. These choices also affect the quality attributes of the system. Modifiability as a quality attribute is related to the structure of the application; including both high and low level structures. An application could be realized as a multitier system where each tier might be deployed on different computers. This design forces a strict separation of functionality between the modules; possibly helping the overall maintainability. Such a design may, however, be suboptimal for performance as it introduces a communication overhead between the tiers. The overhead could, however, be kept at an acceptable level by balancing the frequency and size of the calls between the tiers. Even with such an architectural design in place, maintainability can be compromised by the duplication of functionality between tiers, and performance by doing large amounts of cross tier communication or poorly implemented algorithms in one or more of the tiers.

Architecture is not only about the high level designs like dividing functionality into application tiers and modules, but also about low level design and making functionality possible. As an example; making it possible for the end user to copy, paste, apply picture effects as layers or customize the menus is also architectural issues that influence the usability of the application. Architectural design is all about enabling the application to have the required quality attributes. If performance is important you have to make sure that the architecture does not require such a high level of inter-communication and synchronization that reaching the quality goals gets impossible. Another example is availability; if high availability is required, you have to design the architecture in such a way that it is possible to introduce enough failover, redundancy and other architectural elements to fulfil the requirement.

Bass et al. [7] sums up the relationship between architecture and quality attributes quite nicely:

- "Architecture is critical to the realization of many qualities of interest in a system, and these qualities should be designed in and be evaluated at the architectural level"[7].
- "Architecture by itself is unable to achieve qualities. It provides the foundation for achieving quality, but this foundation will be of no avail if attention is not paid in the details"[7].

Kazman et al. [40] make an even stronger statement by suggesting that the achievement of qualities like performance, availability and modifiability depends more on architecture than other factors like practices at the code level, development language or choice of algorithms. This supports the fact that making sure the architecture is able to support the quality attributes, is important. As the architecture is among the first areas for decision and design in the development of a system, it lays the foundation for the rest of the development. Making sure these choices are correct and making any changes to them as early as possible is crucial. As development progresses and builds upon the choices made, the cost and effort to make changes increases. Redoing the architecture of an application that proved to not fulfil its vital quality requirements during acceptance testing requires much more effort than redoing the architecture while it still is on paper or in its early prototyping stages.

Chapter 5

Interviews

Before the work on this thesis started, my supervisor Dr. Tor Stålhane had collected information from three companies concerning the theme robustness. During the work on this thesis I did seven interviews with other companies in order to determine how the current industry define robustness, and how they think it can be influenced.

First the interview process including areas of focus, selection of interviewees and how the interviews were conducted will be presented. Next, the answers will be summarized and threats to the validity of the results will be discussed. Last, I will present which of the areas mentioned by interviewees that will be my main focus for the rest of this thesis.

5.1 Interview design

5.1.1 Preliminary work

As already mentioned, Dr. Tor Stålhane had talked to representants from three companies. All three companies were familiar to him, as he had cooperated with them for some time. In two of the companies he talked to multiple people, and used a group process where each person wrote down suggestions on pieces of paper, followed by a group discussion. In the third company he conducted an interview with a single person. The results were structured and listed for each individual company when delivered to me.

Before the interview process started, I analyzed the results from the three companies to compare them to each other. Based on this analysis I created a document containing the preliminary results grouped by theme. When starting this process it was uncertain whether results from the three companies were consistent, or if the different companies had different opinions. A theory was that a core definition of robustness exists, but that different business sectors have additions to this core. I will get back to this in more detail in Chapter 6.

To analyze the results I first created a document based on results from one company, listing the main characterizations of robustness on one page, and listing how to achieve a better level of robustness on another page. I then took the results from the second and third company and tried to fill them into the structure. Some adjustments were needed, but they all fitted nicely into a

common structure. The same themes and ideas were represented in all three results with only small differences.

The resulting document provided a list of characteristics of robustness, and suggestions for how architecture and process may result in higher robustness.

5.1.2 Main purpose of interviews

The purpose of the interview process was to get a better insight into the current software development industry's view of robustness. During the interviews, there were two focus areas. First, information on how robustness is defined, and what characterizes a robust application. Next, how robustness can be improved during development. This covers process measures, architectural solutions, guidelines and implementation details. Architecture and process has been the main focus, and this was also important when choosing interviewees.

5.1.3 Interviewee selection

As architecture and process were the main area of focus, talking to persons that work with this on a daily basis was an important criterion when initiating the interview process. From personal experience, I know that these persons have a busy schedule. It can also be difficult to get hold of these persons in a large organization. They are normally not exposed like managers, sales representatives or business consultants are. Based on this I chose to use existing contacts and relations within the industry to get in contact with the right persons. Other strategies could have been used. Sending mail to organizations asking for a interview with the relevant persons was an option. I did not try this as I suspected that the response rate would be low. Devoting time to all enquiries like this can be difficult for an organization. According to Cornford and Smithson [16], the fact that there is no apparent gain for the interviewee also makes it less likely for someone to accept being interviewed.

Using my contacts I managed to book interviews with seven persons from five companies. All interviewees work with architecture and development processes, but from different perspectives. Some work on the business side of things, some on the technical side as senior developers and others work as software architects.

5.1.4 Confidentiality

Interviewees were informed that the interview result would be anonymized, and that neither their name nor their company name would be presented together with the results. If information provided by the interviewees would be traceable back to them and the company, it could make the interviewee afraid to provide information. This was the main reason for choosing to present results anonymously.

This level of confidentiality limits how the result of the analysis can be presented. I, however, felt that the positive effect of being able to speak freely outweighed the benefit of being able to publish details from the individual interviews. Whether the confidentiality led to better answers or not is not known. My personal opinion is that I know I would be more careful if everything I said

would be published together with my name and company name. The fear of saying something wrong would lead to brevity in my answers.

5.1.5 The interview

Interview objects were scattered around the Nordic countries and conducting a face-to-face interview with each of them was not possible during the work on this thesis. This was both due to travel cost and difficulty of finding available time. Use of telephone, text-chat and mail gave more flexibility, and was extensively used. One person was interviewed by phone, one answered questions by mail, one was interviewed in person, and the remaning four were interviewed by the help of text-chat. Interviews were conducted in Norwegian, English and Norwegian/Swedish. All non Norwegian interviews were conducted by text-chat.

All interviewees were informed about the theme of the interview, and got some time to think and make some notes before the interview. Interviews were conducted as semi structured interviews[1], lead by the interview guide presented in Section 5.2.

During the interview the preliminary document created on the basis of the initial three interviews were also used to ask more specific questions about things not explicitly mentioned by the interviewee. As new characterizations of robustness or solutions were mentioned by interviewees this document was also updated so that subsequent interviews could be used to verify the document.

Information from the interviews was collected in several ways. In the face-to-face interview notes was taken by hand, and later a transcript based on these notes were written. The phone interview was recorded, and later transcribed. All text from text-chats was copied, and the mail reply from the last interviewee was copied into a document.

5.1.6 Analysis

After an interview had been conducted and transcribed, the transcript was copied into a new document and parts without interest were removed. After the transcript had been cleaned, information from the transcript was added into the preliminary document created based on Stålhanes existing work. During the first interviews some additions and modifications to the structure of the document was necessary. The part of the document characterizing robustness stabilized and the last interviews fitted nicely into the categories. The second part of the document describing suggestions to how robustness can be influenced or achieved grew after each interview. The characterization of robustness in Section 5.4 is directly based on the first part of this document. Information on how to achieve and influence robustness from the second part of the document was restructured into the form presented in Section 5.5.

As information from the interview process should not be traceable I have not included all the transcripts. Two transcript summaries are, however, with the consent of the interviewees provided as examples in Appendix B.

[1] As defined by [61].

5.2 Interview guide

The questions presented below were only used as a rough guide, and do not focus on guiding the person being interviewed. Most of the interviewees were asked more specific questions if some aspect were not mentioned, in order to determine if the interviewee only had forgotten to mention it.

The questions were either asked in Norwegian or English depending on the person being interviewed, however, only the English version of the questions is listed below:

What is robustness?

- How do you define the term robustness?
- What characterizes a robust software application?
- How robust does a business application have to be?

How to achieve robustness?

- In your opinion, what has to be done to achieve robustness?

In which phases of development is it important to focus on robustness?

- Where in the process is it important to focus on robustness? Why?

What do you do about robustness today, and what plans do you have for the future?

- Do you focus on robustness in development today?
- Why / why not? Do you plan to do do it in the nearest future?
- What do you do today / what do you plan to do in order to focus on robustness?
- What do you plan to do in the future?

What is software architecture?

- What do you consider to be software architecture?

How important is architecture?

- What role does the architecture play in creating a robust solution as you see it?

Architectural solutions

- Is there any architectural solutions/design solutions/patterns that you think contribute to robustness? (Here both high-level as well as low-level on the boundary to code-level solution are interesting.)

5.3 Interview results

Below, interview results are summarized in two sections. The summary also includes results from Stålhane's preliminary work. For simplicity the rest of the paper will also refer to Stålhane's preliminary work as interviews, although information from two of the companies were collected through a group process. The summary is divided into two main sections. First information about what characterizes a robust solution is presented in Section 5.4, then information on how a robust application can be achieved is presented in Section 5.5. Although I have tried to divide information from interviews into two compartments, what characterises a robust application and how robustness can be achieved, some information about characterizations will indicate how robustness can be achieved, and some solutions will indicate what can be defined as robust behaviour.

5.4 What characterizes a robust solution?

Below main aspects for robustness are described based on results from the interviews. Each aspect is discussed, and information on how many interviewees that has mentioned it is provided. If any interviewees have conflicting opinions, this is also mentioned.

5.4.1 Good error handling

The interviewees all agree that the existence of good error handling is important for an application to be robust. Error handling is a large subject. To be more specific, the interviewees have mentioned the following aspects of good error handling:

- Strategy for handling errors.
- Careful handling of error situations.
- Keep errors local.
- Informative error messages.

I will handle each of these in more detail below, but first I will list some general aspects of robust behaviour with respect to error handling mentioned by the interviewees.

- The system is able to handle (all) error situations.
- The system should not terminate uncontrollably.
- The system never crashes.
- How errors are handled is important to determine if an application is robust or not.
- Errors should be handled in a way that is acceptable for the user.
- The system should be able to detect and report potential errors.

- The system should be able to handle unforeseen incidents.
- The user should be given the ability to retry a failed operation in relevant use-cases.
- An application is robust as long as it handles error situations in a reasonable way.

The statements indicate that handling errors is of importance to robustness. To what extent errors has to be handled varies between interviewees, but it seems that recognizing that an error has occurred, and reacting in a sensible way based on the situation is enough to consider a system to be robust. All errors do not have to be handled in such a way that the system continues to run, as long as it terminates in a controlled manner. This means not leaving stored data or dependencies in an inconsistent state, and informing relevant parties about the cause of the error.

Strategy for handling errors

A robust system should have a clear strategy for how errors and failures should be handled, including those who are unexpected. The effect of such a strategy is that:

- Errors should be logged.
- Effects of errors should be minimized.
- Errors should not lead to leakage of resources like database connections, open files etc.
- The system should provide clear information about the error, if an error occur.
- The system is able to monitor itself, and initiate corrective measures.
- All errors should be reported, the application should not silently fail without reporting the error.
- Unacceptable error situations should lead to a controlled termination of the system.

Careful handling of error situations

Error situations should be carefully handled, both at the user interface level and in the interface between components. More specifically interviewees mentioned:

- Data should not be lost.
- If a failure should occur, it should be easy to get back into production.
- If the database connection is lost, unstored data should not be lost. The system should provide the ability to retry operation when the database connection is back.

- The system needs to be aware of the severity of a failure, and react accordingly. For example, there is a difference between getting a SQL syntax failure back from the database and getting a failure from the database telling that the database is corrupt.

Keep local errors local

Local errors should not affect the whole system. This means that:

- The system should consider the effect on other components relying on a component before disabling it.
- An error in a module or plug-in should not harm the rest of the system.
- Errors should not result in consumption of resources like database connections, open cursors that eventually could lead to failure.
- Errors should not be allowed to spread by allowing invalid data to be stored in a database or similar data store.
- If a module or plug-in experiences an error, the error should be reported and the rest of the system should keep running unless the error is critical to the consistency of the system.

Informative errors

Error messages and logged information should be informative to the reader, and contain information so that the error can be located and corrected. More specific the following is important:

- Is should be possible for the user to understand error messages presented.
- Information in error messages should help locate the underlying problem.
- If a simplified error message is displayed to the user, technical information should either be logged or accessible through the message so that technical personnel can locate and correct the problem.
- Do not use error messages only saying ”Something went wrong”.
- Information from error messages should help getting the system back online if an error should result in the system going offline.
- All details should be kept in the error message as it is transferred between parts of the system, and should not be removed by abstractions.
- Error messages should be clear, consistent and not overly complex.

Four of the interviewees mentioned informative errors as an important aspect of a robust application, but one of the interviewees disagreed, stating that error messages are more a user interface issue than a robustness issue.

5.4.2 Clear strategy for logging

The application should have a clear strategy for logging. Logging may be used for both logging errors and keeping audit trails. Logging can be done to several sources like file, event log and database. Three of the interviewees have stated that errors have to be logged when they occur. If the application needs to terminate it should log the cause of termination before exiting so that the cause of the termination is known. It is important that information about the source of the logging also is included in the log itself. A log entry only containing "Input value out of range, reverting to default value" is of little use if the source of the logging is unknown. Initially an event could only be logged from one location, but that might change as time goes by.

It is suggested that a system for logging should be contained in the application; this would make sure that logging is done consistently, and relevant information is included in the log entry.

Further, information must be logged in such a way that the correct persons are informed about any relevant errors that have occurred within the system. To be usefull, the logged information has to be monitored by someone. This is especially important if the error that occurred did not make the system terminate. If the system terminated it would be natural to look in the log to determine the cause of the termination before starting the system. If the system continued to function after the error, no one would know that the fault occurred unless someone monitored the log or got notified about the error in some other way.

An *audit trail* has a different purpose than error and diagnostic logging. One of the interviewees has mentioned that logging of audit trails is important. Audit trails can be used to verify who made changes to data, and performed operations within the system. This can be used both to explain why data has changed, and to diagnose errors in the system by comparing logs and audit trails with the actual data stored.

Diagnostic and error logging is most relevant to server based systems, but can also be beneficial to include in client based applications. Logs can be a valuable source of information when causes of errors in data or results should be diagnosed. It might be that the user has just clicked OK on several error messages without notifying anyone about the errors that has occurred.

5.4.3 Consistent data source

Consistency of the data source is of importance to robustness. Making sure the database is consistent through the use of transactions and referential constraints in the database helps make sure that only consistent and valid information get stored in the database. Using correct data types also helps preventing invalid data from getting stored in the database.

Transactions are helpful since changes will be rolled back automatically if an error should occur in the middle of a process. Although this requires transactions to be used correctly, it is much simpler than to manually undo changes.

Combining a correct normalized data model, with the use of correct data types, referential constraints and correct use of transactions will result in fewer chances for invalid or inconsistent data to be stored in the database. This also

forces other applications writing to the database to adhere to the same rules, as they are enforced by the database.

Developing applications that can be used to verify integrity between different databases, and complex forms of integrity that cannot be enforced by constraints in the database are also helpful to ensure that the data stored in the database is correct.

Data consistency is important, as the application normally trusts information stored in the database and uses it extensively during operation. Errors in this data would lead to incorrect results.

5.4.4 Validity control

Related to consistency control at the database layer, are validity controls inside the application itself. Although only mentioned by two of the interviewees, I find this to be important. By having checks inside the system that informs when data indicates that something is wrong helps preventing errors from spreading. One example of such a check is checking for extreme amounts before charging a credit card. Checks can be realized either as error being reported or as a mechanism that only logs a warning, or make an operator manually verify that the information is correct.

Checking of data is not only important inside a system, verifying both input provided to the system as described in Section 5.4.5, and output returned from the system is important.

5.4.5 Input tolerance

Most of the interviewees (seven) have mentioned input tolerance as being important for an application to be considered robust. A robust application should handle all input variants, including invalid input that should result in the user being informed about the invalid input. An application that crashes due to invalid input is not considered robust. A special class of input tolerance, injection attacks, will be handled together with Security in Section 5.4.12.

Input tolerance is not only relevant to parts of the application communicating with a human user. Component interfaces and integration points also have to be tolerant to all possible input. Some interviewees also include communication between internal modules when talking about input tolerance, but others thinks you should trust other parts of your system. Being careful about input received also internally in the application would make it more difficult for bad data to spread, but could lead to decreased performance.

When checking for input it is essential to balance between trying to handle too much and being too restrictive. Complex logic that tries to interpret data that contain faults can lead to invalid data being accepted. For example, trying to handle various decimal separators can be dangerous as there is no way to know whether a number contains a decimal separator or a thousand separator. Handling both types of decimal separators without the use of a setting or context switch can lead to wrong interpretation of the data.

If you do not perform strict verification on the data entering a system, the data might lead to many error situations inside the system. For instance, instead of checking the length of data entering a system, the system might have to handle errors everywhere data is stored to fields with fixed length. Instead

of checking the length one place, several checks could be required. Also compensation logic might be needed to undo other changes and bring back the system to a consistent state. By verifying input, the complexity of the rest of the system can be reduced by not requiring checks and compensation logic spread around the system.

5.4.6 Good handling of load

A robust application should, according to nine of the interviewees, handle heavy load in an acceptable way. Load might be high volumes of data, large amounts of data, high number of requests or a high number of users. Opinions about how load should be handled vary. Low response time, and being able to handle higher load than specified seems to be a common ground. Some also mention explicitly that the system should be designed so it can make use of connection pooling and load balancing.

When it comes to how the system should handle situations where load exceeds the load the system is able to handle, there are several opinions. Some think the system should have a mechanism that rejects users or in another way prevents the system from getting overloaded. Others think that it is OK that the response time decreases, and having a mechanism that rejects users is not useful as the system is not able to handle the request anyway. Another negative side of having a mechanism that rejects requests is that it can be difficult to design such a mechanism as there are so many contributing factors.

Independent of the solution, it seems like most of the interviewees thinks that the most important thing is that the system does not crash or terminate and handles the overload situation gracefully. Examples of how the overload situation can be handled are:

- Increase in response time.
- Rejecting users or requests.
- Queue requests.

Some also mentions that it should be possible to monitor so that it can be determined what the cause for a decrease in response time is, and identify increasing load early so that measures can be taken to make the application handle the new load.

5.4.7 User friendly

A robust system should be user friendly. A user friendly user interface helps the user using the system correctly, and at the same time checks user input and informs the user about any invalid input before starting to process or to store the input.

- The system is simple to use.
- The system has an intuitive user interface.
- The user interface is built in such a way that it prevents wrong use of the system.

- If a user enters invalid data or in another way uses the system wrongly, the system informs the user through an understandable error message or visual clue.
- The user interface verifies user input before trying to store or to process the input.
- The application uses terms and concepts familiar to the user of the system.
- The user interface guides the user to correct use of the system.

5.4.8 Few errors

Few errors seem also to be a common characteristic of a robust application. It is mentioned by nine of the interviewees. Most of the interviewees only requires few errors to be present, while one said that the system should not contain any errors. Four interviewees were asked whether there is a difference between logical and technical errors when considering whether an application can be considered to be robust. The interviewees responded that there is not much difference between the types of errors. An application containing lots of errors would not be considered to be robust regardless of the errors types.

Another opinion mentioned by four of the interviewees, was that as long as errors are handled gracefully so that it does not severely reduce the user experience, the number of errors does not matter. This indicates that the number of errors is not a direct metric for robustness.

Thoroughly tested

One prerequisite for an application to contain few errors is that it has been thoroughly tested. There is no way to know if an application contains errors or not without testing it. Through testing it is possible to find and correct errors as well as geting an indication of the number of errors in the application. An application that undergoes extensive testing without finding any errors is less likely to have a high number of remaining errors compared to an application where testing reveals a high number of errors. Thoroughly tested or having undergone quality assurance is mentioned as a characteristic by four of the same interviewees that mentions the importance of few errors.

5.4.9 Redundant and/or uses failover

Five of the interviewees have mentioned that the existence redundancy, load balancing or failover can be an indication on a robust application. There are different opinions on to what extent it is necessary and beneficial. Some think it is overkill, other think it can be beneficial, but that third party solution should be used where possible, instead of implementing this functionality into the application itself.

5.4.10 Predictable

Predictability is mentioned as important by half of the interviewees. A predictable application should perform consistently every time it is used. This include that it should handle similar situations consistently, and have a predictable uptime (see also Section 5.4.11).

Some samples of statements from the interviewees related to predictability are:

- The application should satisfy the expected level of availability
- The system should not exhibit any suspicious or unpredictable behaviour.
- If the application is used within the predefined requirements, the application should be predictable

Predefined requirements in this context can be specified versions of the operating systems and databases, specified interfaces of components or applications, or behavioural specifications.

5.4.11 Has low downtime

Three interviewees mentioned low downtime or high availability as a characteristic of a robust application. One of the three mentioned it as especially important for server based applications. Availability can be seen as a behavioural result of other aspects mentioned by interviewees. It is difficult to get high availability if the application contains a high number of faults that leads to failures. To accomplish high levels of availability further measures like redundancy might also be necessary.

One of the three interviewees states that the system's availability should be according to the expectations put upon the application. Based on this statement, required level of availability depends on the type of application in question. Another important aspect to remember when it comes to availability is that the availability of an application or component requires that other services or components the application or component is dependent upon also is available. An application that depends of a web service to function cannot have a higher availability than the web service.

A related aspect mentioned by three interviewees is that the application should not leak resources. Leakage of resources would eventually result in no more resources being available and the application might crash or become unavailable. This is also related to error handling and load situations. Errors can result in resources not being released. The amount of resource leakage would also be higher for a time interval with high load than an interval with low load, which could lead to shorter time between failures.

5.4.12 Security

Interviewees have different opinions when it comes to the relation between security and robustness. Two interviewees think security in general is important for an application to be robust. This includes having a high level of security

in the application through access control, rights and roles. Two other interviewees mentioned access control as being important, but did not define security in general as a part of robustness.

Based on the number of interviewees and the split opinions, it is not possible to conclude whether general security is part of robustness or not. Results, however, indicate that the resistance of a security system is related to robustness. This means that a robust system needs to handle several types of attacks like intrusion attempts. Preventing legitimate users from seeing data they should not have access to, however, might not be related to robustness as long as they do not deliberately try to access the data by using vulnerability in the security system. The relation between robustness and security also heavily depend on the level of security needed for a system. For drawing applications robustness would not be related to security at all, but a online banking system would most likely define security as a part of robustness.

Seven of the interviewees have mentioned that a robust system should not be vulnerable to specific types of attack, giving a clear indication that this part of security is related to robustness. Types of vulnerabilities mentioned are:

- SQL injection.
- Cross site scripting (XSS).
- Buffer overflow.
- Denial-Of-Service attacks.

Except denial-of-service, all these relates to usage of input and data stored in a data store, which could be seen as an aspect of input handling. The notion of access control, however, is something only covered by security. Although there is some uncertainty about the relation between security and robustness, I choose to consider security as a part of robustness as it has been mentioned by several of the interviewees. Which parts of security that is relevant for robustness is, however, kept open.

5.4.13 High maintainability

Some aspect related to maintainability is mentioned by all but one of the interviewees. Maintainability of both the application itself and any database or other data store should be considered. Although some mentions maintainability in general, there are more specific aspects of maintainability mentioned:

- Faults should be simple to find and correct.
- Introduction of new functionality or change of existing functionality should not lead to errors in other parts of the system.
- The system should have a good structure.

All these aspects are related. A well arranged system should only require localized changes when functionality is to be added or changed, and a good system structure should also make it easy to find and correct a fault.

Interviewees indicate that maintainability is an important aspect for creating a robust application. System architecture with high maintainability would

lead to a system with clear interfaces and responsibility. This would make error handling easier to plan and design, it would also make designing error handling as part of component interfaces possible.

One of the interviewees wanted to make a clear distinction between robustness and maintainability, but mentioned that it should be easy to find and correct problems. This indicates that even this person defined parts of maintainability to be relevant to robustness.

5.4.14 Good documentation

Documentation is mentioned in relation to robustness by five interviewees, but the results are difficult to interpret. Two have mentioned that good documentation is an aspect of a robust system. Another one has expressed that documentation is only important with regards to robustness during the development of a system. Further, one of the two interviewees that mentioned good documentation also has mentioned user and system documentation. This is not mentioned by any other of the interviewees. Two interviewees has mentioned documentation, but not been as specific as the two explicitly mentioning it.

Information from interviews indicates that documentation is important, but it is difficult to know if end user documentation, interface documentation, architectural documentation, or system documentation is important based on information from the interviews. However, it is difficult to understand that a system without any documentation at all would be easy to use, support and maintain.

5.4.15 Other aspects

This section will list some aspects mentioned by only one or two interviewees. It is not possible to know if these aspects are only important to some, or represents viewpoints other interviewees have omitted.

Can stand the test of time

Two of the interviewees considered the robustness of the application and its architecture in a broader sense. They indicated that an architecture or application that is able to handle shifting requirements, and changes in technical aspects should be considered robust. This is related to maintainability, but is a much broader sense of maintainability than mentioned by most of the interviewees. It is important to remember that this is not a direct characterization of application behaviour at runtime, or during minor maintenance, but how well the architecture and application are able to handle larger shifts in technological trends and functional requirements. An underlying data model for an application should also be taken into consideration when considering how well the application adapts to required changes.

An application or architecture requiring large fundamental changes when minor changes in business rules are to be changed, or some new functionality is to be added, is not considered to be robust. That is, it might be robust at runtime, but is not robust to changes that should be anticipated when creating an application that will be used for a period of time.

Adapts to the environment

One of the interviewed companies mentioned adaptability to the environment as relevant behaviour of a robust system. This includes both adaptability to the hardware and adapting to environmental changes at runtime. This is mentioned in one of the preliminary interviews, making it difficult to know if this was related to a specific class of systems or meant in a specific context. Hardware adaptability might be related to maintenance, and runtime adaptability to available CPU and memory available. As this is mere speculations from me, and not mentioned explicitly by any other interviewees I will not follow this further.

5.5 How to achieve a robust system

The interviewees gave a number of suggestions for how they think robustness can be improved. The interview included questions about what they currently do, what they think can be done and what they plan to do in the future. As the interviews were conducted, it became apparent that separating what they currently do, plan to do, and suggest to do would be extremely difficult. Asking for each suggestion mentioned by interviewees whether they currently did it, wanted to do it, or just thought it would be beneficial, would require more time and could disturb the flow of the interview. As a result of this I simplified my questions by not asking specifically about what they did, or planned to do. I have thus combined current efforts, suggested, and planned efforts into a combined list of suggestions on how robustness can be achieved or improved.

5.5.1 Process measures

Several measures related to the process, were mentioned during the interviews, and they can be summarized into the following main suggestions which will be discussed separately:

- Work in teams.
- Reduce risks.
- Adhere to a process.
- Automate processes.
- Control the environment.

Work in teams

Working in teams instead of working alone is a suggestion from several of the interviewees. Teams could consist only of people with similar roles, or consist of people with different roles and backgrounds. Users might also be included in these teams. Important aspects related to working in teams mentioned by interviewees are:

- Teamwork should be performed as soon as possible in the project, it is not recommended to do too much thinking and design alone.

- Teams should consist of people with different background making them complement each other. This includes both a mix of roles, and level of experience.
- The effect of teamwork depends on the participants. If participants do not contribute to the team, teamwork does not have the same positive effect

Reduce risks

Risks should be reduced during the project, by designing and implementing the most critical and difficult parts of the system early. Iterative development is one way of accomplishing this, by selecting critical or difficult parts for the first iterations. Further, it is suggested to create prototypes and test parts of the system in the customer environment early. This should reduce the risk that some parts are not possible to implement, or will not run in the customer environment.

Adhere to a process

Development should follow a predefined process. This process should include a set of guidelines, and coding standards should be used to ensure that development and design follow a common standard. The second purpose of guidelines is to reuse well tested and known solutions instead of inventing new solutions.

A development process should focus on being simple, but still be so detailed that it can be used consistently for all development within the company or a development group. An agile development process is one example of such a small but effective process.

Further, a well defined development framework should be used, to make sure all developers use a common platform of utilities and perform the same task in a common way. Such a framework includes functions for logging, handling exceptions, database communication and so on. A framework may also contain written guidelines for how certain things should be done.

One of the interviewees said: ”Without a robust process, you will probably not be able to develop robust applications.” This indicates a strong relation between the development process and the robustness of an application.

Automate processes

It is suggested to automate certain parts of the development process, to make sure the steps are done consistently and correct every time. Candidates of such automation are processes that are repeated several times during a product's lifetime, and are easy to automate. Candidates for such automation are:

- Automate testing through the use of robot testing or unit testing.
- Automate the build process.
- Automate installations through the use of installation programs.
- Automate installation in test environments.

Control the environment

The runtime environment used during testing and production should be controlled. A stabile environment should be used, reducing the risk that something outside the application fails. Most applications will not be robust if installed on hardware that fails randomly.

5.5.2 Focus on robustness

Keeping focus on robustness during design and development may seem like an obvious and too general way of improving the robustness of an application. Interviewees have suggested ways increased focus can help. First I will look at some general ideas, and then move over to look at some more specific suggestions.

- It is too much focus on functionality. To focus more on robustness and the fact that robustness comes with a cost would be beneficial.
- Be critical about robustness already during requirement analysis.
- You need to have focus on robustness from the very beginning of a development project, it cannot be an afterthought.
- Architectural design needs to focus on robustness, it is important that the architecture has to handle error handling, logging, and resource management.
- It is important to balance between flexibility, complexity and robustness. The difficulty of creating a robust solution increases with added complexity.
- Focus on robustness during requirement analysis would be beneficial, as it would lead to better robustness requirements.

Consciousness over error conditions

Being conscious of possible errors and failures during design and development can be beneficial. It is suggested to think through possible error situations, and use this to plan and design the error handling accordingly. The consciousness should be at several levels of abstraction, from the top level design of the system and down to implementation details. Having a clear strategy on how errors and failures should be handled by the system, helps ensuring that the system is thought-through and the behaviour in erroneous situations is well known.

An example is thinking through what happens if the database server should crash, or possible errors that could occur when opening a file. In the database server case, it might be beneficial to add a failover mechanism to a replicated database server instance, or use a database cluster instead of a single database. This has to be evaluated based on the requirements for the system's availability.

Prevent error situations

Related to the consciousness of possible error conditions, is error prevention. Designers and developers should consider if there are alternative solutions or measures that can be taken to reduce the possibility for a failure to occur. It can be reduced through the number of possibilities, or the probability of each possibility occurring.

Possibilities for error prevention are highly dependent on the solution being developed. Some examples mentioned are:

- Use asynchronous communication instead of synchronous communication.
- Keep a local copy of data to prevent communication failures from causing failures in the application.
- Make sure that local errors do not spread to the rest of the system, or in another way influence the rest of the system.
- Instead of using an unreliable data store, use a reliable one and include an asynchronous mechanism that updates the unreliable data store from the reliable data store.

Defensive development is also suggested by one of the interviewees as a way of improving error situations. By being defensive when developing and designing, you try to think that everything can go wrong instead of thinking of errors as something that seldom happen. Being defensive when developing and designing might also lead to better interfaces as errors are consistently handled and exposed through the interface.

Analyze designs

Analysing the design and architecture was mentioned by one interviewee, and four others agreed that it would be beneficial when asked. None of the others were specifically asked about analysis of the architecture would be beneficial or not. All four that either mentioned it or agreed when asked, did not think that starting the testing and analysis on the architecture would be too early. One expressed that starting with testing based on the architecture could even be a bit late. This indicates that testing and analysis also may start before having a suggested architecture.

Interviewees had several ideas for how architectures and systems may be analyzed:

- Problematize the design or architecture; this is more relevant for large, complex distributed systems than small simple systems.
- Evaluate the design by, letting an objective person analyze the architecture.
- Create (proof-of-concept) prototypes.
- Design the architecture and identify problematic areas that might need further investigation or testing.

- Let a group composed of people with varying background analyze the architecture, this could for instance be done through asking a set of "What happens if ..." questions.
- Involve users in the analysis and design phases, and create prototypes to visualise solutions to the users.

The last suggestion is similar to the suggested approach I will describe in Chapter 9.

5.5.3 Thorough requirement analysis

I have already briefly mentioned that focusing on robustness when doing requirement analysis can be beneficial. It would lead to better specified robustness related requirements. Six of the interviewees have mentioned this as being important when designing and implementing a robust application. Four of them were asked if requirements or specifications today are explicit enough on robustness related issues, and none of them think enough information is provided. It seems like current requirements and specifications have too high focus on functionality and that non-functional requirements should be devoted more space in the requirement specification. One of the interviewees expressed a concern related to non-functional requirements, customers do not know what they really need and want. Customers know what functionality they want, but often lack the knowledge to express non-functional requirements.

Some negative effects of not having explicit non-functional requirements are mentioned by interviewees:

- The customer might say that the application is too slow, too unstable, or has too high downtime. If there are no specific requirements stating these expectations, you would have a challenge. You do not want a customer who is not satisfied with your delivery, and the customer might not want to pay more as he might think you should have been able to build an application that was robust, performant and available enough in the first place.
- A solution or part of a solution that is developed only based on functional requirements depends highly on the developers that realize it. Some developers develop robust and performant solutions regardless of whether it is specified or not, others just develop the functionality specified without thinking much of performance or robustness.
- Certain robustness requirements come with a cost, and these requirements will seldom be fulfilled if they are not explicitly stated. You normally do not make a more expensive or complex solution than necessary.

Two of the interviewees made a specific suggestion to how robustness could become a part of the requirements. They suggest that a chapter or section in the requirement specification should be devoted to robustness.

Robustness is not important alone as stated by one of the interviewees; a robust system is useless if it does not fulfil the rest of the customer's requirements.

In addition to explicitly including robustness requirements, there are other suggestions as well mentioned by interviewees:

- Requirements and design should be performed in iterations.
- Use use-cases.
- Make sure requirements focus on what is really important. Achieving robustness becomes more difficult as complexity grows.
- Let the customer and/or someone objective verify the requirements.
- Perform a thorough collection of requirements.
- Document the requirements thoroughly, and make sure they are all written down.
- Include specification of required runtime environments in the specification.
- Do not change requirements during development unless necessary.
- Use diagrams and figures to illustrate parts of the system and interactions with its surroundings.
- Involve users in the requirements phase.

All these suggestions are quite general, but as one of the interviewees pointed out it is important to build the system based on the correct requirements. If the requirements turn out to be wrong, or some functionality is not included in the requirements, and this is discovered in later stages of development or during testing it might be hard to fix. It might be that the underlying design is simply wrong, making it necessary to make suboptimal shortcuts or fixes in fundamental parts of the application. A system that contains a lot of shortcuts and suboptimal tweaks before it is released the first time will be harder to maintain and less likely to be robust than a system that is ”clean” in its first version. The same interviewee concretized this into the following statement: ”The better you succeed with the first version, the more robust the system will be”.

As time goes by, all systems will end up containing various fixes and shortcuts due to added functionality and maintenance. A system that has fixes and shortcuts from the start will possibly be harder to maintain, and can be less robust because functionality has to be fitted into a wrong architecture from the start.

Plan for support and maintenance

One of the interviewees mentioned an important aspect, you should plan for support and maintenance of the system. It is not only important during requirement collection, but also during design, development, and testing. When a system is in production, it is important that also support personnel are able to maintain the system so it keeps running. If this aspect does not have focus during development, it might be difficult to monitor, recover and diagnose problems. Support personnel seldom have access to debuggers and source code like developers have. A system that is only a black box can be difficult to support.

5.5.4 Plan for robustness in architecture and design

Seven of the interviewees were asked whether they thought the architecture have any influence on the robustness. All expressed that the architecture lays the foundation for robustness in the application. Without considering robustness during the architectural design, making a robust solution may become difficult. Robustness is about more than just implementation details. Numerous suggestions for how architecture and design should be used to ensure robustness and possible architectural solutions were mentioned by interviewees. I will start by listing some more general suggestions and then move over to more specific suggestions afterwards.

Some general suggestions were:

- Perform thorough design.
- Do not start the implementation before the design is complete.
- Do not make a more complex design than necessary.
- Focus on robustness during design.
- Make sure the architecture supports the required level of error handling and resource management.
- Coordinate application and database design activities.
- Make relevant parts of the architecture fault tolerant.
- Try to just have a single architecture, and not a composition of several architecture in the application.
- Make the architecture easy to use, easy to understand, thought-through, and well documented.
- Design with performance in mind.
- Try to come up with alternative solutions when the current solution seems to complex.
- Robustness alone is not enough, functionality and other non-functional requirements also need to be considered during design.
- Document the design, and make use of figures and diagrams.
- Assure that the design is correct; starting implementation on the wrong design can turn out to be costly.
- Test and analyze relevant parts of the architecture using prototypes.

I will now look at some more concrete suggestions from the interviewees.

Load handling

During the design stage one should determine how the design is able to cope with load; every design has a limit on how much load it can handle. Multiple alternatives to how load can be handled has already been discussed in Section 5.4.6.

Build a good data model

During the design stage a good data model should be designed. One of the interviewees mentioned this explicitly. In his experience systems with a normalized data model have longer lifetime. In his experience changes in a correct data model are more seldom than in the business logic of the application. This is due to changes in business rules, but the underlying concepts seldom change. Reducing the number of breaking changes in the data model is important, as a change here also requires a change in the application everywhere the data is used or stored.

Complexity and functionality vs robustness

As already mentioned, the solution added complexity makes robustness more difficult to achieve. In addition tradeoffs between robustness and other non-functional properties like performance and flexibility also need to be done. The recommendation from three of the interviewees is to balance complexity and flexibility with the need for robustness, but at the same time also evaluate the effect on other non-functional aspects like performance and maintainability. Making things unnecessary is not recommended, as it increases the difficulty in achieving robustness.

Plan and design for recovery

When designing a system, you should also consider how the system supports recovery. How is it possible to restore the system into a consistent state? If this has not been considered during design and implementation and trained on during testing, it might not be possible. If possible it could take too much time to determine how it should be done when a situation occurs in production. How critical the system is for the company also has to be taken into consideration. If it takes 2 days to perform a recovery using the current design, the design might have to be changed if the company requires no more than 4 hours downtime on the system. Although this is only mentioned by one interviewee, I have chosen to include it. This is because I think a system is not robust, if the system does not have a proper possibility for recovery if it should crash.

Plan transactions

When designing a system, transaction logic has to be thought through. When you implement a function or task, you also have to implement a mechanism that can reverse the task. The task might be a part of a larger composite task, or the task itself might fail during task execution. If this is not properly handled, you may leave the system in an inconsistent state or store incomplete data to a database.

It is possible to solve this by the means of transactions, but it can also be accomplished using compensation logic. Most important is that the whole process needs to be thought through during design. A different order of the steps involved might be needed, for instance reversing a print of a letter or sending of a fax or mail is not possible.

Plan for integrated systems

When designing a system that should be integrated with other systems, you also have to consider how your integration interfaces should look like. Adding integration interfaces as an afterthought might lead to problems. One example could be that you have implemented your system as an asynchronous service, if integration then requires synchronous communication you might struggle with implementing it within the performance requirements set. Requirements for transaction handling, rollback, and compensation functionality can also be important in an integration scenario.

The main idea mentioned by three of the interviewees is that implementing integrations is easier if systems have good integration and well thought through possibilities.

Other important aspects related to integration mentioned by one of the interviewees are:

- Document integration interfaces well.
- Try to let only one system own each kind of data.
- In an integrated chain of systems, the weakest system determines the strength of the chain.
- Try to avoid compensation logic; this adds complexity to a integration scenario.
- Try to make sure that integrated systems do not start using incomplete or inconsistent data.

Error handling

Various forms for achieving better error handling is suggested. Numerous suggestions have already been listed in Section 5.4.1. The following list presents a summarization of suggestions related to error handling mentioned by interviewees:

- Only handle the exception you can handle, do not try to handle all exceptions.
- Report errors, do not fail silently.
- Limit the effect of errors by reducing the ability for the error to spread.
- Use frameworks to achieve consistent error handling.
- Create wrappers around external systems, which handle errors and transactions against the external system.
- Identify how errors should be handled in the system during early design.
- Try to achieve uniform error handling throughout the application.

Logging

Logging seems to be an area many think is important. Support for logging should be a central part of the foundation the application is built upon. Three different ways of logging are mentioned:

- All errors should be logged when they occur.
- The system should log information making diagnosis and monitoring possible.
- Logging should be configurable.

Interviewees suggests that logging has to get more focus than it has today, and that the logged information has to be logged in such a way that it makes diagnosis possible. Just logging random information is less useful, than a log specially designed for diagnosis. Logging should be possible also in the customer environment. Is should not be used only during development and testing, but also in production.

Secure data consistency

It is suggested that an application should contain certain mechanisms for ensuring consistency and validity of data. Interviewees had the following suggestions to how this could be done:

- Use transactions when updating the database.
- Analyze the use of transactions, to make sure that they are used correctly.
- Use the correct data type in the database.
- Use the database's support for defining referential constraints.
- Implement consistency checks inside the application.

5.5.5 Architectural measures

As already mentioned, many of the interviewees said that the architecture lays the foundation for a robust application. In this section I will list some suggestions from the interviewees on functionality that can be built into the architecture in order to increase robustness.

Enforce uniform solutions

The architecture should make sure that common tasks are performed in the same way throughout the whole application. The architecture should contain base functionality that makes this possible. Examples of base functionality that should be supported by the architecture mentioned by the interviewees are:

- Read and write to the database.
- Base functionality for the application domain.

- Error handling.

By enforcing uniform solutions, the application will behave consistently. If each developer were to invent his own method for reading from the database, certain parts of the application could fail due to a change in the database configuration.

Design for self-monitoring

Interviewees has suggested that more monitoring should be built into the application, or its surrounding environment. The main idea is that the application should be proactive and warn about potential errors that occur. No suggestions for how this may be realized in the application were mentioned, but the interviewees had several suggestions for things that could be monitored. Examples were:

- Low disk space.
- High CPU utilization.
- Verification that a particular application runs.
- No log entries for an application within the last x minutes.
- High request rate.

By notifying the support personel about potential problems and errors when they occur, the support personnel might be able to fix the issue before users detect it. As long as users do not experience errors, they think the system is robust. One of the interviewees has used extensive monitoring for a while, and his experience is that it has a positive effect on problem resolvement time and user experience.

Plan and design for fault tolerance and load balancing

Few of the interviewees have experience with use of fault tolerance and load balancing. Most thought it could be beneficial, but at the same time it was considered overkill for most applications they build. These kinds of solutions were also considered expensive and complex to build into the application. It was suggested to design the application in such a way that functionality in existing products could be used to implement failover, load balancing and redundancy. Several products exist, for example load balancers, proxies and database clusters.

When designing a system that should support load balancing, redundancy, or failover there is one important design requirement. Design the service to be stateless. If this is not possible, the necessary state should be available from all nodes.

Use asyncronous patterns

Two of the interviewees suggested that asynchronous queues should be used where possible. Both as a form of communication between systems, and within a single system. Asynchronous communication is beneficial as it does not put the same requirements on response time as synchronous patterns do. At the same time it is suggested that asynchronous interfaces implemented as queues do not have the same challenges as synchronous patterns have when it comes to 2-phase commit. This is because an asynchronous pattern requires that traditional commit is implemented in an alternative way.

Another positive aspect of using asynchronous queues is that it makes interfaces well defined. It is a predefined set of messages that can be placed on the different queues.

Further, asynchronous queues make it possible to implement error handling through the use of retry and dead letter queues. This makes error handling simpler as it is a well defined behaviour that defines what should happen when something goes wrong.

Use known tools and solutions

When designing and implementing architecture, it is suggested to make use of known tools and solutions. The main reason behind this suggestion is that it is difficult to know how new tools and solutions will perform. Possible sources of known solutions are previous experience and existing documented patterns. You can choose to either use a collection of independent tools, components and solution, or use more extensive frameworks or architectural styles.

Some samples of patterns mentioned by interviewees were:

- Watchdog pattern
- Asynchronous communication
- Redundancy

Use components with care

Although components can be beneficial to use, you have to be careful. Components can help save development time, and provide useful functionality. A component or set of components might have been well tested through the use in several systems. When using the component in a new system you might use it in a different way, the component might not have been tested under these circumstances which could result in failures and incorrect functionality. The suggestion from the interviewees is to make sure that the components are well tested under similar conditions as you are going to use them in.

5.5.6 Maintainability

It is suggested that robustness can be improved by ensuring high maintainability. The benefits of maintainability has already been discussed in detail in Section 5.4.13. In addition to the already mentioned benefits, the following have been brought up by interviewees:

- Encapsulation helps robustness by making parts of the system easier to test in isolation.
- Complexity is reduced by using information hiding and encapsulation.
- Layering should be used and abstraction leakage should be avoided.
- Divide a large problem into manageable parts.
- Modularity enforces definition of clear boundaries.

5.5.7 Testing

All interviewees have mentioned testing as being important when building a robust solution. Numerous suggestions both on types of testing and what is important to consider in relation to testing have been mentioned.

Suggestions have been categorized into the following categories which will be discussed briefly:

- Plan testing.
- Test early.
- Test often.
- Spend time testing.
- Use many test methods.
- Involve users in early testing.
- Design with testing in mind.
- Use a relevant environment for testing.
- Use skilled testers.
- Unit testing.
- Regression testing.
- Stress and load testing.
- Extreme testing.
- Negative testing.
- Prototyping.
- Other test methods.

Plan testing Testing should not be an ad hoc task. It should be planned in the beginning of the development project, and written down as a test plan.

Test early Testing should be started as early as possible in the development project. At the latest, during the architectural design stage.

Test often Testing should not be a one-time activity in a development project. It should be repeated often, and testing should be performed in all phases of development.

Spend time testing When a project gets delayed, one should not get back on track by reducing the time available for testing. Time and resources should be devoted to testing; some suggest using more time than they currently do for testing.

Use many test methods You should not rely on a single test method. Various methods like for instance regression testing, integration testing, system testing, load testing and extreme testing should be combined into a test plan. No single test or test method would be able to find all issues and defects in a system.

Involve users in early testing It is suggested to involve the users before the acceptance testing stage. This is because when the acceptance test is performed, fixing certain issues can be too late.

Design with testing in mind When designing the system, you should also remember that it should be possible to effectively test it. Dividing the system into modules, and encapsulation are effective methods to make a system better testable.

Use a relevant environment for testing When testing, a relevant environment should be used. The environment should resemble the runtime environment the customer uses as much as possible. Testing by using the wrong version of databases, operating systems or on completely different hardware will most likely not reveal all environmental related issues.

Use skilled testers Testing is not a simple task that anyone can do. In addition to using end users and regular testers, you should also use especially skilled testers. Skilled testers often show more inventiveness than regular users when testing. You seldom see a user that tries to break the system; they often just test the functionality.

Unit testing It is suggested to make use of automated unit testing. This would lead to reduced effort needed for testing at the class, component and service level. By using automatic unit testing all tests can be run after changes have been made to the system. Although unit testing is beneficial it will not be able to replace other forms of testing, and should only be used as an additional test method.

Regression testing After adding or changing functionality in the application, regression testing of existing functionality should be performed. This ensures that existing functionality is not broken due to the changes.

Stress and load testing As there is no way to really be sure how a solution performs, load and stress testing should be performed. Load and stress tests help to determine how the system really performs, what happens when it gets overloaded, and combined with monitoring it can help pinpoint bottlenecks in the system. Further, you can test the system's resource consumption, and verify that the system is able to handle load over long periods of time without leaking resources or slow down.

Extreme testing Extreme testing should also be carried out. This can be for example testing extreme loads, disconnecting the network or introducing a corrupt file. This kind of testing will determine if the application is able to handle the error situation properly.

Negative testing In addition to testing functionality, it is also important to test on less common situations and perform error testing. Typical approaches are to test extreme values, deliberately use the system wrong or try to exploit common security vulnerabilities. This kind of test should be used at all stages of development, from early architectural design to acceptance testing.

Prototyping Prototypes can be used as a form of testing, to verify that certain solutions work, and test how they perform. This can help to reduce risks, and help to gain more knowledge about alternative solutions.

Other test methods A variety of other test methods and areas of focus for testing might be relevant for robustness. Some other mentioned by interviewees are: Usability testing, measuring code coverage during testing, testing security, testing error propagation, testing for security vulnerabilities, concurrency testing, and static code analysis.

5.5.8 Documentation

As already stated, documentation is an area where the results from the interviews is difficult to interpret. The results do give some indications that documentation is important. Interface and architectural documentation is mentioned in particular by one of the interviewees. It is difficult to maintain and extend a system that lacks proper developer/design documentation, and based on this it seems like documenting the system is important. End user documentation on the other hand was only mentioned by one interviewee. Based on this it is difficult to conclude whether it is important or not to focus on. It could help the users use the system correctly, but if a system requires a user manual to be used it is not user friendly. Fixing a system by creating documentation on how it must be used is probably not the best way to improve a ”broken” system.

When it comes to systems that do not have end users, but systems as the user, user documentation is more important. Documentation would then describe how the system interfaces work. This would help the using system to ensure that the system is used properly, and conforms to the interface specifications set forth by the system.

One interviewee suggests using graphical diagrams to illustrate the system. His experience suggests, that such figures are easier to understand and has given a positive contribution to the overall documentation. Diagrams and figures are often easier to understand than plain text explaining the same concept.

5.5.9 Usability

Focus on usability of the system should be a part of the development process. This includes indentifying the type of users, and their knowledge. Information about the users will be helpful when designing the user interface, and other

forms of interaction with the user. User interfaces should be developed in such a way that it prevents the user from making errors. If the user manages to make an error, the user interface should hinder it from being sent further into the system. Visual clues, wizards, interactive validation, and disabling of fields and buttons are some specific ways the user interface can help the user to correctly use the system. This aspect is mentioned by four of the interviewees.

5.5.10 Input handling

Focus on input handling is mentioned as important during design and development. It is suggested that not only user input should be validated, but also input from other systems and other components. All data entering systems and components should be validated so that only valid data is used. Not all invalid data leads to crashes. Invalid input could just as well result in invalid output. It is better to report invalid input, than to end up producing wrong output, or storing invalid data in a database.

Interviewees did not suggest any special solutions for input validation. This might be because the implementation of the validation itself is not difficult.

5.5.11 Configuration management

In addition to all the suggestions already listed, interviewees suggested that good configuration management is important when designing a robust solution. Two main aspects were mentioned: change control and version control.

Change control Changes needs to be evaluated before implemented in a solution. As the solution grows it also gets more and more complex. Before implementing a change, it is important to check for side effects. A tiny change could lead to a complete failure of a large system. By verifying and analyzing suggested changes such problems can be avoided in advance.

Version control As new versions are released, it is interesting to know what has changed since last version. It might also be necessary to undo some changes because they turned out to have unwanted side effects. As a result of this, it is suggested to have version control on the source code, and keep a log of changes between versions. This makes it possible to revert to the old version and check what has changed between versions.

5.6 Threats to validity

When doing research, a discussion of the validity of the results found is needed to determine to what degree the results can be said to be valid. This section will discuss the validity of the interview results presented in this chapter.

5.6.1 Selection of interviewees

Interviewees were selected through convenience sampling. It is not possible to know whether such a sample is representative or not, according to Robson [61].

This opens for the possibility that the results cannot be generalized to the whole population of architects. Further, only people involved with architecture were interviewed. Other groups like end-users, customers, and support personnel might have a different opinion on what characterizes a robust application. The same applies for the suggestions to what can be done to influence robustness.

Ten interviews were conducted in total and form the basis for the results. This number is low compared to the total number of architects, but 8 different companies are represented providing a span over the total population of companies. The similarity between individual interviews indicates that the definition of robustness is quite similar in different companies.

Moreover, the interviewees work mostly in large companies (companies with 40 or more developers).

All these three factors could threat the generality of the results presented. I advise to test the findings on a larger population.

5.6.2 Language and cultural issues

The interviewees were Norwegian, Swedish and Finnish. I communicated with the Swedish speaking interviewees in Norwegian, while the interviews with Finnish speaking interviewees were conducted in English. Language and cultural differences might threaten the understanding of the questions and answers. To minimize this, all interviews with Swedish and Finnish interviewees were conducted by the use of text chat. My experience with text chat is positive when it comes to talking to foreign people; it helps on pronunciation and talking speed issues. In particular Swedes have problems understanding Norwegian dialects. It may however still be an issue that I or the interviewees have misunderstood something.

Different countries might have different cultures when it comes to development. The sample is heavily biased towards Norwegian companies, so results may not be interpreted as valid for all the Nordic countries.

5.6.3 Multiple forms of interviews

Multiple forms of interviews were conducted: Text chat, phone interviews, in person interviews of both single persons and groups. In addition, three of the interviews were conducted by another interviewee, Dr. Tor Stålhane. All my interviews were limited to one hour; this gave a difference between the information provided in text interviews versus phone and in person interviews. It takes longer time to answer when writing than speaking. This could lead to a difference in the amount of information provided in the different interviews. However, all interviews conducted by me were completed before we ran out of time, so all interviewees had the possibility to express their ideas and opinions.

5.6.4 Interpretation and analysis of results

All interviews were transcribed after the interviews were conducted. The interviewee interviewed by phone got the transcript and agreed that I had understood him correctly. For the in person interview I sent the transcript late, and I had not recieved an answer at the time of delivery of this thesis. When

it comes to Stålhane's interviews, I am not sure whether he verified his interview notes with the interviewees. None of the interviewees interviewed by the help of text chat got the possibility to comment on my transcript. However, the transcript was a copy of the conversation log. I did not feel any need to let them read their own statements in retrospect.

During the interview and the subsequent analysis, there is a possibility that I may have misunderstood something. I could have sent the summarized interview results for the interviewees to comment on, but time did not allow it.

5.6.5 Overall validity

As described above, there are many potential threats to the validity of the interview results. It is safe to say that the results should be used with care. At the same time, they give an indication that can be developed further in subsequent analyses. It is also important to mention that interviews were only conducted with architects, and may not be seen as representative for other groups.

5.7 Selected areas of further focus

Interviewees provided a wide variety of suggestions on how to improve robustness. Suggestions vary from implementation details at one end, to process based at the other end of the scale. The main focus of my work on this thesis has been on architectural design and analysis. As a result of this, I chose to focus on some suggestions from the interviewees related to this area. The selected areas from the interview results are:

- Teamwork.
- Focus on robustness.
- Designing for robustness.
- Analysis of design.
- Gaining experience through prototyping.
- Reducing risk through iterative development and prototyping.

All these areas are relevant when doing design and analysis of software architecture. In addition, these areas are mentioned by more than a couple of the interviewees indicating that they are of some importance to other architects than myself. Due to limited time, and the huge number of suggestions, I had to make a selection. In my opinion, the areas selected form a sensible set of areas to base the further analysis of architectural design and analysis methods on.

These areas will be used together with personal experience to form a set of criteria for design and evaluation of architecture in Chapter 7. Further, the same areas of focus will be used discussing existing methods and suggesting a method in Chapter 9.

As time did not allow for studying patterns and architectural solutions in detail, some of the suggested architectural approaches are summarized and presented as a part of suggestions for further work.

Chapter 6

Robustness defined

Before continuing it is useful to look at the definition of robustness and provide a definition that matches the information collected through the interview process. Robustness as a term is commonly used in both literature and daily speech, so a concrete definition of the term is helpful before shifting focus to how robustness can be achieved.

First, I will illustrate an issue with current definitions in computer research. Next, I will look at some existing definitions of robustness from the literature and the internet, and illustrate their differences. The relation between reliability and robustness will then be presented before moving on to summarize the interview results. As a part of the summary, I will briefly discuss how the definitions cover the wide perception of robustness indicated by the interview results. A table comparing the individual definitions with the robustness perception will be presented before I present a definition for robustness. In the end of the chapter I will briefly discuss the relation between robustness and some other terms found in literature.

6.1 Literature and definitions

It is clear when reading articles, papers, and books that several definitions of common terms and concepts exist. Variations can be found between areas of research where the same term has different meanings, and within the same field or related fields of research where the same term has completely or partly different meanings. One example of such a difference can be seen when comparing the definition of security used by Laprie and Randell [50] with the definition used by Becker et al. [8]. In the paper by Laprie and Randell [50] the security quality attribute is defined as the combination of availability, confidentiality, and integrity. The article by Becker et al. [8] indicates that availability and security are independent quality attributes. The same observation has been done by Bass et al. [7] and is one of the reasons behind modelling generic scenarios for individual attributes. To provide a detailed overview of all quality attributes related to robustness that include a representative selection of definitions would require a huge literature study and is outside of the scope of this project. In order to illustrate robustness's relation to other quality attributes, I have chosen to select a single definition for some related attributes,

but several definitions for robustness.

Other works like Becker et al. [8] provides an overview over some quality attribute definitions and provides a discussion on differences. I have, however, not found any paper or book that provides an updated overview of all major attributes. This is most likely because such an overview would require extensive literature research and knowledge in various fields of research. Quality models is one source of an overview, but different models represent different views of quality. These different views result in different interpretations of the quality attributes used. This makes it necessary to combine definitions from several models in order to create a definition that covers multiple views. This is complicated by the fact that old models like those by Boehm and McCall are not revised to incorporate new concepts. Looking at robustness, it is not even listed as a quality attribute in ISO 9126, McCall's, or Bohem's model. Development of taxonomies similar to the taxonomy for dependability by Laprie and Randell [50] for all major areas of computer research would be an important building block towards the development of taxonomies or glossaries that span several fields of research. The IEEE glossary [10] provides such an overview, but new terms and concepts introduced since its last published revision are missing.

In the lack of established taxonomies for robustness it is necessary to look for definitions in existing literature, and compare them in order to find a definition that fits my needs.

6.2 Current definitions

I will start by briefly discussing some existing definitions of the terms robust and robustness. Each of the definitions is given a number for later reference.

Dictionary.com [77] defines robust as:

1.
   - 1. strong and healthy; hardy; vigorous: a robust young man; a robust faith; a robust mind.
   - 2. strongly or stoutly built: his robust frame.
   - 3. suited to or requiring bodily strength or endurance: robust exercise.
   - 4. rough, rude, or boisterous: robust drinkers and dancers.
   - 5. rich and full-bodied: the robust flavor of freshly brewed coffee.

Taken from a regular dictionary this definition represents a "popular definition" of the word robustness. It describes something that is built in such a manner that it does not easily break, shows strength, and is able to endure difficult conditions. IEEE Standard Glossary of Software Engineering Technology [10] has a definition with similar content:

2. Robustness: The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. See also: error tolerance, fault tolerance.

This definition is for software and more specific than the one from Dictionary.com. It describes two types of issues related to robustness. Invalid inputs

should be handled so that the system can continue to function correctly even when they are present. The other factor is stressful environmental conditions; this could be under high load, low amount of available resources and so on. Tolerance to faults and errors is also mentioned. Error tolerance is defined as ”the ability of a system or component to continue normal operation despite the presence of erroneous inputs” [10], which is basically the first part of the robustness definition. Fault tolerance is a much wider area, defined as ”the ability to continue normal operation despite the presence of hardware or software faults”[10]. Fault tolerance is only mentioned as a cross reference so it might not be intended as a part of the definition.

The definition from ATIS Telecom Glossary 2000 [75] includes fault tolerance as defined by the IEEE glossary. The definition does not specify a category of faults, so this is a less precise definition than the definition given by IEEE.

3. System Robustness: The measure or extent of the ability of a system, such as a computer, communications, data processing, or weapons system, to continue to function despite the existence of faults in its component subsystems or parts. Note: System performance may be diminished or otherwise altered until the faults are corrected.

The definition found on BusinessDictionary.com [76] contains similar elements, although it defines the term in relation to business.

4. Robust: Product, process, or system designed for continuous operation with very low downtime, failure rate, variability, and very high insensitivity to a continually changing external environment.

One difference is the focus on the system's insensitivity to a changing external environment. External environment is in the same dictionary defined as ”Conditions, entities, events, and factors surrounding an organization which influence its activities and choices, and determine its opportunities and risks. Also called operating environment”. This is a wider definition of insensitivity to the environment than found in the IEEE glossary's definition, as that only covers stressful environmental conditions. A new element contained in this definition is the lack of variability. This means that the system should be predictable and not have a large range of possible outcomes of a operation. This could include both normal and erroneous situations.

Webopedia.com contains yet another definition for robustness related to computers and software [2]:

5. Robust: When used to describe software or computer systems, robust can describe one or more of several qualities:

   - a system that does not break down easily or is not wholly affected by a single application failure
   - a system that either recovers quickly from or holds up well under exceptional circumstances
   - a system that is not wholly affected by a bug in one aspect of it
   - a system that comes with a wide range of capabilities. (It should be noted that this last sense of the term robust is not uniformly accepted in technical circles. The term is typically used in this sense

in the marketing of software or computer systems to emphasize a selling point and does not refer to the first three meanings of the term.)

This definition mainly focuses on that failures and bugs should not affect the whole system, and that the system should either recover quickly or not be affected by exceptional circumstances. The notion about a wide range of qualities is mentioned in the definition, but at the same time it is stated that this is more used in marketing than in technical circles.

Boehm et al. [11] in "quantitative evaluation of software quality" have the following definition:

6. Robustness: Code possesses the characteristic robustness to the extent that it can continue to perform despite some violation of the assumptions of its specification. This implies, for example, that the program will properly handle inputs out of range, or in different format or type than defined, without degrading its performance of functions not dependent on the non-standard inputs.

This definition focuses mainly on the code level details, but concerns that the program should perform correctly despite invalid input and other violations of specification assumptions. The next definition is from Hermanson [37]:

7. Robust software is software that is insensitive to errors regardless of the source – software, hardware, or people. In addition, robust software presents information to the user in a way that aids in the decision making process or makes information easier to comprehend. It is easy to use (i.e. the interface is designed to prevent human errors, and their adverse effects) and accommodates different ways the user may elect to use the software. The software does not "lock-up" or abort except through a predefined termination procedure. Robust software should also be easy to modify; it should be structured, documented and written in a language that makes it easy to isolate and remove faults.

This definition is wide, and includes usability and maintainability. In addition it includes the notion of being insensitive to errors, and also specifies that the application should not lock-up or abort.

Another common way to define robustness is in relation to reliability. Freeman [33] divides reliability into correctness and robustness. "A program is correct if it performs properly the functions that we intended and has no unwanted side effects"[33]. Robustness is further defined as:

8. Robustness: A program is robust if it will continue to do something reasonable in the presence of environmental changes (such as hardware failure) and demands (such as bad data) that were not foreseen. In addition to robustness the terms fault-tolerant and error-resistant are often used to describe this property.

This definition is related to the IEEE definition, but explicitly states that any reasonable behaviour is good enough in the presence of unforeseen changes in environment, or input. Buschmann et al. [13] have a different way of relating reliability to robustness, and distinguish fault tolerance and robustness as two aspects of reliability. The definition of robustness given is:

9. Robustness. This deals with protecting an application against incorrect usage and degenerate input, and keeping it in an defined state in the event of unexpected errors. Note than in contrast to fault tolerance, robustness does not necessarily mean that the software is able to continue computation in the event of errors - it may only guarantee that the software terminates in a defined way. [13, p. 408]

This definition emphasises that the application should protect against incorrect use, much like definition 7. This definition also states that keeping the software in a defined state, and protection against incorrect use are important aspects of robustness.

The last definition I will include is based on the dependability definition by Laprie and Randell [50]:

10. Robustness, i.e., dependability with respect to external faults.

This definition requires a little explanation. In Section 4.1.1, I defined fault, error and failure; the definitions have the same content as those used by Laprie and Randell [50]. Further, Laprie and Randell define dependability as: ”1) ability to deliver service that can justifiably be trusted; 2) ability of a system to avoid service failures that are more frequent or more severe than is acceptable”[50]. The robustness definition uses the term external fault. In order to know the difference between internal and external faults, some information from Laprie and Randell about systems, components and service is necessary. A system is something that provides a service. This service is normally used by a different system that either can be a user or another system. A system is constructed by a set of components. These components interact, and each individual component can be considered a system. This creates a recursive definition of a system that consists of communicating systems. Internal faults are faults (causes of errors) internally in a system. External faults can either be that a service provided by a system results in a failure. This failure will be an external fault for the system using the service. The other alternative is that usage of the system can trigger internal faults (vulnerability). An example is that the system is given two numbers as input, and should return the multiplication of these numbers. An internal fault might be that the internal service logic does not properly check that the result fits in the return data type (overflow). This could either result in the wrong value being returned, a service abort, or an exception being raised. Return of the wrong value will be a clear failure. The return of an exception could be a failure depending on how the specification states that this kind of situation is handled. An abort, or result omission is most likely also a failure, but could be acceptable if it is specified behaviour.

The width of this definition is subject to interpretation, as all failures resulting from usage of the system (correct or incorrect) could be covered by this robustness definition. Usage of the system that is in accorance with the specification that results in the triggering of an internal fault, is not explicitly defined as an external fault, but is indicated by the notion of *interaction faults* used by Laprie and Randell [50]. Further it is indicated by the following quote:

> a common feature of interaction faults is that, in order to be ”successful,” they usually necessitate the prior presence of a vulnerability, i.e., an internal fault that enables an external fault to harm the

> system. Vulnerabilities can be development or operational faults; they can be malicious or non-malicious, as can be the external faults that exploit them. There are interesting and obvious similarities between an intrusion attempt and a physical external fault that "exploits" a lack of shielding. A vulnerability can result from a deliberate development fault, for economic or for usability reasons, thus resulting in limited protections, or even in their absence.

This statement in combination with what is considered a failure "A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function"[50], also suggest that correct input can be seen as external faults. The reason is that the implementation is vulnerable as it does not comply with the specified behaviour. This class of faults is also covered if there is another system or component involved. Consider a user that interacts with component A which again calls component B based on input provided by the user. The input triggers a fault in component B that results in a failure. This failure is an external fault to component A. How component A handles this fault is covered by the definition. This is complicated by the fact that an interaction can trigger a fault, which results in an error in the system's internal state. The state can be further altered by other interactions, and not until the error results in the wrong external state being returned to the user of the system, it is a failure.

The list of definitions above is only a small selection of definitions of robustness. I have not conducted an extensive literature research, so the list of definitions might not be fully representative for every definition used in literature. There is a huge span in the definitions. They range from protection against environmental changes and erroneous inputs to defining a totally fault tolerant solution. As a result of this discovery, the interviews were conducted, and I will come back to the analysis of the interview results, but first I will take a brief look at the relation between robustness and reliability.

6.3 Reliability and robustness

Reliability is a term related to robustness. As already illustrated by the definitions for robustness found in Buschmann et al. [13] and Freeman [33], robustness can be defined as a part of reliability. Zhou and Stålhane [79] separate the concepts for web applications by looking at the types of faults caused by a problem. It is stated that reliability problems are caused by internal faults within the system or component, and robustness problems by external faults rising from the operational environment, such as unexpected input. Any fault that makes the system contradict the specification is a reliability issue, and any other fault is a robustness issue. This way of separating robustness from reliability makes a system completely reliable if it is correct. This is in contrast to the definition given by Freeman [33] and Buschmann et al. [13] which defines robustness as a part of reliability, making it impossible to have a program that is reliable unless it is also robust. The relation to reliability can, however, be covered by the approach presented by Zhou and Stålhane [79] if robustness issues is included in the specification. Although it handles the contradiction, it makes it difficult to determine if a problem is related to robustness or reliability

without investigating the specification.

The IEEE glossary defines reliability as:

> Reliability: The ability of a system or component to perform its required functions under stated conditions for a specified period of time. See also: availability, MTBF[1]

This is only one definition of reliability. Becker et al. [8] list no less than ten different definitions for reliability. This illustrates that there does not exist a de facto definition of reliability in software. Based on this observation and the large list of definitions of robustness with huge differences in their scope, it is difficult to draw a clear line between reliability and robustness. This will further be illustrated when I later look at dependability that combines traditional concepts like reliability, maintainability and availability into a new wide field of study. It is my belief that robustness and reliability indeed shares a common set of issues and solutions, but that neither concept is a strict subset of the other, and that robustness share a common set of issues with some of the traditional concepts like security and maintainability. I will get back to this as I look further into the results of the interviews.

6.4 The interviewees's definition of robustness

The full results from the interviews have already been presented in Chapter 5. The interview results clearly show that the concept robustness cover a wide area. I will summarize the results and see how the definitions presented covers the areas mentioned by the interviewees.

Good error handling is an important part of a robust system. A clear strategy for error handling should be present in the system, and error situations should be carefully handled. The system should try to prevent faults and errors from occurring. If a fault or error should occur, the application should be able to handle the error in some predictable way. Another important aspect of error handling is that a robust application should keep errors local. This means that an error in a module should not affect the whole system. Last not but least any error messages in the system should be informative, this means that errors like "an error occurred, press OK to continue" should not occur. The notion of error handling is covered by almost all the listed definitions for robustness, but it is important to notice that the interviewees did not limit error handling to a specific class of errors, fault or failures. At the same time many of the interviewees stated that the error handling does not have to handle the error. Identifying that there is an error and handling it gracefully is good enough for an application to be considered robust. Possible actions depends on the applications requirements. To help with error handling, a robust system should also be able to log important events. This includes possible errors, and failures. This makes it much easier to diagnose errors if the log contains relevant details.

Related to error handling is the use of redundancy or failover to increase robustness. Not all the interviewees are familiar using failover or redundancy, but agree that it can increase robustness. This is merely a way of handling failures, so a definition does not need to cover this area.

[1] Mean time between failures

Predictability is also mentioned as an important aspect of robustness. This implies that the application, if used in an environment it is built for, should show predictable behaviour. It should also satisfy the needed level of availability, having a low downtime. This is also covered by some of the listed definitions, and is a by-product of proper handling of error situations.

Next, input control is an important characteristic of a robust application. An application that crashes in an undefined way due to invalid input is not considered robust. In close relation to this are also internal validity control and validity and consistency control in the data storage of the application. Control of validity assists in the discovery of some types of errors, and helps ensure that the application and data have a valid state. To help with correct input, the application should also be user friendly making it difficult for the user to enter invalid input, and use the same terms and concepts as the user. These aspects of user friendliness help in reducing the level of invalid input, and lower the number of usability issues in the system. The notion of input tolerance is a central part of some of the definitions, but the extension to validity and consistency control is not covered to the same extent.

In order to be robust it is also important for an application to be thoroughly tested, and thus contain few errors. An application does not have to be completely free of errors to be considered robust, but an application that contains a high number of errors is not considered to be robust by a user. Handling any errors gracefully helps on the robustness, but an application with a high number of errors will most likely not be able to handle them in a way that does not influence the use of the application in some way. If a user has to press retry ten times before an operation is complete, the errors has been sufficiently handled, but the user might not consider the application to be robust. None of the definitions listed does explicitly mention a low number of errors, only that errors should be handled. This is partly related, as few errors results in few faults and errors to handle.

Aspects of maintainability are also important for an application to be considered robust. It is important that faults are easy to locate and correct. It is also important that the application is built in such a way that addition of new functionality and changes in existing functionality do not affect other parts of the application. To achieve this, the application has to have a good structure. This aspect of robustness is only mentioned by definition 7.

Security is an area where the interviewees have different opinions. Some define all aspects of security as a part of robustness, others think security does not directly relate to robustness. When asked directly, it seems like there is one common aspect of security that is related to robustness, injection attacks. Some applications also require a high level of security. These security measures need to be robust so that they cannot be easily broken. The degree of security needed varies between different types of applications. A drawing application would not have any needs for security, while a on-line bank service requires a high level of security.

Proper load handling is an important aspect of robustness. An application should not fail after long use because of some resource leak, or fail completely when under high load. This can be defined as changing environmental conditions and is covered by the listed definitions.

Last, interviewees mentioned that a robust application or architecture should be able to cope with changing requirements as time goes by. This is mainly a

maintainability issue. Further, good documentation has been mentioned, but I see no clear relation to robustness besides making the application more user friendly and maintainable. Adaptability to the environment is also mentioned, which is partly covered by maintainability. The other part of environmental adaption is adaption to the runtime environment. This part is mostly handled by high load conditions and error situations.

Table 6.1 shows a schematic overview of how the different definitions cover the aspects mentioned by the interviewees. All definitions are subject to interpretation, and other interpretations of the same definitions might result in a different table. The purpose of the table is not to show the absolute coverage of the different definitions, but to give an indication of their coverage. It can clearly be seen that the definitions have varying coverage and none of the definitions completely cover the wide definition of robustness given by the interviewees.

I want to make a special note about definition 10, as this is the definition where the most information on how it could be interpreted is available. Exact information on how it should be interpreted is, however, not provided in the paper by Laprie and Randell [50]. The definition alone does not give a clear indication of what it covers. It requires more information about the terminology employed by the authors. If interpreted in a wide manner, the definition might cover all aspects mentioned by the interviewees. Becker et al. [8] has reformulated the definition into: "Robustness is the ability of a system to tolerate inputs that deviate from what is specified as correct input". This interpretation indicates a far more narrow definition of robustness, than I have illustrated is possible above. Due to the possibility for conflicting interpretations, I do not find this definition to be optimal for my use.

6.5 The definition of robustness

Robustness as defined by the interviewees is a very wide concept, and it is not possible to define it isolated from other existing concepts. I have already illustrated that there is some common ground between reliability and robustness. An early theory was that robustness had a defined core, with additions for different types of software development. This is still partly true for instance, robust security is more important for some classes of applications than others. A better model for robustness however is based on the way Laprie and Randell [50] has defined dependability and security. Dependability is defined as a combination of availability, reliability, safety, confidentiality, integrity and maintainability. The interview results suggest a similar definition for robustness, a definition that includes parts of other definitions. Specific aspects of the attributes are included in robustness, but not the whole attribute. Requirements defined for the software product and choice of technical platform also define the importance of various attributes. A schematic view of the relation between robustness and other concepts can be seen in Figure 6.1.

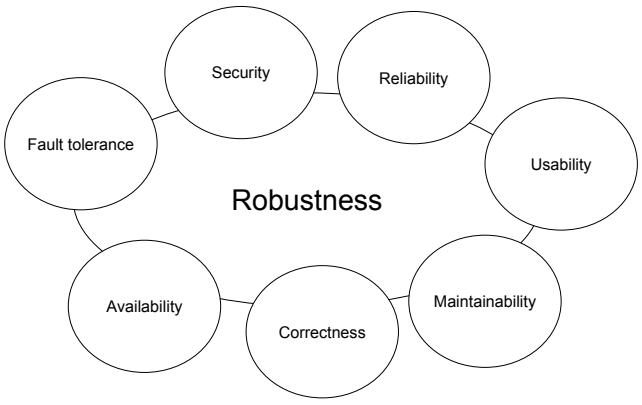

Figure 6.1: Robustness and included concepts

Def	Error handling	Predictable	Low downtime	Validity and consistency	Input tolerance	Handles load	User friendly	Security	Maintainability	Few faults/errors	Comment
1	-	-	-	-	-	-	-	-	-	-	The definition is too general to be evaluated
2	Indirectly, errors and failures from other modules can be seen as (invalid) input, and stressful environmental conditions can result in failures being reported.	Indirectly, the system should be predictable (behave correctly) even when given invalid input and when under stressful environmental conditions.	Indirectly, as invalid inputs and environmental conditions could contribute to downtime.	Not covered	Covered	Not covered, interpreting high load as a stressful condition could include it.	Partly, handling invalid input gives a better usability.	Not covered.	Not covered	Not covered	
3	Yes, faults can lead to a decrease in function and performance.	Indirectly, the system should continue to function in the presence of faults.	Indirectly, as faults is a major cause of downtime.	Not covered	Partly, wrong input due to faults in parts of the systems should be handled.	Not covered, could be said to be partially covered as faults due to high load should be handled.	Not covered	Not covered.	Not covered	Not covered	
4	Yes, errors should not result in failures.	Yes, system should have continuous operation and low variability.	Yes	Not covered	Indirectly, lack of input tolerance would result in failures.	Indirectly, not coping with load situations could lead to downtime.	Not covered	Not covered.	Not covered	Indirectly, due to low failure rate which is hard to accomplish with a large amount of errors/faults.	
5	Yes, single failures should not affect the whole system.	Yes	Yes, system should quickly recover from exceptional situations.	Indirectly, validity and consistency checks are a way of preventing the effect of bugs from spreading.	Partly, bugs related to input handling should not affect the whole system.	Not covered, could be said to be partially covered as extreme high loads can be seen as an exceptional circumstance.	Not covered	Not covered.	Not covered	Not covered	
6	Partly, errors due to invalid input should be handled; this could also cover failures from elements being used by the code.	Yes, undefined inputs, and other violations of specified assumptions are properly handled.	Not covered	Partly, explicit checks of input are a form of validity check at this level of detail.	Yes	Not covered	Indirectly, it contains protection against incorrect use.	Not covered.	Not covered	Not covered	High focus on code level details
7	Yes, insensitive to errors regardless of source.	Yes, does not abort or lock up.	Indirectly, as it is insensitive to errors, there are fewer causes for downtime.	Not covered	Yes	Not covered	Yes	Not covered.	Yes	Not covered	
8	Yes	Indirectly, the system should have predictable behaviour in spite of faults and failures.	Indirectly, due to reasonable behaviours in the presence of faults and failures.	Not covered	Yes	Not covered, interpreting high load as an environmental change could include it.	Not covered	Not covered.	Not covered	Not covered	
9	Yes	Yes, should be in defined states also when unexpected errors occur.	Partly, choice of termination instead of other alternatives could lead to higher downtime.	Yes	Yes	Not covered	Yes, protecting against incorrect use.	Not covered.	Not covered	Not covered	
10	Yes, but the class of faults depends on the interpretation of external fault.	Partly, behaviour should be predictable based on external influence.	Indirectly, downtime should be low due to handling of external faults, but internal faults could cause downtime.	Not covered	Yes, incorrect input is treated as an external fault	Not covered, but certain interpretations of the definition could include it.	Indirectly, external faults (user errors) should be handled.	Not covered.	Not covered	Yes, due to external faults.	

Table 6.1: Robustness definition coverage

Robustness can be defined as the ability to:

- have a high degree of correctness, and gracefully handle any remaining faults, errors and failures
- have predictable behaviour under specified conditions, and fail gracefully if outside these conditions
- provide means to easily detect and remove faults
- accept modifications in modules in such a way that it does not affect other modules
- handle all input from users, components, and other systems in a safe and predictable way
- keep a required level of security despite a changing environment and variations in input
- not be vulnerable to security related faults

This broad definition contains all central elements mentioned by interviewees, and the inclusion of other concepts resembles Figure 6.1. The list of related concepts included in the figure is by no means complete. There exist other concepts like accountability, resilience, dependability, trustworthiness, safety and survivability. The next section will briefly discuss the relation to three of these concepts, but determining the complete list of concepts, and their degree of relation to robustness is a possible direction of further research.

6.6 Other concepts

Figure 6.1 shows the relation to some other concepts. There are, however, three other concepts that shows increasing popularity such as trustworthiness, dependability, and survivability.

Laprie and Randell [50] defines dependability as:

Definition 6.6.1. Dependability 1) ability to deliver service that can justifiably be trusted 2) ability of a system to avoid service failures that are more frequent or more severe than is acceptable

Ellison et al. [27] define survivability as:

Definition 6.6.2. Survivability The capability of a system to fulfill its mission in a timely manner in the presence of attacks, failures, or accidents.

Becker et al. [8] use the following definition for trustworthiness, taken from Schneider [64]:

Definition 6.6.3. Trustworthiness is assurance that a system deserves to be trusted — that it will perform as expected despite environmental disruptions, human and operator error, hostile attacks, and design and implementation errors. Trustworthy systems reinforce the belief that that they will continue to produce expected behaviour and not be susceptible to subversion.

Laprie and Randell [50] give a comparison of these three concepts and conclude that these are ”essentially equivalent in their goals and address similar threats”. Definition 10 for robustness in Section 6.2 is a subset of dependability; this indicates a clear relation between these three concepts and robustness. It is unclear whether robustness is a strict subset of these concepts, or only contains some common ground like reliability. The relation is, however, helpful, as existing research in these areas can also be of help to handle robustness.

Chapter 7

Analysing architecture

There is a large number of approaches to analyzing, evaluating and designing architectures available. A comparison of eight methods has been performed by Dobrica and Niemela [21]. I have chosen to focus on a few methods, as covering all methods is not the main purpose of this project. Formal methods have been excluded. The reasoning behind this selection is that my main focus is on business application development, and formal methods are not extensively used in this development domain. The selection of methods is based on that the methods should be easy to learn, understand and perform; making it more likely that the current industry will use the methods.

First a set of criteria for comparing the methods will be presented. The criteria are based on personal experience and the interview results presented in Chapter 5. Next, a characterization of analysis types will be described before moving on to the methods. I will look at several types of analysis and design techniques like reviews, scenarios, ATAM, SAAM, TRIAD as well as traditional risk and safety methods. In the end of the chapter I will present a comparison of the methods based on the criteria.

7.1 Method criteria

Before starting to present the methods, I will present a list of criteria I later will use to compare the methods presented. The criteria is based on personal experience, and knowledge gained during the interview process.

1. **Is it an evaluation or design method?** This criterion is just to classify the methods.

2. **Which phases of design is the method suited for?** Interviewees have indicated that analysis needs to start early and be a recurring activity trough the whole development process. Based on this a method to evaluate or design for robustness should be usable in various stages of design and development. A good method that can be used during the whole design and implementation phase would be better than using multiple methods.

3. **Is the method for a special type of application?** A method should be usable for all types of applications. This is because I am not focusing on a special

type of applications like distributed systems.

4. **Does the architectural description have to be in a special form?** A method that requires architectural and design documentation to be in a special format, would put requirements on the rest of the development process. This would make the method harder to use, as it would require changing or converting existing architectural descriptions. Depending on a industry standard notation like UML would be ok, but a proprietary notation would not be.

5. **Method supports analyzing single or multiple quality attributes?** Interview results show that robustness includes several other quality attributes. A method should be flexible and handle analyzing both multiple quality attributes and a single quality attribute.

6. **Is the method integrateable into an existing development process?** A method should not put too many requirements on the development process, and be easy to integrate into an existing development process. This is based the assumption that things that are complicated to use, require more effort and is less likely to be used. The best is if the method is stand-alone, then it does not need to be integrated into the process at all.

7. **Is the method demonstrated for robustness analysis?** It is interesting to know if there exist examples where the method is used for robustness. This could provide additional information about the methods applicability to robustness.

8. **Does the method scale with ambition?** Not all companies are willing to devote large amounts of time and resources to robustness design or analysis. Companies and development groups vary in size. It is beneficial for a method to be scalable so that it can be adapted to the project size and amount of resources available to perform the design or analysis. I am looking for a method that scales well with the ambition and the size of the team.

9. **Is the method suited for iterative development?** Iterative development seems to be popular, and a method should support iterative development.

10. **How easy is the method to learn and use?** Based on personal experience, a method that requires large efforts to learn and use is less likely to be used than a method that is easier to both learn and use. This has to be seen in conjunction with the methods ability to scale.

11. **Is the method based on teamwork?** Working in teams has been mentioned as being positive for robustness, and an analysis or design method should therefore include working in a team.

12. **Does the method support analyzing failure behaviour?** Error handling and how the application handles errors is central for robustness. A method should support analysis of how the application deals with various errors.

13. **How does the method document results?** It is interesting to know how the method document design or analysis results.

14. Does the method make use of prior knowledge? It is beneficial if a method is able to make use of prior knowledge. This could help less experienced people use the method more effectively.

When it comes to importance of the different criteria, I find the criteria 2-4, 6, 8, and 10-12 to be most important. The rest of the criteria are less important. Personal experience and information from the interviews are the reason for selecting these as the most important criteria. A method that fulfils these criteria should be easy to adapt and use, support error analysis and be suitable for a wide range of development projects. I find that scalability and ease of use is a key point when looking for methods to use.

The next sections will present a selection of design and analysis methods, and in Section 7.9, I will compare the individual methods to the criteria listed above.

7.2 Analysis types

Abowd et al. [3] divide architectural analysis into two main categories: *questioning techniques* and *measuring techniques*. Questioning techniques can be used to evaluate any quality, while measuring techniques tend to be more focused on a particular quality attribute. Questioning techniques help us to learn and understand more about the architecture's fitness to the requirements placed upon it. This is done by posing questions about the architecture, but the techniques do not help much in answering the questions.

Measuring techniques are used to answer specific questions. Mature areas like performance and modifiability are the most used quality areas for measurements. This is mostly because measurement requires the area of focus to be well defined and mature in order to know what measures to use and how to interpret the measured values. Another important difference is that questioning techniques help in collecting qualitative answers, while measuring techniques is about creating quantitative answers. However, measuring techniques can be used to answer questions given by questioning techniques.

The report also lists main categories of questioning and measuring techniques where questioning techniques consist of:

Scenario is a list of changes/uses of a system or architecture. The scenario can be used to evaluate how well the architecture handles the changes/uses. Different types of scenarios contain different types of actions. Security scenarios contain a list of threat actions, while modifiability scenarios contain a list of changes or modifications.

Questionnaire is a list of general and open questions that apply to most types of architectures. The questions could be about the process and resources involved as well as the described architecture itself. The purpose of the questionnaires is to get an answer to any concerns one might have on the architecture. These concerns often relate to the requirements which the architecture is designed to fulfil.

Checklist can be viewed as a specialization of questionnaires. A checklist is based on experiences with a type of quality and/or class of systems. A

checklist often focuses on aspects relating to a single quality attribute, while questionnaires typically have a more broad focus.

The report divides measuring techniques into:

Metrics that are quantifiable measures which describe some specific aspect. Examples of such metrics can be number of code lines and fan in/fan out. Metrics can be applied for various quality attributes and at several levels of abstraction.

Simulations, Prototypes, and Experiments can be used to both help create and clarify architectures. Prototypes are normally built to either prove some aspect of an architecture, or to learn more about it. Simulations provide answers to specific questions by simulating how the architecture(s) will behave. Performance models are a typical use of simulations.

I will look closer into some of these types of evaluation. Scenarios will be further described in Section 7.3.1. Metrics are a central part of quality models presented in Chapter 4. Some other types of analysis like checklists require extensive experience with multiple systems, and therefore be outside the scope of this project, and will not be handled further.

7.3 Scenario based analysis

First, scenarios in general will be described. Second, two methods that utilizes scenarios will be presented, first SAAM and then ATAM.

7.3.1 Scenarios in general

Scenarios are widely used in many areas; requirements elicitation, performance modelling, and safety inspections to name a few. Scenarios are effective means of communication, easy to learn and effective in use. This is probably the explanation for its extensive use [15].

Scenarios come in several forms, but the kind of scenarios used in architectural evaluation consists of three parts; *stimulus*, *environment*, and *response*. The scenario describes an interaction with a system, where the interaction can be performed by a user, another system, another part of the same system, a developer, an architect, or a customer to name a few. These are commonly called *stakeholders*. Scenarios form a central part of some architectural evaluation methods like SAAM and ATAM, and are used to clarify and concretise quality attribute requirements. Most common quality terms like e.g. reliability, modifiability and availability provides a common platform for communication. However, these common quality terms need to be further elaborated in order to create meaningful requirements. In addition, most of common quality terms have to be evaluated in context, as it for example is not possible to achieve complete maintainability; it will always be in some context. The context can either be a particular system, or a set of changes. Scenarios provide this context [3].

Bass et al. [7] differentiate between *general scenarios* and *concrete scenarios*. General scenarios can pertain to any system, while concrete scenarios are

adapted to a specific system. Normally, a concrete scenario can be formed by adapting the general scenario to a specific system. Another characterization is used in SAAM [15, 42] and an article by Kazman et al. [43]: *direct* and *indirect* scenarios. A direct scenario represents functionality or uses of the system. These are scenarios that are handled with the system and do not require any modification. Indirect scenarios are those requiring a change to the system. Typical indirect scenarios are growth of the system or porting the system to a new platform, while a direct scenario can be processing a message arriving at the system. ATAM uses three types of scenarios that are related to the direct and indirect characterization: *use case scenarios*, *growth scenarios* and *exploratory scenarios* [15]. Use case scenarios are similar to direct scenarios and describe uses of the system. Growth scenarios are similar to indirect scenarios and describe normal growth or modification to the system. Exploratory scenarios are the extreme version of the two former types. This kind of scenario tries to stress the system and represent extreme uses or changes to the system. Samples of such scenarios can be platform changes, implementing completely new functionality like going from only receiving synchronous to handling both synchronous and asynchronous messages.

A complete scenario should consist of the three mentioned parts' stimulus, environment and response. The final formulated scenario does not have to adhere to this particular form, but should contain all three aspects in order to be clear and testable.

Stimulus This describes what the stakeholder does to initiate the interaction with the system. This includes both the stakeholder and what the stakeholder does and what change or input that is used. One stimulus could be that an end user enters his name and presses enter, another could be that the developer implements support for a new product type to the system.

Environment This describes the part of the system that receives the stimulus and the state the system is in when the stimulus is received. If the state of the system is normal and the whole system is the target, this part can be omitted. One example of an environmental condition could be that the primary node is down, and the secondary node is partially failing due to extensive load.

Response This describes the anticipated response the system should give when the stimulus is received by the system. The response should be defined in such way that it is possible to check if the actual response is in accordance with the scenario. Samples could be that the new product type should only require some added configuration to the database and the creation of a new registration form, or that the system should either process the request in 10 seconds or respond with a message that informs the client to try again later.

Apart from concretising and providing a way to analyse quality attributes which I will look more closely at while describing the utility-tree method in ATAM, scenarios in combination with analysis methods also help to provide a better understanding of the requirements, helps stakeholders buy in, and give a shared understanding of how the architecture fulfils the scenarios. Information

on how scenarios are accomplished by the architecture also gives a documentation on how the parts of the system work together to provide functionality and adapt to changes. Last, the mapping of scenarios onto the architecture provides traceability of the requirements down to the components that realize the requirement. [43]

7.3.2 SAAM

SAAM was first presented by Kazman et al. [41], and a later version that also incorporates the evaluation process was presented by Clements et al. [15], Kazman et al. [42]. SAAM is a method for evaluating the quality of architecture, or for comparing alternative architectures. Maintainability and functionality are the main focus areas of the method, but it can also be used to analyze other aspects like performance and safety.

Compared to other methods like ATAM it is easy to learn, and requires less time as the method itself is simple. Scenarios are the method's vehicle of communication, and the use of scenarios stimulates communication among participants. Apart from having a positive effect on communication, it helps to ensure that the architecture is properly documented, as an architectural description is a prerequisite and the method itself enhances and complements the existing documentation. Requirements and business goals will be translated into well defined scenarios and problematic areas of the architecture will be expressed. This will be both areas with high interaction among scenarios, and scenarios that require a modification to a large number of scenarios. The following steps are involved in a typical SAAM analysis [15, 42]:

Step 1 - Develop Scenarios This is a step there the whole group brainstorms for scenarios. The scenarios should represent anticipated changes to the system, major uses of the system and qualities the system should satisfy. Both the present and the foreseeable future should be covered.

Step 2 - Describe the Architecture(s) The architecture should be described to the participants so that it is well understood. Several architectural views should be available and described. The description of the architecture and the development of scenarios is normally a recurring loop. When new scenarios surface, the architecture might have to be modified to take these into consideration, or more documentation might be needed. As more details and knowledge about the architecture is available, the participants discover more relevant scenarios.

Step 3 - Classify and Prioritize the Scenarios Each scenario in the list should now be classified as a direct or indirect scenario. After classifying the scenarios, they should be ranked according to their importance. This prioritization could be done by voting.

Step 4 - Individually Evaluate Indirect Scenarios The most important scenarios from the list should now be mapped onto the architecture. Direct scenarios should be demonstrated by how they execute on the architecture. Indirect scenarios are demonstrated by illustrating how the architecture needs to change to implement the scenario. This step will illustrate the strengths and weaknesses of the architecture and its documentation. The

output of this step should be a list of scenarios with a description of required changes and an estimate of the required effort for all the indirect scenarios.

Step 5 - Assess Scenario Interactions Two indirect scenarios requiring a change to the same component or set of components are said to interact. If multiple semantically different scenarios interact in one component this could indicate a potential problem. The same is true if a scenario requires changes to a large set of components. This step is about investigating interactions between scenarios and analyzing how the architecture overall responds to changes. The goal of this step is to discover if interactions occurs because the documentation is not at the correct level of detail (the documentation shows that interaction occurs, but the component is really divided into subcomponents where interaction does not occur), or if the component or set of components is an area of high coupling and potential complexity that the designers should focus on. Scenarios that require a large number of changes should also be a focus area of further design.

Step 6 - Create the Overall Evaluation This step can be used to compare multiple alternative architectures, or multiple alternative solutions in one architecture. This is done by assigning a subjective weight to the architecture that could indicate costs, complexity, general suitability or other important factors. In addition, each architecture could be assigned a positive, negative or neutral point for each scenario. This combination of weight and scenario-points can be used to compute a suitability rank for the architecture and produce an ordered list that can be used as a basis of decision.

7.3.3 ATAM

ATAM [7, 15, 40] stands for Architecture Tradeoff Analysis Method, which is a structured method to analyze an architecture focusing on multiple quality attributes. The method is based on the SAAM method and is inspired by the quality attribute communities and the notion of architectural styles. The creators of the ATAM method think architectural styles are important as they differentiate different classes of design. Architectural styles are helpfull by providing experimental evidence for how a class of architectures has been used, and qualitative reasoning that explains why the class of architectures has certain properties and when it should be used [15].

The ATAM method focuses on identifying risks, nonrisks, sensitivity and tradeoff points in the architecture, and makes extensive use of scenarios. "A *sensitivity point* is a property of one or more components and/or component relationships that are critical for achieving a particular quality attribute response"[15, p. 36], while "a *tradeoff point* is a property that affects more than one attribute and is a sensitivity point for more than one attribute" [15, p. 36]. A *risk* is a problematic issue in the architecture; it can be a decision that has not been taken, an architectural decision, or something outside the architecture itself. A sensitivity or tradeoff point can be a risk in the architecture, but can also be a nonrisk. A *nonrisk* is a good decision in the architecture based on something

that is fixed or highly unlikely to change. This means that when the underlying assumptions change, the nonrisk might be converted to a risk.

> For example, assigning processes to a server might affect the number of transactions that server can process in a second. Thus, the assignment of processes to the server is a sensitivity point with respect to the response as measured in transactions per second. Some assignments will result in unacceptable values of this response - these are risks. Finally when it turns out that an architectural decision is a sensitivity point for more than one attribute, it is designated as a tradeoff point. [15, p. 57]

After a completed ATAM evaluation, we will have a list of business goals, a presentation of the architecture and a catalogue of the architectural approaches employed. In addition, we will have a list of the most important scenarios linked with relevant architectural approaches, and a list of relevant quality attribute question for the approaches used, and risks, nonrisks, sensitivity, and tradeoff points. The method can be used both in the early stages of development, and on legacy systems that are about to undergo some major changes or additions. While the main objective of the method is to identify risks related to the architecture, sometimes alternative solutions or improvements can be produced as a by-product.

ATAM was first described in an article by Kazman et al. [40], but is presented in a more mature version in Bass et al. [7] and Clements et al. [15]. The steps in the mature version are presented below. Even if the steps are numbered 1-9 the method is not strictly a waterfall method — there is nothing that prevents loops or iterative uses of the method.

Step 1 - Present the ATAM The main focus of this step is to let the evaluation leader present the ATAM method to the participants. This presentation should briefly describe the steps and the methods that will be used. It also helps set the context of the work that should be performed, and set the expectations of the participants. The evaluation leader should also reserve some time in his presentation to answer any questions the participants might have.

Step 2 - Present the Business Drivers A representative of the project, like the project manager or the customer should present the business goals that motivate the development of the system. These are the main goals that represent the main architectural drivers like high security, time to market or high flexibility. The main purpose of the step is to make sure that all the participants, including the evaluation team, fully understand the context of the system, and the reasoning behind its creation. A high level system overview, focusing on the main system functions, and any technical, managerial, economical, or political constraints should also be covered in order to fully understand the system's context.

Step 3 - Present the Architecture Architectural presentation is the main objective of this step, and the lead architect or the architectural team should do the presentation. The presentation should focus on how the architecture supports the main business goals presented in the previous step. Level of

detail and architectural views should be chosen in a way that best present the architectural approaches that helps fulfil the business goals and main quality requirements. Available time and the level of detail the architectural design has, will also have to be taken into consideration when choosing views and level of details. Any constraints like OS, hardware or prescribed middleware and any required interaction with other systems should also be presented so that the participants can understand the underlying constraints of the architectural design.

Step 4 - Identify the Architectural Approaches The main purpose of this step is to identify the architectural approaches used to meet the main quality requirements and goals. The architect will identify and name the architectural approaches and architectural styles used. There will be no analysis of the approaches and styles in this step. The analysis team will just make a list of the approaches presented and add any approaches revealed during the preceding architectural presentation.

The background for this collection is that the architectural approaches and styles lay the architecture's main foundation and structures for addressing the highest ranked quality attributes. These approaches and styles is the architect's main mean to ensure that the finished system will meet its critical requirements in a predictable way. They will lay the foundation for how the architecture can grow, adapt, resist attacks, respond to major changes and so on. By later analyzing the architectural choices, the evaluation team can determine how well the approaches and styles qualifies the system's driving quality attribute goals.

Step 5 - Generate the Quality Attribute Utility Tree If the analysis was to take all possible qualities and approaches into consideration, it would be a tedious task that could take a long time to complete. Building a utility tree is ATAM's way of focusing the analysis on the most important parts of the system. This step is performed by the main decision makers in the project, - normally the architectural team, manager, and customer representatives. A utility tree represent the system's "goodness" and this is represented by the trees top node labelled *utility*. Below this level the team is requested to name the main "'itilities" or quality attributes the system should employ. The list does not have to adhere to any particular naming convention from a quality model, but can use own terms and categorizations. As long as the category can be further refined, any naming and categorization is fine. At the third level, each of the quality attributes should be refined into sub-categories. For instance, availability is separated into hardware and software failures. Below each sub-category a list of well defined scenarios should be created. Each of these scenarios should be important for the system.

After all the relevant scenarios have been created, they have to be prioritized. The relative importance of the scenario, and how difficult it is to realize should be ranked separately. Any ranking can be used, but it is encouraged to use a High/Medium/Low ranking as it is not possible to apply definitive metrics. This step will produce a list of well defined, testable scenarios that define what is most important for the system being developed. All participants are forced to explicify and prioritize current

and future driving forces for the system. This might also facilitate discussions based on that new important goal and driving forces are brought up on the table.

This ranked list of scenarios will guide the rest of the analysis, as the highest ranked scenarios will be analyzed first, and then one goes down the list. Normally the lowest ranked scenarios are not analyzed as they are either easy to ensure in the architecture or are of low importance to the system. Apart from this, time will set the limit for which scenarios are covered and not in the following analysis. An example of a utility tree can be seen in Figure 7.1.

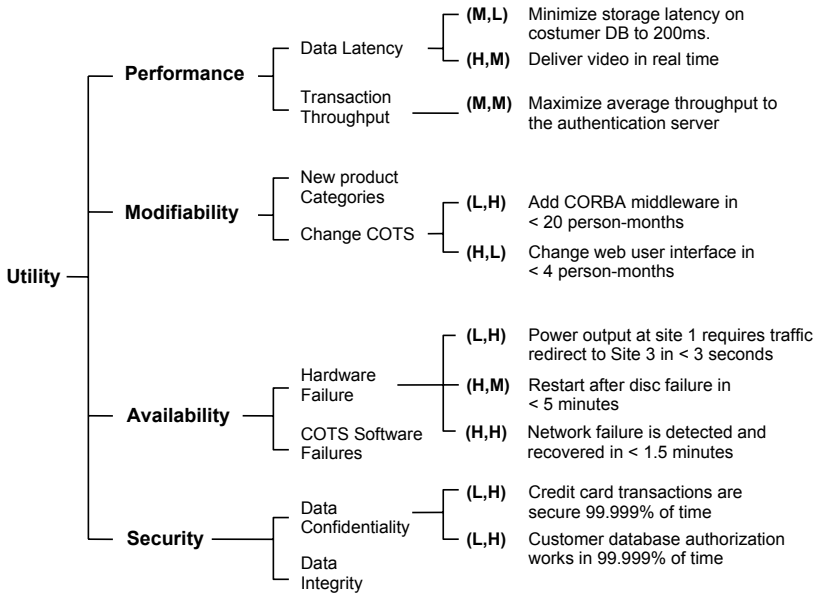

Figure 7.1: Sample of an utility-tree [15, p. 51]

Step 6 - Analyze the Architectural Approaches Now it is time to compare the architectural approaches collected in step 4 with the scenarios from step 5. The purpose of this step is to see how well they match, and identify risks, sensitivity, and tradeoff points. For each individual scenario that is analyzed, the architect will have to identify architectural approaches that helps satisfy the scenario. Hopefully, all the architectural approaches have already been collected. If any new approaches come up, one must analyze why, as it might indicate that step 4 has not succeeded in collecting all relevant approaches.

Once the relevant approaches for the scenario in question have been identified, the evaluation team starts asking questions to understand the approach in detail. Questions are focused on the quality attribute the scenario focuses on and are normally collected from literature and previous experience. The questions help the evaluation team identify known weaknesses with the approach, as well as risks, sensitivity, and tradeoff points. The use of questions is not meant to be a straight Q&A ses-

sion, but a basis for discussion. As possible risks, sensitivity, and tradeoff points arise, further analysis might be needed to investigate it further. These analyses are not meant to be thorough or detailed; a "back of the envelope analysis" is in most cases good enough. The architect might also be unable to answer detail questions about how the approach is applied, and in that case there is not enough information to even perform such an analysis.

At the end of the ATAM analysis, the goal is to have a list of sensitivity points and tradeoffs associated with each scenario, and each of these should either be classified as a risk or nonrisk. The output of this step will for each analyzed scenario be a list of architectural approaches that relates to the scenario, including a reasoning on how the approaches contributes to accomplish the scenario in question. In addition, a list of sensitivity, tradeoff points, and risks related to the scenario and the applied architectural approaches will be provided.

These artefacts will provide architectural reasoning and documentation on how the architecture supports the most important quality attribute goals, as it is created directly on the basis of the utility tree that contains the most important scenarios for the system.

Step 7 - Brainstorm and Prioritize Scenarios At this point, as many as possible of the people involved in the project will be present. The idea is that everyone should be able to provide well-defined scenarios that are important to them. It should be done as a brainstorm, where everyone should be able to get their scenarios on the list. The participants are also encouraged to add scenarios from the utility tree that has not been analysed to the list of brainstormed scenarios. Any similar scenarios are merged before the list is prioritized. Voting may be one way to perform the prioritisation. The top ranked scenarios are then added to the utility tree, and any scenarios related to multiple branches in the tree is split and adapted to each of the branches.

The result of this exercise is that the list of scenarios produced by the smaller group of architects and developers is tested against the larger group of participants. Either the new scenarios add variations of the existing scenarios to the tree, or they create completely new branches in the tree. The creation of new branches, especially if new quality attributes on the second level in the tree have to be added, should be noted as a risk. Additions of such nodes indicate that there exists an area of important quality aspects that has not been considered while designing and analyzing the architecture.

Step 8 - Analyze the Architectural Approaches This is the second testing step, where the top ranked scenarios from the previous step should be demonstrated on the architecture. The architect will guide the group through how the architectural approaches help realizing the scenario. Basically this is a repetition of step 6 but on a different set of scenarios. Hopefully this step will not reveal many new architectural approaches or risks, sensitivity points or tradeoffs. A huge number of new discoveries at this point would indicate that the previous steps have not been carried out properly.

Step 9 - Present the Results Presenting the result of the analysis is the last step of the method. This step recaps previous steps, and lists the outputs and findings. All the steps should have been documented along the way and provide the basis for a report and/or presentation in this step. This step produces a list of risk themes that is produced by the evaluation team. This is based on the fact that risks normally can be grouped together based on an underlying concern or systematic deficiency. Each of these themes are then associated with the business drivers from step 2. This gives closure to the method and brings the discovered risks to the attention of the management.

7.4 Traditional risk/safety methods

A large variety of traditional risk/safety methods exists, and Clifton Ericson lists no less than 22 different techniques in his book Hazard Analysis Techniques for System Safety [28]. In the following sections a few of these methods will be presented.

Later I will present a method where the FMEA technique described below is applied together with Jacobson's robustness analysis.

7.4.1 PHA

Preliminary Hazard Analysis (PHA) is a method used to identify and collect hazards in a system in order to form the initial system safety requirements. The method focuses on early stages of design, when little design is available. The method analyses already identified hazards as well as help discover previously undiscovered hazards. Input to the method is knowledge of the current design, knowledge about potential hazards, and mishaps (typically in the form of checklists and lessons learned from similar systems), and lists of known hazards.

PHA starts by looking at the list of known hazards and then combines the design with checklists and other forms of prior knowledge in order to find any hazards that are not already on the list. Checklists and other materials help stimulate thought and makes discovery of new hazards easier. For each individual function and part of the system one should develop each individual hazard further by identifying the potential causes for the hazard and the worst case effects and consequences the hazard has on the system if it should occur, and in what operating modes of the system the hazard is a concern. Then each hazard should be associated with a severity and probability for the hazard based on the causes and effects. Recommended preventive measures to help reduce the severity or probability through design, safety devices, warnings, or procedures should then be noted and last, a new estimate of severity and probability if the recommended measures are implemented.

PHA is not designed to be a one-time procedure that is carried out from A to Z, but a repetitive procedure where new hazards can be added, and worksheets can be revisited as the design progresses.

The main outputs from this method is a list of safety requirements in the form of preventive measures, an updated list of hazards, sources of hazards and an initial risk assessment for the system.

7.4.2 HAZOP

HAZOP, or Hazard and Operability Analysis is quite similar to PHA in form, but have its origin in the chemical industry. The method can be used both in preliminary design and in detailed design. Instead of using checklists like PHA, HAZOP uses key guide words and system diagrams. It is also a more time consuming method as brainstorming is a key element. A group of multidisciplinary experts carry through the analysis, and having a qualified team leader and a relevant selection of experts is an important key to success. Instead of focusing on hazards directly, the focus is on identifying potential deviations that could lead to hazards.

A worksheet is recommended, and the analysis is done on individual parts of the system. For each item its intended function and purpose is described, then combinations of parameters and a guide words are investigated in order to find possible deviations. For example, the main function of a water pump in a car is to circulate the water between the engine and radiator in order to provide cooling; what if less water is circulating, or the water temperature rises. For each of these combinations of properties/attributes and guidewords a consequence and potential factors that could contribute to the cause should be described. In addition a potential hazard should be identified if the combination has hazardous consequences. Last, an estimate on the severity and probability should be stated together with any recommendations on how to mitigate the hazard.

7.4.3 Fault tree analysis

If you have a single undesired event, and want to know the root causes and probabilities for this event, then the fault tree analysis (FTA) can be used. It helps you analyse large and complex systems to determine what combinations of events that can cause the undesired event, and lets you calculate the probability for a single combination or the probability for the undesired event occurring. FTA is a top-down approach built on logical gates and fault events that models the cause-effect relationships that can lead to an undesired event. The approach can be applied in a recurring fashion in order to improve the model as more information or details are made available, and multiple models can be built to analyse multiple undesired events. Building the model is an iterative process. It starts at the top with the undesired event. Next, a list of normal-events and failures that could lead to this event is described. These are linked through a logic gate telling if all events and failures need to happen in order to produce the undesired event (AND gate), or if only one of them needs to be represented (OR gate). Then each of the individual failures and events are decomposed in the same way. This is repeated until all events are defined in terms of basic identifiable hardware faults, software faults and human faults. The logical gates are not limited to simple AND and OR, but more complex gates like conditionals, Priority AND and Exclusive OR are also possible to use.

Using the strict logical model, one can easily define sets of events that lead to the undesired events, and calculate probabilities for each of them. The method requires some training, but is quite easy to perform once it is understood. One issue with the method is that the graphical model can get quite

large for complex systems.

7.4.4 Event tree analysis

Event tree analysis (ETA) is in a way the opposite of the Fault tree analysis. It lets you start with an initiating event and analyse the sequence of events that might lead to a potential hazard or accident scenario. Like FTA the method uses a graphical tree, although a bit simpler than the FTA trees. The main objective of an ETA is to analyze if the occurrence of an initiating event is sufficiently controlled by the safety mechanisms in the system so that it does not result in a serious mishap.

The method starts with an initiating event that could lead to a potentially dangerous situation, and then the list of pivotal events that should prevent the eventual failure is identified. This requires that the accident scenarios and the pivotal elements are found before the analysis is started. Given the initial event there is normally two possible outcomes of the first pivotal event, either a failure or success. For each of the output one moves on to the next pivotal element to determine if it is able to help preventing and the possible results. This is done for each of the pivotal events and will give an event tree. One example of such a tree can be seen in Figure 7.2

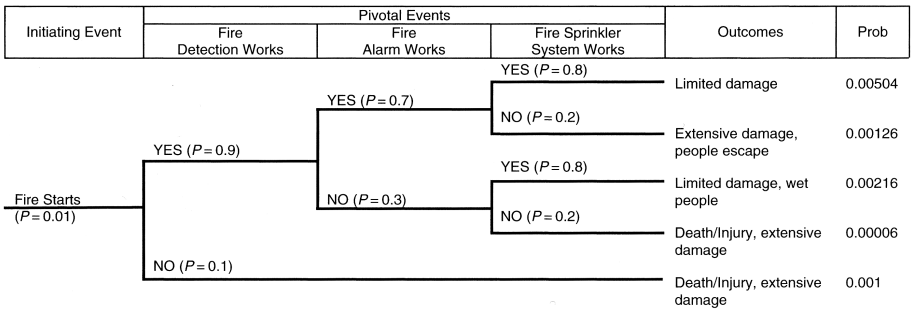

Figure 7.2: Sample of an event tree

Since each of the results from a pivotal event are mutually exclusive, a probability can be associated with the result. That makes it possible to calculate a probability for each of the possible outputs. These can be used to evaluate whether or not the system has an acceptable risk profile or not.

7.4.5 FMEA

FMEA (Failure Mode and Effects Analysis) is a tool to analyse failure modes that can affect the reliability of a system. It can also be extended to analyse failure modes that can lead to an undesired system state like a hazard. FMEA is a bottom-up approach that uses the detailed components and functions to analyse the effect of different failure modes. The fact that the lowest level should be used does not require a detailed design to be in place. The FMEA method can be used at any phase and on any level of abstraction, you just work from the bottom up on the level of detail an abstraction you have chosen. As for PHA

and HAZOP, it is recommended to use a worksheet in order to more easily adhere to the process.

FMEA starts by processing the low-level items in the design (component, function or subsystem), and for each individual item, the possible failure modes should be listed. Sample failure modes for functions are: fails to perform, performs prematurely, and does not fail safe. For hardware, sample failure modes are; cracked, bent, failure to operate, and short circuit. After having noted potential failure modes for each item, the failure rate or probability of failure for the items failure mode should be estimated. Then all the potential factors that could cause the failure mode should be identified. The immediate effect should then be determined; this should be the low-level effect that occurs on the next item in the current design. If there is a system level effect, that should also be noted. Method of detection should be noted to describe how the method of failure can be detected before it has caused any major accident. Current controls should then be noted, to describe any measures that are in place to either prevent the failure from happening, or prevent it from having any major consequences if it should happen. If there are any potential hazards as a result of the failure mode, it should be noted as well as the severity and probability for the hazard or mishap occurring. Last, a recommended action to eliminate or mitigate the effects of the failure mode should be noted.

There are some important shortcomings of the method one should be aware of, and that is that the method is not designed to identify hazards that does not occur as a result of a failure mode. The ability of the method to examine human errors and external influence and interfaces are also limited.

7.4.6 Safety methods applicability to software

None of the methods described above were initially designed to be used on software from the start, and may not be as suitable for software applications as the physical systems they are designed for. Independent of this the described models have methodologies that helps in discovering and analysing systems with focus on causes, effects, events and hazards.

7.5 Combining FMEA and Jacobsons analysis method

Zhou and Stålhane [78, 79] present a robustness analysis framework that can be used in the early analysis and preliminary design phases of web systems. The framework uses the Jacobson's robustness analysis method in combination with FMEA to find system elements that are critical for achieving robustness, and identify preventive actions. Jacobsons analysis method is used to model the system with focus on its behavioural aspects. The proposed method starts by defining the robustness requirements for the system. Once this is done, the system is divided into subsystems by focusing on the most important use cases. Each use case is then analyzed using Jacobsons analysis to identify the objects involved and classify them as one of three types:

- Boundary objects, that are the objects the user interacts with when communicating with the system.
- Entity objects, that represent storage or objects from the domain model.

- Control objects, that link boundary objects to entity objects and normally contain business logic and rules.

Control objects serve as the target of analysis as these are the only way for a boundary object to communicate with the entity objects. For each use case a light-weight FMEA should be performed on all the control objects. The result of this analysis should be connected to the goals of the system and prioritized.

7.6 TRIAD

Trustworthy Refinement through Intrusion-Aware Design, TRIAD [26, 55] is a framework for designing survivable architectures. The approach is based on the creation of a survivability strategy. This strategy should address how to handle mission-compromising threats and attacks. This could be done through resisting, recognizing, recovering from, or adapting the system to attacks. The strategy has to be kept updated to reflect any changes in the threat situation and critical mission goals of the system. Examples of changes that could require an update of the strategy is a higher dependability of the system in the organization or the discovery of a new type of attack. The rationale behind creating the framework is that many existing solutions to handling threats focus too much on details and produces only individual solutions for individual types of threats. TRIAD tries to focus at a higher level and create an overall strategy to handle the threats.

The method uses inspiration from Boehm's spiral model, and has three sectors in the spiral:

1. Architectural strategy.
2. Architectural instantiation.
3. Environmental analysis.

One starts in the first sector and walks through the other sectors before doing a new round in the spiral starting with the first sector again. Each iteration refines and develops the system based on risk analysis, prototyping, risk mitigation, new knowledge, and experience from previous iterations. The iterations stop when the resulting product of each sector is at a level that gives an acceptable risk profile for the stakeholders. The first iterations focus on creating the survivability strategy, once that is complete focus shifts over to technical refinement. Both new development and maintenance on legacy systems can be performed using the model. It can either be fully integrated into the development process or be run as a separate effort early in the early stages of development.

1 - Architectural Strategy Based on a definition of the system's overall mission, this step forms survivability requirements and a high-level conceptual survivability architecture describing structure and functionality. The basis for the requirements and the conceptual architecture is that the system should be able to perform its mission despite penetrations and compromises. Conceptual architecture should be described and presented in a form which is easy to understand for the customer.

A set of survivability tactics forms the conceptual survivability architecture, where each tactic is "a generic representation of an architectural approach to resist, recognize, recover from, or adapt to a pattern of attach in a given context" [55].

2 - Architectural Instantiation This sector transforms the conceptual architecture into a technical implementable architecture. It is done by instantiating the conceptual architecture using low level technical components. Instantiation might result in changes or feedback on the technical feasibility of the conceptual architecture, and the resulting description of function and structure should be at a level of detail that is suited for implementing the architecture.

3 - Environmental Analysis After having a conceptual architecture and a suggestion for its technical implementation, one can evaluate this against the expected threats and the main objective of the system. Based on the suggested survivability strategy one can check if the system will be able to carry out its mission given the expected threats. Threats can be everything from a small denial of service attack to driving a bulldozer into a building.

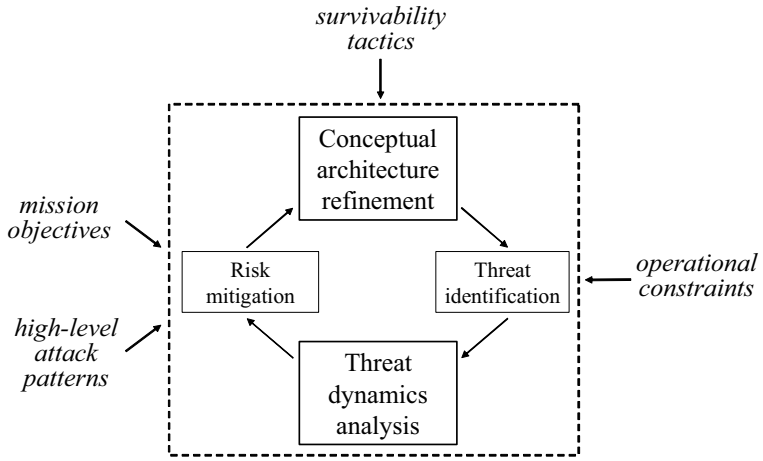

Figure 7.3: Survivability Strategy Refinement Process [55]

TRIAD has a high focus on traceability. Thus, each mission objective has an associated list of possible threats. For each of these threats there is a list of survivability requirements that are realised in the conceptual architecture. The same chain is also traceable from the conceptual architecture and back to the mission objectives. Traceability like this is valuable when changes to the system are to be performed; mission objective changes or new threats are discovered. In all these situations traceability helps by listing the areas that will need analysis or changes.

Survivability strategy consists of the combination of survivability requirements and conceptual architecture. Development and evaluation of these make use of a *system dynamics* approach. Threat dynamics is a modification to the

system dynamic approach that includes hostile actions and the system's operational response to the actions. This special kind of modelling is used to model and understand the structure of complex human-based systems. "... threat dynamics provide an overview of the general influences that the threat environment has on the ability of the system to fulfil its mission and better understanding of strategic responses to counter likely threats" [55]. The approach to refinement of the survivability strategy can be seen in Figure 7.3. This kind of modelling is powerful and helpful in a threat context, but less interesting in a robustness context.

7.7 Reviews

IEEE defines a review as:

Definition 7.7.1. review. A process or meeting during which a software product is presented to project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval. [1]

IEEE [1] has defined five main types of (systematic) reviews:

- Management reviews.
- Technical reviews.
- Inspections.
- Walkthroughs.
- Audits.

Reviews are mainly a tool that can help in verifying that requirements are met, and that the result meets the required levels of quality. The quality attributes and requirements are inputs to the reviews. Reviews can not only be carried out on to the source code, but also on all documents and other by-products of the development process like design documents, contracts and release notes. Reviews can also be performed at various levels in the organization from the management down to a peer review. According to McConnell [53, p. 573], the main idea behind technical reviews is that developers are quite often blind to their own errors. Others, however, spot quite a few of these errors them self, and it is often enough to just try to describe a problem to another individual in order to spot a solution or error. McConnell also emphasize that reviews are effective in discovering bugs. He refers that testing has an average defect detection-rate of 25-45 percent, but reviews has a rate of 55-60 percent. This, in addition to that reviews can be performed at earlier stages than testing, makes reviews a cost-effective solution to keep defect counts down. Because of this it can also be used as a measure to reduce development costs.

Although there is a defined distinction between different kinds of reviews, there is not always a clear way to classify a particular kind of review. In the rest of this thesis, the term review will therefore be used for all different kinds of reviews. The following sections describe the five main types of reviews in the IEEE standard.

7.7.1 Management review

The main objectives of a management review are to ensure and monitor progress, recommend corrective actions and ensure a proper allocation of resources. The typical attendants are management, technical leadership and peer mix. The review can be used to evaluate e.g. software acquisition, development and software maintenance processes. Typical items under review are reports like progress reports, technical review reports, and audit reports and plans like software safety plans and installation plans. The main characteristic of management of reviews is that they operate on a high level, and has focus on evaluating plans, progress and output of other reviews.

7.7.2 Technical review

A technical review evaluates the software product's conformance to specifications, standards, guidelines, plans and procedures. In addition it verifies that changes to a software product are implemented properly and affect only the relevant parts of the system. Another common use of technical reviews is to examine and provide a recommendation from alternatives. Participants in these kinds of reviews are technical leadership and a peer mix. The review evaluates items like software requirements, design descriptions, maintenance manual, and user documentation. The main focus of this kind of review is to look for and handle issues at the design and requirements level.

7.7.3 Inspection

Inspections are more about the details than the technical reviews are. The main objectives are to find anomalies, examine alternatives and verify the product quality. The same items as reviewed by technical reviews can also be reviewed in an inspection. An inspection may in addition look at the actual source code of the product. It is common to use a checklist to help find common errors during an inspection. The typical group in an inspection is a set of peers, possibly accompanied by a trained facilitator. The focus is on detecting abnormalities, and how to handle them, but not on the actual solution. Also, the area of focus tends to be much smaller and more detailed than in a technical review.

7.7.4 Walkthrough

A walkthrough's main objectives are to find anomalies, examine alternatives, improve the product, and it can be a forum for learning. The items under review are the same as for inspections, but the focus is different. In a walkthrough, a solution or part of a solution is presented by its author for a larger collection of people. People involved are normally a peer mix and the technical leadership.

7.7.5 Audits

Audits may cover all types of items, and the main difference from other types of reviews is that they are performed by an external auditor. An audit makes it possible to independently evaluate compliance with objective standards and

regulations. Typical participants are auditors, management and technical personnel. The process is controlled by an external lead auditor.

7.8 Prototyping / proof of concept

Prototyping is a way of testing concepts or designs. A prototype implements some part of a solution. Prototypes can vary from being completely functional to being just a mockup.

Bardram et al. [6] mention that prototyping can be used as a way to explore and experiment with various patterns, features and architectural styles. They illustrate architectural prototyping through three distinct cases. It is also mentioned that it is more the process of building the prototype than the prototype in itself that is of interest. It is during the development and testing of the prototype that you gain knowledge.

Prototyping has also been mentioned by several of the interviewees, and this is the reason why it is included in the overview of architectural design and analysis methods. Prototyping is not as well defined as the rest of the methods, but seems to be commonly used during development.
Possible uses of prototypes according to Bardram et al. [6] are:

- Explore and learn about the architectural design space.
- Test and verify that the design fulfills the quality attribute requirements.
- Prototype parts of the architecture to reduce risks. By prototyping one can test the suggested design, and verify that it is buildable and suitable.
- Prototypes can serve as examples to learn others how a system should be built.

7.9 Evaluation of criteria

In Table 7.1 a comparison of each method with the individual criteria is shown. To reduce the size of the table, some simplifications have been made. Audits and management reviews have been omitted from the table. Management reviews focus more on processes and results than the design, and audits is a special type of review that has a more formal focus than other methods. Further, the other generic review methods have been combined into a common column as their properties are more or less the same.

The details of the comparison are shown in the table, and I will therefore only discuss it briefly. I will get back to the adaptability to robustness for most of the method's in Chapter 9, where I will suggest a method based on a combination of some of the methods presented in this chapter.

The table illustrates that methods have strengths and weaknesses. Based on my analysis, no method seems to be superior. TRIAD is too focused on security; SAAM is very focused on maintainability and flexibility. Reviews are more general characterizations than concrete methods, making them very flexible, but requiring specialization to be useful for robustness. Prototyping is a general activity that can be used to test designs, but does not focus on robustness in particular. FMEA and Jacobson's analysis focus on early stages of

design, where little design information is available. I am looking for a method that is usable during the whole design stage. The traditional safety methods are promising, but need adaption to be used on software. ATAM has many good qualities, but I suspect that it does not scale well for small teams. One possibility is to remove some steps, but the consequences of this are unknown.

Criteria	Importance	SAAM	ATAM	Safety methods					TRIAD	Technical review / Inspection / Walkthrough	Prototyping	Jacobsonsons and FMEA
				PHA	HAZOP	FTA	ETA	FMEA				
Evaluation or design method?	-	Evaluation	Evaluation	Evaluation	Evaluation	Evaluation	Evaluation	Evaluation	Both	Evaluation	Both	Evaluation
Usable in various stages of design	**	Requires some design details	Requires some design details	Method is meant to be used before details are available.	Requires some details, but can adapt to various degree of detail.	Method requires that the design is available and is designed to be used on detailed design, but can adapt to multiple levels of detail.			Spans all stages of design.	Can be used on various levels of design.	Can be used on various levels of design.	Is designed to be used before much detailed design information is available.
General, not for a special kind of application	**	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Does require a special form of architectural description	**	No	No	No	No	No	No	No	No	No	No	No
Supports analyzing a single and multiple quality attributes.	*	Best for maintainability, ATAM is suggested for other uses.	Specialized for analysis of multiple attributes, some steps could be superfluous when used for a single attribute.	Method is designed for safety, but cause of hazards can be related to different attributes like maintainability, safety, performance or security.					Partly, see **	Yes, depends on the reviews focus	Yes, depends on the main focus of the prototype.	No, focus on robustness only.
Integrates in an existing process?	**	Yes, standalone process	Yes, standalone process	Yes, standalone process	Yes, standalone process	Yes, standalone process	Yes, standalone process	Yes, standalone process	Can be hard to integrate	Yes, standalone process	Yes, standalone process	Yes, standalone process
Example of use for robustness is found	*	No	No	No	No	No	No	Yes, in combination with Jacobson's	No	No	No	Yes
Scales with ambition	**	Could be difficult, see *		Yes, the effort used can vary.	Yes, solution can be analyzed in various levels of detail.	Yes, solution can be analyzed at various levels of detail.	Yes, solution can be analyzed at various levels of detail.	Yes, solution can be analyzed at various levels of detail.	Not directly, method requires a substantial base effort due to extensive traceability.	Yes, can adapt to the time available and size of the team.	Yes	Yes, solution can be analyzed in various levels of detail and in groups of varying size.
Supports iterative use	*	Yes, can be used multiple times and at multiple stages	Yes, can be used multiple times and at multiple stages	Yes, can be updated when new information is available.	Possible to re-run the process or parts of it in various iterations.	Possible to re-run the process or parts of it in various iterations.	Possible to re-run the process or parts of it in various iterations.	Possible to re-run the process or parts of it in various iterations.	Yes, method in itself is iterative.	Yes	Yes, prototypes can be built at various stages and be built iteratively.	Yes, can be used multiple times as new or more info is available.
Easy to learn and use	**	Requires some training, and a leader is most likely needed.	Requires some training, and a leader is most likely needed when using the method.	Yes, but requires some experience to be able to list risks and hazards.	Quite easy to learn, but requires a skilled leader.	Quite easy, may require some training. Easy to use when understood.	Yes	Yes, when understood the method is easy to use	No, requires substantial training, and parts of the method is difficult to use (threat modelling)	Yes	Yes, similar to regular development.	Yes
Involves teamwork	**	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Can analyze causes of errors and their result	**	Yes, by focusing on it in scenarios.	Yes, by focusing on it in scenarios.	Focus only on possible effects.	Yes	Analyses causes of a single hazard.	Analyzes effects of a single failure.	Yes	Yes, trough threat modelling.	Yes, by focusing on it	No, but can verify possible designs.	Yes
Documents result	*	Yes, report	Yes, report	Yes, risk and hazard list.	Yes, worksheet	Yes, worksheet	Yes, worksheet	Yes, worksheet	Yes, specifications, strategies, requirements.	Yes, a report can be written.	Yes, a report can be written.	Yes, robustness model and worksheet.
Makes use of prior knowledge	*	Yes, knowledge about benefits and drawbacks of architectural solutions can be used.		Yes, lists of risks and hazards.	Yes, list of guide words and changes.	No, not directly	No, not directly	Yes, possible types of component failure.	Yes, architectural strategies.	Yes	Not directly.	Yes, lists of failure modes.

*) The method scales with number of people involved, but involves a number of steps which would require some effort to complete. Especially for ATAM, some steps might need to be removed when little effort should be used. Another alternative is to focus on just some parts of the architecture.

**) Method is built for trustworthiness. Supports analyzing both single and multiple attributes related to trustworthiness, but is not very suitable for other attributes.

Table 7.1: Comparison of methods

Chapter 8

Related research on robustness

In this chapter I will briefly look at some areas of research that either focus primarily on robustness or is of relevance to robustness. The purpose of the chapter is to provide an overview over the broad spectrum of relevant research being performed. Presented approaches vary too much in scope and form to make an comparison of the methods valuable. Instead I will briefly comment the applicability of the individual approaches.

8.1 N-version programming and robustness

Although an experiment by Knight and Leveson [45] has shown that N-version solutions in software do not increase reliability, many approaches similar to N-version programming can be found. N-version programming is based on creating several independent implementations of some part of a system. It could be a module, an algorithm or the whole system. At runtime, one or more implementations recieve identical input and are run in parallel. The results are then compared, and the majority result is considered correct. Another alternative is comparing results and abort if implementations have not provided the same result. The experiment conducted by Knight and Leveson concludes that different implementations do not fail independently. If individual versions had failed independently, combining multiple versions would reduce the probability of failure drastically. The study indicates that independent versions have similar faults, and independent version thus do not fail independently. Critics have been raised[46], but I have not been able to find any conclusive answer to whether the results shown by Knight and Leveson is also valid for other systems or not. Based on this it is not evident that an N-version approach alone will result in a dramatically reduced probability for failure.

Huhns et al. [38] show various pre and post process approaches for the usage of independent versions of algorithm implementations. Pre-processing adds the possibility of choosing compatible versions and only executing versions that are compatible with the task at hand. The article suggests adding agent capabilities to the algorithm. The use of agents tries to solve two issues with the pre and post processing approaches; faults in the pre or post step

could give wrong results, and the adding and removal of algorithms could be problematic. Using an agent based approach the group of agents can totally or partially replace the pre and/or post process steps. The functionality previously in the pre and/or post process step is replaced by agent collaboration. Other more complex ways to use agents also illustrated by Huhns et al. is collaboration, where individual agents can work on parts of the solution to produce the total result.

N-version approaches using agents or not, require the development of multiple versions. This development comes with added costs. Pullum [59] shows that experiments suggest that the cost of an N version approach is less than n times the cost to develop a single version. Pullum also discusses other issues with the N-version approach. Errors in common specification might influence all versions; different versions might contain common errors and running multiple implementations might affect performance. These are just some of the issues mentioned by Pullum. For a more complete discussion I refer to her book.

In the light of developing more robust systems, the ability to have alternative implementations that possibly collaborate to perform tasks is promising. If one succeeds in the development of alternative versions that fails independently. Just the ability to be able to use an alternative version if the first one terminates unexpectedly will be of great use. The basic notion of alternative versions is the foundation of fault-tolerance and redundancy. However, in fault-tolerance and redundancy all these alternative versions do not have to provide the same functionality. Combining these efforts into an approach that can collaborate, both have redundant and degraded functionality could be valuable to robustness. This combined effort gives multiple possibilities to prevent failure if a fault should occur. The combined effort comes with a price: increased complexity. It is an open question whether the issues that come with the added complexity outweigh the functionality N-version programming gives or not.

8.2 Increasing robustness using self-adaption

A related approach to N-version development and agent based systems is the notion of self-adapting systems. The main idea is that the system should be built in such a way that it is able to monitor its own condition and possibly being able to adapt to improve its condition at runtime [49, 56]. The main goal of self-adaptive software can be formulated as "the creation of technology to enable programs to understand, monitor, and modify themselves"[49]. Multiple approaches to supporting self-adaption exist in literature. Some approaches I have found interesting are the general approaches by Oreizy et al. [56] and Schmerl and Garlan [62]. Both these have a system level focus, where a mapping between the architectural description and the implementation is used actively to make changes in the architecture that is reflected in the running implementation. Also the component centric solution presented by Shin [67] could be an interesting approach for increasing robustness. Shin's approach is that individual components in a distributed system have a service part and a healing part. The healing part monitors objects in the service part based on notifications. If the healing layer detects an abnormality, it switches to healing mode and notifies other components about the situation so they can reconfig-

ure accordingly. In healing mode the component will repair itself, test that the repair succeeded and then move back into normal service mode. The general idea of self-adaption is promising, but seems to be in its early stages of research. Digging deep into the field is outside the scope of my work, and is left for further work.

Systems that monitor themselves and adapt to changes in the environment or based on usage or fault patterns gives increased awareness of own limitations and capabilities. An application that is knows how much load it can stand before performance drops below the acceptable level, and monitors current load in order to reject requests if the load gets to high will have a predictable behaviour. This predictability may lead to a higher robustness if the ability of the application and the expected load is compatible.

8.3 Robust datastructures

Another approach to achieving robustness is adding robustness to data structures as described by Black et al. [9]. The idea is to create data structures that make it possible to correct and detect errors. Traditional data structures have little support for this. For instance, a single-linked list is impossible to repair if a pointer to the next item gets corrupted or erased. It is also not trivial to discover the corruption of the list. A double linked list with information about the number of elements in the head of the list is a more robust structure. In the paper they describe data structures by the number of ”illegal” changes the structure is able to detect (N-detectable) and the number of changes the structure is able to correct (N-correctable). A collection of designs for data structures is also presented, including robust versions of linked lists and b-trees. The paper is from 1995 and since then the amount of memory corruption has lessened as a result of controlled execution environments like Java Runtime and Microsoft's Common Language Runtime where usage of raw memory pointers is less common. The thought about including mechanisms in the structures that allow detection of errors and possibly correction is, however, just as relevant today.

8.4 Testing for robustness

A number of approaches for robustness testing exist. DeVale et al. [19] give an overview of several methods and approaches. Some automatic methods are fault injection introducing hardware and software faults; noise introduction; writing random data to memory before spawning a large number of processes. Most of these methods suffer from lack of repeatability. Traditional testing approaches could also be used, but often testing is based on specifications, requiring that detailed robustness requirements must be included. Specifications, however, normally focus on the functionality to be provided and not on robustness. Another important aspect according to DeVale et al. [19] is that measures like test coverage can be misleading. These are able to verify that error handling code is executed, but fail to discover if some code that should have handled some erroneous situation is missing. Some tools exist that inject

faults and monitors the execution of the application to reveal robustness issues, but this often requires source code which may be unavailable.

Based on the lack of a suitable method that provided repeatability and handled third party components without source code, Ballista was introduced. Ballista [19, 48] is a testing service developed for robustness testing of software with focus on robustness to external input. It consists of a centralized server that generates tests and a lightweight test runner that performs the tests and communicates with the server. The test runner normally spawns new processes to perform tests and monitors the spawned process for hangs or abnormal exits. This design makes the testing service easy to port to new platforms as much of the logic can be contained in the server part. The test runner uses module level testing by calling functions and routines in the module with various inputs.

A typical problem with testing is the large number of possible input values. In order to fully test a function that has one byte argument, 255 tests has to be performed. A function with three 16-bit integer inputs and a string has an enormous number of possible tests. Running all of them for a single function could take years on a fast computer.

Example 8.4.1. A function taking three 16-bit integer inputs would take almost 9 years to fully test on a computer that is able to perform 1 million tests every second.

A complete test could guarantee that a function is free from input based robustness issues, but this is not possible to do in practice. Ballista has employed a different approach. Each function argument has a type. For each type a list of values is created. The list could contain a combination of values that often causes problems like maxint, minint, zero and one, and a random selection of other values. When a function is to be tested, one can generate combinations based on this limited list of values for each argument. For the function in Example 8.4.1, the number of tests would be reduced from $2.8 * 10^{14}$ to 8000 tests if the list of values contains 20 items. Another alternative could have been to use random values. Random numbers do not produce repeatable tests, making it difficult to compare two test runs. The other issue with pure random tests is that if you run a low number of tests, there is no guarantee that problematic values like maxint, minint, zero and one are included in the tests.

Even the limited list of input values used by Ballista generates a huge number of possible test cases and these can take too much time to run. A suggested approach to reducing the number of test cases even more is by doing adaptive testing [20]. Normal static test case generation test all combinations of parameter values and evaluates the results afterwards, adaptive testing analyzes and adapts the list of tests based on previous results. Single parameter values can always cause a fault, by identifying these early in the test run one can ignore other tests involving these values, and reduce the number of tests and effort while maintaining an acceptable level of precision.

Ballista does not make use of specifications. The purpose is not to test if the result is functionally correct, but to test for non-robust behaviour. Any test that does not result in a crash or hang is a successful test. This simplification can be done since the purpose is to test for robustness issues, and a crash or hang is an indication of a robustness problem. The CRASH severity scale is used to characterize the faults [48]:

C atastrophic is when the OS gets corrupted or reboots.

R estart is when the testing process hangs and needs to be killed.

A bort is when the testing process exits abnormally.

S ilent is when the test silently fails. This is when an error code or indication is expected but not occur.

H indering is when the wrong error code is returned.

Ballista is not able to discover Silent or Hindering failures as this requires semantical knowledge of the function.

In order to perform testing, Ballista requires a definition of the functions to be tested. This definition should list the arguments and the data type for each of the arguments should be specified. The data type is not necessarily the physical data type; it could also be the logical data type. For instance, the physical type could be an integer, while the logical type is a file handle represented by an integer pointer. It is possible to make use of hierarchies of data types to simplify implementation of new data types. The file handle type could inherit the value list from integer and then add its own values. As long as the testing server and client have support for the specified data types and the testing client has support for calling the module, no further details or input is needed.

The Ballista approach has been proven to be effective, showing failure rates from 9.99% to 22.69% with a mean of 15.6% across 15 different POSIX based operating systems where 233 function calls were tested [19, 48]. The porting ability has also been demonstrated in a study where Ballista was ported to Windows and used to test the Microsoft Win32 API on six different versions of Windows ranging from Windows CE to Windows 2000 [66]. The tests were done on 237 function and system calls and also include a comparison with Linux. The study indicates that Linux and Windows NT/2000 has less catastrophic failures (none) than Windows 95/98/98 SE/CE and that there are large variations in the number of abort failures across the operating systems and function groups tested. Windows CE has a lower number of abort failures but a higher number of complete system crashes, making it less reliable. The study also indicated that the Windows 95/98/98 SE family of operating systems contained a higher number of silent failures than Windows NT/2000.

A related study testing the robustness of MacOS applications[54] uses a different approach. Instead of using statically generated tests based on parameter types and a limited set of values, it uses random inputs in the user interface (unstructured black box random testing). The method classifies both application crashes and hangs as failures. Both graphical and command line programs were tested. The report is the fourth in a series of test reports. It all started in 1990 with tests of more than 80 command line utilities on six versions of UNIX where a failure rate of 25%-33% was found. A follow-up study was performed in 1995 covering more utilities and operating systems. Graphical applications running under the X-Window system and some standard library interfaces was also included. The study illustrated a failure rate of 15-43% on the command line utilities and 26% on the GUI applications. The open source tools, however, had a lower failure rate; GNU utilities 6%; Linux utilities 9%. In 2005 a

study on Windows NT/2000 was performed and resulted in crashes in 45% of the applications tested. In the MacOS study, 135 command line utilities and 30 GUI applications were tested. Failures were found in 7% of the command line utilities and 73% of the GUI applications.

For applications with available source code, crashes were analyzed and a classification of the types of causes were created (not all categories had failures in the MacOS study).

Failure to check return values - The code assumes that calls cannot fail and does not check return values. It can also be that it is too inconvenient or problematic to handle a failure.

Pointers/arrays - The code use pointers or arrays without having proper bounds and validity checks.

Signed characters - The code uses the numeric form of characters and fails to take into account sign bits and other issues when converting to and from character notation.

Race conditions - The code assumes a sequential execution and fails to handle race conditions that arise when that is not the case.

Input functions - Input from the user is placed into fixed size buffers or variables of specific data types without proper checks of the data.

Bad error handling - Error handling is present, but fails to handle all situations properly.

Interaction effects - The program has some expectation on the input, and feeds it to an interpreter like a format command or SQL query. If the input contains control sequences this might cause errors.

Sub processes - If the application delegates work to another application, it does not help that the app itself can handle the input if the delegated application fails due to the input.

Unit testing is suggested as a testing method for verifying COTS components before using them in a software solution [73]. The Mono implementation of the .Net framework was used to illustrate that a basic suite of tests could find faults in a widely used part of a framework. The Convert class was chosen as the target for testing, and a set of unit tests were written to test it. A total of 25 faults were discovered in the class, ranging from incorrect results to exceptions being thrown. The use of unit tests makes it possible to discover silent failures and incorrect results that testing frameworks like Ballista are not able to discover. On the negative side, the study wrote 2734 lines of test code for a class containing 2463 lines of code, illustrating that writing unit tests requires a substantial effort. Unit tests, however, give the ability to re-run tests without much effort, and also provides a way to test APIs and functionality at a low level. Thus it provides a way to catch errors early in the development cycle.

Three approaches for robustness testing have been presented, and they all have their strengths and weaknesses. Ballista is currently focused on testing single functions and methods, which makes it less suitable for solutions where functions or methods depend on each other. Ballista is able to handle this to

some extent, as demonstrated by the use of a data type for file pointers. If Ballista were to be used for testing an object oriented solution, however, it would most likely not be able to cover enough of the possible scenarios in its current form. Objects often depend on the sequence of calls to their methods since methods and properties of objects often modify their internal state. Just adding the combination and sequences of methods, functions and properties could be a solution, but the number of tests would explode. Unit tests handle this as the tests are manually generated. The type of random testing used for testing Windows, MacOS and Unix described above, does not mind if the solution is object oriented or not as it exercises the external interface of the application. If random testing were to be used for API level testing, one would have to take into account the same issues with state as Ballista does. As development languages and runtime environments have developed, more metadata has been added. Environments like .Net and Java contain information about classes, their methods and arguments. This makes approaches like Ballista easier to use since the list of methods and their arguments can be generated by the use of reflection[1]. A shortcoming of both random testing and Ballista is how they check if a test has succeeded or not. By only looking for hangs and crashes they are not able to discover silent errors, and they are not able to discover if an exception or error code returned is the correct one based on the input. This makes testing methods like these only helpful in determining if the application has non-robust behaviour, not if it is correct with regards to robustness.

8.5 Wrapping for robustness

One of the goals of Ballista was to use the test results to generate or create a robustness wrapper for the API [48]. The efficiency of hardening through wrapping is demonstrated on three systems by members of the Ballista team [18]. In the same paper it is also demonstrated that such checks can be added without a high performance drop through the use of a cache for the argument validity checks. An approach to generating wrappers for improved robustness of Windows software is demonstrated by Ghosh et al. [36]. By the injection of code between the calling code and the library function, the approach gives the ability to modify the return values. This can be used to either test if an application handles the exceptions and return values the function may return, or protecting the application against exceptions or return values the application does not handle. The approach does not require any modification to the original application as the injection of the custom code happens entirely at runtime.

An automatic approach that uses a similar approach for generating wrappers for C libraries called *HEALERS* is described by Fetzer and Xiao [30]. The approach does not modify the library itself, but generates a wrapper for the library based on adaptive fault-injection experiments. By placing the generated wrapper so that it has a greater resolvability than the original library, the wrapper will be called instead of the library. The fault injection focuses on determining robust values for the different arguments of a function, and uses

[1]Reflection is a functionality of the programming language and runtime platform that lets an application introspect itself. It allows for the program to list the classes it contains, the methods each class contains and so on. It also allows manipulation of itself, like calling its own methods dynamically.

approaches like hardware memory protection and adaptive testing to accomplish this. Instead of requiring a list of functions with their definition as input like Ballista, mining of the shared library, manual pages and header files is used to generate the list of functions to test. After testing the list of functions only the functions that have an unsafe behaviour (produce a crash or hangs) is considered for wrapping. The wrappers can be generated directly based on the adaptive testing, or some manual modifications to the generated function declarations can be made. Wrappers work by testing arguments and monitoring the environment so any call with a set of arguments that normally will result in a crash returns a proper error code instead of calling the library function. By only wrapping functions with unsafe behaviour, and providing multiple types of wrappers to be generated, performance overhead can be kept at an acceptable level.

Safety facades are another approach to wrapping. Siedersleben [68] first introduced the method, and Tellefsen [72] has investigated it in more detail. The main idea is that a safety facade is placed in front of components, and acts as the interface for callers. Implementation of the safety facade can be replaced based on demands for error handling placed upon the facade. It is possible to apply safety facades for existing components, or design components so that they are aware of the facade. Building the components so they are aware of the safety facade makes it possible for the component to interact with the facade. If a component is not aware of the facade, the facade will have to do all the work.

The main idea of safety facades is that the facade can centralize the error handling and exception logic. Further, the safety facade should return errors through return parameters on functions or through a predefined set of abstract exceptions. The idea is that either the function call returns and indicates that everything was ok, or it returns an error (either through return arguments, or an exception). If an error is returned, it means that the original call failed, and all attempts to solve the failure failed, and an attempt to cleanup any side effects has been performed. A call with a normal return must not always run without problems inside the facade, but this is ok as long as the facade has been able to resolve the error reported. One example of such an error resolvement is that a transaction is started, a call results in a transient failure, and a rollback is performed before the whole operation is retried and succeeds.

Safety facades can reduce the amount of error handling code required by component users, and provide a structured way to check the limited set of error situations. The compartment approach I will present in Section 8.6 is based on a similar concept, but is more a design approach than safety facades.

Limiting the number of possible errors to handle, reducing amount of error handling code for component usage, and defining errors as a part of the component interface would simplify error handling and force all usage of the component to adhere to the same error handling logic. This would be positive for robustness, as facades can be reused and built upon each other. Further, simplifying error handling code spread throughout the application reduces the chance for errors in the error handling logic.

8.6 Exception handling

Although solving some problems, exception handling is not a perfect cure. Tellefsen [72] investigates issues with exception handling in detail. Interested readers are directed to Tellefsen's thesis for details. I will just list some issues for illustrative purposes here:

- Exceptions are performance intensive, using them for control flow result in suboptimal performance.
- Empty catch clauses prevent exceptions from being raised up the call stack, but might allow the application to continue execution even when the exception has caused an invalid state.
- Too generic catch clauses might catch more exceptions than the code in the catch clause is built to handle.
- Callers of a method have to take into account a wide variety of exceptions that could propagate from a method far down the call chain.
- Incorrect re-throw of exceptions cause that information from the initial exception to be lost.
- Exceptions might disclose internal details through stack and message information to a user[2].
- Exceptions add an additional layer of possible flow of control.

Exceptions are also relevant from an architectural point of view, despite the fact that they are code level constructs. A part of a system that communicates with another part has to handle exceptions in some way. For direct calls between modules created using the same language, this is handled automatically. Techniques used for RPC[3] calls, however, might not handle it automatically. Automatically handled or not is mainly a technical aspect, and transferring this information through return types or arguments is one possible approach.

The interesting part, however, is twofold. First, the list of possible exceptions that could be returned by a method call into a different module could be quite long. Next, the total exception flow in an application can get complicated. This is illustrated by Figure 8.1 that shows possible exception flow in an application at the class level for a single exception type.

In Section 8.5, I presented safety facades that reduces the number of exceptions crossing the safety facade boundary. A different approach are software compartments, first presented for ADA by Litke [51] and later applied on a Java program by Robillard and Murphy [60]. The idea is to divide the application into compartments during system design. For each compartment[4] of the application, a list of exceptions that can be raised to users of the compartment should be defined. These exceptions should be meaningful to the user of the compartment, and be at the same level of abstraction as the compartment itself.

[2] An exception that contain sensitive information like e.g. a database connection string, or part of a concatenated sql statement might give a malicious user enough info to penetrate the system

[3] remote procedure call

[4] the same technique could just as well be used at a component or element level

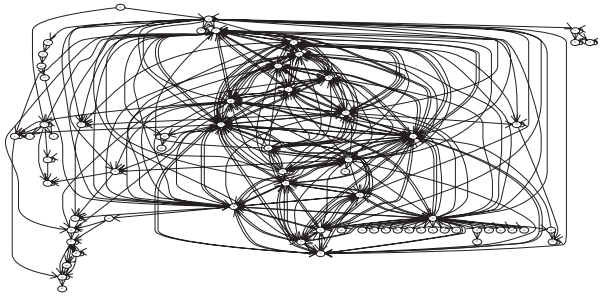

Figure 8.1: Exception flow for one exception type from Robillard and Murphy [60]

The benefit is that users of a compartment have a small defined list of exceptions that they should handle. During the design stage this forces designers to think about possible exceptions that can be reported, and include them as a part of the compartment interface description. To illustrate the idea, think of a component that is used to draw graphs. If the graph component would raise `ArrayOutOfBounds` or `IOException` this would be impossible for the caller to understand or handle. Designing the component to raise `InvalidDataExeption` or `InvalidGraphType` exceptions instead would make the exceptions possible to handle and understand for the caller, and reduce the number of possible exceptions being raised by the component.

Robillard and Murphy [60] have used the technique on three existing applications, and managed to simplify the exception flow. The same technique may also be used at the design stage of the application development. Simplifying the exception structure could lead to simpler exception handling logic, and this could result in less chances for robustness issues to be present in exception handling code. Further, having a list of relevant exceptions as a part of interface descriptions help developers write relevant exception handling code. Informing the user as a result of a `InvalidGraphType` is a logical choice, while catching a `ArrayOutOfBounds` when calling a method to draw a graph is not obvious.

Chapter 9

Observations

Before discussing the current methods further, it is valuable to recap some of the foundations for my study of architectural design and analysis methods.

As discussed in Chapter 6, the interview results clearly indicate that robustness is a wide subject, including existing areas of research like fault-tolerance, maintainability, reliability, security and availability. The definition of robustness and the suggestions on how to achieve a robust solution presented in Chapter 5 vary between the interviewees, but none of them has a narrow definition. Many of the interviewees mentioned that the degree of robustness needed varies between software products, and different customers. This suggests that a method for developing robust applications has to be adaptable and useful for a wide range of applications. Not only does it have to be adaptable to cover the parts of robustness relevant for the application being developed, but the method also has to integrate well into an existing development process. Further, it has to adapt to the effort the organization wants to use on robustness and the size of the team. More important is that a method should be easy to use. A complex method requires substantial training to master and because of this it is harder to adapt in industry. Personal experience from the software development industry indicates that methods that are easy to learn and use, and do not require substantial effort, have a higher probability of being used than methods requiring more time, training, or effort.

Looking through the list of what contributes to the creation of a robust software solution in Section 5.5 reveals that the number of suggestions are high and vary from implementation details to process guidelines. This makes it difficult to find a method that fulfils all the suggestions. Teamwork, focus on robustness, designing for robustness, design analysis, and gaining experience through prototyping are suggestions mentioned by most of the interviewees, and has been the main areas of focus in the evaluation of the architectural design and analysis methods in Chapter 7. Based on the set of criteria presented in Chapter 7.1 I evaluated the design and analysis methods, and concluded that none of the methods fulfilled all the criteria.

This chapter will discuss some of the strengths and weaknesses of the methods presented in Chapter 7 further. I will then suggest a method that combines some of the methods from Chapter 7.

9.1 Current methods

During the literature analysis, several analysis and design methods have been considered. An observation is that robustness is often mentioned as a quality attribute that can be analyzed using a method, but analyzing robustness is not used as an example unless the method is solely made for robustness analysis. It seems like using security, modifiability, performance and the other well established quality attributes is easier and more popular. This makes it difficult to see how the method performs when applied to robustness.

The method based on FMEA and Jacobson's robustness analysis presented in Section 7.5 focuses on control objects and does a safety analysis on these control objects to reveal robustness issues. This method is based on the assumption that relevant issues always are initiated by an interaction with the system. The definition of robustness presented in Chapter 6 also includes internal failures, and they do not always surface at the interaction level but might be incorrectly handled by architectural mechanisms, resulting in corrupt state or behaviour of the system. The difference in the definition of robustness makes the Jacobson's analysis less useful as interactions at a different level of abstraction also becomes interesting. The combination of Jacobson's and the FMEA analysis of control objects is still useful for the class of robustness failures that are initiated as a result of interaction with the system, but does not easily cover the whole broad definition of robustness given by the interviewees. Based on the limited presentation of the method, it seems like it would also scale easily with the size of the group and is easy to use.

In Chapter 7 a variety of architectural design and analysis methods were presented. Some, are mere categorizations of methods, others like ATAM are full analysis processes. Agile and iterative methods are popular these days, and TRIAD is based on an iterative approach starting with a strategy that is transformed to an implementable architecture. The method is designed for designing and architecting systems with respect to threat modelling, and has high focus on traceability between objectives, requirements and the conceptual architecture. The iterative design of a strategy followed by a technical refinement makes it possible to do early analysis on the design and do adjustments and choose alternative solutions before doing a substantial investment in a particular solution. Threat dynamics analysis is rather complex and most likely works best for modelling threats and modeling the ability of techniques to reduce threats. Robustness is more about either handling a case or not, and does not have to cope with an ever changing set of threats. TRIAD in its current form is not directly applicable to design robust software, but the use of a strategy combined with an iterative refinement and structured evaluation is relevant to robustness.

SAAM and ATAM are analysis techniques designed to evaluate quality attributes like robustness. Analysis of robustness is not used as an example in neither of the methods. SAAM is highly focused on the interaction between scenarios in the architecture and how architectural parts contribute to the functionality of the application. This interaction is used to indicate scenarios that require modification of many parts, and parts that require modification from many independent scenarios. This kind of analysis can help participants understand how the architecture works and help spot architectural issues, and scenarios focusing on robustness could be used to further improve this pro-

cess. The use of scenarios is participant friendly because it is easy to understand and feels like a natural way to describe actions and modifications to the system. ATAM is based on SAAM and represents a more complete process that could also be used as an audit. Doing a complete ATAM analysis is quite time and resource consuming due to the combination of a analysis and a testing phase. Focus is on defining the main goals of the systems, form scenarios that represent these goals, and then to identify how the architectural choices interact and form risks, trade-offs and sensitivity points. For a full evaluation of the architecture ATAM stands out as a highly usable method. But to analyze only the robustness of a system, the elicitation phase is not optimal to list robustness scenarios, and the whole process can be time consuming. General ideas presented by the method can be used, and by making some minor adjustments to the elicitation process optimizing it to define relevant scenarios for robustness, the process could be used to analyze all required quality attributes or just robustness.

An alternative to scenario based methods like SAAM and ATAM are traditional safety methods like the methods described in Section 7.4. FTA analysis can be used to determine possible causes and determine probability for a specific situation that might arise, and Event Tree Analysis helps analyzing consequences of a possible initiating event. Both can be useful in determining causes and results of robustness issues that can lead to unfortunate situations in a program system. Robustness and safety share the same challenges, but robustness issues do not always result in hazardous situations that safety analysis is designed to detect and handle. If slightly modified to consider issues due to robustness issues, most safety methods can be used for robustness analysis. The degree of information needed to perform the analysis varies with the methods. Methods like PHA and HAZOP are better suited in early design than FTA and ETA that requires more explicit knowledge about the system's design. The goal is to do analysis based on the current design, and this partly rules out PHA as it is more based on known hazards and prior lists mainly related to the type of system rather than using the design actively. FTA is better for analyzing a hazard than to analyze which hazards the system contains. ETA's main strength is to determine potential hazards an initiating event can lead to, and this can be helpful to determine possible effects of a robustness issue in a component or a part of the system. HAZOP uses properties of system components in combination with guide words to determine how a system responds to deviations. This can be used for analyzing individual components and component interactions in a software architecture. FMEA has been suggested for use in software robustness analysis already and is used in a quite similar way as HAZOP can be used. For each component in the system, lists of possible failure modes are analyzed to determine cause and effect. Failure modes can be triggered by robustness issues, and is these issues that it is interesting to analyze in the software architecture. Are the failure modes gracefully handled, or do unintended side effects occur?

9.2 Proposed method

I suggest a method based on TRIAD (presented in Section 7.6) and (ATAM presented in Section 7.3.3). The method consists of two parts that can be used

together or independently; *a design process* that helps in the design of a robust software architecture, and *an evaluation process* that helps analyzing an architecture for robustness issues. The design process does resemble a typical iterative process, while the analysis process is more specialized toward discovering robustness related issues in architecture. It is a goal of the method to be so flexible that it can be used at several levels of detail and can be integrated into an organization's development process, or used as a separate tool when needed.

The design process is an iterative process starting with a strategy or vision that is transformed into a realizable architecture that can be evaluated. This sequence of steps is repeated in an iterative manner adapting to the analysis results from the previous iterations. The first iterations should focus mainly on a strategy consisting of a rough sketch of the system architecture. As the strategy matures, focus shifts to transforming the strategy into a realizable architectural design.

The evaluation process is a simplified version of ATAM adapted to analysing a system's robustness. It consists of a presentation of the architectural choices that contributes to the system's robustness, and an analysis of these choices with respect to their contribution to the overall robustness of the system. Analysis is done by analyzing a set of "what-if" scenarios inspired by safety analysis on the proposed architecture to reveal shortcomings or issues.

A prerequisite for the method is that the system should have well defined requirements that describe expected system behaviour in erroneous and extreme situations. Requirements should cover the expected external behaviour of the system, which includes both external interfaces and user interfaces. Common questions answered by these requirements are:

- Should the system stop or terminate on a fault?
- Should the system notify the user about faults?
- Should the user be presented with an option to retry a failed operation?
- If the system is overloaded, is a degraded performance acceptable, or should the operation be rejected?
- Are there any situations where a rejected operation is not acceptable?
- Which level of downtime is accepted if an error should occur?
- Should anyone else than the end-user be notified about a fault?

This prerequisite makes the definition of the architectural strategy slightly different from TRIAD. In TRIAD the strategy consists of the combination of the survivability requirements and the conceptual architecture. The suggested method requires that requirements already exist and the task of defining the strategy only needs to consider the conceptual architecture or vision. It is expected that during the creation of the conceptual architecture requirements might need to be refined. This means that the distinction between the architectural strategy and conceptual architecture is kept and that the main focus is on design of the conceptual architecture.

Design and analysis of the architecture need to have these requirements in order to build the architecture in conformance with the expected behaviour.

The requirements form a lower boundary of accepted behaviour, and the designers can choose to build an architecture that performs better than the requirements. It expected that requirements need to be added or refined as the design process progresses, and questions about expected behaviour are raised.

The steps of the design method are:

Step 1: Define an architectural strategy for the system. In the early stages, architectural approaches to the solution at hand should be described without much thought on the realization issues. Typical artefacts created at this stage are rough sketches of the architecture, including the main strategies of the architectural solution.

Step 2: Design the architecture of the system. During this stage, the sketches and strategies are transformed into realizable designs of the architecture. This includes making the changes needed to make the ideas possible to implement, and making a more detailed design.

Step 3: Prototype parts of the architecture. While designing architecture, certain parts are more uncertain than others. It can be an unknown technology, unfamiliar problems or unknown properties of an architectural design. Trying these out is helpful, as it helps evaluating solutions and test concepts.

Step 4: Perform a robustness evaluation. The architectural proposal needs to be evaluated. This step uses an adapted ATAM analysis to check the architecture for robustness related issues, which is done through analyzing a set of what-if scenarios.

Architectural evaluation (step 4) is further refined into:

Step 4.1: Present the architecture. The architecture should be presented with focus on describing architectural solutions handling robustness.

Step 4.2: Define what-if scenarios. A set of what-if / failure scenarios should be created. The purpose of the scenarios is to determine if the suggested architecture responds correctly to erroneous and extreme situations.

Step 4.3: Test scenarios on the architecture. Each of the scenarios should be checked against the architecture. Any scenario which is not handled or handled in an unsatisfactory way should be noted for evaluation and correction.

Step 4.4: Summarize the results. A summarized list of the analyzed scenarios should be created where each scenario should be classified as fully handled, partially handled or not handled. Details from the analysis of scenarions that are not handled or just partially handled should be included on the list.

The summarized list of results and discussions during the analysis process works as input to the next design iteration. A discussion should take place after the analysis is complete, to decide whether a new iteration is needed or not.

This could be to determine whether the architectural strategy is good enough to proceed with designing the architecture, or if the architectural design is ready for implementation.

9.2.1 The design approach

The suggested design approach is based on the TRIAD method presented in Section 7.6. Instead of focusing on system survivability and its threats; focus is on the design of robust software architectures. An explicit step to create prototypes and do explicit trials of the architecture is added. Instead of using threat dynamics modelling to evaluate the system's response to threats, I suggest the use of a simple ATAM like evaluation to evaluate how the system behaves in various situations. An overview of the suggested design process can be seen in Figure 9.1.

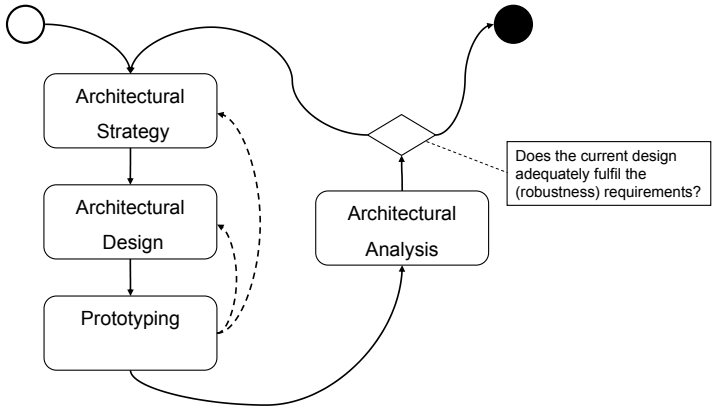

Figure 9.1: Proposed design process

During the design phase, it is important not to over-architect the solution, adding more flexibility or functionality than needed. One should find a balance between complexity and simplicity that handles the robustness requirements. It is important to remember that it is the whole system that should be robust, not necessarily all its parts. For instance, there is no reason to split a web application into a web front-end that calls a set of web services if there only is one application using the exposed web service functionality, and no concrete plans to offer the services to other applications exists. By adding this extra complexity, the exposed surface is larger, resulting in more complex and expensive design, analysis and testing. Developing the core functionality as a set of libraries makes it easy to expose a web service when needed, and keeps core logic clearly separated from GUI logic. Just adding flexibility because it is easy, without thinking about the added complexity and possibility for robustness issues, could result in a suboptimal architecture with regards to robustness. Adding measures to cope with robustness to the architecture is not always the solution. It might be that the gain in robustness does not justify the added complexity, decrease in maintainability and increase in development costs.

Step 1: Define an architectural strategy for the system

By starting with a conceptual architecture defined by an architectural strategy or vision, communicating the architecture can be done at an early stage. By communicating and analyzing the proposed architecture early, it is possible to do major changes and evaluate alternative solutions before large investments in the architecture have been made. Another benefit is that by working at the conceptual level, it is possible to ignore the technical details and limitations until a clear concept has been defined.

A conceptual architecture might define that the system should be distributed and rely on existing open standards for client-server communication. It might also suggest that messages should be encoded as XML according to a schema, and that a standard load-balancing approach should be employed. Details like the choice of communication standard, schema layout and choice of library for XML handling are irrelevant at the conceptual level.

The conceptual architecture should make it possible to draw and discuss the architecture at a level that makes it possible to discuss the main structures of the architecture with regards to robustness. Choice of application layout, distribution, infrastructure, forms of communication, architectural style and suggested implementation language and environment should be the main focus of the conceptual architecture. These choices lay the foundation for which robustness issues that have relevance to the proposed architecture. For instance, a sequential approach has benefits and drawbacks that are different from a asynchronous approach when it comes to robustness.

Important questions the architectural strategy should answer are:

- Which forms of load-balancing should be included?
- Does the application need any form of redundancy?
- Can self-monitoring help making the application more robust?
- What is the strategy for error-handling?
- Which external services does the application depend on?
- Can the architecture make use of asynchronous operations?
- How should the architecture handle transactional operations?

Separating the work on the conceptual architecture in this step from the actual design of the architecture can be difficult. The important difference is that the conceptual architecture does not have to handle all the details and technical realization details. As work progresses and the design has been refined in several iterations, it can be difficult to decide which details should be in the conceptual architecture and what should only be described in the architectural design. As a rule of thumb, all non major details and technical details should be in the architectural design, and the conceptual architecture should be a high level picture of the architecture. It is important to update the conceptual architecture when changes are made in the architectural design due to technical

limitations or realization, and at the same time it is important that the conceptual architecture contains enough details to be useful. A conceptual architecture that only describes that "the architecture is a client-server architecture that accepts multiple clients" is less useful.

Step 2: Design the architecture of the system

The main purpose of this step is to transform the conceptual architecture into an implementable solution. This includes deciding technical aspects and adding details to the design. Several approaches are possible; sometimes it might be important to add some overall detail to the conceptual architecture. In other cases it might be better to start by only detailing parts of the architecture.

By detailing parts of the architecture, the most uncertain parts can be designed early to reveal issues or make it possible to prototype parts of the system. Adding some more details and formalizing the whole architecture can make it easier to discuss and communicate the design. Facilitating discussion, increasing the understanding of the architecture and reducing risk should be the main drivers of choosing which parts to start with. Detailing the whole architecture when it is likely that the overall concept cannot fulfil the architectural goals of the system is most likely a waste of time.

Important elements the architectural design should include are:

- Determine how the application should handle errors.
- Determine which commonly used functions the architecture should support.
- Determine where and how input should be validated.
- How can repeatable tasks be supported or automated?
- Define transactional boundaries.
- Which communication protocols should be used.
- How should external components and dependencies be encapsulated.
- Definition of the data model.

Step 3: Prototype risky parts of the architecture

During the work on the conceptual architecture and architectural design, several areas that need investigation might be revealed. Possible issues can be uncertainty about how a design performs, how a library or other external dependency behaves in various situations, or if a particular design approach is implementable. Prototyping is an alternative to doing design based investigation, and simple proof-of-concept prototypes or partial implementations can be developed. Developing prototypes as a central part of the design process can provide improved understanding of the problems at hand, help reduce risks, provide better understanding of how external solutions work, and be used to test out new ideas.

By using prototyping in the design process, the number of surprises during implementation and testing of the solution can be reduced. It is much easier to make corrections based on discoveries in a prototype during the early design phases than to do the same discovery during the final testing phase.

Another important use of prototypes can be to train developers in the framework that should be used or create small sample applications that can be used as reference when implementing the real application.

Step 4: Perform a robustness evaluation

The main purpose of this step is to evaluate the current architecture or architectural strategy to determine how robust it is to various types of environmental influences and failures. The output of the analysis is used to determine if corrections, investigations or more detailed design are needed.

9.2.2 The analysis process in detail

Analysis can be performed either as a strictly sequential process, an iterative process or somewhere in between. It is recommended to let the architect perform the initial architectural presentation before starting the scenario elicitation. The scenario elicitation and analysis, however, would benefit from an iterative process, as analysis can promote discussion and stimulate discovery of new scenarios. An overview of the suggested analysis process can be seen in Figure 9.2.

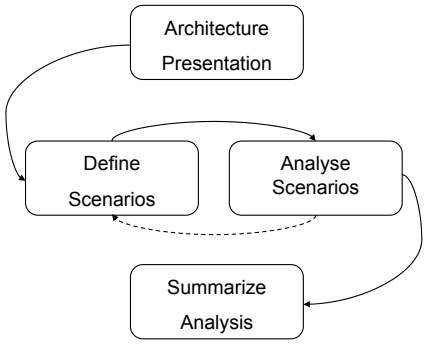

Figure 9.2: Proposed analysis process

Compared to ATAM, step 4.1 resembles step 3 & 4, 4.2 resembles 5 & 7, and 4.3 resembles step 6 & 8. The presentation steps 3 & 4 have been combined into a single step. Further, the analysis and testing stage of ATAM have been combined to form one stage that combines a structured process with brainstorming. The search for robustness related issues should benefit from this combined approach, as robustness is more related to testing for a wide selection of scenarios and understanding the architectural choices with respect to robust behaviour than adhering to business goals that is the focus of the structured utility-tree approach used by ATAM.

Step 4.1: Present the architecture

The architecture should be presented, and the architectural choices contributing to the system's robustness should be highlighted. The expected behaviour of the system in erroneous situations should also be presented. To what extent the architecture should be presented is determined by the prior knowledge the participants have about the solution. There is no need to do a extensive presentation of the architecture if all participants are familiar with it. The same applies for subsequent analysis as the design progresses; it is important to tell the participants about the changes in the current version, not spend time on presenting the whole architecture. At the end of this step all participants should have a firm understanding of the architecture or architectural strategy so they can suggest relevant scenarios to test on the architecture.

It is suggested to have figures and illustrations describing the architecture as they aid the understanding of the architecture and are helpful when participants should describe scenarios and later when the scenarios are going to be analyzed.

Step 4.2: Define what-if scenarios

As already stated, robustness is a wide subject, and as a consequence I suggest using a "what-if approach" to elicit scenarios. The process can take many forms, but a structured approach inspired by FMEA combined with an open brainstorm is my choice.

The reasoning behind the choice is twofold. First, going through the system in an ordered manner helps make sure that all the individual parts have been covered by analysis. Also errors occurring in seemingly unimportant components can cause robustness issues if not handled properly. Next, a brainstorm approach does not limit the creativity of the participants. Ideas from one participant could also trigger others to think of other issues. The downside of using a brainstorm approach is that it requires creativity from the participants. Using a list of scenario categories can, however, help this, but should be used with care. A list could limit the creativity of the participants as they might not think of types of scenarios not represented in the list.

First, the system as a whole should be investigated, the participants here listspossible issues the system can encounter. Typical examples are high number of requests, harmful input like injection attacks or environmental influences like power failure or hardware failures. The purpose of the overall system analysis is to cover scenarios that are hard to relate to a single component or a set of components. Next, the individual components of the architecture should be subject to a similar analysis. Samples of issues are inability to communicate with other components, internal failure in the component itself, call to a component results in a failure and erroneous input to component. A list of sample scenarios is shown in Table 9.1. All the architectural components do not have to be included, but all central components should be covered. It might be that the list of components has to be expanded based on how the architecture under review performs to determine the extent of some issue.

During the structured elicitation, all participants should be allowed to suggest scenarios that are not directly related to a single component or the system as a whole. It is important that all scenarios are considered, it is better to reject

Part	Scenario
Overall	The database server crashes
Overall	A user is registering a new order and during the registration, communication with the server is lost
Overall	The system experiences a higher concurrent load than the database is configured to accept
Overall	Due to a power failure, the server loses power immediately
Overall	User creates a new person with a ' character
Database	The disk volume runs out of available space
Database	The database query contains a syntax error
Component X	Invalid input is provided by component Y
Component Z	An internal failure occurs and is broadcasted to the calling component
Secondary Node	Communication with the primary node is suddenly lost, and now receives a request from a client
TransactionValidator	The transaction in the queue contains invalid XML
WebService	The specified session token is invalid

Table 9.1: List of sample scenarios

a scenario during the analysis phase if it turns out to be irrelevant, duplicate, or not related to robustness than to be negative about the participants' suggestions. One irrelevant scenario could easily trigger another participant to think of a new related scenario.

Lists of scenarios from previous analyses can be used to facilitate discussions, check that the current list covers all relevant aspects, or replace the elicitation step completely. It is important not to start by presenting such a list to the participants, as this might limit the types of faults the participants present scenarios for.

To help the participants, a categorization of scenario types can be provided, an example of such a categorization can be found in Table 9.2. It should, however, as already mentioned be used with care.

Relation to existing methods

Tekinerdogan et al. [71] defines *failure scenarios* based on FMEA, and describe a methodology using a fault domain model for scenario elicitation. The suggestion is to use a general fault domain model like the model defined by Avizienis et al. [5] (Figure 9.3 shows an updated version of the model that resembles the figure used by Tekinerdogan et al. [71]) to analyze possible failure modes for individual components in an architecture. HAZOP (see Section 7.4.2) has a similar approach that combines a set of guide words with properties of the system to be analyzed. The suggested approach above has direct relations to both of these approaches, but has a wider focus area than failure scenarios and is more targeted than the general use of guide words used by HAZOP.

Type	Category
Overall	Loss of service
Overall	Invalid input
Overall	High load
Overall	Hardware fault
Overall	Malicious input
Overall	Data loss
Component	Nonconforming design
Component	Lost communication
Component	Transient failure occurred during delegation
Component	Internal fault
Component	Delegation failure
Component	Invalid result returned from delegation
Component	Invalid input
Component	Invalid state

Table 9.2: Sample of possible scenario categories.

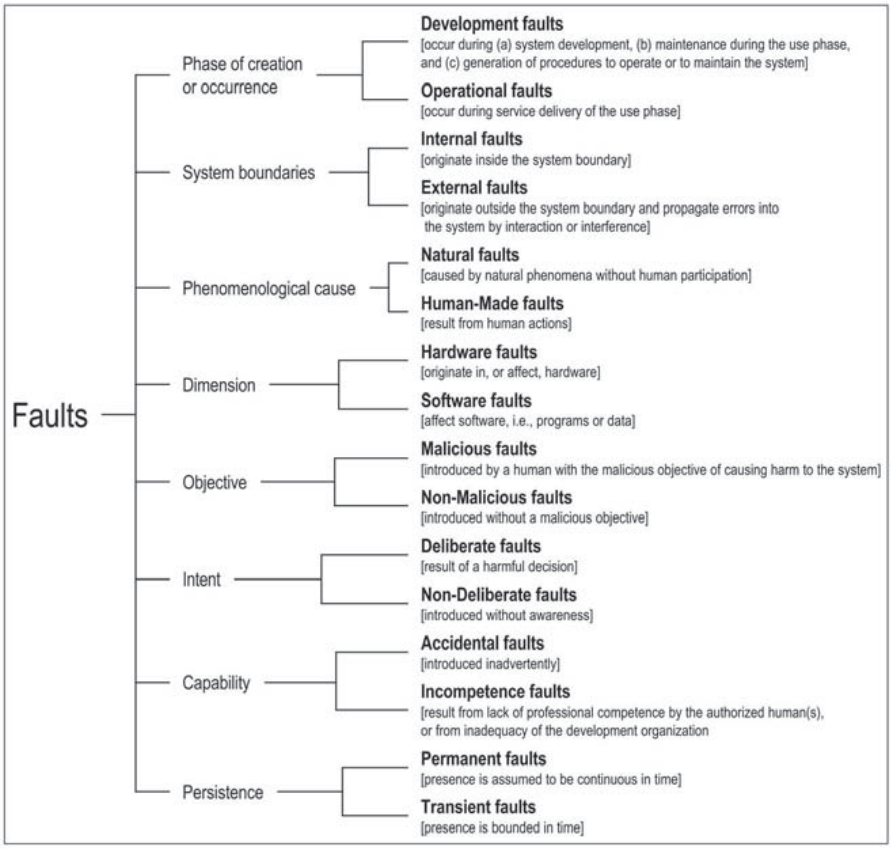

Figure 9.3: Fault domain model defined by Laprie and Randell [50].

Step 4.3: Test scenarios on the architecture

For individual what-if scenarios, the architect should present what happens, and demonstrate how the robustness aspects of the architecture are able to cope with the situation. Preferably, it should be demonstrated on a model of the proposed architecture. For issues involving implementation or detailed design, the architect should describe how the development process, guidelines, and tools make sure that implementation and detailed design are not violating the requirements set forth by the architecture.

A scenario could be only partially handled, handled wrong, or not handled at all by the proposed architecture. This should be noted together with any suggestions to alternative architectural solutions or improvements to the current architecture that could resolve the issue. The same should be the case with any scenarios or issues where it is not clear how the architecture will react. For a scenario that is handled, it is not obligatory to describe how the issue is handled, but a short comment about how the scenario is handled is recommended.

Uncertainties should be further analyzed and are candidates for prototyping.

Step 4.4: Summarize results

The results from the analysis should be summarized. Based on the results, a new iteration can be initiated if the current proposal has issues, or further investigations are needed. A possible template for such a summary is shown in Table 9.3

9.2.3 Main benefits of the method

The main goal was to define a scalable and easy to use method for designing and evaluating software architectures with respect to robustness. An illustration of the combined design and analysis process can be seen in Figure 9.4. The suggested method should be usable both for a small development team consisting of a couple of persons, and be a valuable tool for larger teams. Steps involved should be easy to understand and perform, and should integrate easily into existing development processes like RUP, Scrum and others. The use of iterative design should make it possible to start with a simple vision and transform it safely into a robust software architecture that forms a basis for the implementation of a software application providing robust services to its users.

Prototyping aids learning and helps reduce risks and explore unknown territory. By using a simplified version of ATAM, a well studied and working analysis tool based on scenarios, the analysis part should be helpful in evaluating how robust the proposed architecture is. The combination of a structured approach that considers the system as a whole and all its components, and the open brainstorming for each part should facilitate the discovery of relevant scenarios that need to be tested on the architecture.

The method remains to be tested on a real software project, and the effect would most likely not be breathtaking compared to a similar project that does not use the method. By using the method it will give a structured focus on robustness in the software architecture, and the focus alone should give a positive effect on the robustness. Based on findings and scenarios developed in

Part	Scenario	Status	Problem description
System	Extreme load	Handled	Load-balancer rejects requests when response time from server exceeds 10 secs
System	Input field contains a ’ character	Not handled	If input field is used in a SQL statement, it will result in a syntax error. It is suggested to add a generic database interface that handles special cases like this.
System	System has more concurrent users than the database allows	Unknown	The error returned by the database server in this situation are unknown, and possible effects of this error is unknown
Component Z	Internal fault is not handled	Partially handled	Cached by generic top-level exception mechanism, but the system might be left in a unknown state. Possible exception paths from Component Z must to be verified
Logger	Fault while logging a fault	Handled	Original error is returned to the caller also when logging failures are encountered.

Table 9.3: Sample summary report.

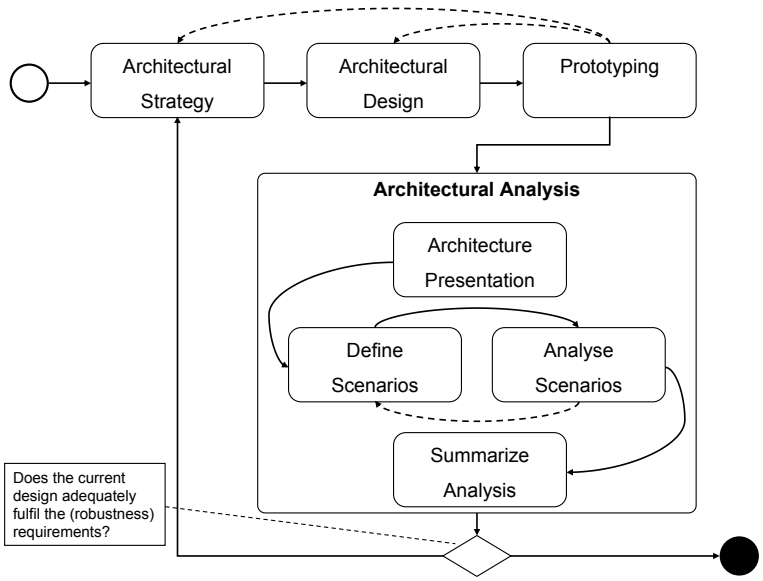

Figure 9.4: Combined design and analysis process

applications of the method, knowledge can be combined into checklists, lists of common scenarios, and robustness description of various solutions. These aids may be integrated into a more mature development process to increase robustness, but shifts in technology transforms robustness into a moving target as new technology brings new types of robustness issues and removes others.

Considering how internet applications were first built using mainly CGI applications written in C++, handling the input data was a major robustness issue. As time went by, frameworks and new languages like ASP, Perl and JSP came along and solved the issues with buffer overflows and parsing of the input data. Decrease in awareness of input resulted in new forms of problems. Issues like SQL injection and cross site scripting are quite common. Frameworks and solutions for handling these exist, and the number of issues decreases, but we are in the middle of a new era with more powerful client side behaviour in web applications making use of client side scripting and Ajax that brings a whole new set of challenges to the table.

A set of guidelines, checklists and solutions for robustness will only be relevant for a limited period of time, until technology, languages, tools and framework has changed the rules of the game.

Chapter 10

Conclusion and further work

10.1 Conclusion

This thesis started by illustrating the importance and complexity of robustness. I then looked at quality and quality models in general before looking at the relation between quality and architecture.

I then presented the results from ten interviews on the theme robustness and architecture. All the interviewees work with software architecture on a daily basis. The interview results indicated that the architects within the software industry have a wider definition of robustness than existing definitions in literature. Based on this, I have in Section 6.5 presented a definition of robustness that reflects the interviewees view of robustness.

The required level of robustness varies with the type of application. In essence, an application that from the users point of view handles faults, errors and failures gracefully, can be said to be robust.

The interviewees were also asked for their opinion about process measures which are important when designing and implementing software applications that should exhibit robust behaviour. These results were then combined with my personal experience to form a set of important criteria for an architectural design or analysis method. A selection of analysis and design methods were then presented and evaluated against the criteria. I did not find any of the methods presented to be optimal as a method to design or analyze for robustness. Based on this I have proposed a design and analysis method that combines elements from several of the methods already presented.

The proposed method is presented in Chapter 9 and it makes active use of iterative development, scenarios and prototyping. It is designed to be easy to use, employable during the whole architectural design stage and should scale well with both ambition and the size of the project team.

Time did not allow for testing the definition of robustness on a larger population, or to test the proposed method in practice. This is left for further work.

10.2 Further work

This section will describe suggestions for further work based on the work presented in this thesis.

10.2.1 Patterns and robustness

As mentioned in the introduction, the plan was to perform a study of patterns to find patterns for analysis that could be used for robustness. Due to lack of time, this analysis was not completed. However, a number of possible patterns were found during the work on this thesis. Some were found during the short review I did on pattern collections, some were suggested during the interviews and others were found as a result of the literature review of research related to robustness, presented in Chapter 8. I will briefly discuss my findings to provide a list over patterns and solutions that can be useful for building applications that should exhibit robust behaviour. I will, however, leave further elaboration and analysis of the suggested patterns and solutions for further work.

I suggest studying the individual groups of patterns or solutions listed below. A more complete list of patterns in each category needs to be developed. Further, how each of the patterns can contribute to robustness needs to be investigated. I suggest collecting patterns through the use of a literature review in combination with case studies of existing applications, interviews or surveys. The individual patterns need to be investigated in detail to determine exactly how they contribute to the system's robustness. This can be by analyzing them through case studies, detailed analysis or laboratory experiments.

A list of patterns where each pattern's contribution to the robustness is known is a valuable tool to use in my proposed design and analysis method. The robustness characteristics for the individual patterns can be used both when designing and analyzing a design. Interaction between the patterns still needs to be analyzed, but knowledge about the individual patterns help to reduce the number of unknown factors in a design.

Failover, Redundancy, and Clustering Adding redundancy to a system might be a way of increasing robustness to failure. The N-version approach was discussed in Section 8.1. Douglass [22] describes several patterns that are related to redundancy: protected single channel, homogenous redundancy, triple modular redundancy and heterogeneous redundancy. The protected single channel pattern is the simplest pattern and just adds checks, much like a monitoring pattern. The heterogeneous redundancy pattern is at the other end of the scale, employing independently designed or implemented channels. Buschmann et al. [13] describe the client-dispatcher-server pattern which is another way of implementing redundancy or failover.

Monitoring and self-awareness Monitoring and self-awareness can vary in range. At one end we have the fully generic approach discussed in Section 8.2 which requires substantial architectural support. At the other end we find the simple watchdog pattern mentioned by one of the interviewees. Douglass [22] describes three monitoring patterns: monitor-actuator pattern, sanity check pattern and watchdog. They vary in complexity from the watchdog pattern, which just monitors that a part of a system is running or proceeding as expected. The Monitor-actuator pattern is the most complex and monitors the input and output of the system in order to discover when something might be wrong. Monitoring patterns can also be used to control redundancy.

Load balancing Load balancing is mentioned as a measure to handle load. Interviewees suggest using third party products instead of building it into the systems themselves. The approach looks helpful for systems that need to handle high load.

Layering One of the interviewees explicitly mentioned layering and reducing abstraction leakage as approaches to achieve more robust systems. Other interviewees have not been as specific, but a good system structure was mentioned by several of them. Trowbridge et al. [74] describe the layering pattern, and explicitly state that it is a pattern that contributes to the robustness of a system.

Transactions, Rollback, and Compensation Transactions can help ensure that some changes are not left after an error has occurred, or when an operation is aborted. Transactions support rolling back changes made. However, they rely on proper use. Changes or actions not handled by a transaction boundary will not be rolled back. Proper use of transactions was mentioned by three of the interviewees as an important approach to improved robustness.

Asynchronous communication Not all communication needs to be synchronous. This applies both between systems and inside systems. Douglass [22] describes a message queuing pattern. The pattern is generic and can be used both for communication on an internal thread level, and between systems. One interviewee suggested that this form of communication also can help handle the 2-phase commit issue[1].

Data consistency Consistency can be enforced at the database level by the use of referential constraints, but also through consistency checks for data structures illustrated by the work by Black et al. [9] presented in Section 8.3 can be beneficial for robustness as it identifies inconsistent data.

Wrapping or encapsulation Wrapping or encapsulation can be performed in several ways. Section 8.5 described safety facades as one pattern, and Schmidt et al. [63] have a pattern they call wrapper facade that hides underlying details for a client. Likely other patterns for wrapping or encapsulation also exist, which can be beneficial for robustness.

Further, the definition of robustness provided in Chapter 6 does not limit robustness to a particular class of faults. This makes it relevant to also look at existing research within fault-tolerance. Pullum [59], Koren and ManiKrishna [47], and Lyu [52] describe several approaches or patterns like e.g. data diversity, recovery blocks and retry blocks, which also could be relevant when designing an application that should exhibit robust behaviour.

10.2.2 Testing the suggested design and analysis method

In Chapter 9, I proposed a design and analysis method. This method needs to be empirically tested in order to determine whether it is useful or not. Testing can be performed in several ways:

[1] If multiple systems are involved in a transaction, they all need to agree to commit the transaction before actually committing.

- An existing application or architecture can be analyzed using the approach.
- A laboratory experiment comparing systems designed with and without the method can be performed.
- The method can be tested in a real development project.

10.2.3 Verifying the definition

Based on the interview results, I have suggested a definition for robustness. This definition differs from definitions found in literature, by being very wide. The definition should be tested against a larger population by conducting more interviews or doing a survey. Ten interviews is a small sample to generalize upon, and there are variations in how the interviewees have defined robustness. By testing against a larger and carefully sampled population, it is possible to test if the proposed definition represents the industry's view of robustness or not. It should also be considered to include other groups like end-users, developers, and customers in such a test.

10.2.4 Looking into other suggestions

In this thesis I have focused on finding and defining a process to analyze and design the architecture with the use of teamwork, scenarios and prototyping. In addition, some patterns are suggested, but there are also other areas mentioned by the interviewees that may be important for robustness. Some suggested areas are:

- Focused testing on robustness issues.
- Requirement analysis in relation to robustness.
- Usability issues related to robustness.

Appendix A

Code examples

A.1 Enhanced version of math

This is a enhanced sample of the simple math application found in Figure 4.1

```
using System;

namespace SimpleMathRobust{
    class Program{
        static void Main(string[] args){
            try{
                string operationName = null;
                if ((args != null) && (args.Length > 0)){
                    operationName = args[0];
                }

                switch (operationName){
                    case "div":
                    case "mult":
                    case "mod":
                        if (args.Length != 3){
                            PrintErrorAndHelp("Missing arguments, format should be 'command numberA numberB'");
                            return;
                        }

                        double numA = 0;
                        double numB = 0;
                        double res = 0;

                        if (!double.TryParse(args[1], out numA)){
                            PrintError("Invalid argument: " + args[1] + " is not a valid number");
                            return;
                        }
                        if (!double.TryParse(args[2], out numB)){
                            PrintError("Invalid argument: " + args[2] + " is not a valid number");
                            return;
                        }

                        if (operationName == "div"){
                            if (numB == 0){
                                PrintError("Invalid argument: division by zero is not possible");
                                return;
                            }
                            res = numA / numB;
                        }else if (operationName == "mult"){
                            res = numA * numB;
                        }else if (operationName == "mod"){
                            if (numB == 0){
                                PrintError("Invalid argument: division by zero is not possible");
                                return;
                            }
                            res = numA % numB;
                        }

                        if (double.IsInfinity(res)){
                            Console.WriteLine("Error: cumputing error, result to large");
                        }else if (double.IsNaN(res)){
                            Console.WriteLine("Error: cumputing error, result is not defined");
                        }else{
                            Console.WriteLine("Result: " + res.ToString());
                        }

                        break;
```

```
                case "help":
                    PrintHelp();
                    break;

                default:
                    PrintErrorAndHelp("Invalid option: " + operationName);
                    break;
            }
        }
        catch (Exception ex){
            System.Console.WriteLine("An unexpected error has occured");
            System.Console.WriteLine(ex.ToString());
        }
    }

    private static void PrintError(string error){
        System.Console.WriteLine(error);
    }

    private static void PrintErrorAndHelp(string error){
        System.Console.WriteLine(error);
        System.Console.WriteLine("");
        PrintHelp();
    }

    private static void PrintHelp(){
        System.Console.WriteLine("Console math, help");
        System.Console.WriteLine("help - displays this help");
        System.Console.WriteLine("div a b - divides a by b");
        System.Console.WriteLine("mult a b - multiplies a by b");
        System.Console.WriteLine("mod a b - a modulus b");
    }
  }
}
```

Appendix B

Example interview results

B.1 Company A

Notater fra intervju med NN i bedrift A
Hva er robusthet?

Robusthet kan sees på oppførselen til en applikasjon når den er overlevert og befinner seg i drifts og forvaltningsfasen.

Applikasjonen skal dekke tilgjengelighetsbehovet den er tenkt å dekke, den skal ha forventet ytelse, og dersom det skjer noe galt eller oppstår en feilsituasjon så skal det være raskt å komme opp tilbake i normal produksjon igjen.

Sikkerhet i tradisjonell forstand oppleves ikke som en del av robusthetsbegrepet.

Antall feil i et system er ikke et måleparameter på grad av robusthet alene. Så lenge et system håndterer feil fornuftig kan det være robust selv om det inneholder feil

Angrepsvinkler og viktige elementer ved robusthet:

Jobbe i team fremfor alene

- Sammensatt gruppe med folk

  Viktig å ha med seg folk som har erfaringer med drift og forvaltning av applikasjoner, sammen med (yngre) folk som er oppdaterte på nyere teknologi og løsninger.
- Er absolutt en fordel å jobbe i team
- Viktig å motta nye drypp fra nye folk, og gjerne folk fra helt andre miljøer
- Det å få satt sammen team mest mulig bredt er veldig lurt
- Det er viktig å ta med brukerne i utviklingsprosessen

Planlegg for drift og forvaltning

- Hensyn til driftsdimensjonen

  Viktig å ta hensyn til driftsdimensjonen under design og utvikling av løsningen, den har lett for å bli utelatt.

Gode feilmeldinger og feilhåndtering

- Enkle, fornuftige feilmeldinger
- Feilmeldinger som har informasjon om den underliggende feilen
- Feilmeldinger som bidrar til å løse kilden til feilen som er oppstått

  Viktig med enkle og forståelige feilmeldinger som pinpointer problemet, feilmeldinger som kun forteller at noe har feiler er av liten verdi.
  Feilmeldingene bør hjelpe til at en er raskt oppe og går igjen dersom det skulle oppstå alvorlige feilsituasjoner

Design for selvmonitorering

- Proaktivitet er sentralt
- Automatisk varsling ved symptomer på at en feilsituasjon kan oppstå
- Automatisk varsling ved symptomer på at en feilsituasjon har oppstått

  Har i senere år lagt inn proaktivitet i applikasjonene. Applikasjonen klarer selv eller ved hjelp av ekstern monitorering å oppfatte symptomer på at noe er galt, eller noe galt er i ferd med å skje. Ved denne type symptomer sendes det varsel til forvaltnings og driftspersonale.
- Varsle driftspersonale før brukerne oppdager at noe er galt

  Ved automatisk monitorering og varsling er situasjonen varslet til forvaltning og drift, og kanskje også rettet før brukeren er i stand til å oppfatte situasjonen.
- Varsling med påfølgende manuell reaksjon gir brukere en opplevelse av høyere robusthet

  Varsling og tett oppfølgning fra drift og forvaltning gir ofte brukerne en opplevelse av en mer robust applikasjon enn hva som egentlig er tilfelle. Dette ved at man oppdager og/eller avverger feilsituasjoner på et tidlig tidspunkt.

Typiske varsler

- Varsel ved fulle disker
- Varsel ved fulle køer
- Varsel ved fulle feilkøer
- Varsel ved akkumulerende køer
- Varsel ved unormalt høy last

Varsel ved mangel på data

Dersom det ikke er kommet en type hendelser fra ett av systemene som fungerer som en av leverandørene til ett større system på x minutter, så varsles det.

Dersom det ikke er kommet forespørsel på søket i løpet av x antall minutter så genereres det et varsel

Varsel på manglende data bygges typisk opp på basis av erfaringstall på normal belastning og bruk av systemene.

Benytte driftserfaring til å etablere nye varsler ved behov, og i nye systemer

Design asynkrone grensesnitt

- Asynkrone grensesnitt gir mer robuste systemer

- Benytt asynkrone grensesnitt der det er mulig
- Synkrone grensesnitt krever ofte mer intrikate løsninger enn asynkrone 2 fase commit problemstillinger er ikke like aktuelle ved asynkrone grensesnitt
- Asynkrone grensesnitt krever at man realiserer tradisjonell commit problematikk på en annen måte
- Asynkrone grensesnitt stiller mindre krav til tilgjengelighet enn synkrone systemer

  Dersom et system er nede vil systemet plukke opp forespørsler når det kommer opp igjen.

  Dersom et system skal spørre 2-3 andre for å gi et aggregert svar stiller det større krav ved synkrone enn asynkrone implementasjoner. Både på tilgjengelighet og svartid.
- Asynkrone grensesnitt gir mindre kobling enn synkrone grensesnitt

Bygg en god datamodell

- Bruk tid på datamodellering

  Gode datamodeller har vist seg å ha stor påvirkning for det videre livet til programsystemer.
- En god og riktig datamodell er viktig for fremtidig vedlikehold

  Gode datamodeller er mer bestandige og gjør at introduksjon av ny funksjonalitet som oftest har begrenset påvirkning på den eksisterende datamodellen.

  Ny funksjonalitet er ofte lokalisert til nye entiteter og innføring av nye attributter på eksisterende entiteter, og har liten innvirkning på eksisterende entiteter.
- Involver brukere for å sikre riktig datamodell i første versjon

  Ved å involvere brukere så kan en sikre seg at ens oppfatning av virkeligheten som danner grunnlaget for datamodellen stemmer. Dette gjør at man ikke sent i utviklingsfasen må gjøre store endringer i datamodellen.
- Jobb mot normaliserte datamodeller

  Dagens maskinvare gjør at datamodellene kan være mer riktig uten at man får problemer med ytelse. Og 3'dje normalform er en god mal å legge seg på.
- Ha god og riktig datamodell i første versjon

  Kommer man ut fra start med feil modell i bunn, så er det ikke noen god start for systemet.
- Store endringer i datamodellen har ofte store konsekvenser for løsningen

Dersom en begynner og splitte og/eller flette sammen entiteter så har det oftest store konsekvenser for løsningen som helhet. Ved denne type endringer så har ofte resten av løsningen en tendens til å kunne bli ustabil eller kreve større endringer. Følgefeil er vanlige i slike situasjoner.

- Endringer i den eksisterende datamodellen krever også endringer i data

  Om man endrer datamodellen så krever det også som oftest at dataene som er lagret må oppdateres også, med de farene det fører med seg.

- Gode datamodeller er ofte bestandige

  Ved god datamodell så er datamodellen ganske bestandig, og systemendringene over tid oftest lokalisert til forretningsreglene og ikke datamodellen.

- Datamodellen bestemmer ofte mye av applikasjonens levetid

  Applikasjoner med en datamodell som har vist seg å være god og bestandig har ofte lengre levetid enn andre systemer.

Sikre datakvalitet

- Inputkontroll er viktig
- Viktig å sjekke all data som kommer inn i et system

  Viktig å sjekke når data kommer inn, dersom hele systemet skal ta høyde for skitten data, så skaper dette unødvendig kompleksitet i systemet, samt økt i teste testen av systemet.

- Viktig å sjekke at data om skrives ut av et system/en prosess er korrekt

  Det er gjerne flere prosesser/systemer som leser data som et system/prosess skriver ut, og det er da viktig å fokusere på at systemet/prosessen leverer korrekte data, etter som det krever mye mer arbeid å sjekke at de øvrige systemene/prosessene oppfører seg riktig på basis av ukorrekte data. Samt å designe og utvikle systemene/prosessene for å takle ugyldige data.

- Behov for kompensasjon av feilsituasjoner som følge av ukorrekt data fører til unødvendig kompleksitet.
- Viktig med referanseintegritet i baser for å sikre datakvalitet
- Viktig å benytte regler i basen for å sikre like regler for alle systemer

  Vanskelig å sikre at alle som legger inn data i et datalager følger de eksakt samme reglene uten at de håndheves av basen.

- RI i base er viktig selv om det begrenser frihetsgraden

  Mer komplekst å gjøre uttrekk av data for test, eller gjøre større masseoppdateringer, men det sikrer et konsistent datagrunnlag.

Fokus på god test

- Vesentlig å ha fokus på testing
- Regresjonstesting er viktig
- Faste tester som kjøres for hver versjon (regresjon)

  Regresjonstesting forhindrer at nye versjoner introduserer feil i eksisterende deler av løsningen. Erfaringsmessig innfører nye versjoner mye feil i eksisterende deler av løsninger.

- Automatisert testing er et ønske, spesielt til bruk for regresjonstesting.
- Utfør stresstesting

  Ikke anta at systemet takler produksjonslast, verifiser det

- Ikke la tidsnød gjøre at man får mindre tid til test

  Det er ofte testperioden som blir forkortet når tidligere faser av prosjektet har tatt lengre tid, ikke la dette skje da testing av systemet er veldig viktig.

- Utfør negative tester både under enhetstest og systemtest

  Det er for stor fokus på å teste at systemet gjør det som står i spesifikasjonen, det var mye mer fokus tidligere på å teste hva som skjedde dersom ikke forutsetningene stemte, det oppstår unormale situasjoner, situasjoner som ikke er beskrevet direkte. Det er mange situasjoner som ikke er beskrevet direkte i spesifikasjonene.

- Sørg for at det finnes kunnskap og erfaring på negativ testing
- Utfør "katastrofetesting"

  Dra ut pluggen på servere som en del av testkjøringen for å verifisere at oppførselen til systemet er akseptabel.

- Bruk gjerne negativ testing tidlig i utviklingsløpet

Endringskontroll

- Viktig å planlegge endringer i produksjon

  Enkelte endringer påvirker andre systemer og det er da viktig å ha rutiner som sørger for at behovene til de øvrige systemene er opprettholdt. Dette kan ivaretas av et endringsforum, som også kan avgjøre om endringen er viktig nok for å kunne gjennomføres med tanke på sesongvariasjoner og andre variabler. Et slikt forum kan også ha ansvar for å sørge for at kunden/eier er bevist på endringene som er tenkt foretatt der det er nødvendig.

- Viktig å hensynta perioder hvor det er ekstra kritisk med feil

Enkelte perioder er mer forretningskritisk med tanke på sårbarhet ved feil. Det gjelder både tilgjengelighet på mannskap til å korrigere og utbedre (f.eks ferie), og perioder hvor avhengigheten av systemet er ekstra høy.

- Viktig å evaluere viktigheten av en endring

  En må evaluere om endringen er viktig nok til å gjennomføres sett opp mot risikoen en tar ved å gjennomføre den. En endring bare for endringens skyld er neppe godt nok.

- Viktig å akkumulere erfaringer med endringer

  Det er viktig å akkumulere den kunnskapen man gjør seg med tanke på hvilke feilsituasjoner ulike endringer kan føre med seg, og benytte dette i evalueringen av nye endringer.

- Sammenflettede systemer krever håndheving av endringshåndtering
- En liten "ubetydelig" endring kan ta ned et helt integrert system

Versjonering og versjonskontroll

- Det er viktig med god versjonskontroll og versjonsstyring
- Viktig å kunne vite hva som er endret mellom versjoner
- Viktig å kunne gå tilbake til en tidligere versjon om det skulle være nødvendig
- Flere systemer og plattformer som er involvert øker viktigheten av å ha god felles versjonskontroll

  Flere miljøer er gjerne involverte og uten versjonskontroll er det veldig vanskelig å ha kontroll over endringer mellom versjonene.

- Viktig med versjonering av grensesnitt og tilhørende beskrivelser

Planlegg og design for recovery

- Viktig å ha gode backup og recoveryløsninger
- Viktig å ha planen for recovery klar dersom noe skulle oppstå
- Viktig å bygge systemet med tanke på at en fornuftig recovery er mulig
- Viktig å se recovery scenarioer opp mot hvor tidskritisk det er for virksomheten

  Det holder ikke at det er mulig med recovery om den tar 2 dager, mens bedriften ikke kan klare seg uten systemet i mer enn et par timer.

Failover / clustring / redundans

- Failover / clustring / lastdeling benyttes i noe grad
- Failover/clustring krever stateless systemer som er laget for det

- Det er svært vanskelig å kombinere state i systemer med failover/redundans

  En må i realiteten da starte helt på nytt igjen om man går over på en annen "node".

- De hardwaremessige aspektene er en viktig faktor til robusthet

Avverg feilsituasjoner

- Duplisere viktige data for å forbedre robusthet ved ustabile linjer
- Duplisere viktige data for å sikre bedre tilgjengelighet

  Dersom det er ustabile telefonlinjer mellom applikasjonen og datakilden kan det være nødvendig å duplisere relevante deler av kritisk data på klienten for å sikre seg at den er tilstede når den er nødvendig. F.eks informasjon nødvendig for pakkeutlevering kan sendes når pakken er bekreftet ankommet på postkontoret.

Sikre spesifikasjon og forventninger til løsningen

- Brukermedvirkning i utviklingsprosessen er viktig
- Bruk av prototyping sammen med brukerne i utviklingsprosessen er bra
- Større endringer i sene faser skaper lettvintløsninger og kompromisser

  Derfor viktig å vite med sikkerhet at systemet man er i ferd med å bygge er det samme som brukeren forventer.

- Desto bedre du treffer med systemet i første versjon, desto mer robust blir det
- Viktig å komme ut med et rent system i første versjon, og ikke et som allerede har begynt å ligne et system som har levd et lengre liv
- Viktig å fremstille systemet og design ved hjelp av figurer da det er langt mer utvetydig enn beskrivelser når man skal kommunisere med andre.

Vær bevist på integrerte systemer

- Bygging av systemer ut fra kildedataprinsippet

  Dersom du har fokus på at få systemer skal ha dataeierskap mens mange skal være brukere, vær da bevist på å følge strategien. Vær også bevist på hvilke følger det har med tanke på integrasjonskrav og tilgjengelighetskrav.

- Det svakeste systemet bestemmer for det totale systemet ved integrering

  En kan bygge ens eget system solid, men det er ikke mye til hjelp dersom det er avhengig av integrasjoner mot andre systemer som er dårlige. Alle vil alltid lide under det svakeste leddet i kjeden.

- Systembevissthet på datakrav og avhengigheter

- Forhindre unødvendig kompensering

  Et system har gjerne en del prosesser og funksjoner som ikke får startet eller bør starte med mindre det nødvendige datagrunnlaget er på plass. Det er viktig at systemet da er bevist på disse avhengighetene og ikke starter opp prosessene på feil datagrunnlag. En unngår da en del situasjoner hvor man er nødt til å rulle tilbake fordi man har startet på feil datagrunnlag. Stikkord: kompanserer, proaktiv.

- Viktig med gode og dekkende beskrivelser av grensesnitt mellom systemer

Problematiser design

- Det å problematisere designet med scenarioer kan være nyttig
- Det å forsvare feilsituasjoner på et system i designfasen kan være nyttig
- Komplekse sammensatte systemer kan med fordel problematiseres

  Ett system gjerne kan falle ut og 5 andre skal bestå

Design relevante deler feiltolerante

- Dead-letter køer er en mekanisme for å fange opp feil
- Meldinger med transiente feil kan gjerne legges på feilkøer for å re-prøves

  Feil rekkefølge på meldinger er her en typisk feilkilde

B.2 Company B

Notater sendt på mail av NN i bedrift B
Hva er robusthet?

Hva legger du i begrepet robusthet?

At det tåler en støyt uten å gå i stykker. feks for en bil er at den tåler hard bruk uten at noe ryker, eller at andre feil oppstår.

Hva kjennetegner en robust applikasjon?

Brukeropplevelse At uansett hva brukeren legger inn så får man skikkelige feilmeldinger og data som godtas har en riktig kontekst. Dvs man kan ikke legge inn data som er ugyldige i en eller annen sammenheng. At brukerfeil ikke fører til at systemressurser brukes opp; DB connections, DB cursors mm.

Belastning Ved stor belastning skal ikke systemet plutselig dø, men ha en akseptabel måte å handtere dette på. Feks økt respons tid, eller feilmelding til brukeren/systemlogger som er forståelig.

Innvending Når noe feiler i applikasjonen skal den ikke dø, men logge hva som er feil slik at man kan finne ut av det. I tillegg skal ikke feil føre til at komponenter disables og ikke kan brukes lenger som fører til feil andre steder i applikasjonen.

Grensesnitt Connection problemer skal føre til driftsalarmer, og data skal køes opp slik at de ikke går tapt (persistent). Nærmer man seg feks disk full må applikasjonen slutte å akseptere input, men ikke dø. Dersom den ikke lengre kan ta imot input må den gi feilmelding tilbake til andre systemer slik at det kan køes opp andre steder.

Hvor robust må en vanlig forretningsapplikasjon være?

Dette er avhengig av hvem som er brukere, men jo flere brukere jo mer robust bør den være.

Hvordan oppnå robusthet?

Hva kan man gjøre for å oppnå robusthet slik du ser det?

- Lagdeling i applikasjonsarkitekturen. Forhindre abstraksjonslekkasje (Alle feilmeldidinger bør vær forståelige for brukeren). Asynkronitet bør ivaretas vha køer.
- Bruk av rammeverk i applikasjonene slik at man får en ensartet feilhåndtering.
- Det første man gjør er å definere hvordan feil skal håndteres. Dette bør være klart ved start av implementasjon.
- Videre oppnår man mye ved å teste skikkelig, men hvordan gjør man dette? Jo ved feks å kjøre automatisk og kontinuerlig testing. (Støttes etterhvert av flere utviklingsverktøy)

- Man oppnår også mye ved å slippe sluttbrukerne til i systemtesten. Å la de komme inn i akseptansetest er vanligvis for sent. Dette sikrer at alle brukstilfellene blir kjørt tidligst mulig.
- Man oppnår også mye ved iterativ utvikling. Man får da mindre moduler å debugge.

Hvilke faser er fokus på robusthet viktig i?

> **Hvor i prosessen er det viktigst med fokus på robusthet og hvorfor (hvilke faser)?**
>
> Viktigst i design av applikasjonsarkitekturen. I mange forskjellige systemer må feilhåndtering og ressurshåndtering være en grunnleggende funksjonalitet.

Hva gjør du/dere med robusthet, og hvilke planer har du/dere?

- Fokuserer du/dere på robusthet utviklingen i dag?

  > Det har høyt fokus og man må sikre at man ikke mister data, samt at applikasjonen oppfører seg "fornuftig" hvis store feilmengder plutselig raser inn - den kan feks stoppe innlesning. Kravet har alltid vært å ha feilhåndtering og logge rammeverket klart ved oppstart av implementasjon.

- Hvorfor / hvorfor ikke? Planlegger du/dere å gjøre det i nærmeste fremtid?
- Hva gjør du/dere i dag / hva planlegger du/dere å gjøre for å fokusere på robusthet?
- Hva planlegger du/dere å gjøre i fremtiden?

  > Mer fokus på forbyggende tiltak som nevnt over.

Hva er arkitektur?

> **Hva mener du arkitektur er?**
>
> Grunnleggende strukturer og rammeverk som man benytter når man utvikler. Dette definerer hvilke deler systemet består av, hvordan de kobles sammen, hvilke ansvar de ulike deler har og hvordan de kommuniserer. Patterns kan være med å definere disse strukturene.

Viktigheten av arkitekturen?

> **Hvilken rolle spiller arkitekturen slik du ser det i hvorvidt løsningen blir robust eller ikke?**
>
> En god arkitektur kan sikre egenskaper som feilhåndtering, lastbalansering og effektiv ressurshåndtering.

Arkitekturmessige løsninger

Er det noen arkitekturmessige/design løsninger/patterns som du mener bidrar til robusthet? Her er høynivå så vel som lavnivå arkitektur/design på grensen til kodeløsninger interessante

- Har egentlig ikke sett mange rammeverk som gir robusthet..... men språkene gir mulighet for å bygge komponenter som kan bidra.
- En god arkitektur gjør applikasjonen mer endringsbar.
- Helt avgjørende for å få til en effektiv utvikling med roller og iterativ/scrum baserte metoder
- Har jobbet en del med Java og har sett at dersom man benytter rammeverk som MVC, STRUTS, DAO, Facade, DTO, Gang of four patterns, samt patterns for feks Web applications, i tillegg til blueprints fra feks SUN/Microsoft mm. så blir applikasjonene mer endringsbare og robuste)

Bibliography

[1] *IEEE Std 1028-1997*, chapter IEEE Standard for Software Reviews, pages i–37. 1998.

[2] Robust - webopedia.com. web. URL `http://www.webopedia.com/TERM/R/robust.html`. visited 2007-08-09.

[3] Gregory Abowd, Len Bass, Paul Clements, Rick Kazman, Linda Northrop, and Amy Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. 1996. URL `http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr025.96.pdf`. Technical Report: CMU/SEI-96-TR-025.

[4] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, USA, 1977. ISBN 0-19-501919-9.

[5] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability, 2001. URL `http://dirc.org.uk/research/pubs/trs/papers/739.pdf`. Research Report N01145, LAAS-CNRS, April 2001.

[6] Jakob Eyvind Bardram, Henrik Baerbak Christensen, and Klaus Marius Hansen. Architectural prototyping: An approach for grounding architectural design and learning. *wicsa*, 00:15, 2004. doi: http://doi.ieeecomputersociety.org/10.1109/WICSA.2004.1310686.

[7] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, Boston, USA, second edition, 2003. ISBN 0-321-15495-9.

[8] Steffen Becker, Wilhelm Hasselbring, Alexandra Paul, Marko Boskovic, Heiko Koziolek, Jan Ploski, Abhishek Dhama, Henrik Lipskoch, Matthias Rohr, Daniel Winteler, Simon Giesecke, Roland Meyer, Mani Swaminathan, Jens Happe, Margarete Muhle, and Timo Warns. Trustworthy software systems: a discussion of basic concepts and terminology. *SIGSOFT Softw. Eng. Notes*, 31(6):1–18, 2006. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1218776.1218781.

[9] J.P. Black, D.J. Taylor, and D.E. Morgon. A Compendium of Robust Data Structures. *Fault-Tolerant Computing, 1995, 'Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*, pages 127–, 27-30 Jun 1995.

[10] IEEE Standards Board. *IEEE Standard Glossary of Software Engineering Technology*. The Institute of Electrical and Electronics Engineers, 1990. ISBN 1-55937-067-X.

[11] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[12] Barry W. Boehm, John R. Brown, Hans Kaspar, Myron Lipow, Gordon J. MacLeod, and Michael J. Merrit. *TRW Series of Software Technology Volume 1: Characteristics of software quality*. North-Holland, Amsterdam, 1973. ISBN 0-444-85105-4.

[13] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, USA, 1996. ISBN 0-471-95869-7.

[14] Joseph P. Cavano and James A. McCall. A framework for the measurement of software quality. In *Proceedings of the software quality assurance workshop on Functional and performance issues*, pages 133–139, 1978. doi: http://doi.acm.org/10.1145/800283.811113.

[15] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating software architectures: methods and case studies*. Addison-Wesley, Boston, 2002. ISBN 0-201-70482-x.

[16] Tony Cornford and Steve Smithson. *Project Research in Information Systems. A Student's Guide.* Palgrave, New York, NY, USA, 1996. ISBN 0-333-64421-2.

[17] Marc-Alexis Côté, Witold Suryn, and Elli Georgiadou. In search for a widely applicable and accepted software quality model for software quality engineering. *Software Quality Journal*, 15(3), 2007. doi: http://dx.doi.org/10.1007/s11219-007-9029-0.

[18] John DeVale and Jr. Philip J. Koopman. Robust software - no more excuses. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 145–154, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1597-5.

[19] John P. DeVale, Philip J. Koopman, and David J. Guttendorf. The Ballista Software Robustness Testing Service. URL `http://www.ece.cmu.edu/~koopman/ballista/tcs99/tcs99.pdf`.

[20] Marcel Dix and Holger D. Hofmann. Automated Software Robustness Testing - Static and Adaptive Test Case Design Methods. *euromicro*, 00:62, 2002. ISSN 1089-6503. doi: http://doi.ieeecomputersociety.org/10.1109/EURMIC.2002.1046134.

[21] Liliana Dobrica and Eila Niemela. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2002.1019479.

[22] Bruce Powel Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Professional, Boston, MA, USA, 2002. ISBN 0-201-69956-7.

[23] R. Geoff Dromey. A model for software product quality. *IEEE Trans. Softw. Eng.*, 21(2):146–162, 1995. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.345830.

[24] RG Dromey. Concerning the Chimera [software quality]. *Software, IEEE*, 13(1):33–43, 1996.

[25] ECMA. ECMA-334 - C# Language Specification 4th edition, ISO/IEC 23270:2006. 1996. URL `http://www.ecma-international.org/publications/standards/Ecma-334.htm`.

[26] Robert J. Ellison and Andrew P. Moore. Trustworthy Refinement Through Intrusion-Aware Design (TRIAD). 2003. URL `http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tr002.pdf`. Technical Report: CMU/SEI-2003-TR-002.

[27] Robert J. Ellison, David A. Fisher, Richard C. Linger, Howard F. Lipson, Thomas A. Longstaff, and Nancy R. Mead. Survivability: Protecting your critical systems. *IEEE Internet Computing*, 3(6):55–63, 1999. ISSN 1089-7801. doi: http://dx.doi.org/10.1109/4236.807008.

[28] Clifton A. Ericson. *Hazard analysis techniques for system safety*. Wiley-Interscience, 2005.

[29] Norman Fenton and Shari Lawrence Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, MA, USA, 1997. ISBN 0-534-95600-9.

[30] Christof Fetzer and Zhen Xiao. An automated approach to increasing the robustness of c libraries. *dsn*, 00:155, 2002. doi: http://doi.ieeecomputersociety.org/10.1109/DSN.2002.1028896.

[31] Martin Fowler. Patterns. *IEEE Software*, 20(2):56–57, 2003. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2003.1184168.

[32] Martin Fowler et al. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, Boston, USA, 2003. ISBN 0-321-12742-0.

[33] Peter Freeman. Software reliability and design: A survey. In *DAC '76: Proceedings of the 13th conference on Design automation*, pages 484–494, New York, NY, USA, 1976. ACM. doi: http://doi.acm.org/10.1145/800146.804850.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1995. ISBN 0-201-63361-2.

[35] Darvin .A. Garvin. What does "product quality" really mean. *Sloan Management Review*, 26(1):25–43, 1984.

[36] Anup K. Ghosh, Matt Schmid, and Frank Hill. Wrapping windows nt software for robustness. *ftcs*, 00:344, 1999. ISSN 0731-3071. doi: http://doi.ieeecomputersociety.org/10.1109/FTCS.1999.781070.

[37] S.E. Hermanson. The software embarrassment: a solution. *Computer Assurance, 1989. COMPASS '89, 'Systems Integrity, Software Safety and Process Security', Proceedings of the Fourth Annual Conference on*, pages 26–30, 19-23 Jun 1989. doi: 10.1109/CMPASS.1989.76034.

[38] Michael N. Huhns, Vance T. Holderfield, and Rosa Laura Zavala Gutierrez. Robust software via agent-based redundancy. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 1018–1019, New York, NY, USA, 2003. ACM. ISBN 1-58113-683-8. doi: http://doi.acm.org/10.1145/860575.860774.

[39] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[40] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. *iceccs*, 00:0068, 1998. doi: http://doi.ieeecomputersociety.org/10.1109/ICECCS.1998.706657.

[41] Rick Kazman, Len Bass, Mike Webb, and Gregory Abowd. Saam: a method for analyzing the properties of software architectures. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 81–90, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X.

[42] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Softw.*, 13(6):47–55, 1996. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.542294.

[43] Rick Kazman, S. Jeromy Carrière, and Steven G. Woods. Toward a discipline of scenario-based architectural engineering. *Ann. Softw. Eng.*, 9(1-4): 5–33, 2000. ISSN 1022-7091.

[44] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: The elusive target. *IEEE Softw.*, 13(1):12–21, 1996. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.476281.

[45] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.*, 12(1):96–109, 1986. ISSN 0098-5589.

[46] John C. Knight and Nancy G. Leveson. A reply to the criticisms of the knight & leveson experiment. *SIGSOFT Softw. Eng. Notes*, 15(1):24–35, 1990. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/382294.382710.

[47] Israel Koren and C. ManiKrishna. *Fault Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 2007. ISBN 0-12-088525-5.

[48] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant*

Computing, page 230, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8470-4.

[49] Robert Laddaga. Creating robust software through self-adaptation. *IEEE Intelligent Systems*, 14(3):26–29, 1999. ISSN 1094-7167. doi: http://doi.ieeecomputersociety.org/10.1109/MIS.1999.769879.

[50] Jean-Claude Laprie and Brian Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004. ISSN 1545-5971. doi: http://dx.doi.org/10.1109/TDSC.2004.2. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.

[51] John D. Litke. A systematic approach for implementing fault tolerant software designs in ada. In *TRI-Ada '90: Proceedings of the conference on TRI-ADA '90*, pages 403–408, New York, NY, USA, 1990. ACM. ISBN 0-89791-409-0. doi: http://doi.acm.org/10.1145/255471.255565.

[52] Michael R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0-471-95068-8.

[53] Steve McConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, Redmond, WA, USA, 1993.

[54] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 46–54, New York, NY, USA, 2006. ACM. ISBN 1-59593-457-X. doi: http://doi.acm.org/10.1145/1145735.1145743.

[55] Andrew P. Moore and Robert J. Ellison. TRIAD: a framework for survivability architecting. In *SSRS '03: Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems*, pages 105–109, New York, NY, USA, 2003. ACM. ISBN 1-58113-784-2. doi: http://doi.acm.org/10.1145/1036921.1036933.

[56] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999. ISSN 1541-1672. doi: http://dx.doi.org/10.1109/5254.769885.

[57] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, 2001. ISBN 0130290491.

[58] Roger S. Pressman and Darrel Ince. *Software Engineering: A Practitioner's Approach: European Adaptation*. McGraw-Hill, Inc., New York, NY, USA, 1997.

[59] Laura L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Artech House, Inc., Norwood, MA, USA, 2001. ISBN 1-58053-137-7.

[60] Martin P. Robillard and Gail C. Murphy. Designing robust java programs with exceptions. *SIGSOFT Softw. Eng. Notes*, 25(6):2–10, 2000. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/357474.355046.

[61] Colin Robson. *Real World Research: A Resource for Social Scientists and Practitioner-Researchers (Regional Surveys of the World)*. Blackwell Publishing Limited, 2002. ISBN 0631213058.

[62] Bradley Schmerl and David Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 241–248, New York, NY, USA, 2002. ACM. ISBN 1-58113-556-4. doi: http://doi.acm.org/10.1145/568760.568804.

[63] Douglas C. Schmidt, Hans Rohnert, Michael Stal, and Dieter Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000. ISBN 0471606952.

[64] Fred B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, Washington, DC, USA, 1998. ISBN 0309065585.

[65] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-182957-2.

[66] Charles P. Shelton, Philip Koopman, and Kobey Devale. Robustness testing of the microsoft win32 api. *dsn*, 00:261, 2000. doi: http://doi.ieeecomputersociety.org/10.1109/ICDSN.2000.857548.

[67] Michael E. Shin. Self-healing components in robust software architecture for concurrent and distributed systems. *Sci. Comput. Program.*, 57(1):27–44, 2005. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2004.10.003.

[68] Johannes Siedersleben. *Errors and Exceptions — Rights and Obligations*, volume 4119 of *Lecture Notes in Computer Science*, pages 275–287. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-37443-5.

[69] Joel Spolsky. Don't let architecture astronauts scare you. web, apr 2001. URL `http://www.joelonsoftware.com/articles/fog0000000018.html`. visited 2007-09-03.

[70] Witold Suryn, Alain Abran, and Alain April. ISO/IEC SQuaRE. The second generation of standards for software product quality. *Seventh IASTED International Conference on Software Engineering and Applications*, pages 807–814, 2003.

[71] Bedir Tekinerdogan, Hasan Sozer, and Mehmet Aksit. Software architecture reliability analysis using failure scenarios. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 203–204, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2548-2. doi: http://dx.doi.org/10.1109/WICSA.2005.65.

[72] Christian Tellefsen. An Examination of Issues with Exception Handling Mechanisms. Master's thesis, Norwegian University of Science and Technology, 2007.

[73] R. Torkar, S. Mankefors, K. Hansson, and A. Jonsson. An exploratory study of component reliability using unit testing. *issre*, 00:227, 2003. ISSN 1071-9458. doi: http://doi.ieeecomputersociety.org/10.1109/ISSRE.2003.1251045.

[74] David Trowbridge, Dave Mancini, Dave Quick, Gregor Hohpe, James Newkirk, and David Lavigne. *Enterprise Solution Patterns for .NET*. Microsoft Press, 2003. ISBN 0735618399.

[75] Unknown. System robustness - atis telecom glossary 2000. web, feb 2001. URL `http://www.atis.org/tg2k/_system_robustness.html`. visited 2007-08-09.

[76] Unknown. Robust - businessdictionary.com. web, . URL `http://www.businessdictionary.com/definition/robust.html`. visited 2007-08-09.

[77] Unknown. Robust - dictionary.com unabridged (v 1.1). web, . URL `http://dictionary.reference.com/browse/robust`. visited 2007-10-24.

[78] Jianyun Zhou and Tor Stålhane. A Framework for Early Robustness Assessment. In *Software Engineering and Applications - 2004*.

[79] Jianyun Zhou and Tor Stålhane. Using FMEA for Early Robustness Analysis of Web-Based Systems. *compsac*, 02:28–29, 2004. ISSN 0730-3157. doi: http://doi.ieeecomputersociety.org/10.1109/CMPSAC.2004.1342662.