Software architecture analysis of usability

Folmer, Eelke

Chapter 1

Introduction

At the start of the 21st century, a world without software systems cannot be imagined anymore. Software systems have become an essential part of our daily life although we are often not aware anymore of their presence. Driving a car, listening to music, washing clothes, buying something on the internet, ordering a pizza, almost everything we do involves the use of software systems.

1.1 Software Systems

Software systems are developed with a particular purpose:

- Provide specific functionality that relieves a person from doing a certain task. For example, it is estimated that, automated teller machines (ATM) have replaced more than 500.000 jobs in North America in the last two decades (one ATM replaces the work of about 35 bank employees).

- Provide specific functionality which supports a person to do a certain task in a specific context. For example, content management systems support a user in doing certain tasks such as publishing documents.

In the last decades, software systems have moved from mainframes and traditional desktop systems to almost all (electronic) consumer products. We can find software in mobile phones, digital cameras, DVD players, cars, microwaves, etc.

1.1.1 Feature race

Next to the increase in the number of products with software, we can also identify an increase in the size of software. More and more features are being added to products during their evolution (feature race) to attract and satisfy new customers. Recently, new types of MP3 players, such as the popular iPod, have been introduced featuring a small color screen which allows one to watch photos and play movies on them. A term such as "MP3 player" will probably be replaced by the term "media player" in the near future as these devices do not only play MP3's but do more than that. Adding a screen to an MP3 player allows for a whole new set of software features such as playing games, watching video etc. More and more software is being added to these products to support the increase in hardware features. Because of this increase in size and complexity, software has sometimes become the most valuable asset of a product. It is claimed by Daimler-Benz car manufacturer that the most expensive part in the development of a new Mercedes car is the software system controlling various systems such as the braking system, the fuel injection system, the climate control system, the onboard computer etc. In economic terms, software has become a very valuable and important asset for organizations.

With the increase of software in size, complexity and number of products on which it is deployed, all sorts of problems have emerged. How do we develop and manage software? How can we develop software for different products? Current software is so big and complex that it cannot be developed and managed by a single person. People have to cooperate but because software is often very complex this has become quite a challenge.

1.1.2 Competition

In addition to the feature race problems, organizations have to compete with other organizations, therefore the software product:

- Needs to be delivered on time: to reach the market before competitors do.
- Should be delivered within budget: to increase market share and maximize profits.
- Needs to be delivered with high quality, to keep existing customers happy and attract new customers. For most software products, providing the required functionality is not an issue; software products can only distinguish themselves from those of competitors if they can also provide a high quality. In the future, as users become more critical, and there is more competition because more software is being developed, poor software quality will be a major barrier to the success of new commercial software applications.

There is a continuous market pressure for product development organizations to improve features, quality and to minimize lead time and costs in order to stay in business (innovate or perish). The drawback of this continuous need for innovation is that a company's whole existence depends on the successful launch of one product, which makes product development often a gambling game.

1.1.3 Engineering challenges

How do the feature race and competition affect software development? Software development is restricted around four concerns:

- Features
- Time to market
- Cost
- Quality

Improvements in one may affect at least one of the others negatively. A design decision which affect these concerns has a tradeoff cost; if more features are added to the product, quality must drop or time to market must slip or more money should be invested. Quality can only be increased by putting more people and time on testing, increasing costs. If time to market or costs are cut, features must drop or quality must drop (Berkun, 2002). These concerns make software engineering projects true

challenges and one of these challenges is to find the optimal tradeoff between the four concerns.

Sometimes it is possible to find a way around these tradeoffs. Software engineering sometimes develops new techniques or tools that can work around these constraints but these solutions often lead to new problems. For example features and quality can be increased and costs and time minimized by reusing existing code. But how can we develop code that can be easily reused? This is only one of the problems software engineering tries to solve.

Software engineering is the science that deals with all problems related to the design, implementation and maintenance of software systems. The development of a software system often relies on the intuition of experienced engineers who know how to solve particular problems in their domain. In order for someone inexperienced to design a system, software engineering aims to make this process more formalized by developing methods and techniques that can aid in this purpose. In this thesis we address the problem of how to ensure that software delivers a particular level of quality. Before we discuss what problems are associated to software quality, we first discuss what (software) quality is.

1.2 Software Quality

Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others, that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! (Pirsig,1994)

1.2.1 What is quality?

Before we discuss software quality we first discuss the notion of quality. Quality is a very fuzzy and elusive concept and finding a good definition of quality has occupied the minds of researchers, scientists, philosophers and writers for over a thousand years. The oldest work on quality dates from Aristotle (384-322 BC) (wikipedia: quality). Though quality is a fuzzy term several definitions of quality have been postulated:

"Quality comprises all characteristics and significant features of a product or an activity which relate to the satisfying of given requirements". - (DIN 55350-11)

"The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs" - (ANSI standard)

"The term quality is used to refer to the desirability of properties or characteristics of a person, object, or process" - (wikipedia: quality)

These definitions imply that quality is composed of characteristics, properties and features. For an everyday object (such as a chair) quality can be measured by measuring the following properties:

- Construction quality (e.g. construction material, strength of the joints).

- Aesthetic value (e.g. do people like its design?)
- Fit for purpose (e.g. Can children sit on it? Does it fit under the table?)

A chair made out of steel is obviously stronger than a wooden chair (preserving that they use the same construction) hence it is safe to conclude a steel chair has a higher quality than a wooden chair. Wrong! The problem with measuring quality is that it is not a measure of a product in isolation. Quality should always be evaluated in a particular context. Users and environments are part of this context; for different users in different environments products may have very different qualities. Some users may not like steel and prefer wood, for them a wooden chair has a higher quality. Hence quality is a subjective entity depending on a person's experiences and preferences. Because of this subjectivity quality measures are often relative. E.g. to express the difference between a wooden and a steel chair it is usually impossible to quantify how much better one chair is than the other because there are no absolute measures (how does one quantify: "I like wood over steel"?).

Quality is therefore a very fuzzy concept. In this thesis the following approach to quality is used where it is related to product's acceptance.

1.2.2 Product acceptance

Quality has all to do with product acceptance. E.g. a stakeholder accepts a product only if it meets the stakeholders needs, if it has good quality and if it's available when the stakeholder needs it, for the cost the stakeholder can afford. Between these values tradeoffs exist: some stakeholders may accept a low quality product which does not meet all requirements if it comes quickly and cheaply (such as French fries at McDonalds). Other stakeholders are willing to wait and pay dearly for a high quality compliant product (such as a Ferrari). Even low quality products may be accepted if some of the qualities that the stakeholder prefers are fulfilled (such as a Harley Davidson motorbike which is not known for its high quality, is very expensive but has certain qualities that a user may value). What "good quality" constitutes in a product's acceptance is a subjective notion depending on the stakeholders' preferences and experiences.

1.2.3 What is software quality?

Software quality was originally synonymous to "error free", but nowadays our understanding of software quality has increased. As discussed earlier, software is developed with a particular purpose: e.g. provide specific functionality which supports or replaces a person to do a certain task in a specific context. Fit for purpose is therefore the most important aspect of software quality:

- Does software do what is needed?

Next to that, other aspects play a major role such as:

- Does it do it in the way users expect it to? (usability)
- Does it do it without errors? (reliability)
- Is it fast enough? (performance)

- Does it not hurt or kill any persons? (safety)
- Can we change it as needs change? (modifiability)
- Etc.

All these factors contribute more or less to software quality. The only definition that comes close to this observation is:

"Software quality is (1) the degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations - (IEEE 610.12-1990)

The first part of this definition defines software quality for non interactive systems and the second part defines it for interactive systems (e.g. systems with users). However these definitions still do not state by which properties or characteristics software quality can be measured. Quality models do.

1.2.4 Software quality models

Quality models are useful tools for quality requirements engineering as well as for quality evaluation, since they define how quality can be measured and specified. Being able to evaluate the quality of software is very important, not only from the perspective of a software engineer to determine the level of provided quality but also from a business point of view, such as when having to make a choice between two similar but competing products. Several views on quality expressed by quality models have been defined.

McCall (McCall et al, 1977) proposes a quality model (see figure consisting of 11 factors; such as correctness, reliability, efficiency, etc. McCall's quality model was developed by the US Airforce Electronic System division (ESD), the Rome Air Development Centre (RADC) and General Electric (GE) with the aim of improving the quality of software products and making quality measurable.

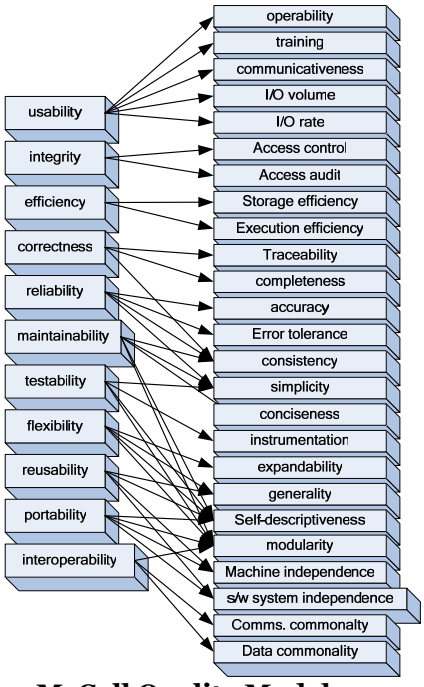


Figure 1: McCall Quality Model

Boehm (Boehm et al, 1981) proposes a quality model which focuses on software quality from the viewpoint of a developer. This model divides quality into the following quality factors: portability, reliability, efficiency, human engineering, testability, understandability and modifiability.

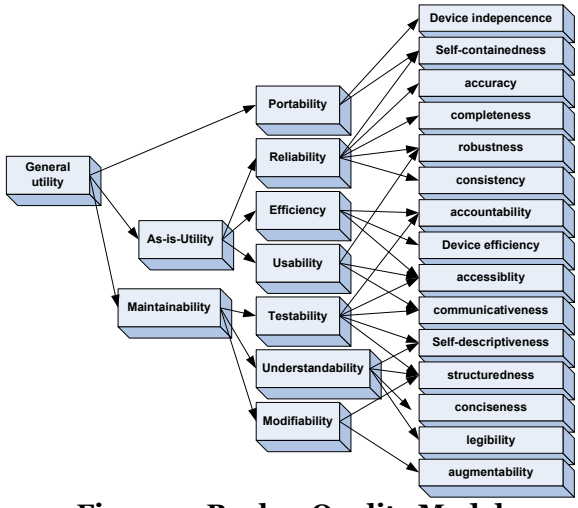


Figure 2: Boehm Quality Model

The ISO 9126 (ISO 9126-1) model describes software quality as a function of six characteristics: functionality, reliability, efficiency, usability, portability, and maintainability. This model takes as a viewpoint the user's perspective. Users mainly

evaluate software quality from how they perceive the software rather than on the basis of internal aspects of the development process.

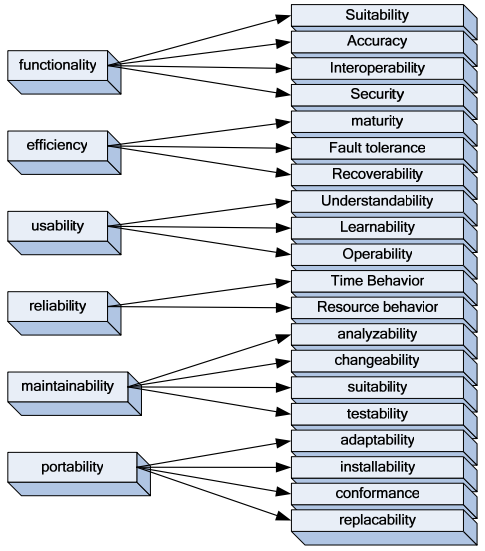


Figure 3: ISO 9126 Quality Model

Though viewed from different perspectives, all these models are based on the idea that software quality is decomposed in a number of "ilities" e.g. high level factors, such as usability and maintainability, which can be further decomposed in a number of sub factors or attributes such as accessibility and testability. These sub factors or attributes can be measured by quality metrics e.g. directly measurable attributes such as "number of errors" or "degree of testing".

There are some differences and overlaps between these quality models. For example they are different in the names they use for the elements in which software quality can be decomposed. McCall and Boehm call them factors whereas ISO 9126 calls them characteristics. Practitioners often just call them "ilities", "qualities" or "quality attributes". There are differences in what names are used for the "ilities". For example usability in the ISO and McCall models is known as human engineering in Boehm's model. The models also differ in how software quality is decomposed; McCall divides it in 11 factors, Boehm into 7 factors, and ISO into 6 characteristics. Sometimes a sub factor in one quality model is a high level factor in another model, for example the sub factor "testability" in the ISO 9126 model is a high level factor in Boehm's quality model.

In general, different authors construct different lists and compositions of what those ilities/qualities are, but they all pretty much agree on the same basic notions. It is therefore very hard to say which quality model has the best quality. The ISO 9126 model provides a complete set of metrics for evaluating quality and is widely adopted by the industry. It further includes attributes such as security which are missing from the other quality models. In addition there is almost a 20 year time lap between the different definitions, recent definition benefit from an increased understanding on the concept of quality.

1.2.5 Usability

One of the qualities that has received increased attention in recent decades is usability. Usability is one of the most important aspects of software quality for interactive software systems. Often software has a high "internal" quality but software becomes useless when it cannot support a user in doing his or her task. As observed earlier on the problem with measuring quality is that it is not a measure of the software product in isolation. Usability is important as it expresses the relationship between software product and the context in which it is deployed e.g. the "external" quality. Usability is also important as it expresses the subjective part of software quality; users and the environment in which they operate are an essential part of this context.

1.2.6 What is usability?

Usability was derived from the term "user friendly" but is often associated with terms such as usefulness, ease of use, quality of use etc. Usability has, similar to many other software engineering terms, many definitions; such as:

The cost/effort to learn and handle a product (McCall et al, 1977)

The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. (ISO 9241-11)

The capability in human functional terms to be used easily and effectively by the specified range of users, given specified training and user support, to fulfill the specified range of tasks, within the specified range of scenarios (Shackel, 1991)

Although there is consensus about the term usability, there are many different approaches to how usability should be measured. As usability is often defined by how it should be measured, many different definitions of usability exist. Basically all these definitions decompose usability into smaller entities e.g. "sub ilities" such as learnability, which are supposed to be more precise and much easier to measure. Various names are used for these entities namely attributes, dimensions, components, scales and factors. It is our opinion that they all mean the same and therefore the term usability attributes is used, which is the term most commonly used.

In addition, authors often provide some model where usability fits in. For example one of the first authors in the field to recognize the importance of usability engineering and the relativity of the concept of usability was (Shackel, 1991). Shackel defines a model where product acceptance is the highest concept. The user has to make a trade-off between utility, usability, likeability. For a system to be usable it has to have high scores on four scales namely: effectiveness, learnability, flexibility and attitude. These scales may be further decomposed. For example; learnability is further decomposed into measurable indicators such as retention and time to learn. Table 1 lists some of the attributes proposed by various authors.

Similar to the software quality models, different authors construct different lists and compositions of what the attributes of usability are, but they all pretty much agree on the same basic notions.

Table 1: Definitions of Usability in Chronological Order

	Shackel 1991	ISO 9126 1991	Hix 1993	Nielsen 1993	Preece 1994	Wixon 1997
Performance	Learnability	Learnability	Learnability	Learnability	Learnability	Learnability
	Effectiveness	Operability	Long-term performance	Efficiency of use	Throughput	Efficiency
	Learnability	-	Retainability	Memorability	-	Memorability
	Effectiveness	Operability		Errors	Throughput	Error rates
User view	Attitude	Attractiveness	Long-term user satisfaction	Satisfaction	Attitude	Satisfaction

All definitions are based on a combination of two different views on how usability should be measured (Bevan et al, 1991):

- Objective / Performance oriented view: usability can be measured by analyzing how a user interacts with the software by for example measuring the time it takes to learn (learnability) or perform (efficiency) a particular task or the number of errors made during a task (reliability).

- Subjective / User view: usability can be measured by analyzing what a user thinks of the product (satisfaction/ attitude).

Since a product has no intrinsic usability of itself, we can only evaluate usability as a function of a particular user performing a specific task with a specific goal in a specific context of use, hence adding a contextually oriented view to these views.

1.2.7 Design for usability

Similar to the different approaches to how usability should be measured there are numerous methods, techniques, guidelines, processes as to how to design and build a usable system. (Keinonen, 1998) identifies two approaches:

1.2.8 Product oriented design

The product oriented design considers usability mainly to be an attribute of the product. It tries to capture existing design knowledge by identifying product properties and -qualities that have proven to have a positive effect on usability. Some examples of the product oriented approach:

Design heuristics

Heuristics such as (Hix and Hartson, 1993, Nielsen, 1993, ISO 9241-11) suggest properties and principles that have a positive effect on usability. For example Nielsen suggests the following "rules of thumb" for interface design:

Table 2: Nielsen's' Heuristics	
Consistency and standards	Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.
Error prevention	Even better than good error messages is a careful design which prevents a problem from occurring in the first place.

Interface guidelines

Guidelines such as (Microsoft, 1992, KDE, 2001, Apple Company, 2004) provide suggestions for low level interface components. For example directions and guidelines for the use of icons, buttons, windows, panels etc. Apple suggests the following detailed guidelines for displaying a menu bar:

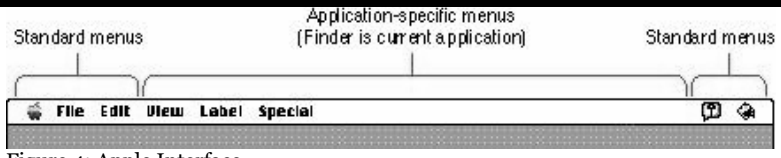
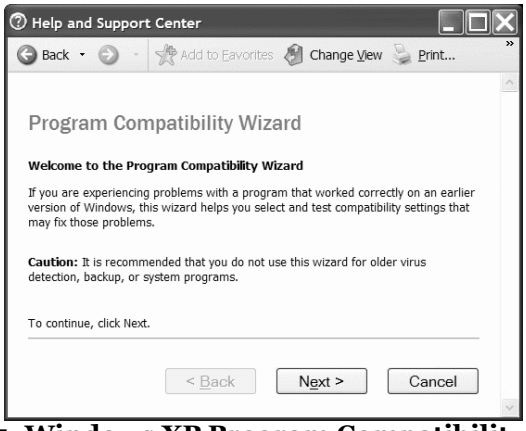
Table 3: Apple Interface Guidelines	
Menu bar	 Figure 4: Apple Interface The menu bar extends across the top of the screen and contains words and icons that serve as the title of each menu. It should be visible and always available to use. Nothing should ever appear on top of the menu bar or obscure it from view. The menu bar should always contain the standard menus--the Apple menu, the File menu, the Edit menu, the Help menu, and the Application menu. The Keyboard menu is an optional standard menu that appears when the user installs a script system other than the Roman Script System.

Table 4: Wizard from Welie Pattern Collection	
Example:	 **Figure 5: Windows XP Program Compatibility Wizard**
Problem	The user wants to achieve a single goal but several decisions need to be made before the goal can be achieved completely, which may not be known to the user.
Use when	A non-expert user needs to perform an infrequent complex task consisting of several subtasks where decisions need to be made in each subtask.
Solution	Take the user through the entire task one step at the time. Let the user step through the tasks and show which steps exist and which have been completed.

Interaction design patterns

Patterns such as (Tidwell 1998, Perzel and Kane 1999, Welie and Trætteberg, 2000) provide solutions to specific usability problems in interface and interaction design. Patterns and pattern languages for describing patterns are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience. A pattern basically is a three-part rule, which expresses a relation between a certain context, a recurring problem, and a solution. An example of such an interaction design pattern is for example a wizard (Welie and Trætteberg, 2000) (See Table 4)

1.2.9 Process oriented design

Process oriented design is bundle of techniques which specifically focuses on analyzing users, analyzing tasks etc, to collect that functionality that makes software usable. User centered design is an iterative process including: analysis, design, testing, and redesign. Techniques are used such as:

- Usability testing techniques such as coaching (Nielsen, 1993), co-discovery learning (Dumas and Redish, 1993, Nielsen, 1993), and question asking (Dumas and Redish, 1993) evaluate usability by observing representative users working on typical tasks using the system or a prototype of the system.

- Usability inspection techniques such as heuristic evaluation (Nielsen, 1994), cognitive walkthroughs (Wharton et al, 1994), or feature inspection (Nielsen, 1994) require usability specialists to examine and judge whether the system follows established usability principles.

- Usability inquiry techniques such as field observation (Nielsen, 1994), Interviews/Focus groups (Nielsen, 1994), surveys (Alreck and Settle, 1994), or questionnaires such as QUIS (Chin et al, 1988) or NAU (Nielsen, 1993) require usability evaluators to obtain information about user's likes, dislikes, needs and understanding of he system by talking to them, observing them or interviewing them using a questionnaire.

1.2.10 Challenges

In the last decades business objectives such as time to market and cost have been preferred over delivering a high quality product; especially in web based systems it was more important to reach the market before competitors and to get a market share than to deliver a high quality product. However in recent years this strategy has proven to lead to failure; because of the intense competition in the software industry and the increasing number of users, if a software product is difficult to use, users will move to a competitive product that is easier to use. Users do not care for nifty features; they are often only interested in getting their work done. Issues such as whether a product is easy to learn to use, whether it is responsive to the user and whether the user can efficiently complete tasks determine to a large extent a product's acceptance and success in the marketplace, apart from other factors such as marketing efforts and reputation.

From product management perspective, having a product with good usability pays off well though initially some money needs to be invested. By initially (before product

release) focusing on usability, companies can see tremendous savings in terms of reduced customer support and reduced product redesign costs. Working usability into a system initially is a very cost effective way to increase customer satisfaction and reduce maintenance / support costs. However, studies of software engineering (Pressman, 1992, Landauer, 1995) projects reveal that organizations do not manage to fulfill this goal. Organizations still spend a relatively large amount of time and money on fixing usability problems during late stage development. Several studies have shown that 80% of total maintenance costs are related to problems of the user with the system (Pressman, 1992). Among these costs, 64% are related to usability problems (Landauer, 1995). A large amount of maintenance costs is spent on dealing with usability issues such as frequent requests for interface changes by users, implementing overlooked tasks and so on (Lederer and Prassad, 1992).

What causes these high costs? Why is ensuring a particular level of usability expensive? The answers to these questions may be found when we look at the software architecture of a system.

1.3 Software Architecture

In recent years the software engineering community has come to the understanding that the software architecture is an important instrument in the fulfillment of quality requirements. The notion of a software architecture was already identified in the late sixties and was related to work on system decomposition and system abstraction. However, it was not until the nineties before architecture design gained the status it has today. Over the past few years several definitions of software architecture have been proposed:

The logical and physical structure of a system, forged by all the strategic and tactical design decisions applied during development (Booch, 1991)

A set of architectural elements that have a particular form. The elements may be processing elements, data elements or connecting elements. (Perry and Wolf, 1992)

Structural models all hold that software architecture is composed of components, connections among those components, plus (usually) some other aspect or aspects, including configuration, style, constraints, semantics, analyses, properties, rationale, requirements, stakeholders' needs (Shaw, 1995)

The structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Bass et al, 1998)

A huge list of all software architecture definitions can be found on (SEI, 2004). The understanding of the abstract concept of software architecture has increased in recent years, though there is still no consensus on one definition. Almost everyone agrees that a software architecture is the product of a series of design decisions each with a rationale leading to some "visible structure" e.g. a set of elements and relations between those elements. These elements may have different abstractions. However whether these elements are subsystems, components, processes etc or whether these relations are data flows, control flows etc. is not really that important. As long as developers in an organization can agree on one definition; there is no point in trying to find the most detailed definition that covers all possible types of software architectures.

Software development is just too divers and complicated to find a definition that suits all. Refining the definition of software architecture is good as long as it increases our understanding of the abstract concept of software architecture, but a definition that is simple to understand and to explain is much likelier to lead to consensus between developers. In recent years one IEEE definition has been formulated which appears to be popular and yet simple.

The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution (IEEE, 1998)

1.3.1 Software architecture description

Apart from that it is hard to find a good definition of software architecture, there is another challenge; how should we describe or document a software architecture? There is always a software architecture, but in order to make it useful it needs to be described. Generally, three arguments for defining an architecture are used (Bass et al, 1998):

- Communication: A software architecture is primarily used as a shared understanding between stakeholders. Different stakeholders depend on the software architecture; software engineers use it as an input to the software design. Project managers use it for estimating effort and planning resources

- Evaluation: A software architecture allows for early assessment of quality attributes (Kazman et al, 1998, Bosch, 2000). E.g. the software architecture is the first product of the initial design activities that allows analysis of quality attributes. Software architects may use an architecture description to analyze the architecture's support for quality.

- Design: Design decisions captured in the software architecture can be transferred to other systems.

The way an architecture needs to be described is closely related to how it is defined. If one uses a definition that includes a rationale then some form of description needs to be used which allows describing the rationale of design decisions. Several types of architecture descriptions have been developed in recent years and these are described in (Clements et al, 2002). In the next section some of these techniques are described.

Source code

The source code is the most basic form of describing a system. The code of a system is often distributed in libraries, modules, components or classes.

```
public class MyButtons extends ... implements ActionListener
...
JButton button1 = new JButton("Hello World");
button1.addActionListener(this);
button1.setActionCommand("hello world");
// add the button to your layout
JButton button2 = new JButton("Exit");
button2.addActionListener(this);
button2.setActionCommand("exit");
// add the button to your layout
...
public void actionPerformed(ActionEvent evt)
{
     if(evt.getActionCommand().equals("hello world")
     {
          System.out.println("Hello world");
     } else
     {
          System.exit(0);
```

Figure 6: Java Source Code

One way to describe the architecture is not to describe it at all e.g. we have only the source code to look at. In Figure 6 some source code is displayed. As can be observed source code is not a good way to describe the architecture as it does not fulfill the goals (Bass et al, 1998) an architecture should fulfill:

- Communication: source code is often too large and often only understood by the ones who wrote it.

- Evaluation: source code is only available when the system has been developed, which makes it useless in a forward engineering perspective. Although some things can be learned from analyzing source code. For example, the coupling between modules or components may indicate how easy a particular component is to modify although such a description is far from sufficient for analyzing other quality attributes.

- Design: Source code is always written in a specific development language such as java or C++ hence if you want to develop software in a different language it is not possible to reuse this code.

Box and line diagrams

A simple technique to describe the architecture is to describe the architecture using boxes and lines which connect the boxes. Boxes and lines are easy to use because they can mean anything the architect desires, for example, the boxes can represent subsystems and the lines and arrows between the boxes can represent expected call relationships between the subsystems. An example box and line diagram of can be found in Figure 7.

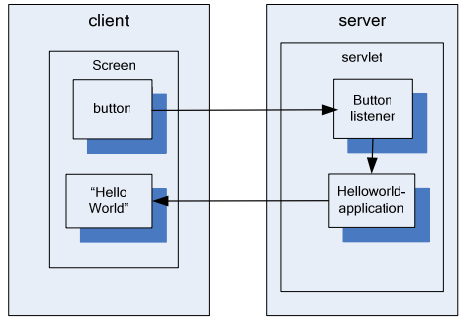


Figure 7: Box and Line Diagram

Although this technique is sufficient to describe a coarse picture of the architecture, this technique has several shortcomings.

- Communication: Box and line diagrams lack being able to describe details such as being able to describe interfaces, control behavior, rationale etc. It lacks semantics e.g. what does a particular box mean? An object? A subsystem? A method? Or merely a logical grouping of functionality? In Figure 7 boxes describe objects (button) and processes (button listener). Arrows can also be interpreted in different ways: is an arrow a control flow? Or a data flow? Without semantics a box and line architecture can lead to ambiguity; i.e. it can be interpreted in different ways which does not contribute to communication

between developers. Behavior (e.g. interaction between elements) is also difficult to describe using box and lines diagrams.

- Evaluation: A box and line diagram may be sufficient for some expert based analysis but for automated evaluation (such as state checking) box and line diagrams are not sufficient because without semantics there is no way to interpret a box and line diagram.

- Design: A box and line diagram is the result of a series of design decisions. Because of the lack of semantics and the lack of being able to describe some rationale it is difficult to transfer design decisions in a box and line diagrams to other systems.

View models

One of the shortcomings of box and line diagrams is that it does not provide a complete picture of the architecture; a better way to describe the architecture is by describing it from different views. Each view has its own description method and styles and addresses a specific set of concerns which are of interest to a specific stakeholder. Because architectures are often too complex to visualize in one view, different views can be used that act as a filter which show what is of interest to a particular stakeholder. In (Bass et al, 1998) a number of different viewpoints are described. For example, to express the interests of project management (such as assigning work), the architecture can be described in a module view. Using this view project management can assign engineers to work on a particular module.

4+1 view model

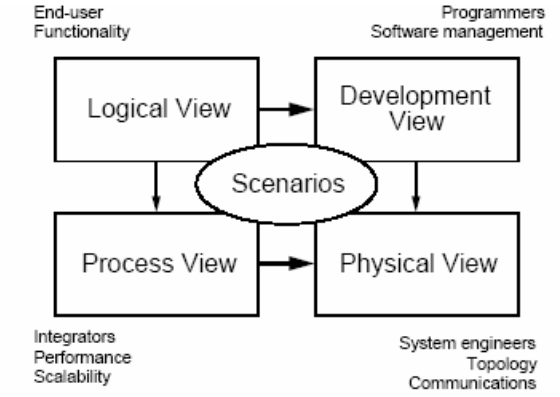


Figure 8: the 4+1 View Model and Stakeholders

A popular view model technique is the 4+1 view (Kruchten, 1995). The 4+1 view uses subsets of the Unified Modeling Language (UML) to describe particular views and consists of the following five views:

- Logical view: describes the object model of the design. This view primarily supports the functional requirements; what the system should provide in terms of services to its users.

- Process view: describes the concurrency and synchronization aspects of the design.

- Physical view: describes the mapping(s) of the software onto the hardware.
- Development view: describes the static organization of the software in its development environment.
- Scenarios: scenarios or use cases (sequences of interactions between objects and processes) can be used to show how the four views work together. Scenarios are an abstraction of the most important requirements.

Because a view merely acts as filter on the software architecture the views are not independent from each other. Changing something in one view is likely to affect something in another view. For example the characterization of logical elements is an indication to define process elements. The autonomy and persistency of objects is an indication for the process view. The logical view and development view are very similar but address different concerns, the same goes for the process and physical view; processes are mapped onto the available physical hardware.

The Soni-Nord-Hofmeister (SNH) view model

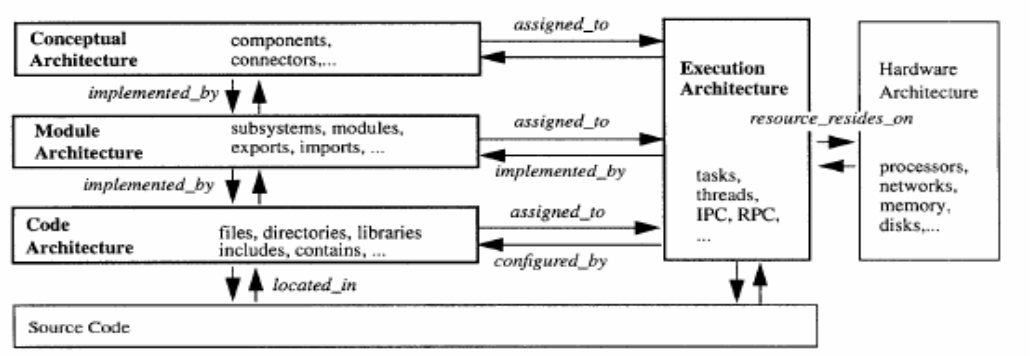


Figure 9: the SNH View Model

Another view model that is used in industry is the one developed by (Hofmeister et al, 1999) and consists of the following four views:

- Conceptual view: describes the architecture and a collection of components and connectors which can later be decomposed in other views.
- Module view: describes the application in concepts that are present in the programming models and platforms that are used.
- Code view: describes the source code organization, the binary files and their relations in different ways.
- Execution view: describes the system in terms of elements that are related to execution e.g. hardware architecture, processing nodes and operation concepts such as threads and processes.

The SNH views are also not independent; a conceptual element is implemented by one or more elements of the module architecture. Module elements are assigned to runtime elements in the execution architecture. Each execution element is implemented by some module elements. Module elements are implemented by elements of the code

architecture. There is a mapping between the execution architecture (threads/ processes) and the hardware architecture (cpu's etc).

The problem with the view models is that it is often difficult to keep the views consistent. For example, in the 4+1 view a change in the development view likely affects something in the logical view (although there is often not a one to one mapping between those views). There are some overlaps and differences between the SNH views and the 4+1 views but on the whole these view models are quite similar. View models provide a more complete description of an architecture than box and line diagrams and they fulfill the following goals:

- Communication: view models can describe details such as interfaces, behavior (by means of scenarios), rationale etc. A multi view model aids communication; only that is shown what is of interest to a particular stakeholder which minimizes confusion. The risk however, is that stakeholders tend to forget a view is part of something complex and changing something in one view certainly may have an effect on other views.

- Evaluation: As view models provide a more complete picture of the architecture it sufficient for expert based analysis. Certain views are described using notation languages such as (Booch, 1991), which makes them suitable for automatic interpretation/analysis.

- Design: View models provide semantics and capture an architecture's rationale. The design decisions captured in the views models can be transferred to other systems.

Architecture description languages (ADL)

ADLs are formal languages that can be used to represent the software architecture by a formal textual syntax but often also as a graphical representation that corresponds with the textual syntax. In order to do that, ADLs are often incorporated in development tools such as code generators, analysis tools and simulation tools. In (Clements, 1996) a survey of different ADLs can be found. Rapide (Luckham et al, 1995, Luckham et al, 1995, Luckham et al, 1995) and Unicon (Shaw et al, 1995) are among the most popular.

Unified modeling language (UML)

UML is a set of ADLs that are used for OO analysis and design. It defines a specific set of views (diagrams) that have been standardized, by the Object Management Group (OMG). Because this language has been standardized, it has gained enough support to make it the industry standard language for visualizing, specifying, modeling and documenting object-oriented and component-based architectures. UML supports classes, abstract classes, relationships, behavior by interaction charts, state machines, grouping in packages, etc. UML defines nine types of diagrams:

- Use case diagram: identifies the primary elements and processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases." The use case diagram shows which actors interact with each use case.

- Class diagram: refines the use case diagram into classes, methods and attributes and defines a detailed design of the system.

- Object diagram: captures the state of different classes in the system and their relationships or associations at a given point of time.
- State diagram: represents the different states that objects in the system undergo during their life cycle.
- Activity diagram: captures the process flows in the system.
- Sequence diagram: represents the interaction between different objects in the system. The important aspect of a sequence diagram is that it is time-ordered.
- Collaboration diagram: groups together the interactions between different objects.
- Component diagram: represents the high-level parts that make up the system.
- Deployment diagram: captures the configuration of the runtime elements of the application.

Concerning the goals of an architecture ADLs fulfill:

- Communication: Some ADLs such as UML have been standardized and therefore widely accepted. ADLs provide a very detailed picture of one aspect (such as behavior) of an architecture. Similar to the view models multiple views are required to provide a complete description of the architecture.
- Evaluation: Because of its standardization, UML has proven to be very useful for the development of all sorts of development oriented tools. UML can be interpreted, generated and analyzed by a variety of tools.
- Design: ADLs have been defined that are able to describe design rationale, behavior, etc) making them suitable for transferring design decisions to other systems. Using model driven engineering UML diagrams can be used to generate systems

Various techniques can be used to describe software architectures. The choice for a particular technique depends on many factors such as the organization, the application domain, the design objectives etc. In this thesis conceptual views (Hofmeister et al, 1999) are used for describing software architectures.

1.3.2 Software architecture design

The software architecture results from set of technical, business and social activities between stakeholders with the common objective to develop software that provides specific functionality such that a balance in fulfillment of requirements is achieved. Stakeholders are persons (such as customers, end users, developers etc) that are affected by, or have an interest in the development outcome. Requirements are provided by the stakeholders. In general, functional requirements define what the system needs to do (e.g. "backup my hard drive") whereas non functional requirements describe how the system will do it (e.g. "backup my hard drive without crashing"). Non functional requirements are often associated with software quality requirements e.g. the "ilities" such as usability, maintainability, etc.

The software architecture is the first product of the initial design activities that allows analysis and discussion about different concerns. As discussed in the software quality section (section 1.2), software design is constrained around four "tradeoffs" namely features, quality, cost and time-to-market. But how do these concerns affect software architecture design?

1.3.3 Features / evolution

A feature is an essential aspect or characteristic of a system in a domain. In the past when a software product was developed it was often assumed that the set of features it should provide was complete. However this assumption has proven to be far from realistic. New features are continuously being added to the product during a product's evolution and existing features change. This is because the context in which the user and the software operate is continually changing and evolving. Software systems are much more dynamic; exchanging data with other software systems. Users may find new uses for a product, for which the product was not originally intended. For example Microsoft Word can now be used to write emails and to create Webpages with, something that could not be imagined a couple of years ago. Requirements are much more dynamic and evolutionary and a software architect should allow for this evolution to happen. When new features need to be added during late stage development or product evolution sometimes the architecture permits these new features from being implemented. We call this the retrofit problem. Often new features are implemented as new architectural entities (such as components, layers, objects etc) or an extension of old architectural entities. If a software architecture has already been implemented then changing or adding new entities to this structure during late stage design is likely to affect many parts of the existing source code. Rewriting large parts of source code is expensive, and it also erodes (Gurp and Bosch, 2002) the original software architecture when features are added in an ad-hoc fashion. Software architects should therefore design their architectures in such a way that to a certain extent, future requirements may still be facilitated. The problem with this requirement is that it is often difficult to predict future requirements. A good intuition of trends, such as the recent shift to use Extended Markup Language (XML) for describing objects is essential to designing a software architecture with a longer expiration date.

1.3.4 Software quality

As already identified by (Parnas, 1972), there is a strong relation between the software architecture and software quality. This relationship is twofold:

Quality is restricted by SA design.

A recognized observation in literature (Buschmann et al, 1996, Shaw and Garlan, 1996, Bass et al, 1998, Bosch, 2000) is that software architecture restricts the level of quality that can be achieved i.e. it often proves to be hard and expensive to make the necessary changes to a running system to improve its quality. Certain quality improving design solutions inhibit a retrofit problem i.e. in order to improve the quality of a system, solutions need to be applied which cannot be supported by the architecture.

Quality must be achieved through architecture design.

The earliest design decisions have a considerable influence on the qualities of a system. Certain architecture design decisions have proven to have a significant impact on

certain qualities; for example, a layered style (Buschmann et al, 1996) improves modifiability but negatively affects efficiency. This shows that qualities are not independent; a design decision that positively affect on one quality might be very negative for another quality. Some qualities frequently conflict; for example design decisions that improve modifiability often negatively affect performance the same goes for security and usability. Tradeoffs between qualities are inevitable and it is important to make these explicit during architecture design because such design decisions are generally very hard to change at a later stage.

1.3.5 Cost & time to market

Delivering a product on time and within budget are often very "hard" constraints enforced by project management. E.g. software engineers are much likelier to cut on features and quality when a project is late and well over budget just to keep management happy.

Next to these four "business-oriented" concerns a software architect has to deal with other constraints:

1.3.6 Complexity

Similar to how an architect of a sky scraper needs to deal with an external force namely gravity (the building should not fall apart), so must a software architect deal with an internal force namely complexity (the software architecture should not become too complex). In order to fulfill quality requirements and to allow evolution, software architects often need to add more flexibility to an architecture.

For example, for a web based content management system it may be unknown which types of databases will be used in two or three years (relational / object oriented). Also it may be unknown whether a particular vendor will still support a particular type of database in the future. In order to increase modifiability an architect may decide to design an architecture which uses flexible data abstraction mechanism which allows the use of different types of persistent storage mediums (for example a text file or different types or brands of databases) instead of using an architecture which uses a fixed type persistent medium (for example a text file or a database).

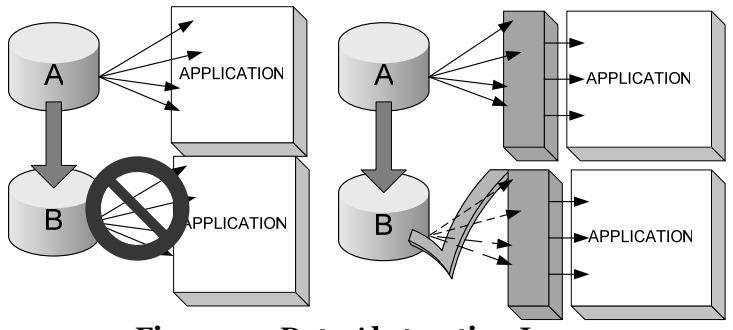


Figure 10: Data Abstraction Layer

Using a data abstraction mechanism (see Figure 10) persistent storage medium dependencies are separated from the rest of the application and are only known to the data abstraction mechanism, which may be a single component or layer. If a new type

of medium is used, only the dependencies between the data abstraction mechanism need to be changed rather than dependencies throughout the application. Though this solution increases modifiability, a data abstraction mechanism often requires the use of new components or new layers, which makes the architecture more complex. Instead of having direct access to the persistent storage medium, all read/write operations must go through the data abstraction layer which may negatively affect performance.

An architect should not forget that the main purpose of a software architecture is that it should aid communication between developers and other stakeholders. An architecture that can only be understood by the architect does not fulfill that purpose. At all times should the software architect maintain a "conceptual integrity" (Brooks, 1995), i.e. knowing when to stop adding complexity to maintain simplicity in design.

1.3.7 Reuse

Another constraint that architects have to deal with is that software should be designed in such a way that certain software artifacts (components, modules etc) can be reused in other contexts to develop new software systems. When code is reused the costs for developing the code can distributed among the different software projects. If code is reused software can be developed in a shorter time. In addition, a higher quality of the software artifacts may be expected since the chance of detecting failures in the artifacts is much higher because they are used in different contexts. Designing for reuse is not simple and generally adds a lot of complexity to the architecture.

1.3.8 Software product families

Related to reuse is the notion of software product families (Bosch, 2000). Software product families are architectures designed to manage and enhance many product variations that may be needed for different markets. The aim of software product families is to have intra-organizational reuse through the explicitly planned exploitation of similarities between related products while preserving ability to support differences. For example, mobile phones come in many different hardware configurations (large screen / small screen) and provide different features (such as a camera). To establish reuse of software, an architecture should be designed in such a way that variations are supported e.g. certain features are included or excluded or can be turned on or off in the final software product. Using software product family techniques, companies such as Nokia, HP, and Philips have reduced time-to-market, engineering costs, and defect rates by factors of 3 to 50.

1.3.9 Design constraints

Architecture design is a complex non-formalized process and there is a lack of methods that specifically guide architecture design. A software system should be compliant to its requirements, yet must have high quality and be delivered on time within budget. In addition, the architecture should not become too complex but yet support evolution and reuse. Because of these informal constraints, architecture design decisions are therefore often taken on intuition, relying on the experience of senior software developers (Svahnberg et al, 2000).

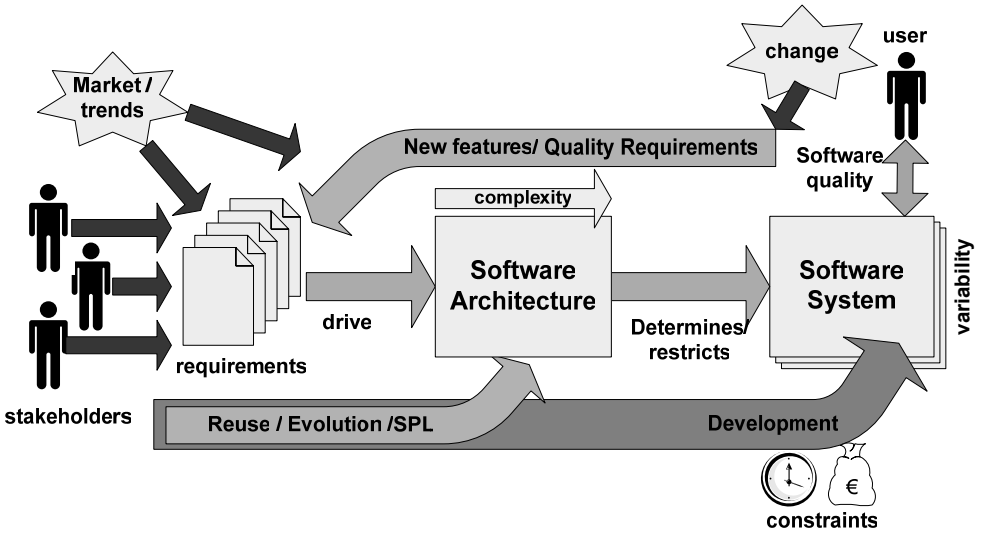


Figure 11: Software Architecture Design Constraints

By capturing what was previously very much the "art" of designing an architecture we can make it less intuitive and more accessible to inexperienced designers. Two approaches are identified:

- Capture existing design knowledge.
- Turn software architecture design into a repeatable process; e.g. provide an architect with a number of steps for performing architecture design.

1.3.10 Capture design knowledge

In order to design or improve the design of an architecture an analyst should know how particular design problems should be solved.

Patterns

Some of the best practices concerning software architecture design have been captured in the forms of patterns. The concept of a pattern was first proposed by (Alexander et al, 1977)

A 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 offers definitive solutions to the problems of urban architecture and design; however we can conclude Alexander has had a greater impact on computer science than on architecture. In 1995 an object-oriented design pattern book (Gamma et al 1995) was published which became a instant classic in the literature of object-oriented (OO) development providing solutions in the form of software patterns to typical OO problems (such as managing object creation, composing objects into larger structures, and coordinating control flow between objects). The goal of software patterns was to

capture design experience in a form that can be effectively reused by software designers without having to address each problem from scratch.

After the success of the "Gang-of-Four" book (Gamma et al 1995) a pattern community emerged that specifies patterns for all sorts of problems:

- Architectural styles (Buschmann et al, 1996),
- Object oriented frameworks (Coplien and Schmidt, 1995),
- Domain models of businesses (Fowler, 1996),
- Interaction patterns (Tidwell 1998, Welie and Trætteberg, 2000)
- Etc.

All sorts of design solutions to functional- and software quality related problems concerning the design of a software architecture have been described by means of design patterns (Gamma et al 1995), architectural patterns and -styles (Buschmann et al, 1996). Though not all architecture related design knowledge has been captured in the form of patterns, most of the design knowledge has been described in a problem solution dichotomy which they share with patterns. Architects often use the following design knowledge:

Architectural styles

An architectural style (Buschmann et al, 1996, Shaw and Garlan, 1996) is a description of an architecture layout of the highest form of abstraction, which generally improves certain quality attributes and impairs others. Imposing an architectural style generally completely reorganizes the architecture. An example of an architectural style is a layered style. A layered architectural style decomposes a system into a set of horizontal layers where each layer provides an additional level of abstraction over its lower layer and provides an interface for using the abstraction it represents to a higher level layer. In general, using a layered style improves reusability; if a layer embodies a well defined abstraction and has a well defined interface the layer can be used in different contexts. Additionally, layers improve modifiability /portability (Buschmann et al, 1996) as individual layer implementations can be replaced by equivalent implementations. Layers may also have negative impact on certain qualities. A layered style is usually less efficient as communication has to go trough a series of layers.

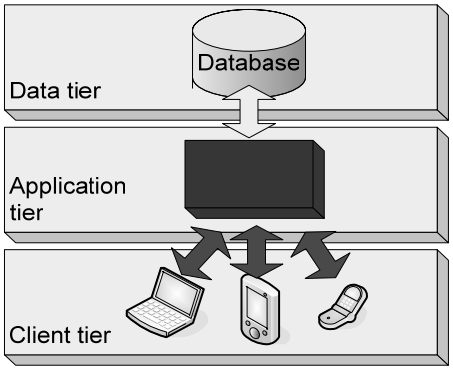


Figure 12: 3-Tier Architecture

Architectural patterns

Architectural patterns are a collection of rules (Richardson and Wolf, 1996) that can be imposed on the system, which require one aspect to be handled as specified. An architectural pattern is different form an architecture style in that it is not predominant and can be merged with architectural styles. An example of an architectural pattern is a database management system (DBMS). A DBMS generally extends the system with an additional component or layers, but it also imposes rules on the original architecture components. Entities that need to be kept persistent (i.e. the ability of data to survive the process in which it was created) need to be extended with additional functionality to support this aspect. The use of a DBMS generally has some effect on qualities such as performance, maintainability/ modifiability and security. For example, performance is negatively affected as all database operations have to go though the DBMS. Modifiability is improved as all database dependencies are localized to the DBMS.

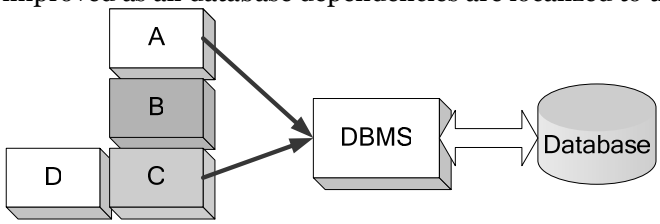


Figure 13: Database Management System

Design patterns

Design patterns (Gamma et al 1995) are a collection of modifications which may improve certain quality attributes. Design patterns do not change the functionality of the system, only the organization or structure of that functionality. Applying a design pattern generally affects only a limited number of classes in the architecture. The quintessential example used to illustrate design patterns is the Model View Controller (MVC) design pattern. The MVC pattern (Buschmann et al, 1996) is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller. This pattern decouples changes to how data are manipulated from how they are displayed or stored, while unifying the code in each component. The use of MVC generally leads to greater flexibility and modifiability. There is a clearly defined separation between components of a program problems in each domain can be solved independently. New views and controllers can be easily added without affecting the rest of the application.

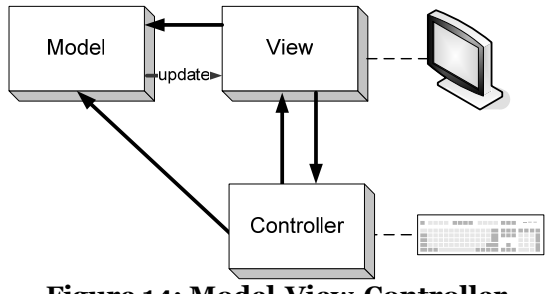


Figure 14: Model-View-Controller

OO Frameworks

OO frameworks are an integrated collection of components that collaborate to produce a reusable architecture for a family of related applications for a particular domain. Often applications are developed by inheriting from and instantiating framework components. An example of such an OO framework is the Apache Struts framework. Struts provides a solution in the domain of Graphical User Interfaces (GUI); the framework implements the MVC design pattern. Struts provides its own controller component and integrates with other technologies to provide the Model and the View. Struts enforces the separation of UI interface components with back end business logic component. Basically it is collection of Java code designed to rapidly and easily build solid web based applications.

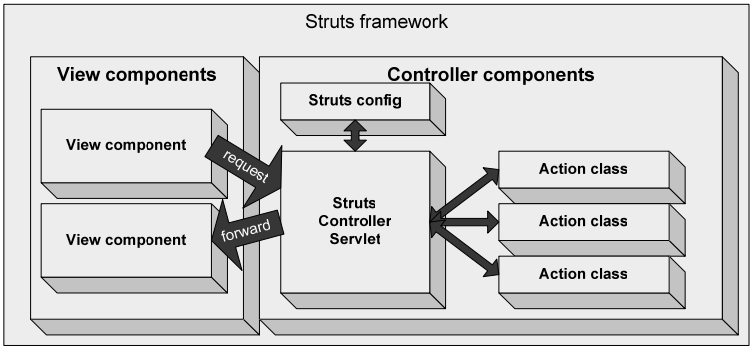


Figure 15: Struts Framework

Role of Architecture Design

The design knowledge that has been captured in the form of patterns, styles, components or frameworks form the building blocks that an architect can use to build a software architecture. Because of increasing reuse, the granularity of the building blocks has increased from design patterns to OO frameworks; therefore the role of software engineering and architecture design has significantly changed. Where a decade ago only some GUI libraries where available to a developer, now there is a wide variety of commercial off-the-shelf (COTS) and open source components, frameworks and applications that can be used for application development. The role of a software engineer has shifted from developing all the code to developing glue code. I.e. code that is written to connect reused components, frameworks and applications. Glue code is often written in scripts such as Tool Command Language (TCL) to parameterize the component with application specific interactions. When the component needs to interact with other components in the system, it executes the appropriate script in the language interpreter.

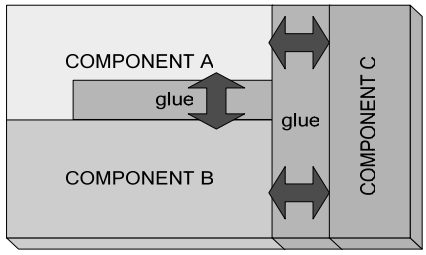


Figure 16: Glue Code

Software architecture design has also changed. Software architecture design has shifted from starting with a functional design and transforming it with appropriate design decisions (such as using an architectural pattern) to selecting the right components and composing them to work together while still ensuring a particular level of quality. Being able to analyze the impact of using a particular component or framework on software quality is essential to developing a high quality system.

The second approach to make software architecture design more formalized is by turning it into a repeatable process.

1.3.11 Turn architecture design into a process

A method provides an analyst with a set of clearly defined steps for performing design and analysis. During architecture design, it is still possible to change the design cheaply. The structure of a method ensures that some reasoning about the architecture and discussion between different stakeholders is taking place. Software architecture analysis, and improvement of the architecture based on the assessment results is a method that aids design.

Software architecture analysis

Instead of relying on intuition or experience, the effect of a design decision on software quality should be (formally) analyzed before this decision becomes too expensive to retract. An architecture description may provide information on the provided quality of the system. The following architecture assessment techniques have been developed:

Metrics

Metrics are a tool for quality control and project management. Metrics provide guidelines on how to measure different attributes of a project or smaller pieces of code. For example, a metric may measure the number of code lines, the complexity of code or the amount of comments. Metrics can be used to find areas that are prone to problems. Metrics can be tracked across multiple teams or projects or they can be used to monitor the development of a single system.

From a forward engineering perspective (e.g. architecture design) code metrics are not useful. Source code is only available when the system has been developed but then it is already too late to retract design decisions which may affect software quality. Metrics however can be useful tools to give an indication of the quality of code that is to going to be reused. Metrics related to the object-oriented features (Chidamber and Kemerer, 1991) of a program (such as the number of abstract data types) can be useful for software architecture analysis of maintainability.

Simulation & Mathematical Models

Simulation of the architecture uses an executable model of the application architecture. This comprises models of the main components of the system composed to form an overall architecture model. It is possible to check various properties of such a model in a formal way and to animate it to allow the user or designer to interact with the model as they might with the finished system. Prototyping is similar to simulation but only implements a part of the architecture; and executes that part in the actual intended context of the system. Closely related to simulation are Mathematical models. Mathematical models have been developed by various research communities such as

high performance computing (Smith, 1990) and real-time systems (Liu and Ha, 1995). These models allow for static evaluation of operational quality attributes of architectural design models.

Simulation and mathematical model based assessment are primarily suited for evaluating operational quality attributes such as performance or reliability but also for evaluating the functional aspects of a design. In order to create a model often a very detailed description of the architecture is needed in the form of ADLs such as UML or message sequence charts (MSC).

Scenario Based Assessment

Another type of assessment technique that is increasingly used is scenario based assessment. To assess a particular quality attribute, a set of scenarios is developed that concretizes the actual meaning of a requirement. For instance, maintainability requirements may be specified by defining change profiles that captures typical changes in the requirements, underlying hardware and so on. A typical process for scenario based technique is as follows: A set of scenarios is elicitated from the stakeholders. After that a description of the software architecture is made. Then the architecture's support for that quality attribute, or the impact of these scenarios (change scenarios) is analyzed depending on the type of scenario. Based on the analysis of the individual scenarios, an overall assessment of the architecture is formulated. Scenario based assessment is a more structured and formalized way to reason about design decisions and tradeoffs and has proven to be successful to asses qualities that are difficult to assess from a forward engineering perspective such as maintainability and modifiability (Bengtsson, 2002).

Several scenario based assessment techniques have been developed in the last decade.

SAAM

The Software Architecture Analysis Method (SAAM) (Kazman et al, 1994) was among the first to address the assessment of software architectures using scenarios. SAAM consists of the following steps:

1. Describe the candidate architecture. The candidate architecture or architectures should be described in a syntactic architectural notation that is well-understood by the parties involved in the analysis.

2. Develop scenarios. Develop task scenarios that illustrate the kinds of activities the system must support and the kinds of changes which it is anticipated will be made to the system over time.

3. Perform scenario evaluations. For each indirect task scenario, list the changes to the architecture that are necessary for it to support the scenario and estimate the cost of performing the change.

4. Reveal scenario interaction. Determining scenario interaction is a process of identifying scenarios that affect a common set of components.

5. Overall Evaluation. A ranking is established between scenarios that reflect the relative importance of the quality factors that the scenarios manifest.

ATAM

From SAAM, the Architecture Tradeoff Analysis Method (ATAM) (Kazman et al, 1998) has evolved. ATAM uses scenarios to identify important quality attribute requirements and tradeoffs between quality attributes for the system. ATAM consists of the following nine steps:

1. Present the steps of ATAM to the stakeholders.
2. A stakeholder with the business perspective presents the business drivers.
3. The software architect presents the architecture.
4. Architectural approaches are recorded by the assessment team.
5. Stakeholders and assessment team generate a quality attribute utility tree.
6. The approaches are analyzed for the most important quality requirements.
7. Stakeholders brainstorm and prioritize scenarios to be used in the analysis.
8. The analysis of the architectural approaches is completed by mapping the scenario to the elements in the architecture description.
9. The results are presented.

QASAR

The Quality Attribute-oriented Software ARchitecture design method (QASAR) (Bosch, 2000) is an architecture design method that employs explicit assessment of, and design for the quality requirements of a software system.

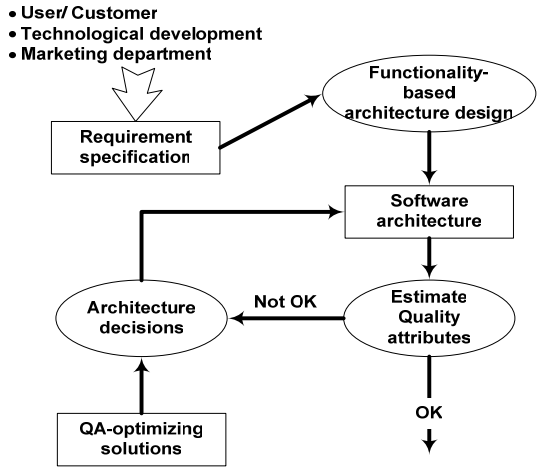


Figure 17: QASAR Architecture Design Method

QASAR consists of the following steps:

1. Collect the requirements from the stakeholders; the users, customers, technological developments and the marketing departments.

2. Start with a design of the software architecture based on the functional requirements.

3. Evaluate the preliminary version of the software architecture design with respect to the quality requirements by using a qualitative (scenarios) or quantitative (simulation/mathematical modeling) assessment technique.

4. Using the assessment results, compare the estimated quality attributes to the values in the specification. If these are satisfactory, the design process is finished.

5. Otherwise, improve the software architecture by selecting appropriate quality attribute optimizing or improving design solutions.

SAAM, ATAM and QASAR do not focus on a single quality attribute but rather provide a generic method and steps for the assessment and reasoning about tradeoffs for different quality attributes. Some specific scenario based quality-attribute assessment techniques have been developed, which have been derived from these generic assessment techniques but have steps or techniques that have been tailored for that specific quality attribute.

SAAMER

The Software Architecture Analysis Method for Evolution and Reusability (SAAMER) (Lung et al, 1997) is an extension to SAAM and addresses quality attributes such as maintainability, modifiability and reusability. SAAMER activities overlap with SAAM, but is different in that it provides a specialized architecture description method and provides criteria for when to stop generating scenarios in the scenario development phase.

ALMA

In (Bengtsson and Bosch, 1999), a scenario based Architecture-Level Modifiability Analysis (ALMA) method is proposed. ALMA specifically focuses on the assessment of modifiability and consists of the following five steps:

1. Determine the aim of the analysis.

2. Describe software architecture.

3. Find the set of relevant change scenarios.

4. Determine the impact of the change scenarios.

5. Draw conclusions from the analysis results.

The goal of an architecture analysis method is to understand and reason about the effect of design decisions on the quality of the final system, at a time when it is still cheap to change these decisions. Which technique to use depends on many factors such as which quality requirements must be fulfilled.

1.3.12 Challenges

One of the key problems with many of today's software is that they do not meet their quality requirements very well, because some qualities are hard to assess during an

early stage or because no assessment techniques have been developed for these qualities. In addition, software engineers have come to the understanding that quality is not something that can be easily "added" to a software product during late stage, since software quality is determined and restricted by architecture design and its often unknown how a particular design decision will affect certain qualities. Software architecture analysis has been successfully applied to improve the quality of systems, but designing an architecture that fulfills its quality requirements is still a difficult task as:

- Quality requirements are often weakly specified.
- Not all qualities can be assessed during architecture design.
- Not enough is known / documented about how architecture design restricts and fulfills quality requirements.
- Not enough is known / documented about tradeoffs in achieving quality requirements.
- There is a lack of design methods that guide architecture design for quality.
- As quality requirements may change during development and deployment, software architectures should provide some flexibility to cost effectively be able to support future quality requirements.

1.4 Research Context

In the previous section several challenges concerning software architecture and software quality, e.g. usability, were identified. In this section we outline the research context in which our research was conducted.

This thesis focuses on one important aspect of software quality, namely usability. Usability is important as the success of a software product to a great extent depends on this quality. Usability similar to other qualities is restricted and fulfilled by software architecture design. The quintessential example that is always used to illustrate this restriction is adding undo to an application. Undo is the ability to reverse certain actions (such as reversing making a text bold in Word). Undo significantly improves usability as it allows a user to explore, make mistakes and easily go some steps back; facilitating learning the application's functionality. However from experience, it is learned that it can be very hard to implement undo in an application because it requires large parts of code to be completely rewritten e.g. undo is hard to retrofit. Software architects are often not aware of the impact of usability improving solutions on the software architecture nor are there any techniques that focus on analyzing software architectures for their support of such solutions. As a result, avoidable rework is frequently necessary. This rework leads to high costs and because tradeoffs need to be made, for example between cost and quality, leads to systems with less than optimal usability.

This and many other of such typical examples, led us to investigate the relationship between software architecture and usability.

1.4.1 STATUS project

The work presented in this thesis has been funded by the European Union funded STATUS (Software Architecture that supports Usability) project. STATUS was an ESPRIT project (IST-2001-32298) financed by the European Commission in its Information Society Technologies Program, Action Line IV.3. The project started on 1-12-2001 and its duration was 36 months. The aim of this project was to gain an in-depth understanding of the relationship between software systems usability and architecture.

1.4.2 STATUS partners

STATUS was a collaborative research project between the following partners from academia and industry across Europe.

University of Groningen - The Netherlands

The University of Groningen (RuG) has about 18.000 students and consists of ten faculties. The SEARCH software engineering research group belongs to the Research Institute of Mathematics and Computing Science. The focus of the research group is on software architecture assessment and design, software product lines and software reuse. Research is typically performed in an empirical manner through extensive co-operation with the software industry.

Universidad Politécnica Madrid - Spain

The Universidad Politécnica de Madrid (UPM) has 2700 students. The Software Engineering Group (SEG), belongs to the School of Computer Science and is commended with teaching and research related to Software Engineering. The group focuses on requirements engineering, methodologies, software process modeling and improvement, usability engineering and reuse and testing.

Imperial College of Science, Technology and Medicine - UK

The Department of Computing at Imperial College (IC) is widely recognized as one of the leading centers of research and advanced training in computer science and software engineering in Europe and has over 100 doctoral students. The Distributed Software Engineering section focuses on methods, tools and techniques for the development of composite, heterogeneous and distributed software-intensive systems.

Imperial Highway Group - Spain

Based in Spain, the Information Highway Group (IHG) is an advanced information technology consultancy, whose expertise includes the following innovative technologies and virtual systems: Electronic Publishing, Distance Learning, Virtual Commerce, Electronic Marketing and Electronic Consulting.

LogicDIS - Greece

LogicDIS is group of companies which provides total quality solutions for private and public establishments. LogicDIS has over 1200 employees and the company is active in the following areas: - development of software packages for large and small companies, - systems integration and development of large-scale customization software for large companies and the government - marketing and distribution of the packaged software

in Greek and overseas markets -customer support for the above markets -training seminars for executives, both in software products and management tools.

The project consortium included research partners specialized in usability (UPM) and software architectures (RuG & IC), as well as industrial partners (IHG & LogicDIS) which provided researchers with feedback on which architectures are being used and what usability attributes were relevant in their experience.

1.4.3 STATUS objectives

The aim of STATUS project was to identify the connections between software architecture and the usability of the resultant software system. The project provides the characteristics of software architectures that improve software usability. The results of the project have been particularized for e-commerce applications because they are of fundamental importance in the information society and because usability is a critical factor for these applications.

The scientific and technological objectives of the project were the following:

1. Identify usability attributes that are possibly affected by software architecture;

2. Study how usability attributes can be influenced by software architecture; development of an architectural style that supports usability;

3. Identify architectural patterns that are repeated in the e-commerce domain to study their relationship with usability, and its improvement with respect to this quality attribute;

4. Propose a development process that integrates traditional software development techniques with techniques proper to the field of usability.

A full motivation and explanation for these research objectives can be found in the STATUS project proposal (STATUS). In order to achieve the project objectives, first the attributes that are possibly affected by software architecture were identified. However this initial research (a literature survey we performed prior to the start of the STATUS project) led to new insights in this relationship. One of these new insights was that usability improving architectural solutions can be added to existing architectures. The focus of the project and objective 2 was revised from finding the optimal architecture that supports usability, to the development of architecture evaluation techniques for usability and finding architectural solutions that can improve the usability of an architecture. The following work packages were defined:

Table 5: STATUS workpackages overview

WP	Objective	Subtasks
1	Project management	-
2	Usability attributes affected by software architecture	2.1 Identification of usability decomposition in literature
		2.2 Identification of usability decomposition from industrial experience
		2.3 Selection of usability attributes affected by software architecture
3	Study of the	3.1 Usability requirement specification technique

	usability/software architecture relationship	3.2 Scenario-based architectural assessment technique
		3.3 Simulation-based assessment technique
		3.4 Architecture-level usability improvement techniques/patterns/styles
4	Proposal of architecture / development of a set of evaluation techniques.	4.1 Identification of architectural patterns for e-commerce applications
		4.2 Evaluation of usability in architectural patterns
		4.3 Improvement on architectural patterns according to results of Work package 3
5	Integrated development process with usability techniques	5.1 Selection of a general user-centered software development process
		5.2 Selection of usability techniques applicable to software development
		5.3 Overview of integration into the general software process
		5.4 Complete specification of the process
		5.5 Feedback from the other partners
6	Development of applications in the e-commerce domain	6.1 Evaluation of usability in typical applications of industrial partners
		6.2 Development by IHG and LogicDIS of applications with STATUS results
		6.3 Evaluation of usability of the applications where STATUS results have been applied
		6.4 Analysis and comparison of the usability tests results before and after applying STATUS results
7	Dissemination	-
8	Exploitation	-

The research in this thesis has followed the planning of the STATUS project. The articles published in this thesis result from work done in workpackages 2 and 3. Our research has been validated in workpackage 6 in the industrial settings provided by the industrial partners.

1.5 Research Questions

This is an article based thesis, therefore a set of general research questions is outlined which are then connected to these articles in the conclusion section. As observed in section 1.2.10, there is a problem with current software development as organizations spend a large amount of time and money on fixing architecture related usability problems during late stage development. To improve upon this situation a software architecture should be analyzed for its support of usability, and improved if the support proves not to be sufficient.

The overall research question that motivates this thesis is:

RQ: Given the fact that the level of usability is restricted and determined by the software architecture how can we design a software architecture that supports usability?

In order to solve this research question, we first need to identify how usability is restricted by architecture design. Unfortunately prior to the start of the STATUS project (1 December 2001) little research had been conducted in this particular area e.g. the relevant design knowledge concerning usability and software architecture needed to be captured and described in order to use it for assessment and design.

1.5.1 Investigating the relationship between SA and usability

The work done in work package 2 has focused on identifying the relationship between usability and software architecture. As discussed in the beginning of this section certain usability improving design solutions were identified to be architecture sensitive and need to be dealt with during architecture design. In addition to the main research question for this topic, four more specific research questions have been defined concerning investigating this relationship:

RQ-1: What is the relationship between software architecture and usability?

- RQ-1.1: How does software architecture design restrict usability?
- RQ-1.2: What does architecture sensitive mean?
- RQ-1.3: How can we describe these architecture sensitive design solutions?
- RQ-1.4: Can we use this knowledge to improve current design for usability?

1.5.2 Development of an SA assessment technique for usability

Initial work on the investigation of the relationship between usability and software architecture revealed that it is possible to design an architecture for usability rather than find one architecture that optimally supports usability. The work done in workpackage 3 and 6 has focused on the development and validation of an architecture assessment technique for usability. Four more detailed research questions have been formulated:

RQ-2: How can we assess a software architecture for its support of usability?

- RQ-2.1: Which techniques can be used for analysis of usability?
- RQ-2.2: How should usability requirements be described for architecture assessment?
- RQ-2.3: How do we determine the architecture's support for a usage scenario?
- RQ-2-4: Can architecture assessment successfully be used for improving the usability of the final system?

The two main research questions directly relate to the two approaches towards formalizing architecture design discussed in section 1.3.9 e.g. capture existing design knowledge (RQ-1) and turn software architecture design into a repeatable process (RQ-2). The articles in this thesis are organized into two parts, each bundling articles that related to one of the two research questions.

1.6 Research Methods

Research methodology is the study of how to perform scientific research. Many sciences have well-defined research strategies providing guidance on how to perform research, however software engineering is still a comparatively young discipline, therefore it does not have this sort of well-understood guidance (Shaw, 2002) yet.

Research in software engineering is fundamentally different from research in other fields of (computer) science. The problem domain of software engineering is that of improving the practice of software manufacturing. Problems are identified by observing how organizations develop software. Solutions to these problems can be tools, techniques, methodology etc. This context puts limitations on how research can be performed:

- In an industrial setting there is little opportunity for using well established research methods such as experiments.
- There are many factors involved in the success or failure of a software project hence making it difficult to model using simulations.
- A solution which addresses a technical factor sometimes affects the software engineering process; a solution must fit in the overall context of business, organization, process and organization and must be accepted by the developers.

Researchers in software engineering therefore rely on research methods that have emerged from empirical sciences such as sociology and psychology. In this thesis, we rely on:

- Case studies for providing us with examples for developing theories and for validating our approaches.
- Action research as the type of research strategy.

1.6.1 Case studies

Case studies are a form of qualitative descriptive research that have as a goal the development of detailed, intensive knowledge about a single "case", or of a small number of related "cases" (Robson, 1993). Information is collected using a range of data collection techniques such as observations, interviews, protocols, tests, etc. Case studies can provide quantitative data, though qualitative data are almost invariably collected. The fields of sociology and anthropology are credited with the primary shaping of the concept as we know it today.

Case studies are a comparatively flexible method of scientific research. Case studies focus on exploration and discovery rather than prescription or prediction. Researchers are free to discover and address issues as they arise in their experiments. Case studies are sometimes criticized as being too subjective because personal biases may creep into the research. Results may not be generalizable; if one relies on one or a few cases as a basis for cognitive extrapolations one may run the risk of inferring too much from what might be circumstance. Conclusions from case studies are not as strong as those from controlled experiments; using multiple cases adds more validity to the conclusions from a study (Robson, 1993).

Another risk with case study research is that researchers change the direction of their research based on the observations made. This may invalidate the original case study design, leaving unknown gaps and biases in the study. To avoid this, researchers should report any preliminary findings to minimize the likelihood of bias. A research approach that fulfils these constraints is called action research.

1.6.2 Action research

Action research is different from other types of research in that it adds change to the traditional research purposes of description, understanding and explanation (Robson, 1993). Action research is an applied research strategy which involves cycles of data collection, evaluation and reflection with the aim of improving the quality or performance of an organization.

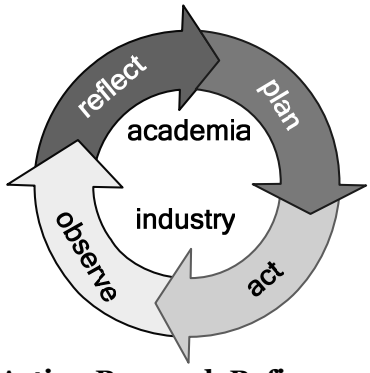


Figure 18: Action Research Refinement Cycle

Central to action research is the collaboration and participation between researchers and those who are the focus of the research (such as an organization). The benefits of a close operation are that a researcher gets a more complete understanding of the research issues. Organizations may get a deeper understanding of the problem when a researcher is involved. On the other hand a close cooperation may result in that the researcher loses some control over the design of the research, as an organization often has a stake in the results of the research.

Software engineering is still a very young discipline, compared to other engineering disciplines. Researchers in software engineering therefore rely on qualitative and empirical research methods for understanding of the problems in the domain. When research matures, research shifts to precise and quantitative models (Shaw, 2002). In our case, investigating the relationship between usability and software architecture, almost no research has been performed, case studies and action research therefore fit well within the flexible "explorative" qualitative research strategy that we adopt.

1.7 Thesis Structure

The body of this thesis consists of 8 articles that have either been published in journals (chapters 2,4,5), conferences (3,7,8) or have been submitted to a journal (6,9). These articles can be found in the next chapters and are divided over the two topics introduced in the research section. The articles within each topic are presented in chronological order.

1.7.1 Investigating the relationship between usability and SA.

Chapter 2, published in January 2004 in the Journal of Systems and Software (Folmer and Bosch, 2004), describes the work we did prior to the start of the STATUS project; it identifies the weaknesses of current usability engineering practice. It argues that usability is restricted and fulfilled by architecture design. It further identifies that no

techniques or tools exist that allow for architectural design and assessment of usability. This survey provided us with valuable insights into the relationship between usability and software architecture and led to the redefinition of STATUS objectives. This survey also led to the definition of our research questions.

Chapter 3, presented at the 10th International Conference on Human-Computer Interaction (Folmer and Bosch, 2003), in 2003 is the first result on investigating the relationship between usability and software architecture by identifying a set of usability patterns that require architectural support and outlining a framework for describing them.

Chapter 4, published in April 2003 in the Journal of Software Process Improvement and Practice (Folmer et al, 2003), is an expansion of the framework presented in chapter 3 which expresses the relationship between usability and software architecture. This framework consists of an integrated set of design solutions, such as architecture sensitive usability patterns and properties, which are costly to retrofit because of their architectural impact. This framework can be used to heuristically evaluate and design software architectures.

Chapter 5, accepted in January 2005 for the Journal of Information and Software Technology (Folmer et al, 2005) further refines the concept of architecture sensitive usability patterns by presenting a new type of pattern called a bridging pattern, which is an extension of an interaction design patterns. In addition to outlining potential architectural implications that developers face implementing this pattern we describe in detail how to generally implement this pattern.

Chapter 6, submitted to the International Journal of Software Engineering and Knowledge Engineering in August 2005 extends the concepts behind the SAU framework (architecture sensitive patterns, attributes and properties) to other qualities (e.g. security and safety). This chapter outlines and presents a framework that formalizes some of the essential relations between software architecture and the qualities usability, security and safety. It presents one architecture sensitive pattern for each quality and shows how tradeoffs between these qualities must be made when implementing such patterns. It also introduces the concept of a boundary pattern; e.g. a pattern that provides a solution to a quality problem (security) but also provides a solution that counters the negative effects on another quality (usability) that traditionally come with implementing this pattern.

1.7.2 Development of an SA assessment technique for usability

Chapter 7, presented at the 9th Working Conference on Engineering for Human-Computer Interaction (Folmer et al, 2004) in 2004, presents the Scenario based Architecture Level UsabiliTy Assessment technique (SALUTA) that we developed. This method is illustrated with a case study (Webplatform) in the domain of web based content management systems.

Chapter 8, accepted for the EUROMICRO conference on SE (Folmer and Bosch, 2005a) in 2005, presents the results of using SALUTA at two case studies (eSuite / Compressor).

Chapter 9, submitted in January 2005 to the Journal of Systems and Software (Folmer and Bosch, 2005b), presents the results of three case studies performed with SALUTA

and presents a set of experiences with performing architecture analysis of usability. A summary of this paper has been accepted as a conference paper entitled "Cost Effective Development of Usable Systems; Gaps between HCI and Software Architecture Design" for the Fourteenth International Conference on Information Systems Development, August 2005 .

Chapter 10 finally presents how the research questions we identified have been answered and presents some conclusions.

1.7.3 Related publications

The following related publications have not been included in this thesis as these papers were published at workshops as a means to get feedback on ongoing research and research results.

- Eelke Folmer, Jilles van Gurp, Jan Bosch, Scenario based assessment of software architecture usability, published in ICSE 2003 workshop "Bridging the Gaps Between Software Engineering and Human-Computer Interaction", Portland, USA, May 2003.

- Eelke Folmer, Jan Bosch, Architectural Sensitive Usability Patterns, VikingPloP workshop, Bergen, Norway, September 2003.

- Lisette Bakalis, Eelke Folmer, Jan Bosch. Workshop: "Identifying Gaps between HCI, Software Engineering and Design and boundary objects to bridge them". Computer Human Interaction 2004, Vienna, Austria, April 2004.

- Eelke Folmer, Jan Bosch, Cost Effective Development of Usable Systems; Gaps between HCI and SE. ICSE Workshop "Bridging the Gaps Between Software Engineering and Human-Computer Interaction-II", Edinburgh, Scotland, May 2004.