A Pattern Framework for Software Quality Assessment and Tradeoff Analysis.

Article *in* International Journal of Software Engineering and Knowledge Engineering · August 2007

DOI: 10.1142/S021819400700332X · Source: DBLP

Jan Bosch
Chalmers University of Technology
619 PUBLICATIONS **22,257** CITATIONS

A Pattern Framework for Software Quality Assessment And Tradeoff Analysis

Eelke Folmer, Jan Bosch
Department of Mathematics and Computing Science
University of Groningen, PO Box 800, 9700 AV the Netherlands

Abstract

The earliest design decisions often have a significant impact on software quality and are the most costly to revoke. One of the challenges in architecture design is to reduce the frequency of retrofit problems in software designs; not being able to cost effectively improve the quality of a system, a problem which frequently occurs during late stage. Software architecture assessment is essential in the design of a high quality system. However, assessing the effect of individual design decisions with respect to quality is often complicated by the fact that it's hard to identify exactly how particular qualities and quality factors are improved or impaired by design decisions. In this paper we present a framework that formalizes some of the relationships between software architecture and software quality; it compiles existing design knowledge (quality improving patterns) in a format suitable for architecture assessment. This framework may prevent the retrofit problem and can assist in reasoning about intra- and inter- quality tradeoffs. We illustrate our framework by creating an instance for it for the qualities usability, security and safety.

Keywords

Software architecture, software quality, tradeoffs, patterns, security, safety, usability.

1. Introduction

A recognized observation in literature [1,2,3,4] is that software architecture restricts the level of quality that can be achieved. The earliest design decisions have a considerable influence on the qualities of a system, therefore it is important to get these right, as they are the most costly to revoke. The effect of such design decisions should therefore be assessed at various phases of the software development life cycle to verify whether quality requirements have been met [4].

One of the challenges in architecture design and analysis is to an extent reduce the frequency of retrofit problems [5] in software designs. One must make the right decisions during architecture design, which is hard as quality requirements frequently change during a product's evolution. As business requirements change the priority of quality requirements changes and earlier design decisions may need to be revoked and new design decisions implemented, which is sometimes not easily supported by the software architecture. Adding quality improving solutions during late stage development is hard as some of these solutions may require the system to be completely reworked. These solutions are often implemented as new architectural entities (such as components, layers or objects) and relations between these entities or an extension of existing architectural entities. If a software architecture has already been implemented then changing or adding new entities during late stage design is likely to affect many parts of the existing source code. Large parts of the code need to be rewritten and restructured which is expensive. Such design solutions are called 'architecture sensitive'. Knowing which quality improving design solutions are architecture sensitive is important, as it can potentially avoid the retrofit problem and hence avoidable rework of the system during late stage. For qualities such as usability, architects are often not aware of these constraints [6]. The frequency of retrofit problems can be reduced by designing an architecture with more flexibility towards implementing such solutions during late stage. Adding flexibility usually adds more complexity to the system, which makes a system harder to change. A good architect should make a tradeoff between these conflicting concerns.

Architecture design is hard as different tradeoffs need to be made; not only between flexibility and complexity but also between other qualities. Qualities are not independent, a design decision that has a positive effect on one quality might be very negative for other qualities [7]. Some qualities frequently conflict: for example design decisions that improve modifiability often negatively affect performance. Usually design solutions describe which high level qualities are affected though it's much more valuable to know in detail how the quality is affected on a lower level. Qualities such as usability can be decomposed into finer grained factors, such as learnability and efficiency. For a particular design decision, also tradeoffs between quality factors need to be made. For example for usability, learnability and efficiency frequently conflict. Tradeoffs between quality factors need to be made within a quality but also between quality factors of different qualities (see Figure 1). Understanding precisely how a particular design solution improves or impairs certain qualities allows an architect to make a more informed decision considering tradeoffs.

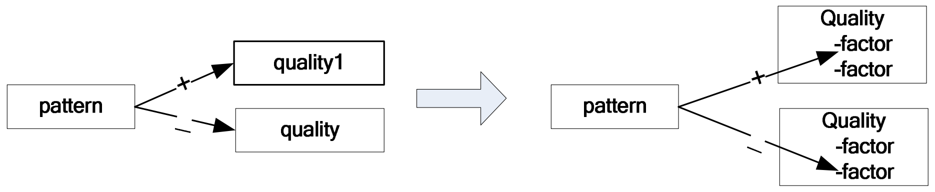

Figure 1: Detailed quality tradeoffs.

We should collect such architecture sensitive solutions to prevent the retrofit problem. To be able to effectively use them for architecture assessment and design they need to be organized in a framework which explicitly connects them to software quality, so we can reason about tradeoffs. In this paper such a framework is presented which organizes existing design knowledge (quality improving patterns) in a format suitable for addressing the two aforementioned problems.

The remainder of this paper is organized as follows. The next section presents the framework that expresses the relationship between quality and software architecture. Section 3 presents an instance of this framework for three qualities namely usability, safety and security. Section 4 discusses some of the issues we encountered during the definition of our framework. Section 5 discusses related work. Section 6 discusses future work and the paper is concluded in section 7.

2. The Software Architecture - Quality Framework

The framework we present in this chapter has been derived from our previous work on describing the relationship between usability and software architecture [8]. This framework has been made more generic to include other qualities. The framework is composed of a layered model consisting of 6 different layers (see Figure 2) each layer is presented in detail below:

2.1 Architecture Sensitive Patterns

The core layer of our framework consists of architecture sensitive patterns; a special category of patterns that inhibit the retrofit [8] problem.

2.1.1 Patterns

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. Design patterns [9,2] are extensively used by software engineers for the actual design process as well as for communicating a design to others. Software patterns first became popular with the object-oriented Design Patterns book [9]. Since then a pattern community has emerged that specifies patterns for all sorts of problem domains: architectural styles [2], object oriented frameworks [10], domain models of businesses [11], but also many software quality related patterns have been identified such as usability patterns [12,13,14], reliability patterns [15], safety patterns [16,17], maintainability patterns [18], security patterns [19,20] and performance patterns [15]. Although design knowledge has been described in different forms such as guidelines or strategies, in this paper we only focus on patterns because of their consistent format and the existence of numerous pattern collections. A plethora of patterns can be found in literature, current practice and on the web [21] but in our framework we only consider those patterns that are "architecture sensitive" and which affect software quality.

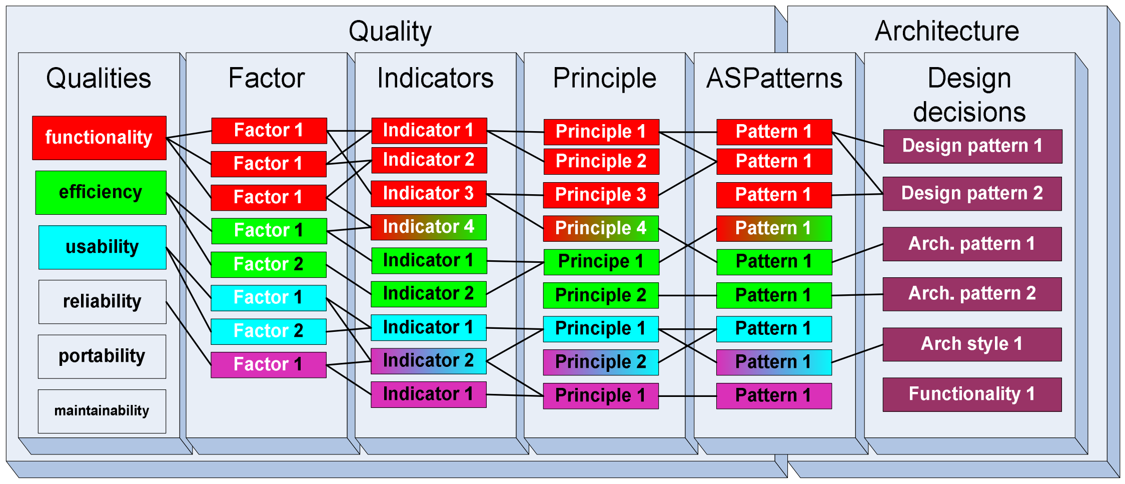

Figure 2: Relation between quality and software architecture.

2.1.2 Architecture sensitivity

To define an architecture sensitive pattern the following criteria are used:

- **Improves quality**: The pattern specifically addresses to one or more quality related problems and its solution improves one or more qualities, or as a side effect, it may impair some other qualities.
- **Architecture impact**: The pattern for a given system has a significant impact on the architecture. The impact can work on different levels in the architecture. In [4] we identify three types of architectural transformations e.g. architectural style, architectural pattern and design pattern. The impact of a pattern can be either one of those transformations or a combination. Each transformation has a different impact on the architecture.
    - Imposing an *architectural style* generally [2] completely reorganizes the architecture [4] 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, which improves maintainability.
    - *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 from an architecture style in that it is not predominant and can be merged with architectural styles. An example of an architectural pattern is for example implementing 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.
    - *Design patterns* [9] 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 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.
- **Effort needed** to implement the pattern: The architectural impact of a pattern can be any of the three aforementioned transformations. However the effort needed to implement such a pattern during late stage may be high or low for any type of transformation. Therefore only those patterns are considered, that for a reasonable system, have a high implementation effort during late stage.

2.1.3 Pattern format

As identified by [22] patterns are an effective way of capturing and transferring knowledge due to their consistent format and readability. To describe the patterns we identified, the following pattern format is used:

Quality: Which quality this pattern aims to improve.

Authors: The authors that have written this pattern.

Problem: Problems are related to a particular quality aspect of the system.

Use when: A situation giving rise to a quality problem. The use when extends the plain problem-solutions dichotomy by describing specific situations in which the problem occurs.

Solution: A proven solution to the problem. However a solution describes only the core of the problem, strategies for solving the problem are not described in this pattern format and for a fully detailed solution we refer to the specific patterns authors' reference.

Why: The rationale (why) provides a reasonable argumentation for the specified impact on a quality when the pattern is applied. The why describe which quality factors are improved or which other factors have to suffer.

Quality principles: Quality principles are guidelines and design principles in the domain of that quality which the pattern addresses. We specifically list the principles the pattern authors themselves mention in the pattern description. (See also the quality principles layer in section 2.4)

Architectural implications: An analysis of the architectural impact of the pattern and which responsibilities may need to be fulfilled by the architecture. Again these are taken from the author’s pattern description.

To be able to use architecture patterns for assessment and design they are connected to specific parts of software quality in the following layers:

2.2 Qualities
The qualities layer is the top layer in our framework and is based on existing 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. Several views on quality expressed by quality models have been defined. McCall [23] proposes a quality model consisting of 11 qualities: correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability and interoperability. Boehm [24] proposes a quality model that divides quality into the following qualities: portability, reliability, efficiency, human engineering, testability, understandability and modifiability. The ISO 9126 [1] model describes software quality as a function of six qualities: functionality, reliability, efficiency, usability, portability, and maintainability. In general, different authors construct different lists and compositions of what those qualities are, but they all agree on the same basic notions, therefore in our framework either one of these quality models can be taken. In our framework we have chosen to use the ISO 9126 quality model.

2.3 Quality factors
The second layer consists of quality factors which explain how to measure that particular quality. All quality models are based on the idea that qualities are decomposed in a number of high level quality factors. Qualities factors are supposed to be much easier to measure than the qualities themselves. For example according to ISO 9126 [1] model usability is decomposed into learnability, efficiency of use, reliability of use etc. How a particular quality should be decomposed into which different factors is still a topic of discussion in some quality research areas and falls our of the scope of this paper.

2.4 Quality indicators
The third layer consists of quality indicators which explain how quality factors may be measured for a particular system. How to measure a quality factor for a given system depends very much on the type of system and on what can be measured. Quality indicators are directly observable principles of software such as the time it takes to perform a specific task, number of hack attempts or number of crashes. These indicators are very application specific; in an application without users (such as control software for a robot arm) you cannot measure how long it takes for a user to perform a certain task. An indicator can be an indicator for multiple quality factors. For example the number of application crashes may be an indicator for the reliability of use factor of safety but also for the stability factor of reliability.

2.5 Quality principles
The fourth layer consists of quality principles which explain how to design for a particular quality. Because telling you how to measure a particular quality factor does not guide design there need to be some sort of guidelines, heuristics or design principles that describe how a particular quality can be improved. For example to improve security one can use the principle of error prevention [16]. We call these quality principles. Guidelines such as ISO/IEC 9126 [25] give standard definitions of quality factors and proposed measures, together with a set of application guidelines. This quality principles form the fourth layer in our framework.

2.6 Design decisions
In addition to the software quality related layers we defined an additional "design decision layer" (patterns form the fifth and design decision the sixth layer) which belongs to the software architecture solution domain. Architecture sensitive patterns may be implemented by an architecture style [2], architectural pattern [2] and or design patterns [9] or by some framework consisting of these elements. The choice to use a style or pattern or design pattern is motivated by a particular design decision. For example a design decision could be: “support platform independence”, this could be implemented by a layered architecture style that provides hardware abstraction in its lowest layer. This last layer was introduced to show that certain patterns share the same implementation. In our work on a framework for usability [8] we identified that with a single mechanism sometimes multiple patterns can be implemented. For example the usability patterns multi level undo and cancel can all be implemented using the Command pattern [8].

2.7 Relationships between the elements of the framework
This layered framework is useless without connections between the layers. The quality factors, indicators and principles layers allow us to connect architecture sensitive patterns to software quality. Relationships between the elements in the framework are the most essential part of our framework. Using these relationships, a designer can analyze when it has a quality requirement which architecture sensitive patterns may need to be implemented to fulfill this requirement. This relationship is twofold; when heuristically evaluating an architecture for a set of quality requirements an architect can identify the support an architecture provides for these requirements, by identifying how particular patterns that have been implemented improve certain aspects of software quality using the relations in the framework.

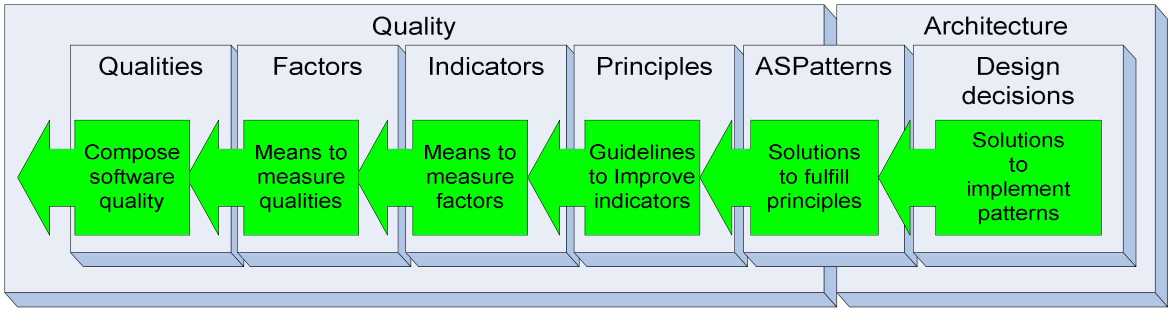

Figure 3: Relationships between the elements in the framework.

The relationships have been defined as follows:

- Software quality consists of several qualities.
- Qualities can be decomposed in factors which are easier to measure than the qualities.
- Observable properties can be defined for a system which are an indicator for a particular quality factor.
- Guidelines can be used to improve specific indicators & factors.
- Architecture sensitive patterns address to one or more principles and improve or impair indicators, factors and qualities
- Design decisions such as the use of design pattern implement one or more architecture sensitive patterns.

In our framework these relationships are not exclusively defined within a particular quality, as is the case with most current pattern descriptions, but they can be defined between any layer for any quality. E.g. a pattern may also improve or impair indicators and factors of different qualities, directly showing the inter quality tradeoffs. In the next section we will illustrate these relationships for an actual instance of the framework.

3. The Security-Safety-Usability Framework

In this section we will illustrate our framework by creating an implementation of this framework for the qualities security, safety and usability. These three qualities are recognized in the ISO 9126 Standard [25], though security and safety are not recognized as high level qualities such as usability, but rather as parts of functionality and quality in use. For simplicity, we consider usability, safety and security to be of the same level and we use the following definitions:

- **Safety:** Software is considered safe if it is impossible that the software produces an output that would cause a catastrophic event for the system that the software controls.
- **Security:** The security of a system is the extent of protection against some unwanted occurrence such as the invasion of privacy, theft, and the corruption of information or physical damage.
- **Usability:** 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 [26].

These qualities were mainly chosen because usually tradeoffs need to be made between these qualities; security and safety frequently conflict with usability. In addition, practice shows that improving these qualities during late stage is often costly. Being able to visualize tradeoffs between those qualities for different design decisions can aid in the decision making process. We identified several security patterns [19,20], safety patterns [16] and usability (interaction) patterns [13,14,27] in literature. Our previous work [8,28] and others [6] has focused on identifying architecture sensitive usability patterns. In this paper we take the same approach but extend it to include other qualities. In the pattern collections we studied, pattern authors present implementation details which allowed us to identify which of those patterns were are architecture sensitive. Although we identified a number of suitable patterns, this paper we show how three of these patterns can be organized in our framework. Part of each pattern is listed below, for more details on the patterns (such as detailed implementation and forces) we refer to the respective authors of the pattern.

3.1 Selected patterns

Single Access Point

Quality	Security
Authors	Yoder and Barcalow [19].

Problem A security model is difficult to validate when it has multiple “front doors,” “back doors,” and “side doors” for entering the application.

Use when Software that has multiple access points.

Solution Set up only one way to get into the system, and if necessary, create a mechanism for deciding which sub-applications to launch.

Why Having multiple ways to open an application makes it easier for it to be used in different environments. (accessibility). An application may be a composite of several applications that all need to be secure. Different login windows or procedures could have duplication code (redundancy). A single entry point may need to collect all of the user information that is needed for the entire application. Multiple entry points to an application can be customized to collect only the information needed at that entry point. This way a user does not have to enter unnecessary information.

Security Principles

- Secure weakest link.
- Principle of least privacy.
- Keep it simple.
- Be reluctant to trust. [29]

Architectural Considerations As claimed by the authors themselves, single access point is very difficult to retrofit in a system that was developed without security in mind [19]. A singleton [9] can be used for the login class especially if you only allow the user to have one login session started or only log into the system one. A singleton could also be used to keep track of all sessions and a key could be used to know which session to use. Introducing Single Access Point is considered to have a **high** impact on the software architecture.

Warning

Quality Safety

Authors Mahemof and Hussay [16]

Problem How can we be confident the user will notice system conditions when they have arisen and take appropriate actions?

Use when Use this pattern when identifiable safety margins exist so that likely hazards can be determined automatically and warnings raised.

Solution Provide warning devices that are triggered when identified safety-critical margins are approached.

Why Software itself is good at monitoring changes in state (better than humans are). Software is good at maintaining a steady stage in the presence of minor external aberrations that would otherwise alter system state. Software is not good at determining the implications of steady state changes and appropriate hazard recovery mechanisms. User's workload is minimized and the hazard response time is decreased. Conditions may be user defined.

Safety Principles

- Automation.
- Monitoring.
- Error Prevention.
- Error reduction. [16]

Architectural Considerations There are a lot of different implementations of the Warning pattern. The implementation depend on many factors: What is there to observe? Who establishes connections between the observer and observable? Who notifies who when something goes wrong? Etc. Some monitoring needs to take place where either the Observer [9] or Publisher Subscriber [2] kind of patterns need to be used. When introducing Warning in an application that does not already use these patterns, it can mean some new relationships between objects need to be established and possibly some new objects (such as an observer) need to be added. Therefore, introducing Warning is considered to have a **high** impact on the software architecture.

Multi-level Undo

Quality Usability

Authors Folmer and Welie [28]

Problem Users do actions they later want reverse because they realized they made a mistake or because they changed their mind.

Use when You are designing a desktop or web-based application where users can manage information or create new artifacts. Typically, such systems include editors, financial systems, graphical drawing packages, or development environments. Such systems deal mostly with their own data and produce only few non-reversible side-effects, like sending of an email within an email application. Undo is not suitable for systems where the majority of actions is not reversible, for example, workflow management systems or transaction systems in general.

Solution Maintain a list of user actions and allow users to reverse selected actions.

Why Offering the possibility to always undo actions gives users a *comforting feeling*. It helps the users feel that *they* are in control of the interaction rather than the other way around. They can explore, make mistakes and easily go some steps back, which facilitates learning the application's functionality. It also often eliminates the need for annoying warning messages since most actions will not be permanent.

Usability Principles

- Explicit user control.
- Explorability.
- Error management.
- Minimize cognitive load. [8]

Architectural Considerations There are basically two possible approaches to implementing Undo. The first is to capture the entire state of the system after each user action. The second is to capture only relative changes to the system's state. The first option is obviously needlessly expensive in terms of memory usage and the second option is therefore the one that is commonly used.

Since changes are the result of an action, the implementation is based on using Command objects [9] that are then put on a stack. Each specific command is a specialized instance of an abstract class Command. Consequently, the entire user-accessible functionality of the application must be written using Command objects. When introducing Undo in an application that does not already use Command objects, it can mean that several hundred Command objects must be written. Therefore, introducing Undo is considered to have a **high** impact on the software architecture.

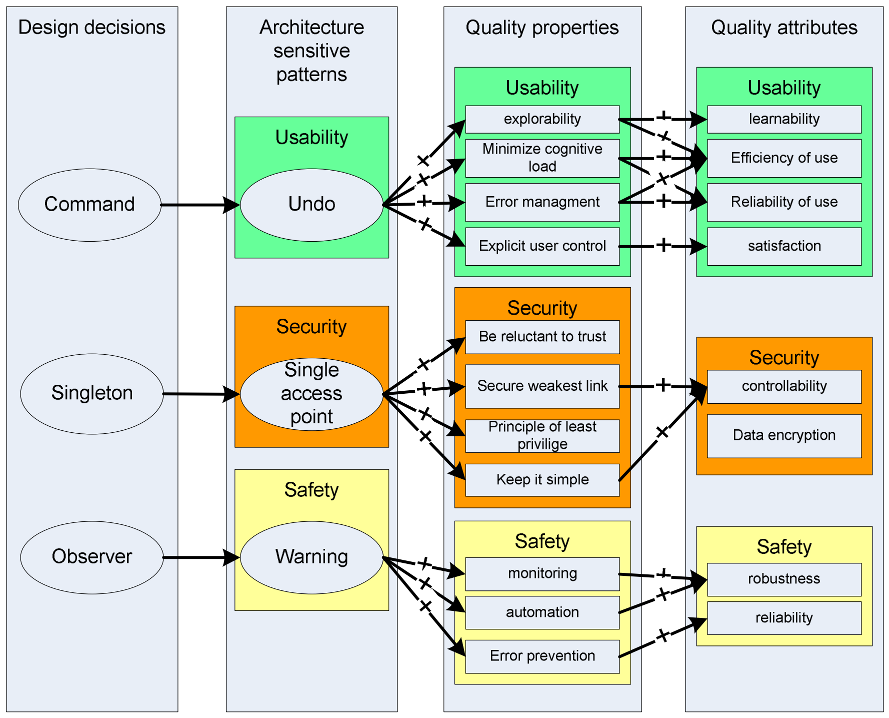

Figure 4: Intra quality relationships between usability, security and safety.

3.2 Patterns in the framework

Figure 4 shows how the patterns, Undo, Warning and Single Point of Access fit in our framework including the relationships between them and various parts of our framework. The qualities and indicators layers has been left out not to make this graph presentation of the framework too cluttered. In this instance we only visualized the relationships of each pattern with one quality. E.g. a particular pattern only improves certain principles and factors of one quality. The security principles, factors and relationships have been derived from [25,29]. the safety principles, factors and relationships have been derived from [16] and the usability principles, factors and relationships have been derived from our previous [8] work on this topic. The relationships can be positive as well as negative for example Undo, improves explorability (part of learnability) which improves learnability but may negatively affect efficiency.

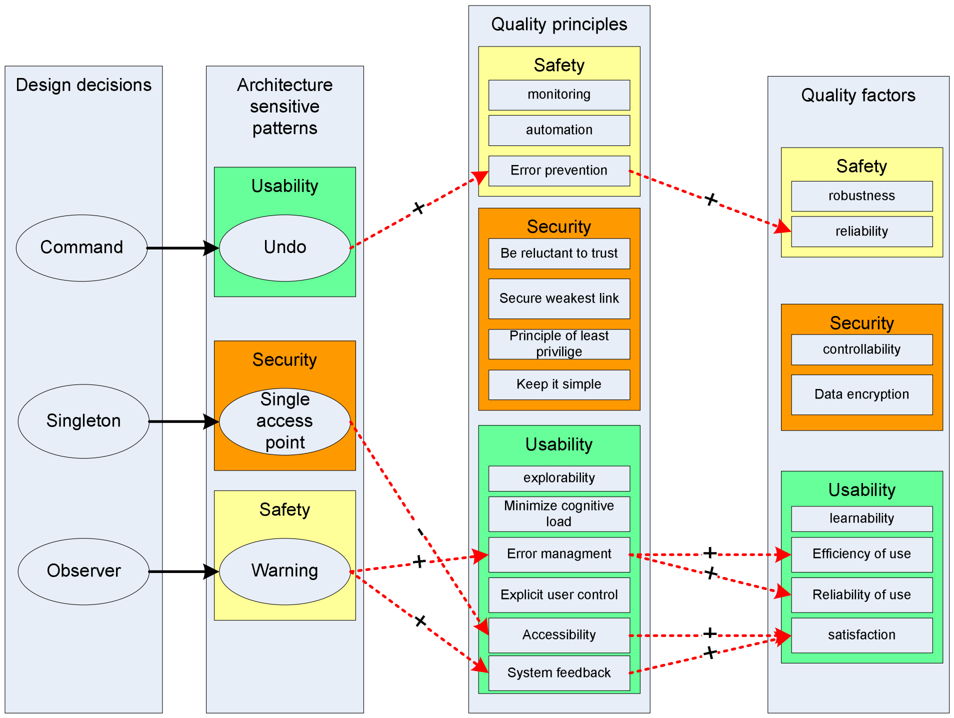

Figure 5: Inter quality relationships between usability, security and safety.

Most pattern descriptions only focus on how a particular pattern improves the quality principles of that particular quality improving pattern. For example in descriptions of undo [14] only those relations with factors of usability are given i.e. they only discuss intra quality tradeoffs (for example between learnability and efficiency factors of usability). However, it's obvious that certain patterns also affect other qualities. For example Single Access Point has a negative influence on the accessibility of a system since the application can only have one entry point. Accessibility is also recognized as a property of usability [30,31]. Single Access Point improves security but to a certain extent impairs usability. In the pattern descriptions these effects on other qualities are often not discussed or provided and it is left up to an architect to analyze such impact on other qualities during an assessment. Our framework formalizes inter and intra quality tradeoffs. Figure 5 shows the relationships that some of the patterns have with other quality factors (see the dashed lines in Figure 5).

3.3 Boundary patterns

An interesting observation that we made while identifying patterns that fit in the usability, safety security framework is that sometimes it is possible to find patterns that provide a way around traditional quality tradeoffs. For example, traditionally usability and security are conflicting; provide a mechanism that improves security such as a password and it will get in the way of usability (users prefer not having to provide a password at all). The warning pattern in our framework is an example of such a pattern, as this pattern primarily improves security but also to some extend improves usability. Warning improves feedback but too much feedback has a negative effect on usability (see 4.2). Another pattern we identified where this effect is more prevalent is Single Sign on (SSO). SSO is an example which provides a solution to a quality problem but also provides a solution that counters the negative effects on another quality that traditionally comes with this pattern. SSO is a pattern that is on the boundary of the traditional usability-security tradeoff. We call such a pattern a boundary pattern.

Single Sign-on

Quality	Usability & Security
Authors	Folmer and Welie [28]
Problem	The user has restricted access to different secure systems for example customer information systems or decision support systems by for example means of a browser.

The user is forced to logon to each different system. The users who are allowed to access each system will get pretty tired and probably make errors constantly logging on and off for each different system they have access to.

Use when You have a collection of (independent) secure systems which allows access to the same groups of users with different passwords.

Solution Provide a mechanism for users to authenticate themselves to the system (or system domain) only once.

Why If users don't have to remember different passwords but only one, end user experience is simplified. The possibility of sign-on operations failing is also reduced and therefore less failed logon attempts can be observed. Security is improved through the reduced need for a user to handle and remember multiple sets of authentication information (However having only one password for all domains is less secure in case the user loses his password). When users have to provide a password only once, routine performance is sped up including reducing the possibility of such sign-on operations failing. Reduction in time taken, and improved response, by system administrators in adding and removing users to the system or modifying their access rights. Improved security through the enhanced ability of system administrators to maintain the integrity of user account configuration including the ability to inhibit or remove an individual user's access to all system resources in a coordinated and consistent manner.

Security Principles

- Be reluctant to trust
- Secure weakest link

Usability Principles

- minimize cognitive load
- error prevention

Architectural Considerations There are several solutions to providing SSO capabilities and some of those have architectural implications. In general the system- or software architecture must fulfill several responsibilities:

Encapsulation of the underlying security infrastructure

Authentication and authorization are distributed and are implemented by all parts of the system or by different systems. Authentication and/or authorization mechanisms on each system should be moved to a single SSO service, component or trusted 3rd party dedicated server. Other systems and/or parts of the system should be released from this responsibility. The SSO server/component/service can act as a wrapper around the existing security infrastructure to provide a single interface that exports security features such as authentication and or authorization. Either some encapsulating layer or some redirection service to the SSO service needs to be established.

Integration of user records

User records (which contains authentication and authorization details) of different systems should be merged in to one centralized system to ease the maintenance of these records. In case of a trusted 3rd party's service some local access control list needs to be created which maps the passports of the 3rd party trusted service to authorization profiles.

Providing SSO has a **high** impact on the software architecture.

Figure 6 shows how SSO fits in our framework and it displays the inter quality relationships with usability and security.

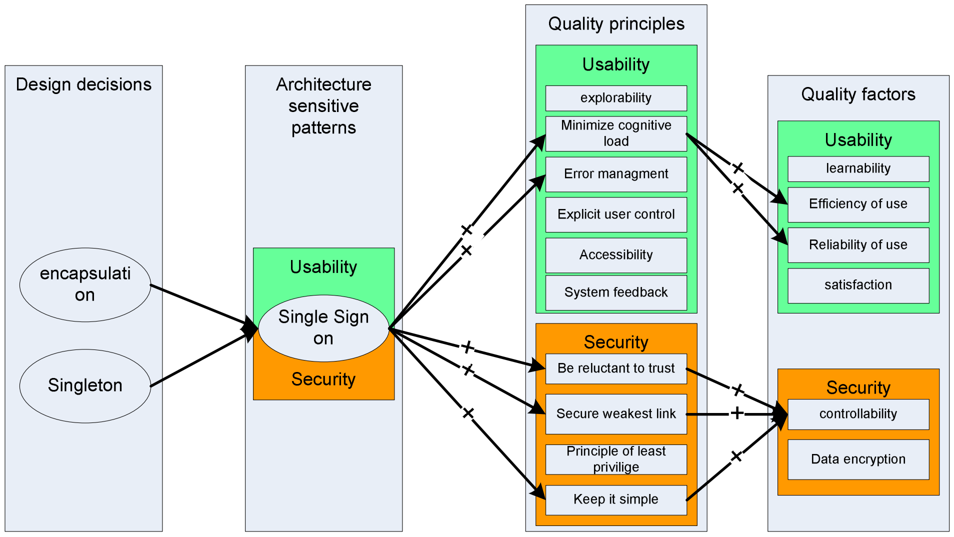

Figure 6: Framework showing SSO as a "boundary" pattern.

3.4 Benefits of using our framework

Our framework assists in two common problems in architecture design namely avoiding retrofit problems and dealing with tradeoffs.

3.4.1 Retrofit problems

Our framework formalizes and raises awareness of the relation that exists between software architecture and parts of software quality that are hard to assess upfront. Existing design knowledge concerning quality improving patterns is filtered and compiled in a format useful for architecture design and evaluation. For example, when a designer has a safety requirement to fulfill, for example "provide robustness", the architect may use the principles in the framework to find suitable patterns (such as robustness →monitoring → warning) to be considered. Architects have often a weak understanding of how to fulfill usage centered qualities such as usability, security or safety, as opposed to development related qualities such as modifiability and scalability. This weak understanding leads to weak architectural support for these qualities and avoidable rework of the system. Studies have shown that a significant part of maintenance costs is spent on fixing architecture related quality problems [4], a significant part of these are usability problems [32]. Increasing awareness of this relationship and guidance provided by our framework can avoid this retrofit problem. One can heuristically evaluate an architecture for its support of our patterns. If it's unknown whether this pattern should be implemented in the final system, one can decide to still design the architecture in such a way that support can be provided for this pattern in the architecture. For example if its unknown whether undo is needed, one can still decide to facilitate the command pattern for different reasons such as improving API design [33]. This, however, comes at the price of additional complexity in the architecture. An architect should find the optimum balance between these constraints (e.g. when additional complexity adds value over simplicity in design [34]). Heuristically evaluating an architecture for its support of these patterns may at first seem a naïve approach, however, we see no other way of raising awareness. An analysis can only be based on information that is available during this stage which is often only architecture diagrams [35,36] and design documentation. The same pattern may be implemented or adopted in many parts of the architecture. In order to be able to analyze the architecture support for quality we must relate them to a set of requirements. This can only be done by using a formalized method which allows one to explicitly define and prioritize the quality requirements of a system. Although alternatives exist, most architectural assessment techniques [37,38,4] employ scenarios. A typical process for scenario based technique is as follows:

1. A set of scenarios is elicitated from the stakeholders and prioritized accordingly.
2. A description the software architecture is made.
3. The architecture's support for, or the impact of these scenarios is analyzed depending on the type of scenario. For qualities such as maintainability metrics [39] can used on an architecture design to identify the support but for other qualities such as usability, security and safety a pattern based analysis is the only feasible approach. During this stage one can identify whether certain quality improving solutions that have been implemented in the software architecture sufficiently support the quality requirement expressed by the scenario.
4. Based on the analysis of the individual scenarios an overall assessment of the architecture is formulated. During this stage architecture sensitive quality improving solutions that

have not been implemented in the architecture can be considered to be implemented based on the results of the analysis.

Our approach is not prescriptive; one must take care with a pattern based design, as not all patterns are applicable to any system; in addition one cannot expect that implementing all patterns results in a high quality system as many factors are involved in the design of a system. Our framework only serves as reminder that if these patterns are considered, that they can only be cost effectively facilitated during architecture design hence raising awareness and discussion about the architecture design. A pattern based framework approach has already been successful for assessing software architecture for their support of usability [40] and creating frameworks for other qualities will yield the same positive results.

3.4.2 Tradeoffs

Our framework also provides exact insight in which tradeoffs must be made for each design solution. Being able to understand precisely how a particular design decision improves or impairs a particular quality on the level of quality factors or quality indicators allows one to make a better informed designed decision and minimizes the risk of designing a system that fails to meet its quality requirements. Identifying exactly how a tradeoff affects a certain indicator also allows one to make a much more detailed analysis of the support of a particular scenario in scenario based assessment. Suppose we consider implementing a warning for a particular system, but suppose the system is only being used by novice users. A usage scenario for this system could be: "novice user inserts order into database". Suppose a usability requirement for this scenario could be that for novice users quality factors such as learnability and reliability of use are very important and other quality factors such as efficiency and reliability are not important. Being able to analyze precisely how warning affects usability and safety allows one to reason that although warning increases quality indicators such as task length and task execution times (if such a popup occurs) that this is not bad as efficiency is not important for this scenario. With this precise analysis one can better reason about tradeoffs and decide that this scenario is supported by the architecture. In our framework an analyst can immediately identify tradeoffs without having to revert to separate quality models. Our framework only focuses on tradeoffs that are relevant to make during architecture design, making sure the architecture provides support for fulfilling these quality requirements.

4. Discussion

4.1 Contents of the framework

Crucial to the success of our framework is an accurate filling of the framework for different qualities. At this moment only a framework for usability [8] exists. Future work will develop such frameworks for the qualities security and safety. Concerning how a particular quality is composed of which factors, how it can be measured by which indicators and which principles improve that quality, we do not make any hard claims. The elements and relationships we identified for usability are based upon existing quality models and literature studies. For security and safety we analyzed the patterns descriptions and some quality models to identify factors and principles for each quality. These factors and principles we identified are far from complete. However we are convinced that any existing quality model such as ISO 9126 [1] can be used.

4.2 Relationships in the framework

Some relations in the framework are hard to express and may need to be quantified. For example warning is a form of feedback but an application which gives too many warnings is not considered usable i.e. too much feedback impairs the usability factor of efficiency. A warning, for example a popup, increases task length and consequently task execution times. So some feedback is good but too much not. Such relations cannot be expressed in our framework. A solution could be to directly relate one of the patterns to an indicator for a quality factor, but because indicators are very system dependent it would make the framework less generally applicable. A similar argumentation holds for the principles. Warning improves the error prevention property of safety but also the error management property of usability. Possibly error management and error prevention should be grouped into one property that affects usability as well as safety, however we need to know more about the exact definition of these principles before we can decide to do that. The relationships in our framework only indicate positive or negative relationships. Effectively an architect is interested in how much a particular pattern or property will improve a particular aspect of quality in order to determine whether requirements have been met. Having quantifiable data for these relationships would greatly enhance the use of our framework. In order to get quantitative data we may need to substantiate these relationships and to provide models and assessment procedures for the precise way that the relationships operate. However, we doubt whether identifying this kind of quantitative information is possible, as some relationships may be very context dependent. In order to overcome this problem we are currently developing a tool that allows architects to put weights on the relationships which would allow them to tailor the framework to their specific systems.

4.3 Scenario based assessment

An interesting observation is that usability, security and safety requirements can all be expressed using a usage scenario. A usage scenario describes a particular interaction that a stakeholder has with the system in a particular context. For example "novice users inserts order using a mobile phone". With architecture assessment of usability [41] we have successfully annotated such a usage scenario to express usability requirements. This annotation has been done by relating the scenario to particular usability factors. As it is often impossible to design a system that has high values for all the factors intra quality tradeoffs must be made. With this annotation we determine a prioritization between usability factors for that scenario. Preliminary research shows that we can use the same technique to express safety and security requirements. For example a security requirement could be: mobile devices should always use encryption. In order to express this requirement one could give a higher priority to the encryption security factor to express that this factor is more important than other security factors. In the analysis step the analyst uses these values to determine the architecture's support for a scenario using the relations in the framework and the patterns that have been implemented.

The benefits of having a usage scenario as a common starting point for specifying scenario is that comparing scenarios becomes easier. How does one compare a change scenario (maintainability) expressed as "Port to another operating system X" with a performance scenario (efficiency) "inserting order in database must take less then 20 seconds"? If different quality scenarios share a usage scenario as a common ground it will be much easier to compare scenarios if tradeoffs need to be made. Consider comparing "novice user inserts order using a mobile phone should be easy to learn" and "novice user inserts order using a mobile phone should be encrypted". Future case studies will focus on defining usage scenarios for these qualities and collecting experiences with architecture assessment.

4.4 Scope of the framework

The patterns in our framework may also improve or impair qualities other than usability, security and safety. But for simplicity we only formalized the relationships with these qualities. A complete framework should express the relationships with all qualities. Concerning harvesting design knowledge relevant to architecture design we should not only consider patterns as they are only a subset of all existing design knowledge. There are certain design decisions that are not expressed by means of a pattern such as the choice to use a particular application framework (although these can be decomposed into patterns), which has an architectural impact.

4.5 Terminology used

Our framework is different from an Object oriented (OO) application framework. We use the term framework to express a relationship between software architecture and software quality. This framework is used to store design knowledge and consist of elements which exist in layers. A OO framework is a reusable, "semi-complete'' application that can be specialized to produce custom applications [42]. Though such OO frameworks may consists of architecture sensitive patterns and patterns in our framework may be supported by OO frameworks. For the elements in our framework we chose the terms factors and indicators and principles but these may also be replaced by terms such as attributes [23], measures and design heuristics/guidelines.

4.6 Selection rather than design

The role of architecture design has changed significantly the past few years. The design knowledge that has been captured in the form of patterns, styles, components or OO 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 components and object oriented frameworks. 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, OO frameworks and applications that can be used for application development. Software architecture design has shifted from starting with a functional design and transforming it with appropriate design decisions, to selecting the right components & framework and composing them to work together while still ensuring a particular level of quality. Within this selection process our framework is still useful. Determining the support for quality can be based upon the amount of evidence for patterns that is extracted from the architecture of such a framework or component. For example, one may choose to use the JAVA swing GUI library as it has support for undo.

4.7 Implementation clusters

Multiple patterns can be facilitated using the same implementation framework. In [28] we identified that usability patterns such as Auto save [43], Cancel [44,6] and Macros [45] can be implemented using the same implementation framework needed for Undo (the Command pattern [9]). There is evidence that this also holds for different patterns for different qualities see Figure5 and 6 where single sign on and Single Access Point both are based upon the singleton design pattern [9]. Future research should focus on investigating the implementation similarities.

5. Related work

There is a large amount of software pattern documentation available today, with the most well known and influential possibly being the book Design Patterns: Elements of Reusable Object-Oriented Software [9]. This book provides a collection of patterns that can be used for software design. However in our opinion this book is focusing more on problems related with functional design.

An architecturally sensitive usability pattern as defined in our work is not the same as a design pattern [9]. Unlike the design patterns, architecturally sensitive patterns do not specify a specific design solution in terms of objects and classes although for usability patterns we have done this [28]. Instead, we outline potential architectural implications that developers face looking to solve the problem the pattern represents.

Concerning patterns for qualities a lot of work has been done in the area of usability patterns (also known as interaction patterns) [13,46,14]. Considerable work has been done on other qualities such as security patterns [19,20], reliability patterns [15], safety patterns [16], maintainability patterns [18], security patterns [19,20] and performance patterns [15] etc.

The layered view on usability presented in [47] inspired several elements of the framework model we presented in Section 2. For example their usability layer inspired our factor layer. Their usage indicators layer inspired our indicators layer and their means layer forms the basis for our principles and patterns layer. One difference with their layered view is that we have made a clear distinction between patterns (solutions) and principles (requirements) and focus on other qualities too to make it more generic.

In [48] a NFR (Non Functional Requirements) Framework is presented which is similar to our approach. Structured graphical facilities are offered for stating NFRs and managing them by refining and inter-relating NFRs, justifying decisions, and determining their impact. Our approach is similar to theirs but differs in that we take as a basis for our framework existing patterns and we focus on different qualities.

Related to our work to establish a clear relation between software quality and software architecture design is the architectural tactics approach by [49,50]. They define a tactic to be a means to satisfy a quality-attribute-response measure by manipulating some aspect of a quality attribute model through architectural design decisions. Our approach shares some similarities with theirs apart from that they focus on performance and modifiability and we focus on usability, security and safety. A quality attribute response (such as mean time between failure) is similar to our quality indicators. Our framework for usability, security and safety is somewhat similar to their analytical models for quality factors although ours is based upon existing quality models [25]. Though in our framework we explicitly connect architectural patterns to different qualities to be able to reason about inter- and intra- quality tradeoffs. Our quality principles are similar to their tactics.

The main difference between both approaches is that we did not try to be innovative when defining our framework, we merely organized existing patterns and related them to existing quality models and design principles to show that in such a format they can be useful for informing architectural design.

6. Conclusions

The framework presented in this paper is a first step in formalizing and raising awareness of the complex relation between software quality and software architecture. The key element of the framework is the notion of an architecture sensitive pattern. An architecture sensitive pattern is a design solution that in most cases is considered to have some effect (positive or negative) on one or more quality aspects but which is difficult to retrofit into application, because this pattern has an architectural impact. Collecting and communicating such patterns is important as it may prevent the retrofit problem. In order to be able to effectively use a collection of such patterns for software assessment and design we need to organize them into a framework which connects them to software quality. Our framework consists of:

- Design decisions; motivate the choice for a particular pattern.
- Architecture sensitive patterns; quality improving design solutions.
- Quality indicators; how to measure that particular quality for a specific system.
- Quality factors; how to measure that particular quality.
- Qualities; different qualities that define software quality.

Relations are defined between the elements of our framework so they can be used to:

- Avoiding irreversibility: The framework explicates for a set of qualities which design solutions should be considered during architecture design. When heuristically evaluating an architecture for a set of quality requirements an architect can decide whether an architecture provides sufficient support for these quality requirements based on the evidence for patterns and principles support that can be extracted from the architecture of the system under analysis. Facilitating support for these patterns avoids the retrofit problem at the cost of complexity but still allow some quality tuning during detailed design, which may save some of the high costs incurred by adaptive maintenance activities once the system has been implemented.

- Reasoning about tradeoffs: Our framework also provides exact insight in which tradeoffs must be made for each design solution. Being able to understand precisely how a particular design decision improves or impairs a particular quality on the level of quality factors or quality indicators allows one to make a better informed designed decision and minimizes the risk of designing a system that fails to meet its quality requirements.

At this moment only a validated framework for usability [8] has been developed but in this paper we outline such a framework for security and safety. Future work will focus on completing frameworks for these and other qualities and collecting experiences with this framework when using it in software architecture design and analysis.

References

[1] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, New Jersey, 1996.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Son Ltd, New York, 1996.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison Wesley Longman, Reading MA, 1998.

[4] J. Bosch, *Design and use of Software Architectures: Adopting and evolving a product line approach*, Pearson Education (Addison-Wesley and ACM Press), Harlow, 2000.

[5] M. Fowler, *Who Needs an Architect*, http://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf

[6] L. Bass, J. Kates, and B. E. John, Achieving Usability through software architecture, Technical Report CMU/SEI-2001-TR-005,1-3-2001.

[7] P. O. Bengtsson, *Architecture-Level Modifiability Analysis*, Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Sweden, 2002.

[8] E. Folmer, J. v. Gurp, and J. Bosch, *A framework for capturing the relationship between usability and software architecture*, Software Process: Improvement and Practice, Wiley, 2003, pp. 67-87.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns elements of reusable object-orientated software.*, Addison-Wesley, Reading, Massachusetts, 1995.

[10] J. O. Coplien and D. C. Schmidt, *Pattern Languages of Program Design*, Addison-Wesley (Software Patterns Series), 1995.

[11] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, Reading, Reading MA, 1996.

[12] Pointer, *PoInter: Patterns of INTERaction collection*, http://www.comp.lancs.ac.uk/computing/research/cseg/projects/pointer/patterns.html

[13] J. Tidwell, Interaction Design Patterns, *Conference on Pattern Languages of Programming 1998*, 1998.

[14] M. Welie and H. Trætteberg, Interaction Patterns in User Interfaces, *7th Conference on Pattern Languages of Programming (PloP)*, 2000.

[15] Microsoft, *Performance and Reliability Patterns*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/EspPerformanceReliabilityPatternsCluster.asp

[16] M. Mahemof and A. Hussay, Patterns for Designing Safety-Critical Interactive Systems, 1999.

[17] F. Bitsch, Safety Patterns - The Key to Formal Specification of Safety Requirements Source, *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*, 2001.

[18] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Robers, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, 1999.

[19] J. Yoder and J. Barcalow, Architectural patterns for enabling application security, *In Proceedings of the Pattern Languages of Programming (PLoP) Workshop*, 1-9-1997.

[20] D. M. Kienzle and M. C. Elder, *Security Patterns for Web Application Development*, http://www.modsecurity.org/archive/securitypatterns/dmdj_final_report.pdf

[21] *Patterns Library*, http://www.hillside.net/

[22] Å. Granlund, D. Lafrenière, and D. A. Carr, Pattern-Supported Approach to the User Interface Design Process, *9th International Conference on Human-Computer Interaction (HCI 2001)*, 2001.

[23] J. A. McCall, P. K. Richards, and G. F. Walters, Factors in software quality Vol. 1,2,3. AD/A-049-014/015/055. Nat.Tech. Inf. Service. Springfield, 1977.

[24] B. Boehm, J. R. K. H. Brown, M. Lipow, G. J. Macleod, and M. J. Merrit, *Characteristics of Software Quality*, North Holland, Netherlands, 1981.

[25] ISO, ISO 9126-1 Software engineering - Product quality - Part 1: Quality Model, 2000.

[26] ISO, ISO 9241-11 Ergonomic requirements for office work with visual display terminals (VDTs) -- Part 11: Guidance on usability., 1994.
[27] D. K. van Duyne, J. A. Landay, and J. I. Hong, *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience*, Addison-Wesley, Boston, 2002.
[28] E. Folmer, M. Welie, and J. Bosch, *Bridging Patterns - an approach to bridge gaps between SE and HCI*, Journal of information & software technology, Kluwer, 2006, pp. 69-89.
[29] J. Viega and G. McCraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Aw Professional, New York, 2001.
[30] J. Nielsen, *Usability Engineering*, Academic Press, Inc, Boston, MA., 1993.
[31] R. Holcomb and A. L. Tharp, What users say about software usability., *International Journal of Human-Computer Interaction, vol. 3 no. 1*, 1991.
[32] R. S. Pressman and Pres, *Software Engineering: A Practitioner's Approach, 5th edition*, McGraw-Hill Higher Education, New York, 2001.
[33] wiki., *The Command Pattern*, http://en.wikipedia.org/wiki/Command_pattern
[34] F. P. jr. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Twentieth Anniversary Edition*, Addison-Wesly, Reading, MA, 1995.
[35] P. B. Kruchten, The 4+1 View Model of Architecture, IEEE Software, 1995.
[36] C. Hofmeister, R. L. Nord, and D. Soni, *Applied Software Architecture*, Addison Wesley Longman, Reading, MA, 1999.
[37] R. Kazman, G. Abowd, and M. Webb, SAAM: A Method for Analyzing the Properties of Software Architectures, *16th International Conference on Software Engineering*, 1994.
[38] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, The Architecture Tradeoff Analysis Method, *International Conference on Engineering of Complex Computer Systems*, 8-1-1998.
[39] P. O. Bengtsson and J. Bosch, Architecture Level Prediction of Software Maintenance, *EuroMicro Conference on Software Engineering*, 1999.
[40] E. Folmer, J. v. Gurp, and J. Bosch, Software Architecture Analysis of Usability , *The 9th IFIP Working Conference on Engineering for Human-Computer Interaction*, 2004.
[41] E. Folmer, J. v. Gurp, and J. Bosch, Scenario-based Assessment of Software Architecture Usability, *ICSE 2003 workshop on "Bridging the Gaps Between Software Engineering and Human-Computer Interaction"*, 2003.
[42] R. Johnson and B. Foote, *Desiging Reusable Classes*, Journal of Object-Oriented Programming, SIGS, 1988, pp. 22-35.
[43] S. A. Laakso, *User Interface Design Patterns*, http://www.cs.helsinki.fi/u/salaakso/patterns/
[44] W. van der Aalst and A. ter Hofstede, *Workflow patterns*, http://tmitwww.tm.tue.nl/research/patterns/
[45] J. Tidwell, *Common ground: Pattern Language for Human-Computer Interface Design*, http://www.mit.edu/~jtidwell/interaction_patterns.html
[46] K. Perzel and D. Kane, Usability Patterns for Applications on the World Wide Web, *Pattern Languages of Programming Conference*, 1999.
[47] M. Welie, G. C. van der Veer, and A. Eliëns, Breaking down Usability, *Proceedings of Interact 99*, 1999.
[48] L. Chung, B. A. Nixon, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishing., Boston Hardbound, 1999.
[49] F. Bachman, L. Bass, and M. Klein, Illuminating the Fundamental Contributors to Software Architecture Quality, 2002.
[50] F. Bachman, L. Bass, and M. Klein, Deriving Architectural Tactics: A Step Toward Methodical Architectural Design, 2003.