Software architecture analysis of usability

Folmer, Eelke

Chapter 6

Reusable SA Tradeoff Solutions

Published as: A Framework for Reusable Architectural Tradeoff Solutions. Eelke Folmer, Jan Bosch, Submitted to the International Journal of Software Engineering & Knowledge Engineering, August, 2005

Abstract: It often proves to be hard to make the necessary changes to a system to improve its quality. A reason for this is that certain changes require changes to the system that cannot be easily accommodated by the software architecture. Architects are often not aware that certain quality improving solutions cannot be added during late stage development (a retrofit problem). In addition architecture design, which includes activities such as deciding on tradeoffs between qualities, is a complex non formalized process, much relying on the experience of senior software engineers. This paper outlines and presents a framework that formalizes some of the essential relations between software architecture and software quality. It consists of an integrated set of design solutions that may improve a particular quality but that require much effort to be implemented during late stage design. Furthermore this framework provides insights into how particular design solutions may improve or impair other qualities. This framework can assist software architects in the design and decision making process; existing design knowledge and experience can be consolidated in a form that allows us to guide and inform architectural design. The use of this framework is illustrated using four architecture sensitive quality improving patterns, namely Undo (usability), Warning (safety), Single point of access (security) and Single Sign on (usability/security).

6.1 Introduction

In the last decades it has become clear that the most challenging task for a software architect is not only to design for the required functionality, but also to focus on designing for specific attributes such as performance, security or maintainability, which contribute to the quality of software (Bosch and Bengtsson, 2002). Quality is not something you can easily add to a system during late stage; it has to be built into the system from the beginning (Svahnberg et al, 2000).

During architecture design, those design decisions are made that you wish to get right early on in a project (Fowler, 2003) since these are very costly to revoke, however, software engineers in general have few techniques available for predicting the quality attributes of a software system before the system itself is available (Bengtsson, 2002). Therefore systems are often developed and the provided quality is measured only when the system is completed. The problem with this type of development is that sometimes a retrofit problem (Folmer et al, 2003) occurs. I.e. In some cases, if you want to improve the quality of a system, certain design solutions need to be applied which cannot be supported by the software architecture. I.e. such solutions are often implemented as new architectural entities (such as components, layers, objects etc) and relations between these entities or an extension of old architectural entities. If a certain software architecture i.e. a fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the

principles guiding its design and evolution (IEEE, 1998) has already been implemented then changing or adding new components to this structure 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 very expensive.

Software development is restricted around the natural tradeoffs of resources, time and quality. 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, than to deliver a high quality product. However in recent years this strategy has lead to failure; because of the intense competition in the software industry and the increasing number of users, if a software product has low quality, users will move to a competitive product with a higher quality.

When a particular software architecture is chosen the qualities are determined and restricted by that same architecture, therefore one of the challenges in architecture design and analysis is to an extent reduce the frequency of irreversibility problems (Fowler, 2003) in software designs. Using this strategy certain design solutions can still be applied during late stage to improve the quality (to a certain extent).

Determining the level of quality that a system needs to provide is essential to the design of any architecture but for some qualities this is very hard to determine. During product evolution often new levels of quality need to be supported which are hard to predict upfront. Adding flexibility (e.g. making things easier to change) is a way to deal with this but it adds additional complexity to a system. Adding complexity makes a system harder to change; therefore a good architect should know when more complexity adds value over simplicity in design. Designing means making decisions, and making the right decisions. Architecture design should include:

- Making tradeoffs between qualities: most design decision have a considerable influence on the qualities of a system such as reliability, performance and modifiability. These qualities are not independent, a design decision that has a positive effect on one quality might be very negative for other qualities (Bengtsson, 2002). Some qualities frequently conflict: for example design decisions that improve modifiability often negatively affect performance. Tradeoffs between qualities are inevitable and it is important to make these explicit during architecture design because design decisions are generally very hard to change at a later stage.

- Deciding where to put flexibility. For some architecture designs it is still possible to add specific solutions which allow for fine tuning of some of the qualities, however this adds additional complexity to the architecture.

- Maintain a conceptual integrity (Brooks, 1995) i.e. knowing when to stop adding complexity to maintain simplicity in design.

Architecture design is a complex non formalized process. To make tradeoffs requires assessment of how a particular design decision may improve or impair other qualities which sometimes is unknown. Design decisions are often taken on intuition, relying on the experience of senior software developers (Svahnberg et al, 2000). Most of the knowledge needed for the decision making process is tacit knowledge stored in the heads of experienced software architects. In addition software architects are often

unaware which quality improving mechanisms cannot be added easily during late stage.

The effect of a design decision should be assessed before it becomes too expensive to retract. Although alternatives exist, most architectural assessment techniques (Kazman et al, 1994, Kazman et al, 1998, Bosch, 2000) employ scenarios. A typical process for scenario based technique is as follows: A set of scenarios is elicitated from the stakeholders. Then a description the software architecture is made. Then the architecture's support for, or the impact of these 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. Though scenario based assessment is a more structured and formalized way to reason about design decisions and tradeoffs it still has some problems. It still requires an experienced engineer to assess whether an architecture supports a particular scenario or not (although for some qualities metrics are provided(Bengtsson, 2002)). Another problem is that and individual assessment may be rather subjective. I.e. One architect may decide that a scenario is supported and another may decide to reject the scenario.

The goal of this paper is to outline and present a framework that formalizes some of the essential relations between software architecture and software quality in order to assist software architects in the design and decision making process. Existing design knowledge and experience is consolidated in a coherent form that allows us to guide and inform architectural design. The remainder of this paper is organized as follows. In the next section, we outline 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.

6.2 The Software Architecture - Quality Framework

Our framework is composed of a layered model consisting of 6 different layers. Each layer is discussed in detail below:

Architecture sensitive patterns

The core layer of our framework is composed of a special category of design solutions that inhibit the retrofit (Folmer et al, 2003) problem which are called architecture sensitive 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 (Gamma et al 1995, Buschmann et al, 1996) 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 (Gamma et al 1995). Since then a pattern community has emerged that specifies patterns for all sorts of problem domains: architectural styles (Buschmann et al, 1996), object oriented frameworks (Coplien and Schmidt, 1995), domain models of businesses (Fowler, 1996), but also software quality related patterns have been identified such as usability patterns (Tidwell 1998, Welie and Trætteberg, 2000, PoInter, 2003), safety patterns (Mahemof and Hussay, 1999), maintainability patterns (Fowler et al, 1999), security patterns (Yoder and Barcalow, 1997, Kienzle and Elder, 2002) etc. A plethora of patterns can be

found in literature, current practice and on the web (Hillside group) but in our framework only those patterns are considered that are "architecture sensitive".

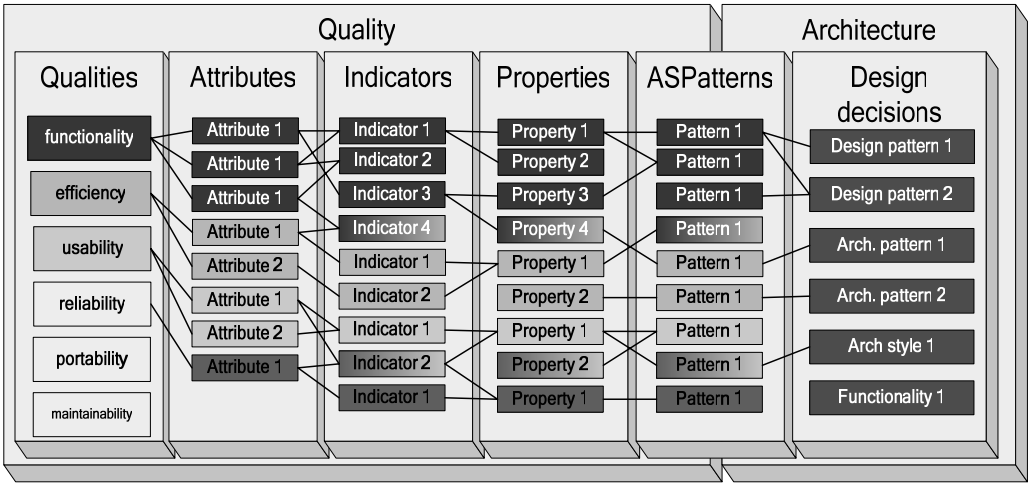

Figure 54: Relation between Quality and Software Architecture

To define an architecture sensitive pattern we use the following criteria:

- Improves quality: The pattern specifically addresses one or more quality related problems and its solution improves one or more qualities or as a side effect impairs one or more qualities.

- Impact on the architecture: The impact can work on different levels in the architecture. In (Bosch, 2000) we identify four types of architectural transformations e.g. architectural style, architectural pattern, design pattern and transformation of quality requirements to functionality. 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 (Buschmann et al, 1996) is a transformation with a major architecture wide impact (Bosch, 2000). An architectural pattern does not generally reorganize the fundamental components but rather extends and changes their behavior as well adds one or a few components that contains functionality needed by the pattern. Applying a design pattern (Gamma et al 1995) generally affects only a limited number of components in the architecture; however the impact can still be very high depending on the size of the components. The last transformation deals with extending the system with additional functionality not concerned with the applications domain but primarily with improving the quality attributes. This type of transformation may require minor reorganizations of the existing architecture but may still be expensive to do during late stage design.

- Effort needed to implement the pattern: The architectural impact of a pattern can be any of the four 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.

We defined the term architecture sensitive pattern with the purpose of:

- Architecture design; e.g. use the pattern collection as an informative source that may guide architecture design.

- Architecture evaluation; when a specific pattern has been implemented in the architecture we can analyze how it affects certain qualities and which tradeoffs are made.

In order to fulfill in that purpose we need to explicate the relationship of these patterns to specific parts of software quality, we therefore defined the following layers (see also figure 1).

Qualities

The qualities layer forms the topmost layer concerning software quality and is based on 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 (McCall et al, 1977) proposes a quality model consisting of 11 factors: correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability and interoperability. Boehm (Boehm et al, 1981) proposes a quality model that divides quality into the following quality factors: portability, reliability, efficiency, human engineering, testability, understandability and modifiability. The ISO 9126 [1] model describes software quality as a function of six characteristics: functionality, reliability, efficiency, usability, portability, and maintainability. In general, different authors construct different lists and compositions of what those qualities are, but they all pretty much agree on the same basic notions, therefore in our framework either one of these quality models can be taken as a starting point. In figure 1 we use the ISO 9126 quality model.

Quality attributes

All quality models are based on the idea that qualities are decomposed in a number of high level quality attributes (or factors). Qualities attributes are supposed to be much easier to measure than the qualities themselves. For example according to ISO 9126 usability is to be composed of learnability, efficiency of use, reliability of use etc. How a particular quality should be decomposed in which different attributes is still a topic of discussion in some quality research areas.

Quality indicators

How to measure a quality attribute for a given system depends very much on the type of system and or what can be measured. Quality indicators are directly observable properties 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 attributes. For example the number of application crashes may be an indicator for the reliability of use attribute of safety but also for the stability attribute of reliability.

Quality properties

Because telling you how to measure a particular quality attribute 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. We call these quality properties. Guidelines such as ISO/IEC 9126 (ISO 9126-1) give standard definitions of quality attributes and proposed measures, together with a set of application guidelines. This quality properties form the fourth layer in our framework.

Design decisions

In addition to the software quality related layers we defined an additional "design decision layer", which belongs to the software architecture solution domain. Architecture sensitive patterns may be implemented by an architecture style (Buschmann et al, 1996), architectural pattern (Buschmann et al, 1996) and or design patterns (Gamma et al 1995) 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. This last layer was introduced to show that certain patterns share the same implementation. In our work on a framework for usability (Folmer et al, 2005) we identified that with a single mechanism sometimes multiple patterns can be implemented. For example the UI patterns multi level undo and cancel can all be implemented using the Command pattern (Folmer et al, 2005).

Relationships between the elements of the framework

The quality attributes, indicators and properties layers allow us to connect architecture sensitive patterns to software quality. Relationships between the elements in the framework are an essential part of our framework. Using the relationships, a designer can analyze when he or she has a quality requirement which architecture sensitive patterns may need to be implemented to fulfill this requirement. On the other hand when heuristically evaluating an architecture for a set of quality requirements an architect can decide whether an architecture has sufficient support based on whether certain quality improving architecture sensitive patterns have been implemented.

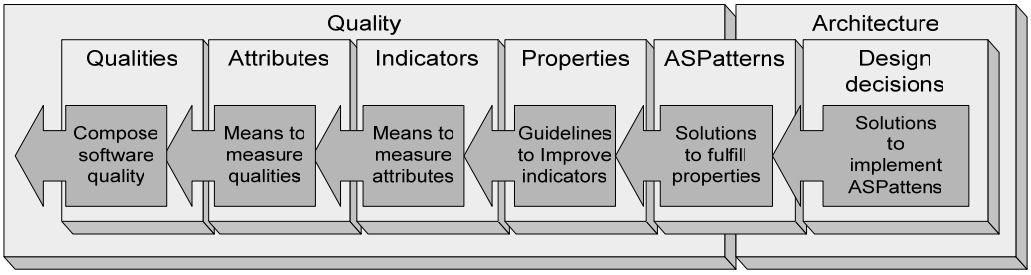

Figure 55: Relationships between the Elements in the Framework

The relationships have been defined as follows:

- Software quality consists of several qualities.
- These qualities can be decomposed in attributes which are much easier to measure than the qualities.

- For a system observable properties can be measured which are an indicator for a particular attribute.
- Guidelines can be used to improve specific indicators & attributes.
- Architecture sensitive patterns fulfill address to one or more properties.
- Certain design decisions such as the use of an architectural style or framework implement an architecture sensitive pattern.

We illustrate these relationships for one specific architecture sensitive pattern for usability namely a wizard (see Figure 56). A wizard is a solution for dividing a complex task into a set of smaller subtasks. When users are forced to follow the order of tasks, users are less likely to miss important things and will hence make fewer errors. A wizard can be implemented by a small wizard framework (Folmer et al, 2005), a wizard on its turn addresses the usability property of guidance as it guides the user through the complex task. Guidance may decrease the time it takes to learn a task. The time to learn a task is an indicator of the usability attribute learnability. Learnability is an attribute of usability. Using these relationships a relationship between software quality and software architecture design is established.

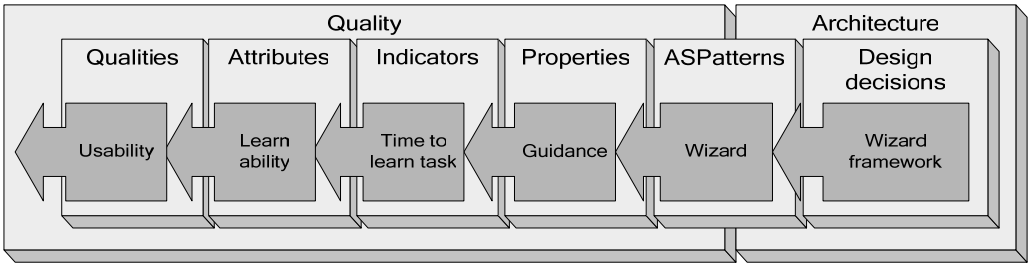

Figure 56: Relationships for Wizard/ Usability

In our framework these relationships are not exclusively defined between the attributes, indicators and properties of a particular quality attribute, as is the case with most pattern descriptions. Some architecture patterns have an effect on multiple qualities, hence directly showing which possible tradeoffs need to be made. We will illustrate this effect in the next section.

6.3 The Security-Safety-Usability Framework

In this chapter we will illustrate the use of the framework by presenting how different quality improving patterns fit in this framework. The qualities we use to illustrate this framework are: security, safety and usability. These three qualities are recognized in the ISO 9126 Standard (ISO 9126-1), though security and safety are not recognized as high level qualities such as usability, but rather as parts of functionality and quality in use. We consider usability, safety and security to be of the same level and we use the following definitions for them:

- **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 (ISO 9241-11).

These qualities were chosen because often tradeoffs need to be made as security and safety frequently conflict with usability. In order to determine whether an architecture provides sufficient support or to be able to design an architecture we first need to specify a necessary level for these quality requirements. For architecture design & evaluation often scenarios are used. 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". We can annotate such a usage scenario to express usability, safety and security requirements. In order to do that we can relate such as scenario to specific attributes of that quality. For example usability is often decomposed into learnability, efficiency of use, reliability of use etc attributes (Figure 57 shows the attributes we identified). As it is often impossible to design a system that has high values for all the attributes often intra quality tradeoffs must be made. A purpose of quality requirements is therefore to somehow specify a necessary level for each quality. 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 attribute to express that this attribute is more important than other security attributes. The benefits of having a usage scenario as a common starting point for defining scenarios for different qualities is that often when one has completely different scenarios for different qualities and one has to for example decide on tradeoffs between scenarios one is often comparing apples with pears. When we defined scenarios that are all based on a usage scenario the scenarios are more alike and one can easier compare one scenario with another.

In literature several security patterns (Yoder and Barcalow, 1997, Kienzle and Elder, 2002), safety patterns (Mahemof and Hussay, 1999) and usability (interaction) patterns (Tidwell 1998, Welie and Trætteberg, 2000, Duyne et al, 2002) have been described. In our own work (Folmer et al, 2003, Folmer et al, 2004) and in others (Bass et al, 2001) already several "architecture sensitive" usability patterns had been identified. Using the same process (e.g. analyzing specific implementations of these patterns in different case studies) architecture sensitive patterns for other qualities have been collected. In our work we only focus on the architecture sensitive patterns i.e. non architecture sensitive patterns can be easily added afterwards and do not require architectural modifications; therefore there is no need to consider these during architecture design. As identified by (Granlund et al, 2001) patterns are an effective way of capturing and transferring knowledge due to their consistent format and readability.

To describe our patterns the following format is used:

Quality: for which quality problem this pattern provides a solution.
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 should describe which properties should have been improved or which other properties have to suffer.
Quality properties: Quality properties are guidelines and design principles in the domain of that quality which the pattern addresses. We specifically list the properties the pattern authors themselves mention in the pattern description.
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 pattern description by the authors.
The three patterns we use to illustrate our framework with are listed below. For more details concerning the patterns implementation and forces etc one should consult the specific pattern authors' reference.

Table 36: Single Access Point

Quality	Security
Authors	Yoder and Barcalow (Yoder and Barcalow, 1997).
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 Properties	• Secure weakest link • Principle of least privacy • Keep it simple • Be reluctant to trust. (Viega and McCraw, 2001)
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 (Yoder and Barcalow, 1997). A singleton (Gamma et al 1995) 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.

Table 37: Warning	
Quality	Safety
Authors	Mahemof and Hussay (Mahemof and Hussay, 1999)
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 Properties	• Automation • Monitoring • Error Prevention • Error reduction (Mahemof and Hussay, 1999)
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 (Gamma et al 1995) or Publisher Subscriber (Buschmann et al, 1996) 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.

Table 38: Multi-Level Undo	
Quality	Usability
Authors	Welie and Folmer (Folmer et al, 2004)
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 Properties	Explicit user control, explorability, error management, minimize cognitive load (Folmer et al, 2003)
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 (Gamma et al 1995) 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.

Patterns in the framework

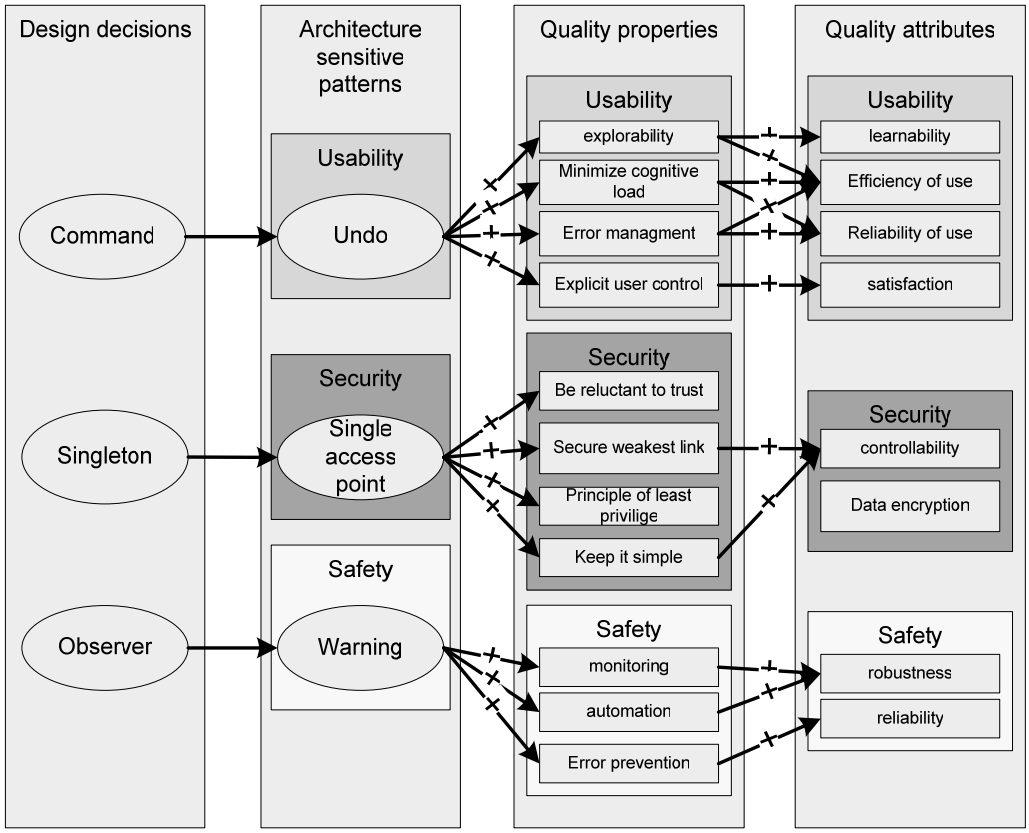

Figure 57: Intra Quality Relations between Usability, Security and Safety

Figure 57 shows how the patterns, Undo, Warning and Single Point of Access fit in 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 properties and attributes of one quality. The security properties, attributes and

relationships have been derived from (ISO 9126-1, Viega and McCraw, 2001). the safety properties, attributes and relationships have been derived from (Mahemof and Hussay, 1999) and the usability properties, attributes and relationships have been derived from our previous (Folmer et al, 2003) work on this topic. The relationships can be positive as well as negative for example Undo improves explorability (e.g. testing out the application) but explorability may negatively affect efficiency and may positively affect learnability.

Most pattern descriptions only focus on how a particular pattern improves the quality properties of that particular quality improving pattern. For example in descriptions of undo only those relations with attributes of usability are given i.e. they only discuss intra quality tradeoffs (for example between learnability and efficiency attributes 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 (Holcomb and Tharp, 1991, Nielsen, 1993). 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 identify the impact on other qualities. Our framework also tries to formalize these intra quality attribute tradeoffs.
Figure 58 shows the relationships that some of the patterns have with other quality attributes. These relationships and the properties that are involved are depicted with red lines. For example warning not only improves security also improves error management (which is also recognized as a usability property).

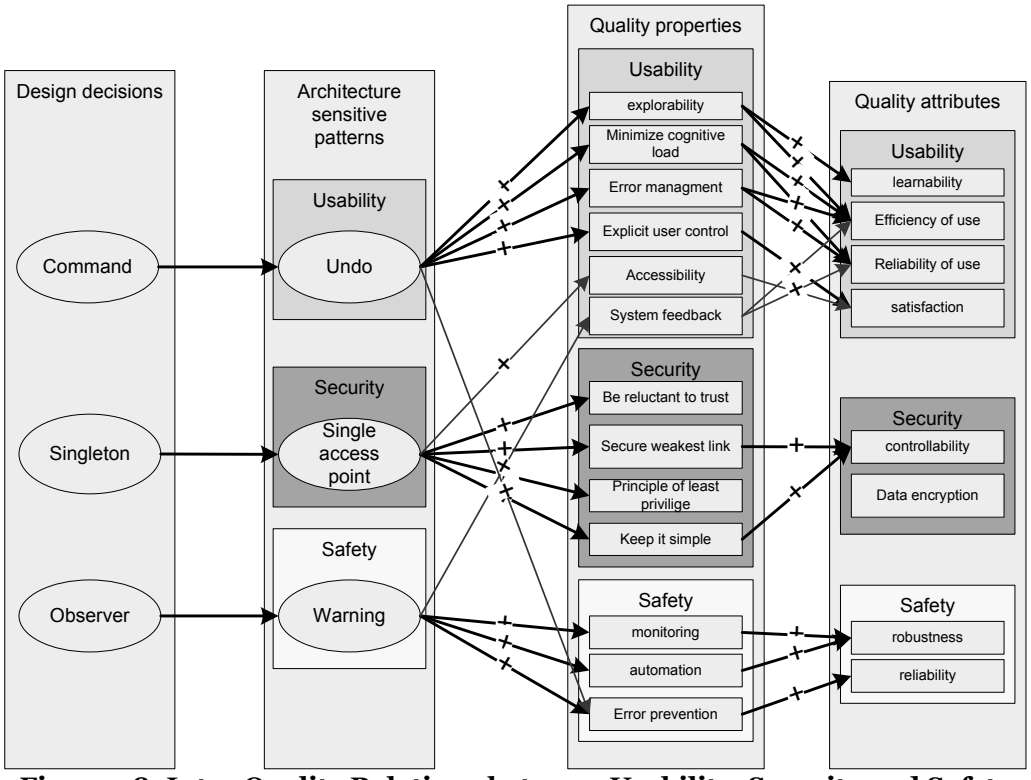

Figure 58: Inter Quality Relations between Usability, Security and Safety

Guide design

The first purpose of the framework is to consolidate existing design knowledge in a form that allows useful guidance of architectural design. For example, when a designer has a safety requirement to fulfill, for example "provide robustness", he or she may use the properties in the framework to find a suitable pattern (such as warning) to implement during architecture design. Of course one cannot design a high quality system by just implementing all the patterns in our framework, there are many other factors involved. The framework emphasizes that if provision of a pattern is considered it should be implemented during architecture design, as it requires architecture support. This may reduce development costs as it is very expensive to modify the software architecture during late stage development. Discussing these concerns with quality engineers such as an interaction designer or a security engineer will raise awareness of the restrictions that exist between architecture design and fulfilling quality requirements.

Reason about tradeoffs

The second purpose is related to the first purpose in the sense that the framework provides guidance in reasoning about tradeoffs. The software architecture is the first product of the initial design activities that allows analysis and discussion about different concerns. Being able to understand how a particular design decision may improve or impair particular qualities during architecture design is very important as it this stage it is still cheap to change design decisions. The framework serves in this purpose as it explicitly connects quality improving design solutions to different qualities. This allows an architect to explicitly identify the tradeoffs that must be made for the implementation of each pattern.

Architecture assessment

The third purpose of the framework is to assist in the decision process of architecture assessment. During architecture design an analysis can only be based on information that is available during that stage for example architecture designs and documentation used with in the development team. At such a stage there is only the vague concept of a software architecture either existing in the head of a software architect or formalized by means of architecture diagrams (Kruchten, 1995, Hofmeister et al, 1999) and design documentation.

As already mentioned in the introduction most architectural assessment techniques such as ATAM (Kazman et al, 1998) or QASAR (Bosch, 2000) employ scenarios. For some qualities such as security and usability which are often only measurable for a completed system it is very hard to determine whether an architecture provides sufficient support for these scenarios. As in the best case there are only architecture diagrams, there is only one way of evaluating, namely a pattern/design solution based heuristic evaluation of the architecture. Our framework can assist in that process.

When heuristically evaluating an architecture for a set of quality requirements an architect can decide whether an architecture has sufficient support, based on whether certain quality improving architecture sensitive patterns from our framework have been implemented. The framework also provides insights in intra- and inter- quality tradeoffs. The result of such an assessment does not result in a system with a necessary higher quality but it ensures that at least the architecture provides support for these solutions so during late stage such solutions may still be easily implemented thus

saving on development costs. For example when evaluating an architecture for its support of usability, an architect may come to the conclusion to implement a command framework (Folmer et al, 2004) even if it is unknown if undo is needed. A pattern based framework approach has already been successful for assessing software architecture for their support of usability (Folmer et al, 2004). In our opinion creating frameworks for other qualities will lead to the similar results.

Boundary patterns

Another 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 users will find such a mechanism not usable (users prefer not having to provide a password at all). However, a solution such as Single Sign on (SSO) provides a solution to improve security without negatively affecting usability. SSO is an example of a solution that provides a solution to a quality problem but also provides a solution that counters the negative effects on another quality that traditionally come with this pattern. SSO is a pattern that is on the boundary of the traditional usability-security tradeoff. Figure 59 shows how SSO fits in our framework and it displays the inter quality relationships with usability and security.

Table 39: Single Sign-on	
Quality	Usability & Security
Authors	Folmer and Welie (Folmer et al, 2004)
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 Properties	Be reluctant to trust Secure weakest link
Usability Properties	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.

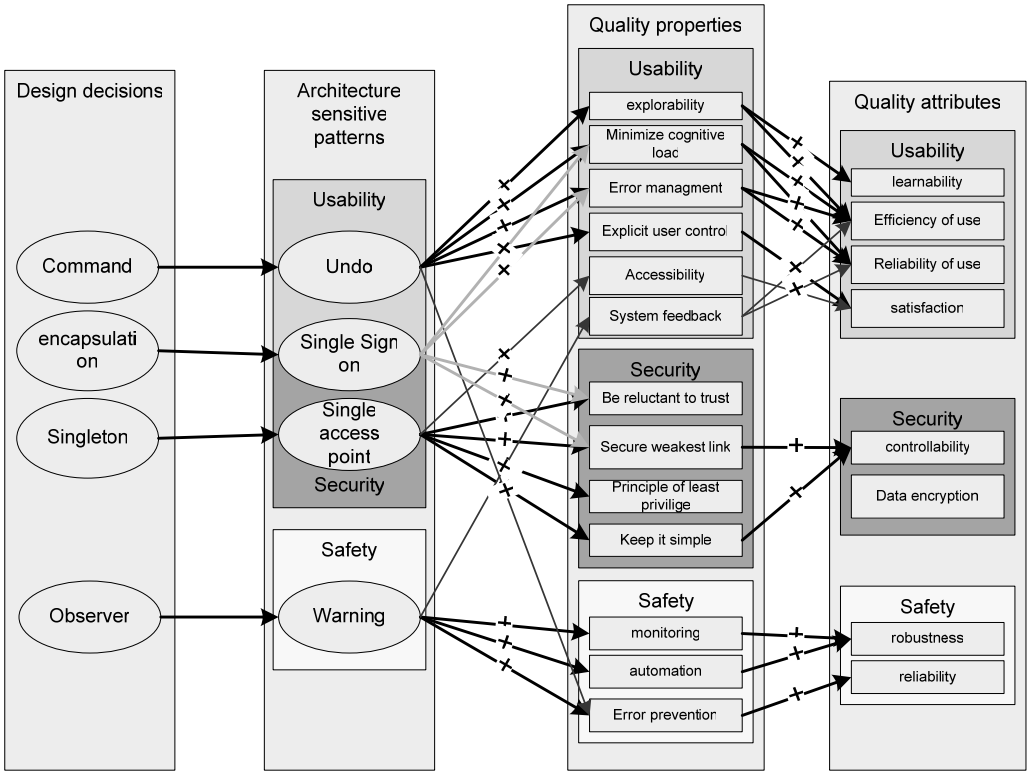

Figure 59: Framework Showing SSO as a "Boundary" Pattern

6.4 Discussion

Contents of the framework

Crucial to the success of the framework is an accurate filling in of the framework. At this moment only a framework for usability (Folmer et al, 2003) has been developed and we provided the skeleton for extending this framework to the qualities security and safety.
Concerning how a particular quality is composed of which attributes, how it can be measured by which indicators and which properties 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 quality models to identify some attributes and properties. These attributes and properties are far from complete. However we are convinced that any quality model can be used for that. For a given system different indicators may be defined. The indicators we propose in our framework are only a suggestion based upon the quality models. It is up to an analyst using our framework to select and find those indicators that for his or her system are valid.

Relationships

Some relations in the framework are hard to express. 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 attribute of efficiency. A warning for example a popup increases task length and consequently task execution times. Such things are hard to express in our framework since there is no direct property related to these issues. A solution could be to directly relate one of the patterns to an indicator for a quality attribute, but because indicators are very application dependent it would make the framework less generally applicable. A similar argumentation holds for the properties. 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.

Relationships have been defined between the elements of the framework. However these relationships only indicate positive 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. Being able to quantify these relationships would greatly enhance the use of our framework. In order to get quantitative data we 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 (generic) quantitative information is possible, as some relationships may be very context dependent. In order to overcome this we consider putting this framework in a tool and allow architects and engineers to put weights on the patterns and properties that they consider to be important.

Applicability of the framework

Our pattern format and the example patterns we have shown are not intended to be exhaustive. For specific application domains, the architecture sensitiveness of a pattern's implementation may be different. Therefore a number of domain specific architectural considerations may need to be provided. For now the patterns architectural sensitivity is based upon examining web based applications. For certain

systems the patterns in our framework may not be applicable. For example for an emailing system, undo may not make any sense, as sending an email cannot be undone.

The same pattern may be implemented or adopted in many parts of the target system. In order to be able to analyze the impact on qualities somehow this must be related to a set of requirements. In order to accurately determine an architectures support for quality we encourage the use of some formalized method such as scenario based assessment that allows one to explicitly define the required quality of the system. When a set of scenarios has been defined that expresses the required quality of the system one can analyze for each scenario whether certain patterns and properties influence the required quality that is expressed by the scenario. In this case also multiple instances of the pattern implementation can be taken into account.

Completeness

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.

Architecture sensitivity

Architecture sensitivity of a specific pattern for a specific system can be a relative notion and depends on the context in which a pattern is being applied. The architectural impact of a pattern on a given system very much depends on what already has been implemented in the system. If some toolkit or framework is used that already supports a pattern, the impact may be low while the impact can be high if no pattern support is there. In that case, that pattern for that system is no longer considered to be "architectural" because with little effort it could be implemented. When writing architecture sensitive patterns it is therefore the responsibility of the pattern writers to identify the main dependencies together with an explanation of how they influence the architectural sensitivity.

Terminology

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. This is something different than Object-oriented (OO) application frameworks. A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications (Johnson and Foote, 1988). 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 attributes and indicators and properties but these may also be replaced by terms such as factors (McCall et al, 1977), measures and design heuristics.

Creating / adding patterns

When creating new patterns that fit in our framework only a subset of all existing quality improving patterns can be meaningfully converted. Certain patterns do not require any significant programming that can generically be described as an architecture sensitive pattern. For example mode Cursor (Welie and Trætteberg, 2000) is an example of an interaction design pattern that does not require any significant programming nor has any architectural impact.

We intend to publish our pattern collection as a WIKI on the web. In that case developers can add their own patterns to the collection and/ or modify existing patterns.

Selection rather than 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 engineering 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. The role of 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/design patterns) to selecting the right components and composing them to work together while still ensuring a particular level of quality. Within this selection process our framework can also be useful. Determining whether a particular framework provides sufficient support for quality can be based upon the amount of evidence for patterns and properties support that can be extracted from the architecture of such a framework. For example, one may choose to use the JAVA struts framework based on that it supports the Model 2 approach; a variation of the classic Model view controller paradigm. Using MVC multiple views and or controllers can be defined to support multiple devices which increases the accessibility property of usability.

Implementation clusters

As identified when analyzing architecture sensitive usability patterns with the Undo pattern often some sort of framework is necessary that uses design- or architectural patterns. Practice shows that a number of usability patterns such as Auto save (Laakso 2004), Cancel (Bass et al, 2001, Workflow patterns) and Macros (Common ground, 1999) are often implemented using the same implementation framework needed for Undo. This is very interesting since often engineers are not aware multiple patterns can be facilitated using the same implementation framework. It is our assumption that for other qualities also clusters of patterns can be found that share the same implementation.

6.5 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 (Gamma et al 1995). 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 (Gamma et al 1995) Unlike the design patterns, architecturally sensitive patterns do not specify a specific design solution in terms of objects and classes. Instead, we outline potential architectural implications that face developers looking to solve the problem the architecturally sensitive pattern represents.

One aspect that architecturally sensitive patterns share with design patterns is that their goal is to capture design experience in a form that can be effectively reused by software designers in order to improve the quality of their software, without having to address each problem from scratch.
Concerning patterns for qualities a lot of work has been done in the area of usability patterns (also known as interaction patterns) (Tidwell 1998, Perzel and Kane 1999, Welie and Trætteberg, 2000). Considerable work has been done on other qualities such as security patterns (Yoder and Barcalow, 1997, Kienzle and Elder, 2002), safety patterns (Mahemof and Hussay, 1999), maintainability patterns (Fowler et al, 1999), performance and reliability (Microsoft, 2004) etc. But to the best of our knowledge none of these patterns discuss architecture related issues (as far as that applies to certain patterns).

The layered view on usability presented in (Welie et al, 1999) inspired several elements of the framework model we presented in Section 2. For example their usability layer inspired our attribute layer. Their usage indicators layer inspired our indicators layer and their means layer forms the basis for our properties and patterns layer. One difference with their layered view is that we have made a clear distinction between patterns (solutions) and properties (requirements). In addition we have tried to explicitly define the relationships between the elements in our framework.

In (Chung et al,) 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 (Bachman et al, 2002, Bachman et al, 2003). 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 attributes. 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 properties 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. Also the focus in our work lays more on web based applications and theirs is more focused on the generic domain.

6.6 Conclusions

The framework we outline and present 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. With our framework we aim to consolidate existing design knowledge in order to:

- **Guide architectural design**: The framework shows for a set of qualities which design solutions should be considered during architecture design. Deciding to implement some of these patterns still allows for some quality tuning during detailed design, this may save some of the high costs incurred by adaptive maintenance activities once the system has been implemented. The framework may also reveal that certain patterns can be facilitated by the same design decision.

- **Reason about tradeoffs**: For each design decision tradeoffs must be made. A specific solution may improve a particular quality but may also impair others. It is very important to explicate these relationships to inform architects in the decision making process as it is still cheap to change such decisions during architecture design. Our framework explicates the tradeoffs that are made when applying a pattern.

- **Architecture assessment**: When heuristically evaluating a software 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 properties support that can be extracted from the architecture of the system under analysis.

Crucial to the success of the framework is an accurate filling in of the framework. At this moment only a framework for usability (Folmer et al, 2003) has been developed. But we intend to expand and refine this framework in the future also for other quality attributes. The relationships depicted in the framework indicate potential relationships. Further work is required to substantiate and quantify as far as possible these relationships and to provide models and assessment procedures for the precise way that the relationships operate. Some patterns such as single sign on have been identified that somehow do not necessarily lead to "traditional" tradeoffs between qualities.