Software architecture analysis of usability

Folmer, Eelke

Chapter 4

The SAU Framework

Published as: A Framework for capturing the Relationship between Usability and Software Architecture, Eelke Folmer, Jilles van Gurp, Jan Bosch, Software Process: Improvement and Practice, Volume 8, Issue 2, Pages 67-87, June 2003.

Abstract: Usability is increasingly recognized as an essential factor that determines the success of software systems. Practice shows that for current software systems, most usability issues are detected during testing and deployment. Fixing usability issues during this late stage of the development proves to be very costly. Some usability improving modifications such as usability patterns may have architectural implications. We believe that the software architecture may restrict usability. The high costs associated with fixing usability issues during late stage development prevent developers from making the necessary adjustments for meeting all the usability requirements. To improve upon this situation we have investigated the relationship between usability and software architecture to gain a better understanding of how the architecture restricts the level of usability. Our paper makes a number of contributions; a framework is presented that expresses the relationship between usability and software architecture. The framework consists of an integrated set of design solutions such as usability patterns and usability properties that have been identified in various cases in industry, modern day software, and literature surveys. These solutions, in most cases, have a positive effect on usability but are difficult to retrofit into applications because they have architectural impact. Our framework may be used to guide and inform the architectural design phase. This may decrease development costs by reducing the amount of usability issues that need to be fixed during the later stages of development.

4.1 Introduction

In recent years, usability has increasingly been recognized as an important consideration during software development. Issues such as whether a product is easy to learn, to use, or whether it is responsive to the user and whether the user can efficiently complete tasks using it, may greatly affect a product's acceptance and success in the marketplace. In the future, as users become more critical, poor usability may become a major barrier to the success of new commercial software applications. Therefore, software developing organizations are paying more and more attention to ensuring the usability of their software.

One of the problems with many of today's software systems is that they do not meet their quality requirements very well. In addition, it often proves hard to make the necessary changes to a system to improve its quality. A reason for this is that many of the necessary changes require changes to the system that cannot be easily accommodated by the software architecture (Bosch, 2000). In other words, the software architecture; "the 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) does not support the required level of quality.

The work in this paper is motivated by the fact that this also applies to usability. Many well-known software products suffer from usability issues that cannot be repaired without major changes to the software architecture of these products. Studies of software engineering projects reveal that organizations spend a relatively large amount of time and money on fixing usability problems. Several studies have shown that 80% of total maintenance costs are related to problems of the user with the system (Pressman, 1992). Among these costs, 64% are related to usability problems (Landauer, 1995). This is called adaptive maintenance by (Swanson, 1976). These figures show that a large amount of maintenance costs are spent on dealing with usability issues such as frequent requests for interface changes by users, implementing overlooked tasks and so on (Lederer and Prassad, 1992).

An important reason for these high costs is that most usability issues are only detected during testing and deployment rather than during design and implementation. Consequently, a large number of change requests to improve usability are made after these phases. This makes meeting all the usability requirements expensive. Potential causes for this problem are:

- Evaluation of usability requires a working system/prototype. In (Folmer and Bosch, 2004) we observed that in order to do a usability evaluation, most existing techniques require at least an interactive prototype and a representative set of users present to assess the usability of a system. Some techniques such as rapid prototyping (Nielsen, 1993) allow for early testing, for example by using a prototype or simulation of an interface. Early prototyping, even on paper, of what the customer's experience will be like, always has great value. However, prototypes have a limited ability to model the application architecture, since they only model the interface. Interaction issues such as whether a task can be undone, or the time it takes to perform a specific task or system properties such as reliability have a great influence on the level of usability. Such issues are hard to simulate with a prototype.

- Limitation of requirements engineering: Usability-engineering techniques such as usage-centered design (Constantine and Lockwood, 1999) have a limited ability to capture or predict all usability requirements. During development, neither the uses of the system are often not fully documented nor a definition is made of exactly who the users are. Users themselves lack understanding of their own requirements. No sooner do they work with a first version of the software do they realize how the system is going to be used. Usability experts miss about half of the problems that real users experience using traditional techniques (Cuomo and Bowen, 1994). Therefore, some usability requirements will not be discovered until the software has been deployed.

- Change of requirements: During or after the development, usability requirements change. The context in which the user and the software operate is continually changing and evolving, which makes it hard to capture all possible (future) usability requirements at the outset (Gurp & Bosch, 2002). Sometimes users may find new uses for a product, which changes the required usability.

These are three of the main reasons that some usability problems are not discovered until testing and deployment. The problem with this late detection is that sometimes it is very difficult to apply design solutions that fix these usability problems, because some of these design solutions may be 'architecturally sensitive'. Some changes that

may improve usability require a substantial degree of modification. For example, changes that relate to the interactions that take place between the system and the user, such as undo to a particular function. Such a modification cannot be implemented easily after implementation without incurring great costs.

A first step in solving these problems is to investigate which solutions are architecturally sensitive. The primary motivation behind the STATUS project that sponsored the research presented in this paper is to gain a better understanding of this relationship between usability and software architecture.

The contribution of this paper, which is one of the cornerstones of this effort, is a framework that expresses the relationship between usability and software architecture. This framework describes an integrated set of 'design solutions', that, in most cases, have a positive effect on the level of usability but that are difficult to retro-fit into applications because these design solutions may require architectural support. For each of these design solutions we have analyzed the usability effect and the potential architectural implications.

The remainder of this paper is organized as follows. The next section presents the relationship between software architecture and usability and presents a framework that expresses this relationship. Section 4.3, 4.4, and 4.5 discuss the elements that compose the framework: usability attributes, usability properties and architecturally sensitive usability patterns. Section 0 discusses the relationships between the elements of our framework and how this framework may be used to inform design. Finally, related work is discussed in section 4.7 and the paper is concluded in section 4.8.

4.2 The Relationship between SA and Usability

Our investigation of the relationship between usability and software architecture has resulted in the definition of a framework, which has the following purposes:

- Express the relationship between usability and software architecture.
- Inform design: existing design knowledge and experience in the software engineering and usability community is consolidated in a form that allows us to inform architectural design.

The framework consists of three layers:

- **Attribute layer**: a number of usability attributes (see section 4.3) have been selected from literature that appear to form the most common denominator of existing notions of usability.
- **Properties layer**: the usability properties embody the heuristics and design principles that researchers in the usability field consider to have a direct influence on system usability. The usability properties (see section 4.4) are a means to link architecturally sensitive usability patterns to usability attributes.
- **Patterns layer**: consists of architecturally sensitive usability patterns (see section 4.5) that we have identified from various industrial case studies and modern software applications as well as from existing (usability) pattern collections. We have focused on selecting patterns that are architecture-

sensitive and have omitted patterns in which the architectural sensitiveness was not clear. In addition, we have abstracted from patterns that are similar from the point of view of a software architect.

The framework expresses the relationship between usability and software architecture. The following two examples illustrate this relation:

An architecturally sensitive usability pattern that we identified from a case presented by one of our industrial partners in the STATUS project is the wizard pattern (Figure 31). The ESUITE product developed by LogicDIS is a system that allows access to an ERP (Enterprise Resource Planning) system, through a web interface. Part of this application is a shopping cart, which uses a wizard for checking out items that are purchased. The checkout procedure uses a wizard that helps users to accomplish the actual purchase with all possible assistance.

The wizard pattern guides the user through a complex task by decomposing the task into a set of manageable subtasks. This usability pattern is described in several usability pattern collections such as (Welie, 2003). In the cases we studied, where this pattern had been implemented, to implement a wizard pattern the following architectural considerations must be made:

- There needs to be a provision in the architecture for a wizard component, which can be connected to other relevant components, the one triggering the operation, and the one receiving the data gathered by the wizard.

- The wizard may improve usability because it relates to the principle of guidance to "assist" the user through performing the task. The concept of "guidance" is an example of a usability property. Guidance may have a positive effect on the attribute learnability but may negatively effect the attribute efficiency (Scapin and Bastien, 1997); (Ravden and Johnson, 1989).

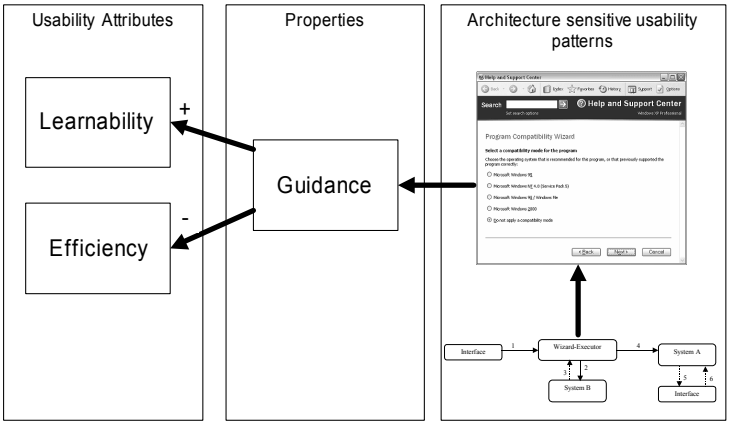

Figure 31: Wizard Pattern

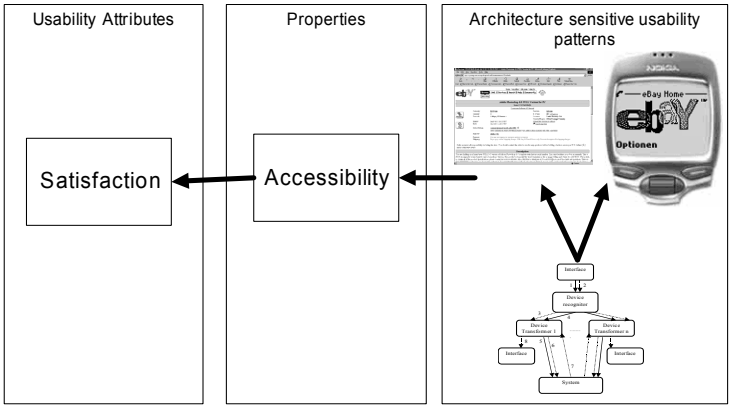

Figure 32: Multi Channeling Pattern

Another architecturally sensitive usability pattern that we identified from a case study is the multi-channeling pattern (Figure 32). The compressor catalogue application is a product developed by imperial highway group (IHG) for a client in the refrigeration industry. It is an e-commerce application, which makes it possible to search for detailed technical information about a range of equipment (compressors). This is a multichanneling application offering access through a variety of devices (e.g. desktop computer, WAP phone). Multi-channeling refers to the capability of the software to be accessed using different types of devices. The architectural considerations that must be made are:

- There must be a component that monitors how users access the application. Depending on which medium is used, the system should make adjustments. For example by presenting a different navigation menu or by limiting the number of images that is sent to the user.

- We identified that the pattern multi-channeling relates to the usability property of accessibility. Accessibility may improve the usability attribute satisfaction (Holcomb and Tharp, 1991, Nielsen, 1993).

The next sections enumerate the concepts of usability attributes, properties, and architecturally sensitive usability patterns that comprise our framework. The full relationships between these concepts are presented in section 0.

4.3 Usability Attributes

Our investigation of the relationship between software architecture and usability started with a survey of existing literature (Folmer and Bosch, 2004) to find a commonly accepted definition of usability in terms of a decomposition of usability into usability attributes. Our survey revealed that different researchers have different definitions for the term usability attribute, but the generally accepted meaning is that a usability attribute is a precise and measurable component of the abstract concept that is usability. For our attributes we have merely taken the subset of attributes most commonly cited amongst authors, (Constantine and Lockwood, 1999), (Hix and Hartson, 1993), (ISO 9126-1), (Nielsen, 1993), (Preece et al, 1994), (Shackel, 1991), (Shneiderman, 1998) and (Wixon & Wilson, 1997) in the usability field. Table 7 gives a short overview of the attributes that are most commonly cited. We wanted to be able to

compare the different decompositions, so we have grouped attributes that refer to the same concept in the same row. Some authors use more attributes than the ones stated in the table.

Table 7: Overview of Authors

Constantine	ISO9126	Nielsen	Preece	Shackel	Shneiderman
Learnability	Learnability	Learnability	Learnability	Learnability	Time to learn
Efficiency in use	Operability	Efficiency of use	Throughput	Effectiveness	Speed of performance
Rememberability	-	Memorability	-	Learnability	Retention over time
Reliability in use	Operability	Errors	Throughput	Effectiveness	Rate of errors by users
User satisfaction	Attractiveness	Satisfaction	Attitude	Attitude	Subjective Satisfaction

We have not tried to innovate in this area, since abundant research has already focused on finding and defining the optimal set of attributes that compose usability. We have merely taken the set of attributes most commonly cited, the four attributes that we selected are:

- Learnability: how quickly and easily users can begin to do productive work with a system that is new to them, combined with the ease of remembering the way a system must be operated.
- Efficiency of use: the number of tasks per unit time that the user can perform when using the system.
- Reliability in use: the error rate (i.e. user errors, not system errors) when using the system and the time it takes to recover from errors.
- Satisfaction: the subjective opinions of the users of the system.

In our definition of usability attributes, we have grouped memorability, retention over time (Nielsen, 1993, Shneiderman, 1998) rememberability (Constantine and Lockwood, 1999) with learnability. Some definitions such as (ISO 9126-1) and (Preece et al, 1994) do not mention this attribute and (Shackel, 1991) also links retention over time with learnability. We have also used the term reliability to refer to the error rate in using the system.

4.4 Usability Properties

When we identified our architecturally sensitive patterns, we discovered that it was extremely difficult to draw a direct relationship between the four usability attributes we identified and these patterns. Some usability patterns that we considered; for example, a wizard (see Table 28) have an influence on the usability attributes learnability and efficiency, but many usability patterns do not have this direct relationship. For example, multi-channeling (Table 24) or multiple views (Table 23) do not appear to have a direct relationship to usability attributes.

However, we want to have this direct relationship between usability and software architecture, to be able to describe and categorize our usability patterns in such a way that they can be used as requirements that can support architectural design. The problem with usability requirements is that traditionally these are specified such that these can only be verified for an implemented system. Such requirements are largely useless in a forward engineering process. For example, a usability requirement that specifies: “the system should be easy to learn”, or “new users should require no more than 30 minutes instruction” does not help guide the design process since there is no obvious way in which the 30-minute time span can be measured when evaluating a design for usability. Usability requirements need to take a more concrete form expressed in terms of the solution domain to be able to influence and guide architectural design.

The concept of usability property links architecturally sensitive usability patterns to requirements and creates a direct relationship between usability attributes and our architecturally sensitive usability patterns. Essentially, these properties embody the heuristics and design principles that researchers in usability field have found to have a direct influence on system usability. For example, one of the concepts that we defined as a usability property is guidance. In order to help the user understand and use the system, the system should provide informative, easy to use, and relevant guidance and support in the application as well as in the user manual. From various literature sources (Scapin and Bastien, 1997), (Ravden and Johnson, 1989) it is known that guidance may improve learnability, but may negatively affect efficiency.

These properties can be used as requirements at the design stage, for instance by specifying, "The system must provide feedback". However, these are not strict requirements that should be fulfilled at all costs. They should be considered as high-level design primitives that have a known effect on usability and most likely have architecture implications, for example by using architecturally sensitive usability patterns, which relate to such a property. It is up to the software architect to decide how and at which levels these properties are implemented. For instance providing feedback when printing in an application is considered to have a positive effect on the usability of the system because it keeps the user informed about the state of the system, which increases user satisfaction. However, if every possible user action would result in feedback from the system it would just be annoying and negatively affect usability. Therefore, these properties should be implemented with care.

To fulfill usability requirements during architectural design architecturally sensitive usability patterns may be applied that relate to specific usability properties, which is discussed in section 0.

Our properties have been derived from an extensive survey of design heuristics and principles of various authors (Shneiderman, 1998), (Nielsen, 1993), (Constantine and Lockwood, 1999), (Ravden and Johnson, 1989), (Hix and Hartson, 1993), (Norman, 1988), (Polson and Lewis, 1990), (ISO 9241-11), (Holcomb and Tharp, 1991), (Rubinstein and Hersh, 1984) and step by step analysis and discussion with the STATUS partners.

Similarly to identifying the usability attributes we have limited ourselves to merely taking a set of properties most commonly cited amongst authors in the usability field that addressed all of the patterns that we identified, rather than inventing our own properties. However, for some heuristics, we have used different names and we have

grouped similar heuristics together under a new name. For example we grouped 'speak the users language' (Nielsen, 1993) and 'allow access to operations from other applications' (Holcomb and Tharp, 1991) under the property of accessibility.

Table 8 - Table 16 presents the properties in a pattern-like format (Alexander et al, 1977) with the following aspects listed for each property:

Name: the name of the property.
Intent: a short statement that describes the properties rationale and intent (Gamma et al 1995).
References: lists some of the authors that consider this property and that describe the effect on usability.
Usability attributes affected: which usability attributes identified in section 4.3 are affected by this property. Only the relationships that in our experience are the strongest are indicated, positive as well as negative relationships. In some systems, the indicated relationships may not apply. These relationships have been derived from our literature survey.
Example: an illustration of the property (not necessarily implemented in an architecture-sensitive fashion)

Table 8: Providing Feedback	
Intent:	The system should provide at every (appropriate) moment feedback to the user in which case he or she is informed of what is going on, that is, what the system is doing at every moment.
Usability attributes affected:	+ Efficiency: feedback may increase efficiency, as users do not have to wonder what the system is doing. + Learnability: feedback may increase learnability, as users know what the system is doing.
References:	(Nielsen, 1993), (Constantine and Lockwood, 1999)
Example:	Progress indication during a file download.

Table 9: Error Management	
Intent:	The system should provide a way to manage user errors. This can be done in two ways: • By preventing errors from happening, so users make fewer mistakes. • By providing an error correcting mechanism, so that errors made by users can be corrected.
Usability attributes affected:	+ Reliability: error management increases reliability because users make fewer mistakes. + Efficiency: efficiency is increased because it takes less time to recover from errors or users make fewer errors.
References:	(Nielsen, 1993), (Hix and Hartson, 1993)
Example:	Red underline for a syntax error in Eclipse (a popular Java development environment).

Table 10: Consistency	
Intent:	Users should not have to wonder whether different words, situations, or actions mean the same thing. An essential design principle is that consistency should be used within applications. Consistency might be provided in different ways: • Visual consistency: user interface elements should be consistent in aspect

	and structure. • Functional consistency: the way to perform different tasks across the system should be consistent, also with other similar systems, and even between different kinds of applications in the same system. • Evolutionary consistency: in the case of a software product family, consistency over the products in the family is an important aspect.
Usability attributes affected:	+ Learnability: consistency makes learning easier because concepts and actions have to be learned only once, because next time the same concept or action is faced in another part of the application, it is familiar. + Reliability: visual consistency increases perceived stability, which increases user confidence in different new environments.
References:	(Nielsen, 1993), (Shneiderman, 1998), (Hix and Hartson, 1993)
Example:	Most applications for MS Windows conform to standards and conventions with respect to e.g. menu layout (file, edit, view, ..., help) and key-bindings.

Table 11: Guidance	
Intent:	In order to help the user understand and use the system, the system should provide informative, easy to use, and relevant guidance and support in the application as well as in the user manual.
Usability attributes affected:	+ Learnability: guidance informs the user at once which steps or actions will need to be taken and where the user currently is, which increases learnability. (Welie, 2003) - Efficiency: guidance may decrease efficiency as users are forced to follow the guidance. (For example when following a wizard) + Reliability: when users are forced to follow a sequence of tasks, users are less likely to miss important things and will hence make fewer errors. (Welie, 2003)
References:	(Scapin and Bastien, 1997), (Ravden and Johnson, 1989)
Example:	ArgoUML, a popular UML modeling tool auto generates a to-do list based on lacking information in models under construction.

Table 12: Minimize Cognitive Load	
Intent:	Humans have cognitive limitations, designers should keep these limitations in mind i.e. presenting more than seven items on the screen is an overload of information. Therefore, systems should minimize the cognitive load.
Usability attributes affected:	+ Reliability: as users are less distracted by objects or functions not of their interest they are less likely to make errors. + Efficiency: minimize cognitive load may increase efficiency, as users are not distracted by objects or functions, which are not of their interest. - Efficiency: for expert users this argument goes the other way around. (see the auto hide feature in office applications).
References:	(Nielsen, 1993), (Hix and Hartson, 1993)
Example:	The auto hide menu-items feature in Office applications.

Table 13: Explicit User Control	
Intent:	The user should get the impression that he is "in control" of the application.
Usability attributes affected:	+ Satisfaction: interaction is more rewarding if the users feel that they directly influence the objects instead of just giving the system instructions to act.
References:	(Hix and Hartson, 1993), (Shneiderman, 1998)
Example:	The cancel button when copying a large file allows users to interrupt the operation.

Table 14: Natural Mapping	
Intent:	The system should provide a clear relationship between what the user wants to do and the mechanism for doing it. This property can be structured as follows: • Predictability: the system should be predictable; e.g. to the user the behavior of the system should be predictable. • Semiotic significance: systems should be semiotically significant; Semiotics, or semiology, is the study of signs, symbols, and signification. It is the study of how meaning is created, not what it is. • Ease of navigation: it should be obvious to the user how to navigate the system.
Usability attributes affected:	+ Learnability: if the system provides a clear relationship between what the user wants to do and the mechanism for doing it, users have less trouble learning something that is already familiar to them in the real world. + Efficiency: a clear relationship between what needs to be done and how, may increase efficiency. + Reliability: a clear relationship between what and how minimizes the number of errors made performing a task.
References:	(McKay, 1999), (Norman, 1988)
Example:	The recycle bin on the desktop is an easy to remember metaphor.

Table 15: Accessibility	
Intent:	Systems should be accessible in everyway that is required. Such property might be decomposed as follows: • Disabilities: systems should provide support for users that are disabled (blind/deaf/short sighted). • Multi-channeling: the system should be able to support access via various media. Multi channeling (accessing) in this way is a very broad concept varying from being able to browse a website via a phone or being able to browse a website through audio (support for audio output). • Internationalization: systems should provide support for internationalization, because users are more familiar with their own language, currency, ZIP code format, date format etc.
Usability attributes affected:	+ Satisfaction: accessibility may increase satisfaction by allowing the use of the system adapted to their (familiar) context (access medium, language, disability etc). + Learnability: learnability may be improved for internationalization because users are more familiar with their own language, currency etc.
References:	(Nielsen, 1993), (Holcomb and Tharp, 1991)
Example:	The w3c CSS (Cascading Style Sheets) standard supports multi-channeling by allowing developers to make specific style rules for printer layout, web layout etc.

Table 16: Adaptability	
Intent:	The system should be able to satisfy the user's needs when the context changes or adapt to changes in the user. Such property might be decomposed as follows: • User experience: Ability to adapt to changes in the user's level of experience. • Customization: Ability to provide certain customized services. • System memorability: capacity of the system for remembering past details of the user-system interaction.

Usability attributes affected:	+ Satisfaction: satisfaction may be increased because users can express their individual likes and preferences. + Efficiency: adaptability allows the system to adept to the skills or preferences or details of the user, which may increase user's efficiency.
References:	(Scapin and Bastien, 1997), (Norman, 1988), (McKay, 1999)
Example:	Customization: Winamp allows skinning. Users can apply a skin they have downloaded or created themselves to the interface of the Winamp application.

4.5 Architecturally Sensitive Usability Patterns

One of the products of the research into the relationship between software architecture and usability is the concept of an architecturally sensitive usability pattern. The implementation of a usability pattern is a modification that may solve a specific usability problem in a specific context, but which may be very hard to implement afterwards because such a pattern may have architectural implications. An "architecturally sensitive usability pattern" refers to a technique or mechanism that should be applied to the design of the architecture of a software system in order to address a need identified by a usability property at the requirements stage (or an iteration thereof).

The purpose of identifying and defining architecturally sensitive usability patterns is to capture design experience to inform architectural design and hence avoid the retrofit problem. There are many different types of patterns. In the context of this paper, we use the term pattern in a similar fashion as (Buschmann et al, 1996): "patterns document existing, well-proven design experience". With our set of patterns, we have concentrated on capturing the architectural considerations that must be taken into account when deciding to implement a usability pattern.
Our architecturally sensitive usability patterns have been derived from three sources:

- Internal case studies at the industrial partners in the STATUS project.
- Existing usability pattern collections (Tidwell 1998),(Brighton, 1998),(Welie and Trætteberg, 2000), (PoInter, 2003).
- An study into the relationship between usability and software architecture (Bass et al, 2001).

Only those patterns are selected or defined that require architectural support. We have merely annotated existing usability patterns for their architectural sensitiveness. When necessary we have defined new patterns or grouped patterns together to define a pattern at the highest possible level of abstraction. For example the usability patterns "progress indication" (Welie, 2003) and "wizard/alert" (Brighton, 1998) have been combined in a pattern called "system feedback" since it covers both patterns. This has also been done for defining the usability properties.

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:

Name: whenever possible we use the names of existing patterns. However, some patterns are known under different names and some patterns are not recognized in usability pattern literature.

Usability context: a situation giving rise to a usability problem, the context extends the plain problem-solutions dichotomy by describing situations in which the problems occur. This is similar to the context used in the patterns defined in (Buschmann et al, 1996).

Intent: a short statement that answers the following questions: what does the pattern do and what are its rationale and intent. Similar to the patterns in (Gamma et al 1995).

Architectural implications: it may be possible to use a number of different methods to implement the solution presented in each usability pattern. Some of our architecturally sensitive usability patterns such as undo (table 5.9) may be implemented by a design pattern. For example, the Memento pattern should be used whenever the internal state of an object may need to be restored at a later time (Gamma et al 1995) Alternatively, an architectural pattern may be used. For example providing multiple views (table 0) by using a model view controller pattern (Buschmann et al, 1996). Our patterns do not specify implementation details in terms of classes and objects. We specify a level abstracted from that. However, we are contemplating a case study to analyze how companies implement the patterns we discuss in this paper. Furthermore, we present the architectural considerations that must be taken into account to implement the pattern. In the case of the wizard example there may need to be a provision in the architecture for a wizard component, which can be connected to other relevant components, the one triggering the operation and the one receiving the data gathered by the wizard. This leads to architectural decisions about the way that operations are managed.

Usability Properties affected: for each pattern we specify its relation to one or more usability properties. For example, the wizard pattern (Table 28) relates to the property of guidance. The cancel pattern (Table 19) relates to the property of explicit user control but is also related to the property of error prevention. These relations are also acknowledged in usability pattern literature. The usability properties as we defined them may be used as requirements to inform design. For each architecturally sensitive usability pattern, we specify to which usability properties it relates. There is no one-to-one mapping between patterns and the usability properties that they affect. A pattern may be related to any number of properties, and each property may be improved (or impaired) by a number of different patterns. A complete overview of the relationships is presented in section 0.

Examples: similar to patterns described in (Gamma et al 1995) and (Buschmann et al, 1996) we present three examples of the use of the pattern in current software (not necessarily implemented in an architecture-sensitive fashion).

Our pattern format is not intended to be exhaustive. We intend to add to the collection in future work and actively engage in discussions with the usability and software engineering communities through e.g. workshops and our website (http://www.designforquality.com). Future work will lead to the expansion and reworking of the set of patterns presented here. This includes work to fill out the elements of each pattern to include more of the sections, which traditionally make up a pattern description, for instance what the pros and cons of using each pattern may be, forces that lead to the use of the pattern, aliases etc.

Table 17: System Feedback	
Usability context:	Situations where the user performs an action that may unintentionally lead to a problem (Welie, 2003)
Intent:	Communicate changes in the system to the user.
Architectural implications:	To support the provision of alerts to the user, there may need to be a component that monitors the behavior of the system and sends messages to an output device. Furthermore, some form of asynchronous messaging (e.g. events) support may be needed to respond to events in other architecture components. (Buschmann et al, 1996) suggests several architectural styles to implement asynchronous messaging (e.g. the blackboard style).
Usability properties affected:	+ Feedback: alerts help to keep the user informed about the state of the system, which is a form of feedback. + Explicit user control: giving an indication of the system's status provides feedback to the user about what the system is currently doing, and what will result from any action they carry out.
Examples:	• If a new email arrives, the user may be alerted by means of an aural or visual cue. • If a user makes a request to a web server that is currently off line, they will be presented with a popup window telling them that the server is not responding. • If a user is running out of disk space, windows XP will alert the user with a popup box in the system tray.

Table 18: Actions for Multiple Objects	
Usability context:	Actions need to be performed on objects, and users are likely to want to perform these actions on two or more objects at one time (Tidwell 1998).
Intent:	Provide a mechanism that allows the user to customize or aggregate actions.
Architectural implications:	A provision needs to be made in the architecture for objects to be grouped into composites, or for it to be possible to iterate over a set of objects performing the same action for each.
Usability properties affected:	+ Explicit user control: providing the user with the ability to group the objects and apply one action to them all "in parallel" increases explicit user control. + Error management: if each object has to be treated individually, errors are more likely to be made.
Examples:	• In a vector based graphics package such as Corel Draw, it is possible to select multiple graphics objects and perform the same action (e.g. change color) on all of them at the same time. • Copying several files from one place to another. • Outlook allows the selection of different received emails and forward them all at once.

Table 19: Cancel	
Usability context:	The user invokes an operation, then no longer wants the operation to be performed. (Bass et al, 2001)
Intent:	Allow the user to cancel a command that has been issued but not yet completed, to prevent reaching an error state.
Architectural implications:	There needs to be provision in the architecture for the component(s) monitoring the user input to run independently from and concurrently with the components that carry out the processing of actions. The components processing actions need to be able to be interrupted and the consequences of the actions may need to be rolled back.

Usability properties affected:	+ Error management: the ability to cancel commands is a form of error management, if the user realizes that he or she has initiated an incorrect action then this action can be interrupted and cancelled before the error state is reached. + Explicit user control: it also gives the user the feeling that they are in control of the interaction (Explicit user control)
Examples:	• In most web browsers, if the user types a URL incorrectly, and the web browser spends a long time searching for a page that does not in fact exist. The user can cancel the action by pressing the "stop" button before the browser presents the user with a "404" page, or a dialog saying that the server could not be found. • When copying files with windows explorer the user is able to press the cancel button to abort the file copy process. • Norton antivirus allows the user to interrupt or cancel the virus scanning process.

Table 20: Data Validation	
Usability context:	The user needs to supply the application with data, but does not know which data is required or what syntax should be used. (Welie and Trætteberg, 2000) Users have to input data manually which may result in errors.
Intent:	Verify whether (multiple) items of data in a form or field have been entered correctly.
Architectural implications:	To ensure that the integrity of the data stored in the system is maintained, a mechanism is needed to validate both the data entered by the user and the processed data. Solutions that may be employed include the use of XML and XML schemas. Furthermore, a data integrity layer consisting of business-objects may be implemented to shield application code from the underlying database. Finally, there may be some client or server components that verify the data entered by users.
Usability properties affected:	+ Error management: this pattern relates to a provision for the management of errors.
Examples:	• This pattern is often employed in forms on websites where the user has to enter a number of different data items, for example, when registering for a new service, or buying something. • Large content management systems often use XML to define objects. Some WYSIWYG tools that allow the user to edit these objects use the XML definition (DTD or schema) to prevent users from entering invalid data. • Use of a data integrity layer in multi tiered applications to shield user interface code from database.

Table 21: History Logging	
Usability context:	How can the software help save the user time and effort? (Tidwell 1998) How can the artifact support the user's need to navigate through it in ways not directly supported by the artifact's structure? (Tidwell 1998) The user performs a sequence of actions with the software, or navigates through it. (Tidwell 1998)
Intent:	Record a log of the actions of the user (and possibly the system) to be able to look back over what was done.
Architectural implications:	In order to implement this, a repository must be provided where information about actions can be stored. Consideration should be given to how long the data is required. Actions must be represented in a suitable way for recording in the log. Additionally, such features may have some privacy/security implications.

Usability properties affected:	+ Error management: providing a log helps the user to see what went wrong if an error occurs and may help the user to correct that error.
Examples:	• Web browsers create a history file listing all the websites that the user has visited. Most web browsers also include functionality for purging this data. • Windows XP keeps track of recently accessed documents. • Automatic Form completion in Mozilla and Internet Explorer based upon previously inserted information.

Table 22: Scripting	
Usability context:	The user needs to perform the same sequence of actions over and over again, with little or no variability (Tidwell 1998).
Intent:	Provide a mechanism that allows the user to perform a sequence of commands or actions to a number of different objects.
Architectural implications:	A provision needs to be made in the architecture for grouping commands into composites or for recording and playing back sequences of commands in some way. There needs to be an appropriate representation of commands, and a repository for storing the macros. Typically, some sort of scripting language is often used to implement such functionality. This implies that all features must be scriptable.
Usability properties affected:	+ Minimize cognitive load: providing the ability to group a set of commands into one higher-level command reduces the user's cognitive load, as the user does not need to remember how to execute the individual steps of the process once the user has created a macro, the user just need to remember how to trigger the macro.
Examples:	• Microsoft's Office applications provide the ability to record macros, or to create them using the Visual Basic for Applications language. • Mozilla Firebird allows users to install extensions that extend the features of the program using scripts. • Open Office has java bindings that allows users to write Java programs that extend open office. Open office also supports a subset of VB.

Table 23: Multiple Views	
Usability context:	The same data and commands must be potentially presented using different human-computer interface styles for different user preferences, needs or disabilities. (Brighton, 1998)
Intent:	Provide multiple views for different users and uses.
Architectural implications:	The architecture must be constructed so that components that hold the model of the data that is currently being processed are separated from components that are responsible for representing this data to the user (view) and those that handle input events (controller). The model component needs to notify the view component when the model is updated, so that the display can be redrawn. Multiple views is often facilitated through the use of the MVC pattern (Buschmann et al, 1996)
Usability properties affected:	+ Consistency: separating the model of the data from the view aids consistency across multiple views when these are employed. + Accessibility: separating out the controller allows different types of input devices to be used by different users, which may be useful for disabled users. + Error management: having data-specific views available at any time will contribute to error prevention.
Examples:	• Microsoft Word has a number of views that the user can select (normal view, outline view, print layout view…) and switch between these at will, which all represent the same underlying data.

	• Rational Rose, uses a single model for various UML diagrams. Changes in one diagram affects related entities in other diagrams. • Nautilus file manager of the Gnome desktop software for Linux allows multiple views on the file system.

Table 24: Multi Channeling	
Usability context:	Users want or require (e.g. because of disabilities) access to the system using different types of devices (input/output). Increasing the number of potential users (customers) and usage of a system.
Intent:	Provide a mechanism that allows access using different types of devices (input/output).
Architectural implications:	There may need to be a component that monitors how users access the application. Depending on which device is used, the system should make adjustments. For example, by presenting a different navigation menu or by limiting the number of data/images sent to the user.
Usability properties affected:	+ Accessibility: this pattern improves system accessibility by users using different devices (accessibility)
Examples:	• Auction sites such as eBay can be accessed from a desktop/laptop, but this information can also be obtained using interactive TV or a mobile phone. • Some set top boxes allow users to surf the internet using an ordinary TV. • Some Word processors allow voice input, which allows (disabled) users to control the application by voice.

Table 25: Undo	
Usability context:	Users may perform actions they want to reverse. (Welie, 2003)
Intent:	Allow the user to undo the effects of an action and return to the previous state.
Architectural implications:	In order to implement undo, a component must be present that can record the sequence of actions carried out by the user and the system, and sufficient detail about the state of the system between each action so that the previous state can be recovered.
Usability properties affected:	+ Error management: providing the ability to undo an action helps the user to correct errors if the user makes a mistake. + Explicit user control: allowing the user to undo actions helps the user feel that they are in control of the interaction.
Examples:	• Microsoft Word provides the ability to undo and redo (repeatedly) almost all actions while the user is working on a document. • Emacs allows all changes made in the text of a buffer to be undone, up to a certain amount of change. • Photoshop provides a multilevel undo, which allows the user to set the number of steps that can be undone. This is necessary because storing the information required to do the operations requires a substantial amount of memory.

Table 26: User Modes	
Usability context:	The application is very complex and many of its functions can be tuned to the user's preference. Not enough is known about the user's preferences to assign defaults that will suit all users. Potential users may range from novice to expert (Welie, 2003).
Intent:	Provide different modes corresponding to different feature sets required by different types of users, or by the same user when performing different tasks.

Architectural implications:	Depending on the mode, the same set of controls may be mapped to different actions, via different sets of connectors, or different user interface components may be displayed. Using e.g. (Buschmann et al, 1996) Broker style may implement this.
Usability properties affected:	+ Adaptability: supporting different modes allows personalization of the software to the current user's needs or expertise. + Minimize cognitive load: expert users can tweak the application for their particular purposes.
Examples:	• WinZip allows the user to switch between "wizard" and "classic" modes, where the wizard mode gives more guidance, but the classic mode lets the expert user work more efficiently. • Many websites have different modes for different users, e.g. guests, normal, logged-in users or administrators. • ICQ allows the user to switch from novice user (limited functionality) to advanced mode thus enabling all functionality.

Table 27: User Profiles	
Usability context:	The application will be used by users with differing abilities, cultures, and tastes. (Tidwell 1998)
Intent:	Build and records a profile of each type of user, so that specific attributes of the system (for example, the layout of the user interface, the amount of data or options to show) can be set and reset each time for a different user. Different users may have different roles, and require different things from the software.
Architectural implications:	A repository for user data needs to be provided. This data may be added or altered either by having the user setting a preference, or by the system. User profiles often have a security impact that has major architectural implications.
Usability properties affected:	+ Adaptability: providing the facility to model different users allows a user to express preferences.
Examples:	• Many websites recognize different types of users (e.g. customers or administrators) and present different functionality tailored to the current user. • Amazon.com builds detailed profiles for each of its customers so it can recommend products the user might like. • .NET security model. By means of attribute oriented programming users can set security modes for three types of profiles.

Table 28: Wizard	
Usability context:	A non-expert user infrequently needs to perform a complex task consisting of several subtasks where decisions need to be made in each subtask (Welie, 2003).
Intent:	Present the user with a structured sequence of steps for carrying out a task and guide them through the sequence one by one. The task as a whole is separated into a series of more manageable subtasks. At any time, the user can go back and change steps in the process.
Architectural implications:	A wizard component can be connected to other relevant components, the one triggering the operation and the other receiving the data gathered by the wizard.
Usability properties affected:	+ Guidance: the wizard shows the user each consecutive step in the process is. + Minimize cognitive load: the task sequence informs the user which steps will need to be taken and where the user currently is.
Examples:	• The install wizard used by most Windows programs guides the user through choosing various options for installation.

	• When partitioning hard disks during Mandrake Linux install a user can use Disk Druid, which is a disk partition wizard. • Blogger.com allows a user to create a new web log (online publishing system) in four simple steps using a wizard. Advanced users may customize their web log afterwards by editing templates.

Table 29: Workflow Model	
Usability context:	A user who is part of a workflow chain (based on some company process), should perform its specific task efficiently and reliable.
Intent:	Provide different users only the tools or actions that they need in order to perform their specific task on a piece of data before passing it to the next person in the workflow chain.
Architectural implications:	A component or set of connectors that model the workflow is required, describing the data flows. A model of each user in the system is also required, so the actions that the user needs to perform on the data can be provided (see also user profile).
Usability properties affected:	+ Minimize cognitive load: targeting the user interface specifically to each user, dependent on the task that they need to perform in the workflow minimizes the user's cognitive load (minimize cognitive load). + Natural mapping: if workflow model is based upon business models, users switching to automation have less trouble switching over.
Examples:	• Most CMS and ERP systems are workflow model based. • A typical example of an administrative process that is workflow based is the handling of an expense account form. An employee fills in the proper information; the form is routed to the employee's manager for approval and then on to the accounting department to disburse the appropriate check and mail it to the employee. • Online publishing: a journalist writes an article and submits it online to an editor for review before it is published on the website of a newspaper. This process is often automated in a workflow model.

Table 30: Emulation	
Usability context:	Users are familiar with a particular system and now require consistency in terms of interface and behavior between different pieces of software.
Intent:	Emulate the appearance and/or behavior of a different system.
Architectural implications:	Command interfaces and views but also behavior needs to be replaceable and interchangeable, or there needs to be provision for a translation from one command language and view to another in order to enable emulation. This differs from the providing multiple views diagram because the behavior of the application should be replaceable.
Usability properties affected:	+ Consistency: emulation can provide consistency in terms of interface and behavior between different pieces of software.
Examples:	• Microsoft Word 97 can be made to emulate WordPerfect, so that it is easier to use for users who are used to that system. • Windows XP offers a new configuration menu; however, it is possible to switch to the "classic view" for users more familiar with windows 2000 or windows 98. • Jedit (Open Source programmer's text editor) can have EMACS and VI key bindings modes.

Table 31: Context Sensitive Help	
Usability context:	When help in the context of the current task would be useful.
Intent:	Monitor what the user is currently doing, and make documentation available that is relevant to the completion of that task.
Architectural implications:	There needs to be provision in the architecture for a component that tracks what the user is doing at any time and targets a relevant portion of the available help.
Usability properties affected:	+ Guidance: the provision of context sensitive help can give the user guidance.
Examples:	• Microsoft Word includes context sensitive help. Depending on what feature the user is currently using (entering text, manipulating an image, selecting a font style) the Office Assistant will offer different pieces of advice (although some users feel that it is too forceful in its advice). • Depending upon what the cursor is currently pointing to; Word will pop up a small description or explanation of that feature. • Eclipse (a popular Java development environment) allows the user to consult context sensitive info (such as specific API specifications)

4.6 Putting it Together: Relation between SA and Usability

Figure 33 summarizes the different usability attributes, usability properties and architecturally sensitive usability patterns that have been considered in the previous sections of this paper and the (positive) relationships between them.

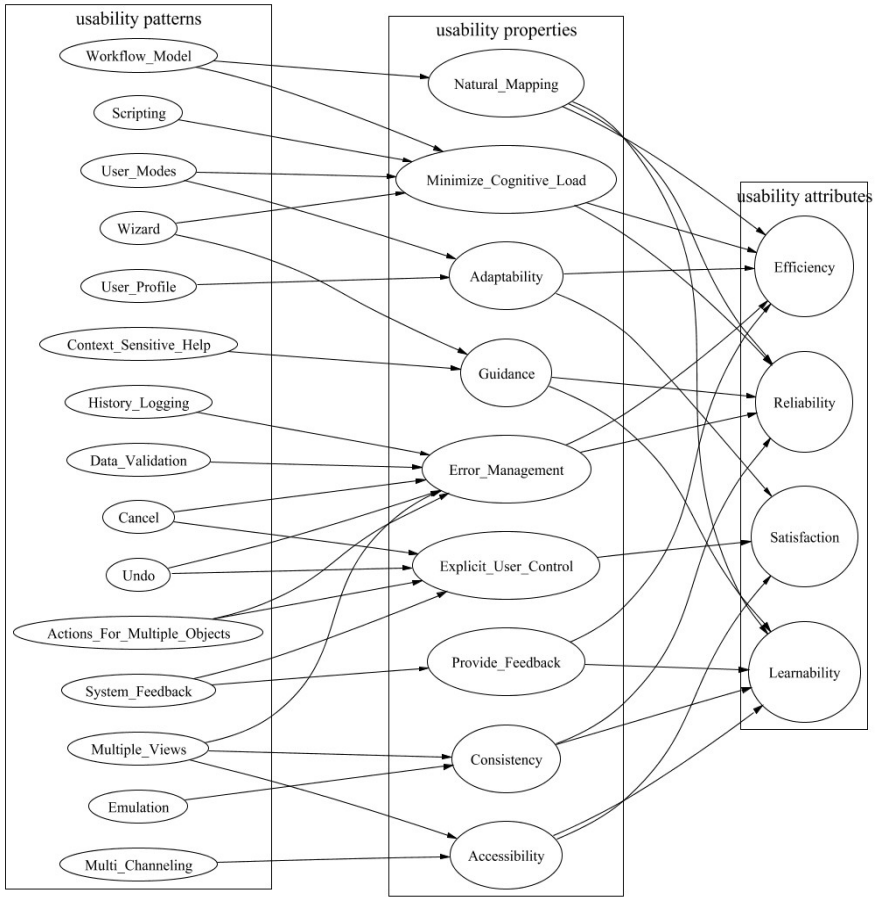

Figure 33: Usability Framework

The usability properties in the framework in this figure may be used as requirements during design. For example, if the requirements specify, "the system must provide feedback", or "minimize cognitive load", we use the framework to identify which usability patterns may be implemented to fulfill these properties by following the arrows in Figure 33. For each architecturally sensitive usability pattern that we identified we linked it to certain usability properties, as discussed in section 0. Usability properties are linked to usability attributes as discussed in section 4.4. The relationships link architecturally sensitive usability patterns to requirements so that when the designer has a usability property as a requirement, they can easily find candidate patterns to meet these requirements.

Consider for example that we want provide guidance to improve learnability. Using the relationships in the framework, we can identify that the patterns "wizard" and "context sensitive help" relate to guidance. During architectural design, we may adjust our

architecture to facilitate such patterns. The choice of which pattern to apply may be made based on cost and trade-off between different usability properties or between usability and other quality attributes such as security or performance. In some systems however, the indicated relationships may not apply. It is up to the software architect to decide how and at which levels these patterns and properties are implemented. Using the framework in this way may guide architectural design and may avoid the retrofit problem we identified in the introduction.

Ongoing work in the STATUS project focuses on developing architectural assessment techniques based on this framework. By evaluating a software architecture for its support of architecturally sensitive usability patterns and/or usability properties, we get an indication of the architectures support of usability. If the architecture does not provide sufficient support for usability (for example, learnability), the framework may be consulted to select usability attribute improving properties and patterns.

4.7 Related Work

Many authors for example: (Constantine and Lockwood, 1999), (Hix and Hartson, 1993), (ISO 9126-1), (Nielsen, 1993), (Preece et al, 1994), (Shackel, 1991), (Shneiderman, 1998), (Wixon & Wilson, 1997) have studied usability. Most of these authors focus on finding and defining the optimal set of attributes that compose usability and on developing guidelines and heuristics for improving and testing usability. Several techniques such as usability testing (Nielsen, 1993), usability inspection (Nielsen, 1994) and usability inquiry (Nielsen, 1993) may be used to evaluate the usability of systems. However, none of these techniques is focused on evaluating software architectures for usability. These authors do not explicitly define a relationship between usability and software architecture or elaborate on how usability requirements may be fulfilled during architectural design.

The layered view on usability presented in (Welie et al, 1999) inspired several elements of the framework model we presented in Section 4.2. For example their usage indicators and usability layer inspired our attribute layer and our architecturally sensitive usability patterns and usability properties are present in their means 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. The terms “usability factors” and “usability criteria” in (Abowd et al, 1992) are similar to our notion of usability attributes".

In our work, the concept of a pattern is used to define an architecturally sensitive usability pattern. 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 problems (e.g. architectural styles (Buschmann et al, 1996) and object oriented frameworks (Coplien and Schmidt, 1995).

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 usability 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 usability of their software, without having to address each problem from scratch. The aim is to capture what was previously very much the "art" of designing usable software and turn it into a repeatable engineering process.

Previous work has been done in the area of usability patterns, by (Tidwell 1998), (Perzel and Kane 1999), (Welie and Trætteberg, 2000). Several usability pattern collections (Brighton, 1998), (Common ground, 1999), (Welie, 2003), (PoInter, 2003) can be found on the web. Most of these usability patterns collections refrain from providing or discussing implementation details. Our paper is not different in that respect because it does not provide specific implementation details. However, we do discuss potential architectural implications. Our work has been influenced by their work, but takes a different standpoint, concentrating on the architectural effect that patterns may have on a system. We consider only patterns that should be applied during the design of a system's software architecture, rather than during the detailed design stage.

(Nigay and Coutaz, 1997) discuss a relationship between usability and software architecture by presenting an architectural model that can help a designer satisfy ergonomic properties. [Trætteberg 2000] presents model fragments that can be used for UI patterns. (Bass et al, 2001) give several examples of architectural patterns that may aid usability. They have identified scenarios that illustrate particular aspects of usability that are in their opinion architecture-sensitive and suggest architectural patterns for implementing these scenarios. Our approach has been influenced by their work. However we have taken a different approach towards investigating the relationship between usability and software architecture because of the following reasons:

- Instead of heuristically evaluating an architecture for a list of usability scenarios we wanted to be able to guide architectural design by investigating how usability requirements can be expressed in a more concrete form. The properties in our framework serve that purpose.

- The number of usability scenarios [Bass et al, 2001] have identified is limited and for some of their scenarios it is debatable whether they are related to usability (Evaluating the system, Verifying resources) or whether they are architecture sensitive (Retrieving forgotten passwords).

This was our motivation to take a top-down approach to investigating the usability - software architecture relationship. We started by a survey from the definition of usability and gradually refined this definition until we defined usability properties and finally related those to architecturally sensitive usability patterns. We focus less on software architecture details, such as possible implementation patterns, but focus more on detailing the relationship those patters have with usability. For defining our framework we used as much as possible usability patterns and design principles that where already defined and accepted in HCI literature and verified the architectural-sensitivity with the industrial case studies we conducted.

We present several new patterns and properties that are not listed in the work of [Bass et al, 2001]. To an extent some of our framework overlaps with their usability scenarios

though we make distinction between the concept of a architecture sensitive pattern and the concept of a usability property. Some of their patterns are defined in our framework as a usability properties, for example the scenarios "working in an unfamiliar context" and "operating consistently across views" are defined in our framework as the usability property "consistency". Some usability scenarios such as "predicting task duration" and "observing system state" are defined in our framework as a pattern "system feedback". We defined architecture sensitive patterns and properties with the highest possible level of abstraction. We believe both approaches are complementary and present different views on a set of "usability improving design solutions" that require architecture support.

A number of papers discuss the relationship between SE and HCI. (Willshire, 2003) and (Milewski, 2003) focus on educational issues. (Walenstein, 2003) discusses what bridges need to be built between the SE and HCI community. (Constantine et al, 2003) and (Ferre, 2003) discuss how HCI engineering should be integrated with software engineering. However, with the exception of (Bass et al, 2001) and (Nigay and Coutaz, 1997), few authors focus on the essential relation with software architecture. The notion of software architecture was already identified in the late sixties. However, it was not until the nineties before architecture design gained the status it has today. Publications such as (Shaw and Garlan, 1996) and (Bass et al, 1998) that discuss definitions, methods and best practices have contributed to a growing awareness of the importance of an explicit software architecture.

4.8 Conclusions

In this article, we present a framework that captures the relationship between usability and software architecture. The framework consists of usability attributes, usability properties, architecturally sensitive usability patterns and the relationships between these elements. The purpose of this framework is the following:

- Express the relationship between usability and software architecture.
- Guide the architecture design process. Existing design experience from the usability and software engineering communities is consolidated in a form that allows us to inform architectural design.

This framework describes an integrated set of 'design solutions', that, in most cases, are considered to have a positive effect on the level of usability but that are difficult to retro-fit into applications because these design solutions may require architectural support. For each of these design solutions we have analyzed the usability effect and the potential architectural implications. The architecturally sensitive usability patterns and properties that we identified have been derived from internal case studies at the industrial partners in the STATUS project and from existing usability pattern collections.

The framework may be used to guide design; however, we do not claim that a particular pattern or property will always improve usability. It is up to the architect to assess whether implementing a property or pattern at the architectural level will improve usability. In addition, the architect will have to balance usability optimizing solutions with other quality attributes such as performance, maintainability or security.

We believe that it is vital that usability issues are taken into account during the architecture design phase to as large an extent as is possible to prevent high costs incurring adaptive maintenance activities once the system has been implemented. This not only holds for usability patterns but also for the usability properties we identified. Usability properties such as consistency need to be considered during architectural design. Future research should focus on how usability properties may be fulfilled during architecture design stage (apart from implementing architecturally sensitive patterns that address them).

This framework has allowed us to develop an architectural assessment technique that may solve some of these problems discussed in the introduction. Using this technique software architects may analyze their architectures for their support of usability without a prototype. Issues such as performance or reliability which affect usability but that are hard to model with a prototype can be analyzed in an architecture, by analyzing the presence of properties such as error management or patterns such as data validation or workflow modeling.

Using the assessment technique, software architectures may become more flexible towards unanticipated usability requirements caused by the limitations of requirements engineering techniques or changes of usability requirements. We have already used this framework at three case studies. These cases studies, which will be published as part of the STATUS deliverables and in a pending article, show that it is possible to use the framework for assessing software architectures for their support of usability. The industrial partners in the STATUS projects are using the results of the assessment and the framework to improve their architectural designs.

Empirical validation is important when offering new techniques. Our framework is a first step in illustrating the relationship between usability and software architecture. The list of architecturally sensitive usability patterns and properties we identified are substantial but incomplete. The relationships depicted in the framework indicate potential relationships. Further work is required to substantiate these relationships and to provide models and assessment procedures for the precise way that the relationships operate.

Future research should focus on verifying our assumptions concerning the architectural sensitiveness of the usability patterns and properties. Proving the architecture sensitivity of a usability pattern is difficult because the patterns we presented may be implemented in different ways, influencing architectural sensitiveness. Practice shows that patterns such as cancel, undo and history logging may be implemented by the command pattern (Gamma et al 1995), emulation and providing multiple views may be implemented by the MVC pattern (Buschmann et al, 1996). Actions for multiple objects may be implemented by the composite pattern (Gamma et al 1995) or the visitor pattern (Gamma et al 1995). Investigating how our usability patterns may be implemented by design patterns or architectural patterns is considered as future work.

In addition to the patterns that we identified, there are some techniques that can be applied to the way that the development team designs and builds the software and that may lead to improvements in usability for the end user. For example, the use of an application framework as a baseline on which to construct applications may be of benefit, promoting consistency in the appearance and behavior of components across a number of applications. For instance, using the Microsoft Foundation Classes when

building a Windows application will provide “common” Windows functionality that will be familiar to users who have previously used other applications build on this library. This is not a pattern that can be applied to the architecture in the same way as those presented in section 4.5, but it is nonetheless something which will be considered during the further study of the relationship between software architecture and usability during the remaining parts of this project.

4.9 Acknowledgements

This work was sponsored by the IST STATUS project. We would like to thank the partners in the STATUS project for their input and their cooperation.