Chapter 9

Experiences with SALUTA

Published as: Architecting for Usability; a Survey, Eelke Folmer, Jan Bosch, Submitted to the Journal of Systems and Software, January 2005. A summary of this journal paper has been accepted as a conference paper entitled "Cost Effective Development of Usable Systems; Gaps between HCI and Software Architecture Design" at Fourteenth International Conference on Information Systems Development - ISD´2005 Karlstad, Sweden, 14-17 August, 2005

Abstract: Studies of software engineering projects show that a significant large part of the maintenance costs of software systems is spent on dealing with usability issues. Fixing usability problems during the later stages of development often proves to be costly as some changes are not easily accommodated by the software architecture. These high costs often prevent developers from meeting all the usability requirements. Explicit evaluation of a software architecture for its support of usability is a tool to cost effectively develop usable systems. Previously few techniques for architecture analysis of usability existed. Based on our investigations into the relationship between usability and software architecture and experiences with architecture analysis of usability, a Scenario based Architecture Level UsabiliTy Analysis technique (SALUTA) was developed. The contribution of this paper is that it provides experiences and problems we encountered when conducting architecture analysis of usability at three industrial case studies performed in the domain of web based enterprise systems. For each experience, a problem description, examples, causes, solutions and research issues are identified.

9.1 Introduction

One of the key problems with most of today's software is that it does not meet its 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 its software architecture (Bosch, 2000), i.e. 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).

The work in this paper is motivated by that this shortcoming also applies to usability. Usability is increasingly recognized as an important consideration during software development; however, many well-known software products suffer from usability problems that cannot be repaired without major changes to the software architecture of these products. Studies (Pressman, 1992, Landauer, 1995) confirm that a significant large part of the maintenance costs of software systems is spent on dealing with usability issues. A reason for these high costs is that most usability issues are only detected during testing and deployment rather than during design and implementation.

This is caused by the following:

- (Usability) requirements are often weakly specified.
- Usability requirements engineering techniques often fail to capture all requirements.
- (Usability) requirements frequently change during development and product evolution.

As a result, a large number of change requests to improve usability are made after these phases. Discovering requirements late is a problem inherent to all software development which cannot be fully avoided. The real problem is that it often proves to be hard and expensive to implement certain changes. Some usability improving solutions such as adding undo, user profiles and visual consistency have for particular application domains proven (Bass et al, 2001, Folmer et al, 2003) to be extremely hard to retrofit during late stage development.

The level of usability is, to a certain extent, restricted by software architecture design. However few software engineers and human computer interaction engineers are aware of this constraint; as a result avoidable costly rework is frequently necessary. During design different tradeoffs need to be made, for example between cost and quality. At a certain point it becomes too expensive to fix certain usability problems.

The software architecture is the first product of the initial design activities that allows analysis and discussion about different concerns. The goal of an architecture analysis method is to understand and reason about the effect of design decisions on the quality of the final system, at a time when it is still cheap to change these decisions. Software architecture analysis of usability is a technique to come up with a software architecture that allows for more “usability tuning” on the detailed design level, hence, preventing part of the high costs incurred by adaptive (Swanson, 1976) maintenance activities once the system has been implemented.

In (Folmer and Bosch, 2004) we provide an overview of usability evaluation techniques. Unfortunately, no assessment techniques exist that explicitly focus on analyzing an architecture's support for usability. Based upon successful experiences (Lassing et al, 2002a) with scenario based assessment of maintainability, we developed a Scenario based Architecture Level UsabiliTy Assessment technique (SALUTA) (Folmer et al, 2004).

A method provides a structure for understanding and reasoning about how a design decision may affect usability but it still requires an experienced engineer to determine how an architecture can support usability. E.g. to assess for usability an analyst should know whether usability improving mechanisms should be implemented during architecture design. In order to make architecture design accessible to inexperienced designers the relevant design knowledge concerning usability and software architecture needs to be captured and described (Folmer and Bosch, 2004). In (Folmer et al, 2003) we investigated the relationship between usability and software architecture. The result of that research is captured in the software-architecture-usability (SAU) framework, which consists of an integrated set of design solutions that in most cases have a positive effect on usability but are difficult to retrofit into applications because they have architectural impact.

In (Folmer et al, 2004) we defined SALUTA which uses the SAU framework to analyze a software architecture for its support of usability. We applied SALUTA at three different case studies in the domain of web based enterprise systems (e.g. e-commerce-, content management- and enterprise resource planning systems). During these case studies several experiences were collected. The contribution of this paper is as follows: it provides experiences and problems that we encountered when conducting architecture analysis of usability. Suggestions are provided for solving or avoiding these problems so organizations that want to conduct architecture analysis facing similar problems may learn from our experiences.

The remainder of this paper is organized as follows. In the next section, the SAU framework that we use for the analysis is presented. The steps of SALUTA are described in section 9.3. Section 9.4 introduces the three cases and the assessment results. Our experiences are described in section 9.5. Finally, related work is discussed in section 9.6 and the paper is concluded in section 9.7.

9.2 The SAU Framework

A software architecture allows for early assessment of quality attributes (Kazman et al, 1998, Bosch, 2000). Specific relationships between software architecture entities (such as - styles, -patterns, -fragments etc) and software quality (maintainability, reliability and efficiency) have been described by several authors (Gamma et al 1995, Buschmann et al, 1996, Bosch, 2000). Until recently (Bass et al, 2001, Folmer et al, 2003) such relationships between usability and software architecture had not been described nor investigated.

In (Folmer et al, 2003) we defined the SAU framework that expresses relationships between Software Architecture and Usability. The SAU framework consists of an integrated set of design solutions that have been identified in various cases in industry, modern day software, and literature surveys (Folmer and Bosch, 2004). These solutions are typically considered to improve usability but are difficult to retro-fit into applications because these solutions require architectural support. The requirement of architectural support has two aspects:

- Retrofit problem: Adding a certain solution has a structural impact. 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 software architecture is already implemented then changing or adding new entities to this structure during late stage design is likely to affect many parts of the existing source code.

- Architectural support: Certain solutions such as providing visual consistency do not necessarily require an extension or restructuring of the architecture. It is possible to implement these otherwise for example by imposing a design rule on the system that requires all screens to be visually consistent (which is a solution that works if you only have a few screens). However this is not the most optimal solution; visual consistency, for example, may be easily facilitated by the use of a separation-of-data-from-presentation mechanism such as XML and XSLT (a style sheet language for transforming XML documents). A template can be defined that is used by all screens when the layout of a screen needs to be modified only the template should be changed. In this case the best solution is

also driven by other qualities such as the need to be able to modify screens (modifiability).

For each of these design solutions we analyzed the effect on usability and the potential architectural implications. The SAU framework consists of the following concepts:

9.2.1 Usability attributes

We needed to be able to measure usability; therefore the first step in investigating the relationship was to decompose usability into usability attributes. A number attributes have been selected from literature that appear to form the most common denominator of existing notions of usability (Shackel, 1991, Hix and Hartson, 1993, Nielsen, 1993, Preece et al, 1994, Wixon and Wilson, 1997, Shneiderman, 1998, Constantine and Lockwood, 1999):

- 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 using the system.
- Reliability in use - the error rate in using the system and the time it takes to recover from errors.
- Satisfaction - the subjective opinions that users form when using the system.

9.2.2 Usability properties

As we needed some way to design for usability, for example by following certain heuristics and design principles that researchers in the usability field have found to have a direct positive influence on usability, a set of usability properties have been identified from literature (Rubinstein and Hersh, 1984, Norman, 1988, Ravden and Johnson, 1989, Polson and Lewis, 1990, Holcomb and Tharp, 1991, Hix and Hartson, 1993, Nielsen, 1993, ISO 9241-11, Shneiderman, 1998, Constantine and Lockwood, 1999). Properties are high-level design primitives that have a known effect on usability and typically have some architectural implications. The usability property consistency is presented in Table 52:

Table 52: Consistency

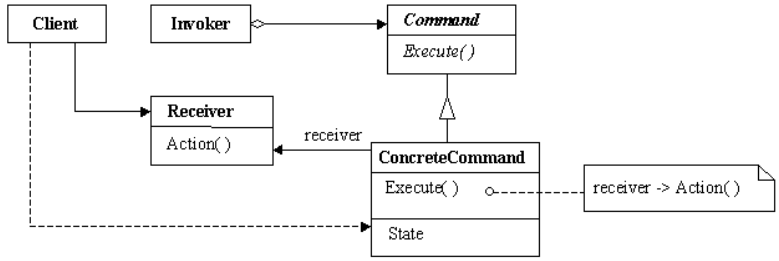
Intent:	Users should not have to wonder whether different words, situations, or actions mean the same thing. An essential design principle is that consistency should be used within applications. Consistency might be provided in different ways: • Visual consistency: user interface elements should be consistent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

9.2.3 Architecture sensitive usability patterns:

In order to be able to design an architecture that supports usability, a number of architecture sensitive usability patterns have been identified that should be applied during the design of a system's software architecture, rather than during the detailed design stage. 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. Our set of patterns has been identified from various cases in industry, modern software, literature surveys (Shackel, 1991, Hix and Hartson, 1993, Nielsen, 1993, Preece et al, 1994, Wixon and Wilson, 1997, Shneiderman, 1998, Constantine and Lockwood, 1999) as well as from existing usability pattern collections (Brighton, 1998, Tidwell 1998, Welie and Trætteberg, 2000, PoInter, 2003).
We defined architecturally sensitive usability patterns with the purpose of capturing design experience in a form that allows us to inform architectural design so we are able to avoid retrofit problems. With our set of patterns, we have concentrated on capturing the architectural considerations that must be taken into account when deciding to implement the pattern. For some patterns however we do provide generic implementation details in terms objects or classes or small application frameworks that are needed for implementing the pattern. An excerpt of the multilevel undo pattern is shown in Table 53:

Table 53: Multi-Level Undo	
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.** Each 'action' the user does is recorded and added to a list. This list then becomes the 'history of user actions' and users can reverse actions from the last done action to the first one recorded.
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
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 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.
Implementation	Most implementations of multi-level undo are based on the Command (Gamma et al 1995) pattern. When using the Command pattern, most functionality is encapsulated in Command objects rather than in other controlling classes. The idea is to have a base class that defines a method to "do" a command, and another method to "undo" a command. Then, for each command, you derive from the command base class and fill in the code for the do and undo methods. The "do" method is expected to store any information needed to "undo" the command. For example, the command to delete an item would remember the content of the item being deleted. The following class diagram shows the basic Command pattern structure:  **Figure 66: Multi Level Undo UML Model** In order to create a multi-level undo, a Command Stack is introduced. When a new command is created, its 'Do' function is called and the object is added to the top of the stack if the command was successful. When undoing commands, the 'Undo' function of the command object at the top of the stack is called and the pointer to the current command is set back.

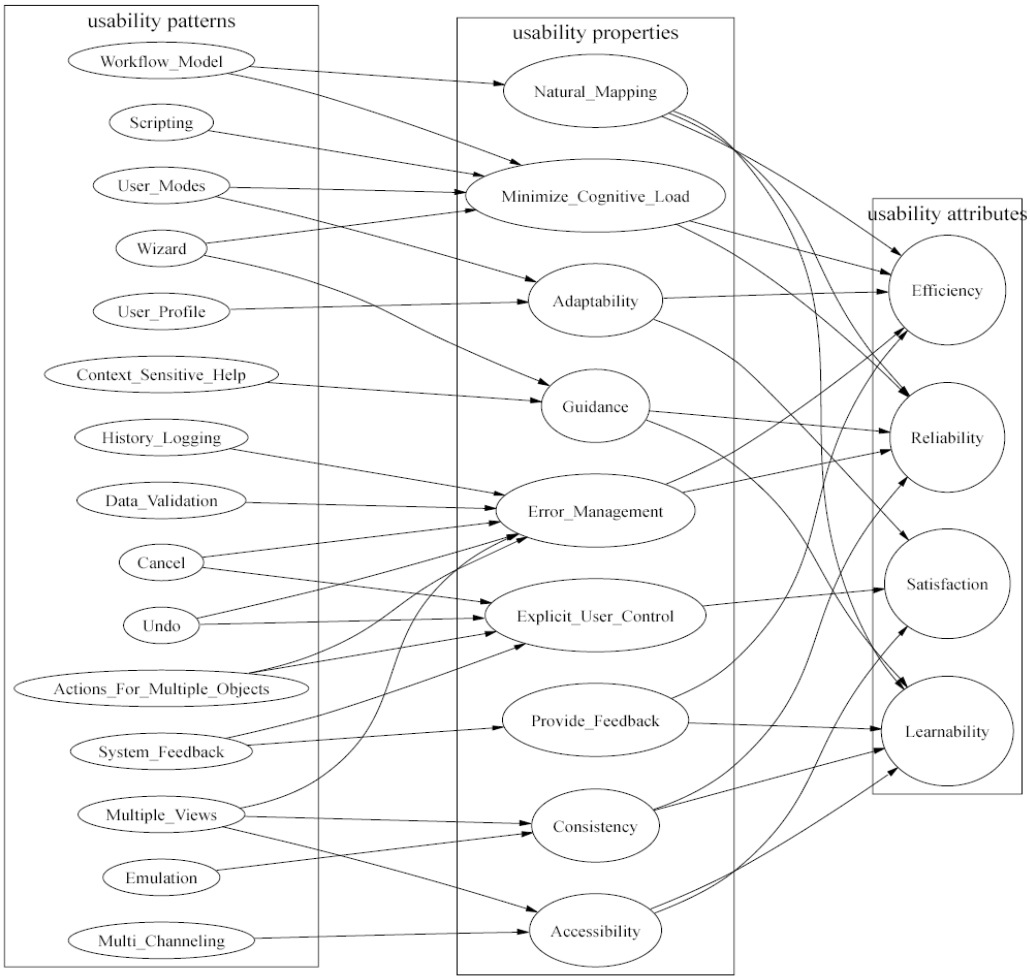


Figure 67: Relationships between Attributes, Properties and Patterns.

9.2.4 Relationships in the SAU framework

Relationships, typically positive, have been defined between the elements of the framework that link architecturally sensitive usability patterns to usability properties and attributes. These relationships have been derived from our literature survey (Folmer et al, 2003), and industrial experiences. Defining relationships between the elements serves two purposes:

- Inform design: The usability properties in the framework may be used as requirements during design. For example, if a requirement specifies, "the system must provide feedback", we use the framework to identify which usability patterns should be considered during architecture design by following the arrows in Figure 67. The choice for which patterns and properties to apply depends on several factors:

  - Not all patterns & properties may improve usability or a are relevant for a particular system. It is up to a usability expert or engineer to decide whether a particular pattern or property may be applied to architecture of the system under analysis.

  - Cost and trade-off between different usability attributes or between usability and other quality attributes such as security or performance.

- Software architecture analysis: Our framework tries to capture essential design solutions so these can be taken into account during architectural design and evaluation. The relationships are then used to identify how particular patterns and properties, that have been implemented in the architecture, support usability. For example, if undo has been implemented we can analyze that undo improves efficiency and reliability.

SALUTA uses the SAU framework for analyzing the architecture's support for usability. A complete overview and description of all patterns and properties and the relationships between them can be found in (Folmer et al, 2003).

9.3 Overview of SALUTA

In (Folmer et al, 2004) the SALUTA method is presented. A method ensures that some form of reasoning and discussion between different stakeholders about the architecture is taking place. SALUTA is scenario based i.e. in order to assess a particular architecture, a set of scenarios is developed that concretizes the actual meaning of that quality requirement (Bosch, 2000). Although there are other types of architecture assessment techniques such as metrics, simulations and mathematical models (Bosch, 2000) in our industrial and academic experience with scenario based analysis we have come to understanding that the use of scenarios allows us to make a very concrete and detailed analysis and statements about their impact or support they require, even for quality attributes that are hard to predict and assess from a forward engineering perspective such as maintainability, security and modifiability.

SALUTA has been derived from scenario based assessment techniques such as ALMA (Bengtsson, 2002), SAAM (Kazman et al, 1994), ATAM (Kazman et al, 1998) and QASAR(Bosch, 2000). Although it's possible to use a generic scenario based assessment technique such as ATAM or QASAR a specialized technique (such as ALMA) is more tailored to a specific quality and will lead to more accurate assessment results. For example guidelines and criteria are given for creating specific scenarios. To assess the architecture a set of *usage scenarios* are defined. By analyzing the architecture for its support of each of these usage scenarios we determine the architecture's support for usability. SALUTA consists of the following four steps:

1. Create usage profile; describe required usability.

2. Analyze the software architecture: describe provided usability.

3. Scenario evaluation: determine the architecture's support for the usage scenarios.

4. Interpret the results: draw conclusions from the analysis results.

A brief overview of the steps is given in the next subsections, a more detailed elaboration of and motivation for these steps can be found in (Folmer et al, 2004).

9.3.1 Usage profile creation

One of the most important steps in SALUTA is the creation of a usage profile. Existing usability specification techniques (Hix and Hartson, 1993, Nielsen, 1993, Preece et al, 1994) are poorly suited for architectural assessment, therefore a scenario profile (Lassing et al, 2002a, Bengtsson, 2002) based approach was chosen. The aim of this step is to come up with a set of usage scenarios that accurately expresses the required usability of the system. Usability is not an intrinsic quality of the system. According to the ISO definition (ISO 9241-11), usability depends on:

- The users (e.g. system administrators, novice users)
- The tasks (e.g. insert order, search for item X)
- The contexts of use (e.g. helpdesk, training environment)

Usability may also depend on other variables, such as goals of use, etc. However in a usage scenario only the variables stated above are included. A usage scenario describes a particular interaction (task) of a user with the system in a particular context. A usage scenario specified in such a way does not yet specify anything about the required usability of the system. In order to do that, the usage scenario is related to the four usability attributes defined in the SAU-framework. For each usage scenario, numeric values are determined for each of these usability attributes. The numeric values are used to determine a prioritization between the usability attributes. For some usability attributes, such as efficiency and learnability, tradeoffs have to be made during design. It is often impossible to design a system that has high scores on all attributes. A purpose of usability requirements is therefore to specify a necessary level for each attribute (Lauesen and Younessi, 1998). For example, if for a particular usage scenario learnability is considered to be of more importance than other attributes (for example, because of a requirement), then the usage scenario must reflect this difference in the priorities for the usability attributes. The analyst interprets the priority values during the analysis phase to determine the level of support in the software architecture for that particular usage scenario. An example usage scenario is displayed in Figure 68.

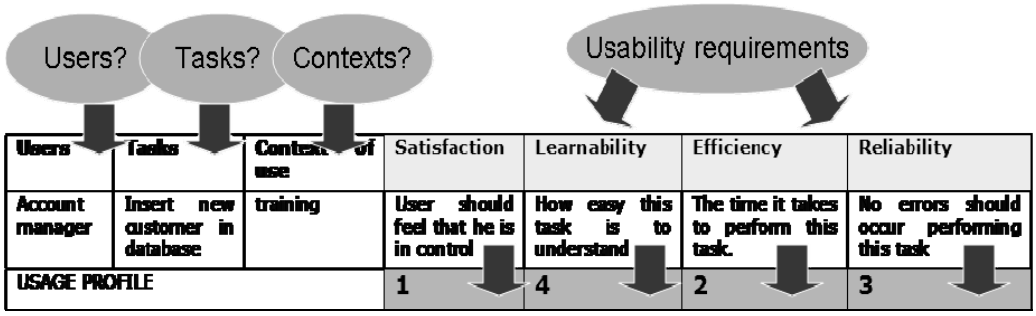


Figure 68: Example Usage Scenario

Usage profile creation does not replace existing requirements engineering techniques. Rather it is intended to transform (existing) usability requirements into something that

can be used for architecture assessment. Existing techniques such as interviews, group discussions or observations (Hix and Hartson, 1993, Nielsen, 1993, Hackos and Redish, 1998, Shneiderman, 1998) typically already provide information such as representative tasks, users and contexts of use that are needed to create a usage profile. The steps that need to be taken for usage profile creation are the following:

1. Identify the users: rather than listing individual users, users that are representative for the use of the system should be categorized in types or groups (for example system administrators, end-users etc).

2. Identify the tasks: Instead of converting the complete functionality of the system into tasks, representative tasks are selected that highlight the important features of the system. An accurate description of what is understood for a particular task and of which subtasks this task is composed, is an essential part of this step. For example, a task may be "search for specific compressor model" consisting of subtasks "go to performance part" and "select specific compressor model".

3. Identify the contexts of use: In this step, representative contexts of use are identified. (For example, helpdesk context or disability context.)

4. Determine attribute values: For each valid combination of user, task and context of use, usability attributes are quantified to express the required usability of the system, based on the usability requirements specification. Defining specific indicators for attributes may assist the analyst in interpreting usability requirements. To reflect the difference in priority, numeric values between one and four have been assigned to the attributes for each scenario.

5. Scenario selection and weighing: Evaluating all identified scenarios may be a costly and time-consuming process. Therefore, the goal of performing an assessment is not to evaluate all scenarios but only a representative subset. Different profiles may be defined depending on the goal of the analysis. For example, if the goal is to compare two different architectures, scenarios may be selected that highlight the differences between those architectures. To express differences between scenarios in the profile, properties may be assigned to the scenarios, for example: priority or probability of use within a certain time. The result of the assessment may be influenced by weighing scenarios, if some scenarios are more important than others, weighing these scenarios reflect these differences.

This step results in a set of usage scenarios that accurately express the required usability of the system.

9.3.2 Analyze the software architecture

In the second step of SALUTA, the information about the software architecture is collected. Usability analysis requires architectural information that allows the analyst to determine the support for the usage scenarios. The process of identifying the support is similar to scenario impact analysis for maintainability assessment (Lassing et al, 2002a) but is different, because it focuses on identifying architectural elements that may support the scenario. For architecture analysis, the SAU framework in section 9.2 is used. Two types of analysis are performed:

- Analyze the support for patterns: Using the list of architecturally sensitive usability patterns we identify whether these are implemented in the architecture.

- Analyze the support for properties: The software architecture is the result of a series of design decisions (Gurp and Bosch, 2002). Reconstructing this process and assessing the effect of individual design decisions with regard to usability provides additional information about the intended quality of the system. Using the list of usability properties, the architecture and the design decisions that lead to this architecture are analyzed for these properties.

The quality of the assessment very much depends on the amount of evidence for patterns and property support that can be extracted from the architecture. SALUTA does not dictate the use of any specific way of documenting a software architecture. Initially the analysis is based on the information that is available, such as architecture designs and documentation used with in the development team for example Figure 69 lists a conceptual view (Hofmeister et al, 1999) that was used to identify the presence of patterns in the Compressor case (see section 9.4.2 for a description of Compressor).

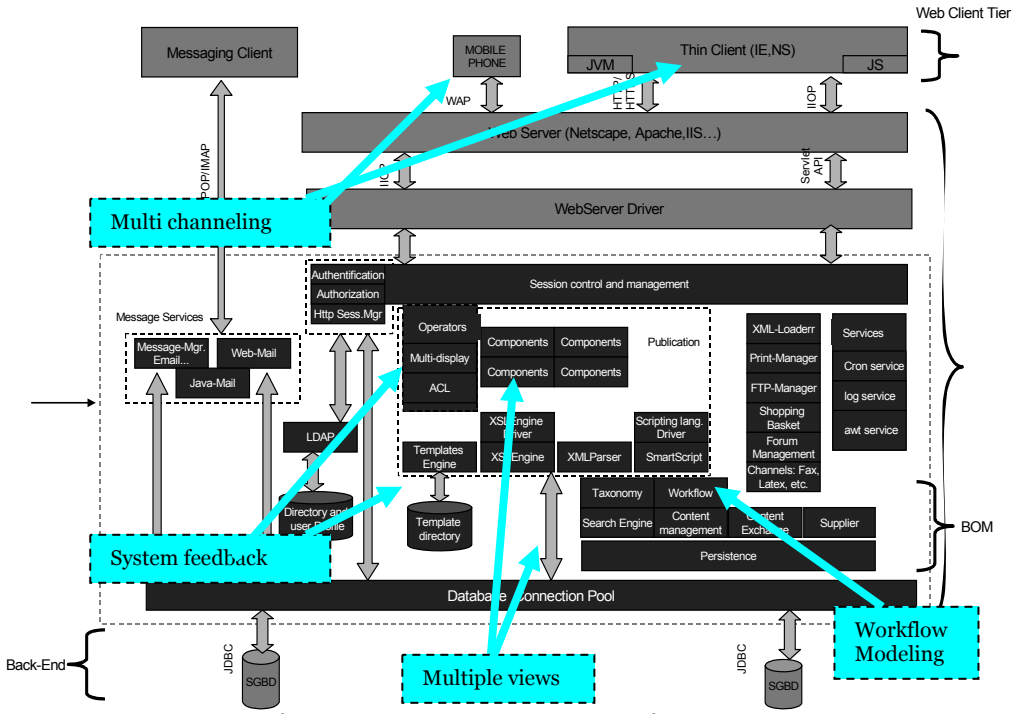


Figure 69: Compressor Architecture

9.3.3 Scenario evaluation

The next step is to evaluate the architecture's support for each of the scenarios in the usage profile. For each scenario, we identify by which usability patterns and properties that are implemented, it is affected. In the next step we identify using the SAU framework how a particular pattern or property improves or impairs certain usability

attributes for that scenario. For example, if it has been identified that undo affects a certain scenario, the relationship between undo and usability are analyzed to determine the support for that particular scenario. Undo improves error management and error management may improve reliability and efficiency. This step is repeated for each pattern and property affecting that scenario. The number and type of patterns and properties that support a particular attribute of a scenario are then compared to the required attribute values to determine the support for this scenario.

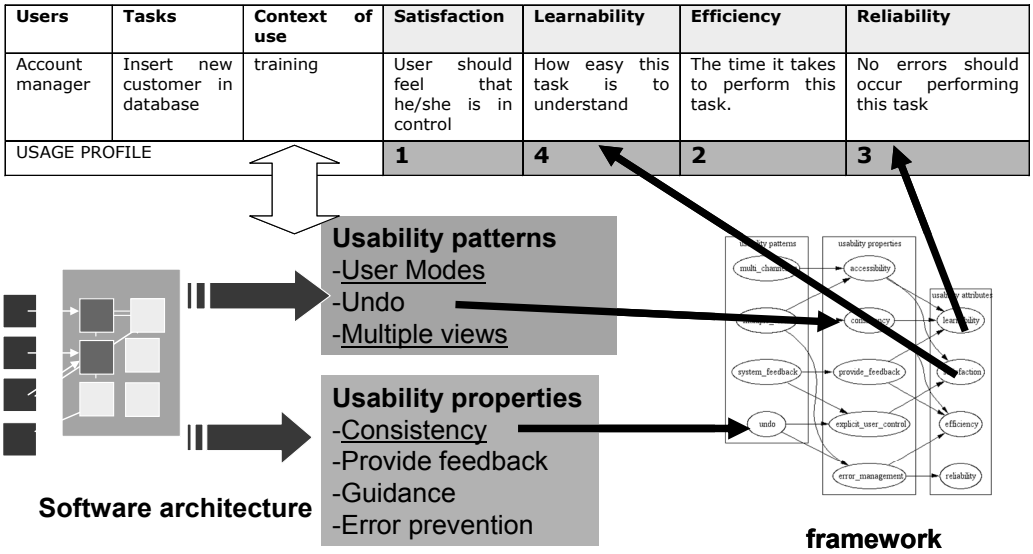


Figure 70: Snapshot Assessment Example

For each scenario, the results of the support analysis are expressed qualitatively using quantitative measures. For example, the support may be expressed on a five level scale (++, +, +/-,-,--). The outcome of the overall analysis may be a simple binary answer (supported/unsupported) or a more elaborate answer (70% supported) depending on how much information is available and how much effort is being put in creating the usage profile.

9.3.4 Interpretation of the results

After scenario evaluation, the results need to be interpreted to draw conclusions concerning the software architecture. If the analysis is sufficiently accurate the results may be quantified. However, even without quantification the assessment can produce useful results. If the goal is to iteratively design an architecture, then if the architecture proves to have sufficient support for usability, the design process may be finalized. Otherwise, architecture transformations need to be applied to improve the support for usability. For example in the eSuite case (see section 9.4.3) the architecture's support for usability was improved by adding three patterns to it. Qualitative information such as which scenarios are poorly supported and which usability properties or patterns have not been considered may guide the architect in applying certain design solutions. An architect should always discuss with a usability engineer which solutions need to be applied. The SAU framework is then used as an informative source for design and improvement of the architecture's support for usability.

9.4 Case Descriptions

In this section we introduce the three systems used in the case studies. The goal of the case studies was to conduct a software architecture analysis of usability on each of the three systems.

As a research strategy action research (Argyris et al, 1985), was used. Action research is an applied research strategy which involves cycles of data collection, evaluation and reflection with the aim of *improving* the quality or performance of an organization. Close cooperation and participation which are important aspects of this type of research allowed us to get a more complete understanding of the research issues. The first case study (Folmer et al, 2004) was performed at a software organization which is part of our university. This provided us with valuable insights and made us revise some of the SALUTA steps. The other two case studies were performed at our industrial partners in the STATUS project. Between those cases again our method was revised and refined. The last two case studies been published as part of the STATUS deliverables (STATUS).

All case studies have been performed in the domain of *web based* enterprise systems, e.g. content management- (CMS), e-commerce- and enterprise resource planning (ERP) – systems. Web based systems have become an increasingly popular application format in recent years. Web based systems have two main advantages: Centralization: the applications run on a (central / distributed) web server, there is no need to install or maintain the application locally. Accessibility: The connectivity of the web allows anyone to access the application from any internet connection on the world and from any device that supports a web browser. From a usability point of view this is a very interesting domain: anyone with an internet connection is a potential user. A lot of different types of users and different kinds of usages must therefore be supported. An overview of the differences between the applications (See Table 54) illustrates the scope of applicability of our method.

Table 54: Comparison of System Characteristics

Aspect	Webplatform	Compressor	eSuite
Type of system	CMS	E-commerce	ERP
Number of users	> 20.000	> 100	> 1000
Goal of the analysis	Analyze architecture's support for usability / Risk assessment: analyze SA related usability issues.	Selection: Compare old versus new version of Compressor.	Design: iteratively design & improve an architecture.
types of users	3	3	2
Characterization of interaction	Information browsing and manipulation of data objects (e.g. create portals, course descriptions)	Information browsing (e.g.) Comparing and analyzing data of different types of compressors and compressor parts.	Typical ERP functionality. (e.g. insert order, get client balance sheet)
Usage contexts	Mobile / desktop/ Helpdesk	Mobile/Desktop/Standalone	Mobile/Desktop

The remainder of this section introduces the three systems that have been analyzed and presents the assessment results.

9.4.1 Webplatform

The Webplatform is a web based content management system for the university of Groningen (RuG) developed by ECCOO (Expertise Centrum Computer Ondersteunend Onderwijs). The Webplatform enables a variety of (centralized) technical and (de-centralized) non technical staff to create, edit, manage and publish a variety of content (such as text, graphics, video etc), whilst being constrained by a centralized set of rules, process and workflows that ensure a coherent, validated website appearance.

The Webplatform data structure is object based; all data from the definitions of the CMS itself to the data of the faculty portals or the personal details of a user are objects. The CMS makes use of the internet file system (IFS) to provide an interface which realises the use of objects and relations as defined in XML. The IFS uses an Oracle 9i database server implementation with a java based front end as search and storage medium. The java based front-end allows for the translation of an object oriented data structure into HTML. The oracle 9i database is a relational based database. On top of the IFS interface, the Webplatform application has been build. Thus, the CMS consists of the functionality provided by the IFS and the java based front-end. Integrated into the Webplatform is a customised tool called Xopus, which enables a content-administrator to create, edit and delete XML objects through a web browser.

As an input to the analysis of the Webplatform, we interviewed the software architect, the usability engineer and several other individuals involved in the development of the system. In addition we examined the design documentation and experimented with the newly deployed RuG site.

9.4.2 Compressor

The Compressor catalogue application is a product developed by the Imperial Highway Group (IHG) for a client in the refrigeration industry. It is an e-commerce application, which makes it possible for potential customers to search for detailed technical information about a range of compressors; for example, comparing two compressors.

There was an existing implementation as a Visual Basic application, but the application has been redeveloped in the form of a web application. The system employs a 3-tiered architecture and is built upon an in-house developed application framework. The application is being designed to be able to work with several different web servers or without any. The independence of the database is developed through Java Database Connectivity (JDBC). The data sources (either input or output) can also be XML files. The application server has a modular structure, it is composed by a messaging system and the rest of the system is based on several connectable modules (services) that communicate between them. This potential structure offers a pool of connections for those applications that are running, providing more efficiency on the access to databases.

As an input to the analysis of Compressor, we interviewed the software architect. We analyzed the results from usability tests with the old system and with an interface prototype of the new system and examined the design documentation such as architectural designs and requirements specifications.

9.4.3 ESuite

The eSuite product developed by LogicDIS is a system that allows access to various ERP (Enterprise Resource Planning) systems, through a web interface. ERP systems generally run on large mainframe computers and only provide users with a terminal interface. eSuite is built as an web interface on top of different ERP systems. Users can access the system from a desktop computer but also from a mobile phone. The system employs a tiered architecture commonly found in web applications. The user interfaces with the system through a web browser. A web server runs a Java servlet and some business logic components, which communicate with the ERP.

As an input to the analysis of ESuite, we interviewed the software architect and several other individuals involved in the development of the system. We analyzed the results from usability tests with the old system and with an interface prototype of the new system and examined the design documentation such as architectural designs and, usability requirements specifications.

9.4.4 Assessment results

Table 55: Assessment Results

	No of scenarios	Strong reject	Weak reject	Accept/reject	Weak accept	Strong accept
Webplatform	11	-	-	-	8	3
Old Compressor	14	2	2	8	-	-
New Compressor	14	-	-	5	6	3
eSuite	12	-	-	3	4	3

Table 55 lists the results of the assessment. The table lists the number of scenario defined and lists whether these scenarios are strongly rejected, weakly rejected, accepted/rejected, weakly accepted or strongly accepted. Our impression was that overall the assessment was well received by the architects that assisted the analysis. Based on the assessment results the Esuite architecture was improved[1] by applying patterns from the SAU framework. In the other cases we were not involved during architecture design but the assessments provided the architects with valuable insights till which extent certain usability improving design solutions could still be implemented during late stage without incurring great costs. This emphasized and increased the understanding of the important relationship between software architecture and usability and the results of the assessments were documented and taken into account for future releases and redevelopment of the products.

9.5 Experiences

This section gives a detailed description of the experiences we acquired during the definition and use of SALUTA. We consider SALUTA to be a prototypical example of an

[1] The decision to apply certain patterns was not solely based on the result of the assessment but also as result of user tests with prototypes were these patterns were present.

architecture assessment technique therefore our experiences are relevant in a wider context. These experiences will be presented using the four steps of the method. For each experience, a problem description, examples, possible causes, available solutions and research issues are provided. The experiences are illustrated using examples from the three case studies introduced before.

9.5.1 Usage profile creation

In addition to experiences that are well recognized in the domain of SE and HCI such as:

- **Poorly specified usability requirements** e.g. In all cases, apart from the web platform case (some general usability guidelines based on Nielsen's heuristics (Nielsen, 1993) had been stated in the functional requirements) no clearly defined and verifiable usability requirements had been collected or specified.
- **Changing requirements** e.g. in all case studies we noticed that during development the usability requirements had changed. For example, in the Webplatform case it had initially been specified that the Webplatform should always provide context sensitive help texts, however for more experienced users this turned out to be annoying and led to a usability problem. A system where help texts could be turned off for more experienced users would be much better.

The following experiences were collected:

Difficult to transform requirements

Problem: To be able to assess a software architecture for its support of usability we need to transform requirements into a suitable format. For SALUTA we have chosen to use usage scenarios. For each scenario, usability attributes are quantified to express the required usability of the system, based on the requirements specification. A problem that we encountered is that sometimes it is difficult to determine attribute values for a scenario because usability requirements and attributes can be interpreted in different ways.

Example: What do efficiency or learnability attributes mean for a particular task, user or user context? Efficiency can be interpreted in different ways: does it mean the time that it takes to perform a task or does it mean the number of errors that a user makes? It can also mean both. Usability requirements are sometimes also difficult to interpret for example in the Webplatform case: "*UR1: every page should feature a quick search which searches the whole portal and comes up with accurate search results*". How can we translate such a requirement to attribute values for a scenario?

Causes: Translating requirements to a format that is suitable for architecture assessment is an activity that takes place on the boundary of both SE and HCI disciplines. Expertise is required; it is difficult to do for a software architect since he or she may have no experience with usability requirements.

Solution: In all of our cases we have let a usability engineer translate usability requirements to attribute values for scenarios. To formalize this step we have let the usability engineer specify for each scenario how to *interpret* a particular attribute. For example, for the web platform case the following usage scenario has been defined: "*end user performing quick search*". The usability engineer formally specified what should be understood for each attribute of this task. Reliability has been associated with the

accuracy of search results; efficiency has been associated with response time of the quick search, learnability with the time it takes to understand and use this function. Then the usability requirements (UR1) were consulted. From this requirement we understand that reliability (e.g. accuracy of search results is important). In the requirements however it has not been specified that quick search should be performed quickly or that this function should be easy to understand. Because most usability requirements are not formally specified we discussed these issues with the usability engineer that assisted the analysis and the engineer found that this is the most important aspect of usability for this task. Consequently, high values have been given to efficiency and reliability and low values to the other attributes (see Figure 71) Defining and discussing specific indicators for attributes (such as number or errors for reliability) may assist the interpretation of usability requirements and may lead to a more accurate prioritization of usability attributes.

Research issues: The weakness in this process is that is inevitably some guesswork involved on the part of the experts and that one must be careful not to add too much value to the numerical scores. E.g. if learnability has value 4 and efficiency value 2 it does not necessarily mean that learnability is twice as important as efficiency. The only reason for using numerical scores is to reflect the difference in priority which is used for analyzing the architecture support for that scenario. To improve the representativeness of a usage scenario possibly a more fine grained definition of a scenario needs to be developed.

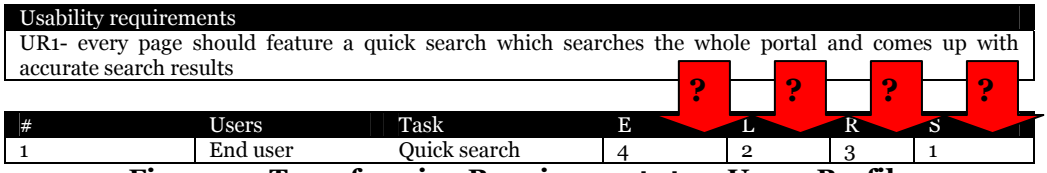


Figure 71: Transforming Requirements to a Usage Profile

Specification of certain quality attributes is difficult during initial design

Problem: A purpose of quality requirements is to specify a *necessary* level (Lauesen and Younessi, 1998). In section 9.2 four different usability attributes have been presented which we use for expressing the required usability for a system in a usage scenario. Specifying a necessary level for satisfaction and specifying how satisfaction should be interpreted has proven to be difficult during initial design. In addition we could not identify specific usability requirements that specify a necessary level for this attribute during initial design.

Example: In the compressor case we defined the following usage scenario: "S*uppliers get the performance data for a specific model*". What does satisfaction mean for this scenario? What is the necessary level of the satisfaction for this scenario? Attributes such as learnability, efficiency and reliability are much easier interpreted and it is therefore much easier to specify a necessary level for them.

Cause: Satisfaction to a great extent depends on, or is influenced by the other three usability attributes (efficiency, reliability and learnability) it expresses the subjective opinions users have in using the system, therefore satisfaction can often only be measured when the system is deployed (for example, by interviewing users).

Solution: The importance of satisfaction in this context should be reevaluated.

Research issues: Satisfaction has been included in our usability decomposition because it expresses the subjective view a user has on the system. We are uncertain if this subjective view is not already reflected by the definition of usability. Which software systems are not usable but have high values for their satisfaction attributes?

Cost benefit tradeoffs

Problem: The number of usage scenarios in the usage profile easily becomes a large number. Evaluating and quantifying all scenarios may be a costly and time-consuming process. How do we keep the assessment at a reasonable size?

Example: For example for the web platform case we initially had identified 68 scenarios. For the Compressor case we identified 58 different usage scenarios.

Cause: The number of scenarios that are identified during the usage profile creation stage can become quite large since many variables are included; users, user contexts and tasks.

Solutions: Inevitably tradeoffs have to be made during usage scenario selection, an important consideration is that the more scenarios are evaluated the more accurate the outcome of the assessment is, but the more expensive and time consuming it is to determine attribute values for these scenarios. We propose three solutions:

- Explicit goal setting: allows the analyst to filter out those scenarios that do not contribute to the goal of the analysis. Goal setting is important since it can influence which scenarios to include in the profile. For example for the Web platform case we decided, based on the goal of the analysis (analyze architecture's support for usability), to only to select those scenarios that were important to a particular user group; a group of content administrators that only constituted 5% of the users population but the success of Webplatform was largely dependent on their acceptance of the system. This reduced the number of scenarios down to a reasonable size of 11 usage scenarios.

- Pair wise comparison: For most usage scenarios, concerning expressing the required usability there is an obvious conflict between attributes such as efficiency and learnability or reliability and efficiency. To minimize the number of attributes that need to be quantified techniques such as pair wise comparison should be considered to only determine attribute values for the attributes that conflict.

- Tool support: It is possible to specify attribute values over a particular task or context of use or for a user group. For example for the user type "expert users" it may be specified that efficiency is the most important attribute for all scenarios that involve expert users. For a particular complex task it may be specified that learnability should be the most important attribute for all scenarios that have included that task. We consider developing tool support in the future which should assist the analyst in specifying attribute values over contexts, users and tasks and that will automatically determine a final prioritization of attribute values for a usage profile.

9.5.2 Architecture analysis

Non-explicit nature of architecture design

Problem: In order to analyze the architecture support for usability, some representation of the software architecture is needed. However, the software architecture has several aspects (such as design decisions and their rationale) that are not easily captured or expressed in a single model or view.

Example: Initially the analysis is based on the information that is available. In the Compressor case a conceptual architecture description had been created (see Figure 69 in section 9.3). However to determine the architectural support for usability we needed more information, such as which design decisions were taken.

Cause: Because of the non-explicit nature of architecture design, the analysis strongly depends on having access to both design documentation and software architects; as design decisions are often not documented the architect may fill in the missing information on the architecture and design decisions that were taken.

Solution: Interviewing the architect provided us with a list *if* particular patterns and properties had been implemented. We then got into more detail by analyzing the architecture designs and documentation for evidence of *how* these patterns and properties had been implemented. Different views on the system (Kruchten, 1995, Hofmeister et al, 1999) may be needed to access such information. A conceptual view (Hofmeister et al, 1999) on the system of the Compressor (see Figure 69) was sufficient for us to provide detailed information on how the patterns (Folmer et al, 2003) system feedback, multi channeling, multiple views and workflow modeling had been implemented. For the other systems that lacked architecture descriptions we let the software architects create conceptual views.

Validation of the SAU framework

Problem: Empirical validation is important when offering new techniques. The analysis technique for determining the provided usability of the system relies on the framework we developed. Initially the SAU framework was based on discussions with our partners in the STATUS project and did not focus on any particular application domain. The list of patterns and properties that we had identified then was substantial but incomplete. Even the relation of some of the patterns and properties with software architecture was open to dispute. For particular application domains the framework may not be accurate.

Example: Our case studies have been performed in the domain of web based systems. Initially our SAU framework contained usability patterns such as multitasking and shortcuts. For these patterns we could not find evidence that they were architecturally sensitive in this domain. Other patterns such as undo and cancel have different meanings in web based interaction. Pressing the stop button in a browser does not really cancel anything. Undo is generally associated with the back button. Web based systems are different from other types of applications.

Causes: the architecture sensitivity of some of our usability patterns depends on its implementation which depends on the application domain.

Solution: The applicability of our analysis method is not excluded to other application domains but the framework that we use for the analysis may need to be specialized for different application domains in the future. As discussed in section 9.2 the "best implementation" of a particular pattern may depend on several other factors such as which application framework is used or on other qualities such as maintainability and flexiblity. Some patterns do not exist or are not relevant for a particular domain. Some patterns may share similar implementations across different domains these patterns can be described in a generic fashion.

Research issue: 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, it does not yet provide a complete comprehensive coverage of all potential architecturally sensitive usability issues for all domains. The case studies have allowed us to refine and extend the framework for the domain of web based enterprise systems, and allowed us to provide detailed architectural solutions for implementing these patterns and properties (based on "best" practices).

Qualitative nature of SAU framework

Problem: 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 usability in order to determine whether requirements have been met. Being able to quantify these relationships and being able to express negative relationships would greatly enhance the use of our framework.

Example: The pattern wizard generally improves learnability but it negatively affects efficiency. Until now it is not known how much a particular pattern or property improves or impairs a particular attribute of usability e.g. we only get a qualitative indication.

Causes: Our framework is a first step in illustrating a relationship between usability and software architecture. Literature does not provide us with quantitative data on how these patterns may improve usability.

Solution: 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. Eventually 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.

9.5.3 Scenario evaluation

Evaluation is guided by tacit knowledge

Problem: The activity of scenario evaluation is concerned with determining the support the architecture provides for that particular usage scenario. The number of patterns and properties that support a particular usability attribute required by a scenario, for example learnability, provide an indication of the architecture's support for that scenario however the evaluation is often guided by tacit knowledge.

Example: For example in the eSuite case the following scenario was affected by four usability patterns and six usability properties. The scenario requires high values for learnability (4) and reliability (3). Several patterns and properties positively contribute to the support of this scenario. For example, the property consistency and the pattern context sensitive help increases learnability as can be analyzed from Figure 67. By analyzing for each pattern and property, the effect on usability, the support for this scenario was determined. However sometimes this has proven to be difficult. How much learnability improving patterns and properties should the architecture provide for deciding whether this scenario is supported?

Table 56: ESuite Usage Scenario						
User	**User context**	**Task**	**S**	**E**	**L**	**R**
Novice	Mobile	Insert Order	1	2	4	3

Cause: Although SALUTA provides the steps for identifying the support determining whether a scenario is accepted or rejected is still is very much guided by tacit knowledge i.e. the undocumented knowledge of experienced software architects.

Solution: Our framework has captured some of that knowledge (e.g. the relationships between usability properties and patterns and usability attributes) but it is up to the analyst to interpret these relationships and determine the support for the scenarios.

Research issues: Since evaluating all the scenarios by hand is time consuming, we consider developing a tool that allows one to automatically determine for a set of identified patterns and properties which attributes they support and to come up with some quantitative indication for the support. Although it may not be possible to give an absolute indication of an architectures support for usability, when iteratively designing and evaluating an architecture we are able to express relative improvements.

9.5.4 Interpretation

Lacked a frame of reference

Problem: After scenario evaluation we have to associate conclusions with these results. However initially we lacked a frame of reference to interpret the results.

Example: In our first case study (Webplatform) the result of the evaluation was that three scenarios were weakly accepted, and eight were strongly accepted. How should this be interpreted and which actions need to be taken?

Cause: Interpretation is concerned with deciding whether the outcome of the assessment is acceptable or not. The experiences that we have, is at initially we lacked a frame of reference for interpreting the results of the evaluation of Webplatform. Were these numbers acceptable? Could we design an architecture that has a better support for usability? The results of the assessment were relative, but we had no means or techniques to relate it to other numbers or results yet. Another issue was that we doubted the representativeness of the usage profile. Did this profile cover all possible usages by all types of users?

Solution: The three case studies have provided us with a small frame of reference. We have seen architectures with significant better and significantly weaker support for usability. This provided us with enough information to judge whether a particular

architecture could still be improved. In order to refine our frame of reference more case studies need to be done within the domain of web based application. Certain patterns such as multiple views were present in all architectures we examined, whereas other patterns such as user modes were only present in one system. We need more info on which patterns are already integrated in application frameworks such as STRUTS (Mercay and Gilbert, 2002) and which patterns have not.

In addition to the architecture assessment related experiences the following general experiences were collected.

9.5.5 General experiences

Some general experiences that are well recognized in the SE and HCI domains which are of cultural and psychological nature have been identified such as:

- Lack of integration of SE and HCI processes e.g. Processes for software engineering and HCI are not fully integrated. There is no integration of SE and HCI techniques during architectural design. Because interface design is often postponed to the later stages of development we run the risk that many assumptions may be built into the design of the architecture that unknowingly may affect interface design and vice versa. The software architecture is seen as an intermediate product in the development process but its potential with respect to quality assessment is not fully exploited.

- Technology driven design: Software architects fail to associate usability with software architecture design e.g. the software architects we interviewed in the case studies were not aware of the important role the software architecture plays in fulfilling and restricting usability requirements. When designing their systems the software architects had already selected technologies (read features) and had already developed a first version of the system before they decided to include the user in the loop. A software product is often seen as a set of features rather then a set of "user experiences".

In addition to these experiences the following experiences were collected:

Impact of software architecture design on usability

Problem: One of the reasons to develop SALUTA was that usability may unknowingly impact software architecture design e.g. the retrofit problem discussed in section 9.2. However, we also identified that it worked the other way around; architecture design sometimes leads to usability problems in the interface and the interaction.

Example: In the ECCOO case study we identified that the layout of a page (users had to fill in a form) was determined by the XML definition of a specific object. When users had to insert data, the order in which particular fields had to be filled in turned out to be very confusing.

Causes: Because interface design is often postponed until the later stages of design we run the risk that many assumptions are built into the design of the architecture that unknowingly affect interface/interaction design and vice versa.

Solution: Interfaces/interaction should not be designed as last but as early as possible to identify what should be supported by the software architecture and how the architecture may affect interface/interaction design. We should not only analyze whether the architecture design supports certain usability solutions but also identify how the architecture design may lead to usability problems.

Research issues: Usability is determined by many factors, issues such as:

- Information architecture: how is information presented to the user?
- Interaction architecture: how is functionality presented to the user?
- System quality attributes: such as efficiency and reliability.

Architecture design does affect all these issues. Considerable more research needs to be performed to analyze how a particular architecture design may lead to such kind of usability problems.

Accuracy of the analysis is unclear

Problem: Our cases studies show that it is possible to use SALUTA to assess software architectures for their support of usability, whether we have accurately predicted the architecture's support for usability can only be answered after the results of this analysis are compared to the results of final user testing results when the system has been finished. Several user tests have been performed. The results of these tests fit the results of our analysis: the software architecture supports the right level of usability. Some usability issues came up that where not predicted during our architectural assessment. However, these do not appear to be caused by problems in the software architecture.

We are not sure that our assessment gives an accurate indication of the architecture's support for usability. On the other hand it is doubtful whether this kind of accuracy is at all achievable.

Causes: The validity of our approach has several threats:

- Usability is often not an explicit design objective; SALUTA focuses on the assessment of usability during architecture design. Any improvement in usability of the final system should not be solely accounted to our method. More focus on usability during development in general is in our opinion the main cause for an increase in observed usability.
- Accuracy of usage profile: Deciding what users, tasks and contexts of use to include in the usage profile requires making tradeoffs between all sorts of factors. The representativeness of the usage profile for describing the required usability of the system is open to dispute. Questions whether we have accurately described the systems usage can only be answered by observing users when the system has been deployed. An additional complicating factor is the often weakly specified requirements, which makes it hard to create a representative usage profile.

Solution: To validate SALUTA we should not only focus on measuring an increase in the usability of the resulting product but we should also measure the decrease in costs

spent on usability during maintenance. If any usability issues come up which require architectural modifications then we should have predicted these during the assessment.

Research issues: architectural assessment saves maintenance costs spent on dealing with usability issues. However at the moment we lack figures that acknowledge this claim. In the organization that participated in the case studies these figures have not been recorded nor did they have any historical data. To raise awareness and change attitudes (especially those of the decision makers) we should clearly define and measure the business advantages of architectural assessment of usability.

Design rather than evaluate

Problem: The usage profile and usage scenarios are used to evaluate a software architecture, once it is there.

Solution: A much better approach would be to design the architecture based on the usage profile e.g. an attribute/property-based architectural design, where the SAU framework is used to suggest patterns that should be used rather than identify their absence post-hoc.

9.6 Related Work

Many authors (Shackel, 1991, Hix and Hartson, 1993, Nielsen, 1993, Preece et al, 1994, Wixon and Wilson, 1997, Shneiderman, 1998, Constantine and Lockwood, 1999, ISO 9126-1) 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 focuses on the essential relation with software architecture.

(Nigay and Coutaz, 1997) discusses a relationship between usability and software architecture by presenting an architectural model that can help a designer satisfy ergonomic properties. (Bass et al, 2001) gives several examples of architectural patterns that may aid usability. Previous work has been done in the area of usability patterns, by (Tidwell 1998, Perzel and Kane 1999, Welie and Trætteberg, 2000). For defining the SAU 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. The framework based approach for usability is similar to the work done on quality attribute characterizations (Bass et al, 2003) in (Folmer et al, 2003) the most important differences between their approach and ours are outlined.

The Software Architecture Analysis Method (SAAM) (Kazman et al, 1994) was among the first to address the assessment of software architectures. SAAM is stakeholder centric and does not focus on a specific quality attribute. From SAAM, ATAM (Kazman et al, 2000) has evolved. ATAM also uses scenarios for identifying important quality attribute requirements for the system. Like SAAM, ATAM does not focus on a single quality attribute but rather on identifying tradeoffs between quality attributes. Some specific quality-attribute assessment techniques have been developed. In (Alonso et al, 1998) an approach to assess the timing properties of software architectures is discussed

using a global rate-monotonic analysis model. The Software Architecture Analysis Method for Evolution and Reusability (SAAMER) (Lung et al, 1997) is an extension to SAAM and addresses quality attributes such as maintainability, modifiability and reusability. In (Bengtsson and Bosch, 1999) a scenario based Architecture-Level Modifiability Analysis (ALMA) method is proposed.

We use scenarios for specification of quality requirements. There are different ways to interpret the concept of a scenario. In object oriented modeling techniques, a scenario generally refers to use case scenarios: scenarios that describe system behavior. The 4+1 view (Kruchten, 1995) uses scenarios for binding the four views together. In Human Computer Interaction, use cases are a recognized form of task descriptions focusing on user-system interactions. We define scenarios with a similar purpose namely to user-system interaction that reflect the usage of the system but we annotate it in such a way that it describes the required usability of the system.

9.7 Conclusions

Software engineers and human computer interaction engineers have come to the understanding that usability is not something that can be easily "added" to a software product during late stage, since to a certain extent it is determined and restricted by architecture design.

Because software engineers in industry lacked support for the early evaluation of usability we defined a generalized four-step method for Software Architecture Level UsabiliTy Analysis called SALUTA. This paper reports on 11 experiences we acquired developing and using SALUTA. These experiences are illustrated using three case studies we performed in the domain of web based enterprise systems: Webplatform, a content management system developed by ECCOO, Compressor, an e-commerce application developed by IHG and eSuite, an Enterprise resource planning system developed by LogicDIS.

With respect to the first step of SALUTA, creating a usage profile we found that transforming requirements to a format that can be used for architectural assessment is difficult because requirements and quality attributes can be interpreted in different ways. In addition specifying a necessary level for certain quality attributes is difficult during initial design since they can often only be measured when the system is deployed. To keep the assessment at a reasonable size we need set an explicit goal for the analysis to filter out those scenarios that do not contribute to this goal, tool support is considered for automating this step.

With respect to the second step of SALUTA, architecture analysis we found that some representation of the software architecture is needed for the analysis however some aspects such as design decisions can only be retrieved by interviewing the software architect. The applicability of SALUTA is not excluded to other application domains but the SAU framework that we use for the architectural analysis may need to be specialized and the relationships quantified for different application domains in order to produce more accurate results.

Concerning the third step, scenario evaluation is often guided by tacit knowledge. Concerning the fourth step, interpretation of results we experienced that initially the lack of a frame of reference made the interpretation less certain. In addition we made some general experiences; not only does usability impact software architecture design

but software architecture design may lead to usability problems. The accuracy of the analysis and the representativeness of a usage scenario can only be determined with results from final usability tests and by analyzing whether costs that are spent on usability during maintenance have decreased. Rather than identify the absence or presence of patterns post-hoc we should use the SAU framework to suggest patterns that should be used. In our view the case studies that have been conducted have provided valuable experiences that have contributed to a better understanding of architecture analysis and scenario based assessment of usability.

9.8 Acknowledgements

This work is sponsored by the STATUS project under contract no IST-2001-32298. We would like to thank the companies that enabled us to perform the case studies, i.e. ECCOO, IHG and LogicDIS. We especially like to thank Lisette Bakalis, Roel Vandewall of ECCOO, Fernando Vaquerizo of IHG and Dimitris Tsirikos of LogicDIS for their valuable time and input.