Experiences with Software Architecture Analysis of Usability

Eelke Folmer, Jan Bosch
Department of Mathematics and Computing Science
University of Groningen, PO Box 800, 9700 AV the Netherlands
mail@eelke.com, Jan.Bosch@cs.rug.nl

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 has proven to be costly since many changes cannot be easily accommodated by the software architecture. These high costs prevent developers from meeting all the usability requirements, resulting in systems with less than optimal usability. Explicit evaluation of a software architecture for its support of usability is a tool to cost effectively develop usable systems. It allows for more "usability tuning" on the detailed design level, hence, preventing part of the high costs incurred by adaptive maintenance activities once the system has been implemented. Based on our investigations into the relationship between usability and software architecture, we developed a Scenario based Architecture Level UsabiliTy Analysis technique (SALUTA). 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 (e.g. e-commerce-, content management- and enterprise resource planning systems). We make some general observations and some architecture assessment related observations. For each experience, a problem description, examples, causes, solutions and research issues are identified.

Keywords
Software architecture, usability, architecture analysis.

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 [1]. The software architecture, 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 [2] does not support the required level of quality.

The work in this paper is motivated by the fact 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.

This is a problem for software development because it is very expensive to ensure a particular level of usability *after* the system has been implemented. Studies [3,4] 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 and is something that cannot be fully solved or avoided. The real problem is that it often proves to be hard and expensive to make the necessary changes to a system to improve its usability. Many of the necessary usability changes to a system cannot be easily be *accommodated* by the software architecture. Certain usability improving solutions such as undo, user profiles and visual consistency have for particular application domains proven [5,6] to be extremely hard to retrofit during late stage development because they require architectural support. Restructuring the system during the later stages of development has proven to be an order of magnitude higher than the costs of an initial development [1].

Usability is to a large extent restricted by the software architecture. However, software engineers and human computer interaction engineers are often not aware of this important constraint and as a

result avoidable rework is frequently necessary. This rework leads to high costs and because during design different tradeoffs have to be made, for example between cost and quality, this leads to systems with less than optimal usability.

The challenge is therefore to *cost effectively* develop usable software e.g. avoid the high costs of reworking the system. Assessing a software architecture for its support of usability is first step in this direction. Software architecture assessment is a technique to come up with a more "usable" software architecture that allows for more "usability tuning" on the detailed design level, hence, preventing part of the high costs incurred by adaptive [7] maintenance activities once the system has been implemented.

Most engineering disciplines provide techniques and methods that allow one to assess and test quality attributes of the system under design. In [8] an overview is provided of usability evaluation techniques that can be used during software development. Unfortunately, no usability assessment techniques exist that focus on the assessment of software architectures. Based upon successful experiences [9] with architectural assessment of maintainability, we developed architectural analysis of usability as an important tool to cost effectively develop usable software. To be able to analyze a software architecture for its support of usability, we first investigated the relationship between usability and software architecture in [6]. 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 [10] we developed a Scenario based Architecture Level UsabiliTy Assessment technique (SALUTA) which is based on the Software Architecture Analysis Method (SAAM) [11]. SALUTA uses the SAU framework to analyze a software architecture for its support of usability. Our technique has been applied 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 we collected several experiences.

We consider SALUTA to be a prototypical example of an architecture assessment technique. 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 framework that we use for analysis is presented. Our method for software architecture analysis is described in section 3. Section 4 introduces the three cases. Our experiences are described in section 5. Finally, related work is discussed in section 6 and the paper is concluded in section 7.

2. The SAU Framework

A software architecture description such as a decomposition of the system into components and relations with its environment may provide information on the support for particular quality attributes. Specific relationships between software architecture (such as - styles, -patterns etc) and quality attributes (maintainability, reliability and efficiency) have been described by several authors [12,13,1]. For example, [12] describes the architectural pattern *Layers* and the positive effect this pattern may have on exchangeability and the negative effect it may have on efficiency.

Until recently [5,6] such relationships between usability and software architecture had not been described nor investigated. In [6] we defined a framework that expresses relationships between Software Architecture and Usability (SAU) based on our comprehensive survey [8]. The framework consists of an integrated set of design solutions that have been identified in various cases in industry, modern day software, and literature surveys. These solutions are typically considered to have a positive effect on the level of 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. E.g. it requires the architecture to be organized in a particular structure with relationships between those structures. If some parts of the system have already been implemented at the time that changes are made, modification will likely affect many parts of the existing source code, which is very expensive to modify.
- Structural support: Some solutions such as providing visual consistency do not necessarily require a particular architecture structure. In practice it is noticed that such 'architecturally sensitive' design solutions are implemented, but business constraints cause such changes to be implemented in an ad-hoc fashion, rather than structurally. For example, it is possible to make all screens consistent without any architectural modification. Such modifications then may erode original architectural design [14] making it harder to modify a particular screen.

Such solutions can be much easier provided, enforced and much easier maintained when a structural solution is chosen. Visual consistency, for example, may be easily facilitated by the use of a separation-of-data-from-presentation mechanism such as the use of XML and XSLT (a style sheet language for transforming XML documents).

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

2.1 Usability attributes

Usability attributes: A number of usability attributes have been selected from literature that appear to form the most common denominator of existing notions of usability [15,16,17,18,19,20,21]:

- 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.

2.2 Usability properties

A number of usability properties have been selected from literature [20,15,16,22,17,23,24,25,26,27] that embody the heuristics and design principles that researchers in the usability field consider to have a direct positive influence on system usability. These should be considered as high-level design primitives that have a known effect on usability and typically have architectural implications. An example can be found in Table 1:

Table 1: 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.

2.3 Architecturally sensitive usability patterns:

A number of 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. A set of architecturally sensitive usability patterns have been identified from various cases in industry, modern software, literature surveys [15,16,17,18,19,20,21] as well as from existing usability pattern collections [28,29,30,31]. 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. 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. Unlike the design patterns [13], architecturally sensitive patterns do not specify a specific design solution in terms of objects and classes. Instead, potential architectural implications that face developers aiming to solve the problem the architecturally sensitive pattern represents are outlined. An example is shown in table 2:

Table 2: 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. [29]
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 [12]
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 view and controller allows different types of input and output 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.
Example:	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 data.

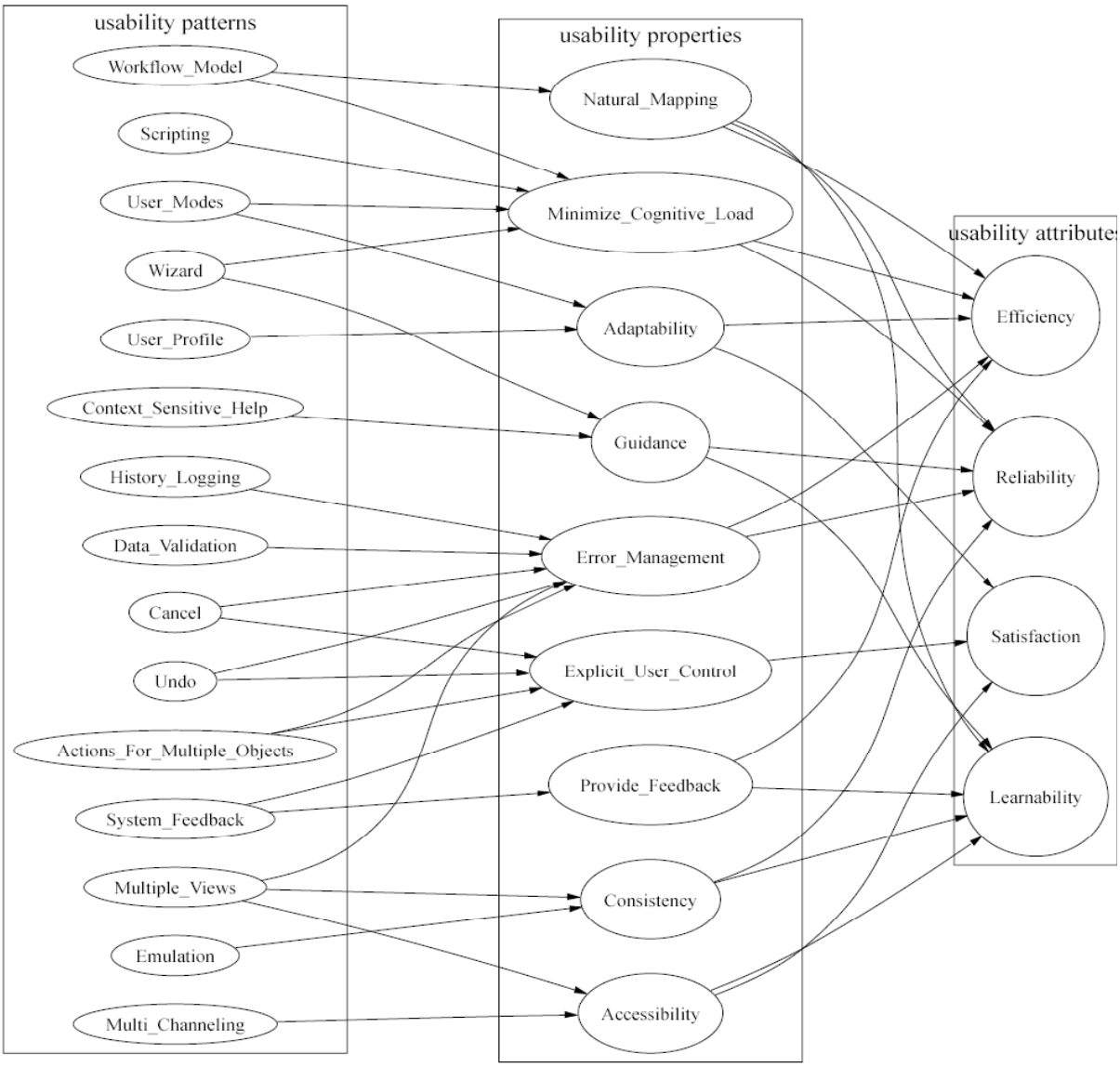

Figure 1: Relationships between attributes, properties and patterns.

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 [6], 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 may be implemented during architecture design to fulfill these properties by following the arrows in Figure 1. The choice of which design solution to apply may be made based on 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 an architecture, may support usability. For example, if undo has been implemented we can analyze how undo improves efficiency and reliability.

Our assessment technique uses this framework to analyze 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 [6].

3. Method overview

In [10] we developed a method for architecture analysis of usability. This method is based on scenario based assessment i.e. in order to assess a particular architecture, a set of scenarios is developed that concretizes the actual meaning of a requirement [1]. For that purpose *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 five 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 [10].

3.1 Usage profile creation

One of the most important steps in SALUTA is the creation of a usage profile. Because we found that traditional usability specification techniques [15,18,17] are poorly suited for architectural assessment, we decided to use scenario profiles [32,9] for this purpose. 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 [25], usability depends on: the users (e.g. system administrators, novice users), the tasks (e.g. insert order, search for item X) and 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 our 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 [33]. For example, if for a particular usage scenario learnability is considered to be of more importance than other usability attributes (maybe 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 2.

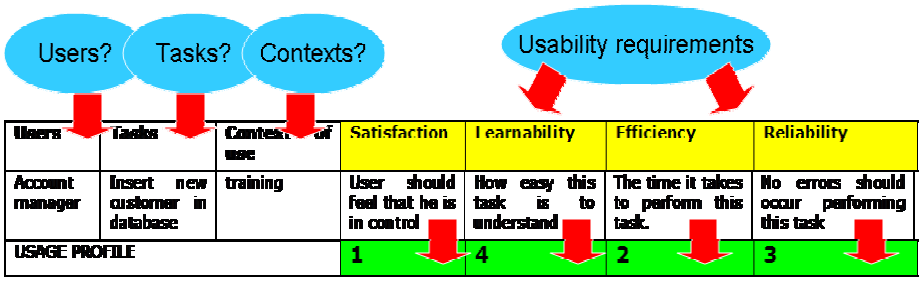

Figure 2: Example usage scenario

Usage profile creation is not intended to 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 [15,17,20,34] 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. Other techniques such as pair wise comparison may also be used to determine a prioritization between attributes.
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. If the goal is to analyze the level of usability support for an architecture, scenarios may be selected that are important to the users. To express differences between usage scenarios in the profile, properties may be assigned to 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.

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 [9] but is different, because it focuses on identifying architectural elements that may support the scenario. For architecture analysis, the SAU framework in section 2 is used to analyze the architecture for its support of usability. Two types of analysis are performed:

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

- Analyze the support for properties: The software architecture can be seen as the result of a series of design decisions [14]. Reconstructing this process and assessing the effect of such individual decisions with regard to usability attributes may provide 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 is 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. The software architecture of a system has several aspects (such as design decisions and their rationale) that cannot easily be captured or expressed in a single model. Different views on the system [35,36] or interviews with the software architect are needed to access such information.

3.3 Scenario Evaluation

The next step is to evaluate the support for each of the scenarios in the usage profile. For each scenario, it is analyzed by which usability patterns and properties, that have been identified in the previous step, it is affected. A technique we have used for identifying the provided usability in our cases is the usability framework approach. The relations defined in the SAU framework are used to analyze how a particular pattern or property affects a specific usability attribute. For example, if it has been identified that error management affects a certain scenario, the relationship between error management and usability are analyzed to determine the support for that particular scenario. Error management may increase reliability and efficiency. These values are then compared to the required attribute values to determine the support for this scenario.

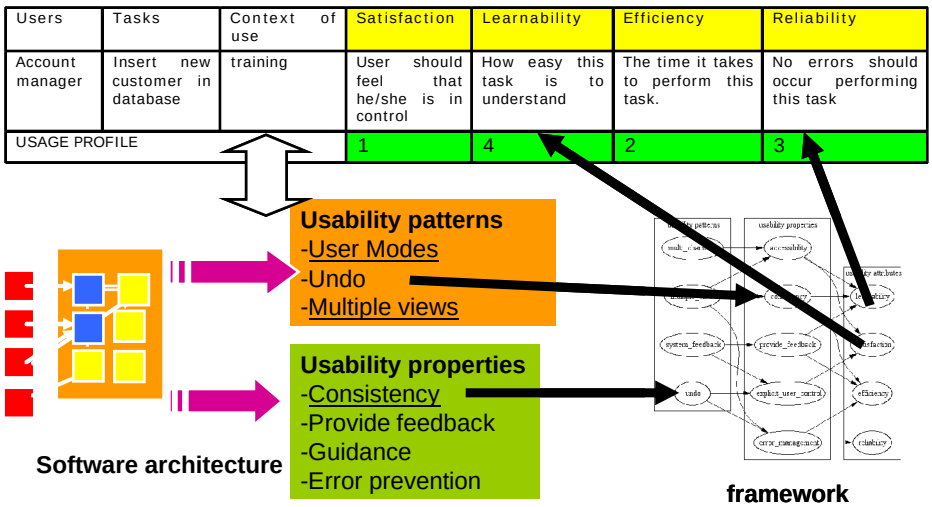

Figure 3: Snapshot assessment example

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.

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. 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 particular transformations. The SAU framework can then be used as an informative source for design and improvement of the architecture's support of usability.

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 method we used action research [37], i.e. we took upon our self the role of external analysts and actively participated in the analysis process and reflected on the process and the results. One case study has been performed at a software organization which is part of our university; the other two case studies are performed at our industrial partners in the STATUS[1] project. These case studies, have been published as part of the STATUS deliverables and in [38] and in a pending article [39]

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 application 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 3) illustrates the scope of applicability of our method. The remainder of this section introduces the three systems that have been analyzed.

Table 3: 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.	Selection: Compare old versus new version of eSuite.
Different 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
System release status	Fully deployed	Completed not yet deployed	In development

4.1 Webplatform

The Webplatform is a web based content management system (CMS) 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 data of 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

[1] STATUS is an IST project (IST-2001-32298) financed by the European Commission in its Information Society Technologies Program. This project studies the relationship between software architecture and the usability of a software system. The partners are Information Highway Group (IHG), Universidad Politecnica de Madrid (UPM), University of Groningen (RUG), Imperial College of Science, Technology and Medicine (ICSTM), LOGICDIS S.A.

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.

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 examined the design documentation such as architectural designs and requirements specifications.

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 examined the design documentation such as architectural designs and usability requirements specifications.

5. Experiences

This section gives a description of the experiences that we acquired during the definition and use of our method. We consider SALUTA to be a prototypical example of an architecture assessment technique. Our experiences are relevant in a wider context than just architecture assessment of usability. Our experiences are therefore categorized as follows:

- General experiences
- Architectural assessment experiences

For each experience, a problem description, examples, possible causes, available solutions and research issues are identified. The experiences are illustrated using examples from the three case studies introduced before.

5.1 General Experiences

(E1) 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 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: HCI and SE processes are not fully integrated (E2). As a result interface design is often postponed until the later stages of design. Hence we run the risk that many assumptions may be built into the design of the architecture that unknowingly may affect interface/interaction design and vice versa. This leads to avoidable rework of the system.

Solution: Closer integration of HCI and SE processes (E2): 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. In addition we should not only analyze whether the architecture design can support usability solutions but also how the architecture design may restrict usability.

Research issues: Our framework only captures the relationships between usability improving design solutions and software architecture, therefore SALUTA can only assess an architecture for its support of these solutions. However there are cases where architecture design leads to usability problems. 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 is required to be able to analyze whether a particular architecture design may lead to these kinds of usability problems.

(E2) SE and HCI processes are not fully integrated

Problem: 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 (E1). This leads often to avoidable rework of the system.

Example: In all case studies we studied, before we started the assessment, no attention was paid to making sure the software architecture supported all usability requirements. Only in one case (Webplatform) it was made sure that the software architecture facilitated visual consistency. Interface design was in all cases at the later stages of design, when it was already too late to fix architecture related usability problems.

Causes: Awareness and Attitude (E3). Software architects fail to associate usability with software architecture design. In addition there is a lack of early assessment techniques [8] that can specifically assess software architectures for their support of usability.

Solutions: Raising awareness and changing attitudes: stressing out the importance of the relationship between usability and software architecture.

(E3) Awareness and Attitudes

Problem: Software architects fail to associate usability with software architecture design. As a result HCI and SE processes are not fully integrated (E2). Functional requirements are very important however non functional requirements are all so very important. Architectural design should not be dominated by the functional requirements.

Example: 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.

Causes: there are two main causes for this problem: First, developers tend to concentrate on the functional features of their architectures and seldom address the ways in which their architectures support quality concerns [11]. The software engineering community often considers usability to be primarily a property of the presentation of information; the user interface [40]. This is a false assumption. The most reliable and performing system architecture is not usable if the user can't figure out how to use the system. But on the other hand a slow and buggy system architecture with a usable interface is not usable either. Software architects should be aware that the software architecture also plays an important role in fulfilling and limiting the level of quality. Second, software architects tend to optimize technological considerations over usability considerations. A software product is often seen as a set of *features* rather then a set of "*user experiences*". When design is dominated by a technological view, it's natural for decision makers (including software architects) to make decisions that optimize technological considerations over all others [40]. As a result, software architecture analysis is an ad-hoc activity (E4).

Solutions: Raising awareness of the importance of the relationship between usability and software architecture but raising the importance of usability as the most import quality attribute and software architecture as an important instrument to *fulfill* this attribute. By raising the awareness of this relationship eventually software engineers and usability engineers must recognize the need for a closer integration of practices and techniques. Changing attitudes: the technological view of a product is only one of many views; usability is an important design objective which should be fulfilled and facilitated by software architecture design. The best software comes from teams, or from team leaders, that are able to see the work from multiple perspectives, balancing them in accordance with the project goals, and the state of the project at any given time [40].

Research issues: architectural assessment saves maintenance costs spent on dealing with usability issues. However at the moment we lack figures that acknowledge this claim. To raise awareness and change attitudes (especially those of the decision makers) we should clearly define and measure the business and competitive advantages of architectural assessment of usability.

(E4) Software architecture analysis is an ad hoc activity

Problem: Generally, three arguments for defining an architecture are used [41]. First, it provides an artifact that allows discussion by the stakeholders very early in the design process. Second, it allows for early assessment of quality attributes [42,1]. Finally, the design decisions captured in the software architecture can be transferred to other systems. As identified by [9] early assessment is least applied in practice.

Example: In all organizations where we conducted case studies, architecture assessment was not an explicitly defined process and there was no integration and cooperation with existing (usability) requirements collection techniques. We were called in as an external assessment team mostly at the end of the software architecture design phase and in one case (Webplatform) even after that phase to assess whether any architecture-usability problems were to be expected.

Causes: Attitude and Awareness (E3). HCI and SE processes are not fully integrated (E2). There is no need to assess a software architecture for quality concerns if developers are not aware that the software architecture plays a major role in fulfilling them. The software architecture is often seen as an intermediate product in the development process but its potential with respect to quality assessment still needs to be exploited [9]. In addition for some quality attributes developers have few or no techniques available for predicting them before the system itself is available.

Solution: Raising the importance of software architecture as an important instrument for quality assessment. If a software architecture analysis technique such as SALUTA was an integral part of the development process earlier phases or activities would result in the necessary information for creating usage profiles and provide for the necessary architectural descriptions. For example existing usability engineering techniques such as interviews, group discussions, rapid prototyping or observations [15,17,20,34] typically already provide information such as representative tasks, users and contexts of use and requirements for these scenarios that are needed to create a usage profile. The results of an architecture assessment should influence the architecture design process.

Research issues: The usage profile and usage scenarios are used to evaluate a software architecture, once it is there. However a much better approach would be to design the architecture based on the usage profile e.g. an attribute-based architectural design, where the SAU framework is used to suggest patterns that should be used rather than identify their absence post-hoc.

(E5) 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. In the case of the Webplatform, several user tests have been performed recently. The results from the assessment seem reasonable and do not conflict with the user tests. The results of the user tests of the other two cases are scheduled for summer 2004. 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 the main cause for an increase in usability of systems. 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. In addition usability requirements are often weakly specified (E6) and change (E7) which makes it hard to predict the right usability requirements.

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 at that time that require architectural modifications then these should have been predicted during the assessment. In general, our assessment approach was overall well received by the software architects. In some cases the software architect had not considered the use of some of our patterns or properties but they considered to implement them in the system based on the result of the assessment.

5.2 Architecture assessment experiences

Software architects should design their architectures in such a way that it will support all functional and all quality requirements. In order to do so they need to be able to *extract* requirements from all stakeholders, they need to have techniques to *realize* these requirements and they need to be able to *assess* whether the resulting product actually meets the requirements. Three types of architecture assessment have been identified [1]:

- Scenario based assessment: In order to assess a particular architecture, a set of scenarios is developed that concretizes the actual meaning of a requirement. For each scenario the architecture is assessed for its impact or support of this scenario.
- Simulation: Simulation of the architecture uses an executable model of the application architecture. It is possible to check various properties of such a model in a formal way and to animate it to allow the user or designer to interact with the model as they might with the finished system.
- Mathematical modeling: By using mathematical models developed by various research communities such as high performance computing, operational quality attributes can be assessed. Mathematical modeling is closely related to, or an alternative to simulation.

In our industrial and academic experience with scenario based analysis we have come to understand that scenario based analysis is a good technique to analyze software architectures. 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 usability. In the context of architecture analysis we collected the following experiences:

(E6) Requirements are often poorly not formally specified

Problem: Requirements are often poorly or weakly specified during *initial* design. This is especially true for usability. Our usage profile technique depends on *specified* usability requirements. Usage profile creation does not replace existing requirements engineering techniques. Rather it transforms existing usability requirements into something that allows for architectural assessment.

Example: In all cases, apart from the web platform case (some general usability guidelines based on Nielsen's heuristics [15] had been stated in the functional requirements) no clearly defined and verifiable usability requirements had been collected or specified. Often usability requirements are specified as: *the system shall be usable.* However this statement does not define for which tasks, users or contexts of use this requirement should be interpreted.

Causes:

- Most software developing companies still underestimate the importance of usability engineering and postpone the activity of usability engineering collection until there is a running system. Usability is often not defined as an explicit project goal. Decision makers do not see the tradeoffs and are more concerned with time to market issues rather than software quality.
- Traditionally, usability requirements are specified such that these can be verified for an implemented system. For example: "new users should require no more than 30 minutes instruction". However, assessing an architecture for such a requirement is difficult because such requirements can only be measured when the system has been completed.

Solutions: Existing usability engineering techniques such as interviews, group discussions, rapid prototyping or observations [15,17,20,34] typically already provide information such as representative tasks, users and contexts of use and requirements for these scenarios that are needed to create a usage profile. For more accurate usage profiles and more accurate assessment results we advocate the use and adaptation of such existing usability requirements engineering techniques. In addition close cooperation between the analyst and the person responsible for the usability

requirements (such as a usability engineer) is required during the analysis. The usability engineer may fill in the missing information on the usability requirements.

(E7) Requirements change

Problem: During or after the development usability requirements change.

Cause: The context in which the user and the software operate is continually changing and evolving. Sometimes users may find new uses for a product, for which the product was not originally intended. During development, the uses of the system are often not fully documented nor is a definition made of exactly who the users are. Users themselves often lack understanding of their own requirements, only when they work with a first version of the software they realize how they are going to use the system. Usability experts miss about half of the problems that real users experience using traditional techniques [43]. This makes it virtually impossible to capture all possible (future) usability requirements during initial design (8).

Example: 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 show context sensitive help texts, however for more experienced users this turned out to be annoying and led to a usability problem. It would be much better if a system had been created where help texts could be turned off for more experienced users.

Solution: requirements that change is a problem inherent to software development and it is not necessarily a problem that can be fully solved or avoided. Designing for change is considered a challenge. By analyzing the software architecture for its support of the properties and patterns (even those that may not be required from the usage profile) in our framework, unforeseen usability requirements may still be facilitated by the software architecture.

(E8) Difficult to transform requirements

Problem: To be able to assess a software architecture for its support of usability we need to transform requirements in a format that can be used for architectural assessment. 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 is encountered is that sometimes it is difficult to determine attribute values for a scenario because requirements and attributes can be interpreted in different ways. In addition most usability requirements are not formally / poorly specified (E6).

Example: What does efficiency or learnability 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 should this requirement be translated 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 4) 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. Possibly techniques such as pair wise comparison would be better suited to determine a prioritization.

Usability requirements						
UR1- every page should feature a quick search which searches the whole portal and comes up with accurate search results						
#	Users	Task	E ?	L ?	R ?	S ?
1	End user	Quick search	4	2	3	1

Figure 4: Transforming requirements to a usage profile

(E9) Specification of certain quality attributes is sometimes difficult

Problem: A purpose of quality requirements is to specify a *necessary* level [33]. In section 2 four different usability attributes have been presented which we use in our definition of usability and in expressing the required usability for a system in a usage scenario. However, specifying a necessary level of the satisfaction attribute of usability has proven to be difficult during initial design. It is very hard to specify how this attribute should be interpreted 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) 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?

(E10) Non-explicit nature of architecture design

Problem: In order to be able to evaluate the usage scenarios, some representation of the software architecture is needed. However, the software architecture has several aspects (such as design decisions and their rationale) that cannot easily be 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 5). However to determine the architectural support for usability we needed more information, such as which design decisions were taken.

Cause: Due to the non-explicit nature of architecture design, the analysis strongly depends on having access to both design documentation and software architects; 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 [35,36] may be needed to access such information. A conceptual view on the system of the Compressor provided us with detailed information on how the patterns [6] system feedback, multi channeling, multiple views and workflow modeling had been implemented.

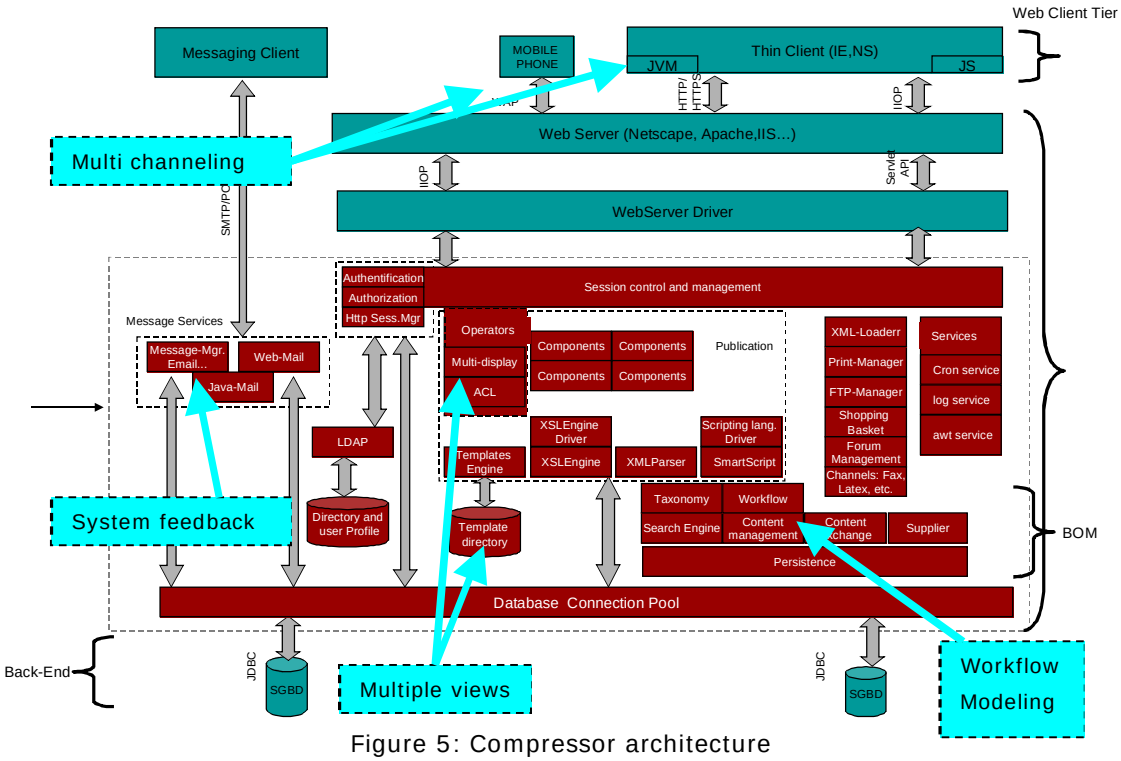

Figure 5: Compressor architecture

(E11) Validation and specialization 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. When we initially defined this framework it 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 and hence the assessment result may not be accurate.

Example: Our case studies have been performed in the domain of web based enterprise systems. Initially our 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.

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.

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).

(E12) 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 two 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 represented 5% of the users but the success of the Webplatform was largely dependent on their acceptance of the system. This brought the number of scenarios down to a reasonable size of 11 usage scenarios. In the compressor case we analyzed the frequency with which tasks were executed. Particular tasks had very low task execution frequencies, leaving these out reduced the number of scenarios from 58 to 14.
- Pair wise comparison: For most usage scenarios, concerning expressing required usability there is an obvious conflict between efficiency and learnability attribute. 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 obviously conflict.

Research issues: 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.

(E13) 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, may be 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 4 usability patterns and 6 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 1. 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 have to decide whether this scenario is supported?

Table 4: eSuite usage scenario

User	User context	Task	S	E	L	R
Novice	Mobile	Insert Order	1	2	4	3

Cause: Due to the lack of formalized knowledge at the architecture level, this step 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. But to do that we need to substantiate these relationships and to provide models and assessment procedures for the precise way that the relationships operate.

(E14) 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 are accepted, six are weakly accepted and two scenarios are weakly rejected (mainly caused because was low support for learnability as required by the content administrators' usage scenarios). 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.

Solution: The three case studies have provided us with a small frame of reference. We have seen architectures with significant better support for particular aspects of usability, which allows us to judge whether a particular architecture could be improved. However to refine our frame of reference more case studies need to be done.

6. Related work

Many authors [16,17,44,15,18,19,20,21] 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 [15], usability inspection [45] and usability inquiry [15] may be used to evaluate the usability of systems. However, none of these techniques focuses on the essential relation with software architecture.

[46] discusses a relationship between usability and software architecture by presenting an architectural model that can help a designer satisfy ergonomic properties. [5] gives several examples of architectural patterns that may aid usability. Previous work has been done in the area of usability patterns, by [28], [47], [30]. 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 Software Architecture Analysis Method (SAAM) [11] 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 [48] 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 [49] an approach to assess the timing properties of software architectures is discussed using a global rate-monotonic analysis model. In [50] a scenario based technique for the assessment of the maintainability of software architectures is proposed. SALUTA can be integrated into these existing techniques.

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 [35] 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.

7. Conclusions

Software engineers in industry lack support for the early evaluation of quality requirements such as usability. In [38] we have defined a generalized five-step method for Software Architecture Level UsabiliTy Analysis (SALUTA). This paper reports on 14 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. Concerning development in general we made the following five observations.

Not only does usability impact software architecture design but software architecture design may also impact usability. Software architects are not aware of these constraints or they do not care because they tend to concentrate on the functional features of their architectures. 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; as a result architecture assessment is an ad-hoc activity that is not explicitly planned for. The accuracy of the analysis can only be determined answered with results from

final usability tests and by analyzing whether costs that are spent on usability during maintenance have decreased.

Concerning architecture analysis we made the following nine observations. First requirements are often poorly or weakly specified during *initial* design. During or after development requirements may change. 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. 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 framework that we use for the architectural analysis may need to be specialized for different application domains in order to produce more accurate results. 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. In addition we should consider using other quantification techniques to minimize the number of attributes that need to be quantified for each scenario. Scenario evaluation is often guided by tacit knowledge; however, our framework however tries to capture some of that knowledge. Finally we experienced that initially the lack of a frame of reference made the interpretation less certain.

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.

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.

References

1. J. Bosch, *Design and use of Software Architectures: Adopting and evolving a product line approach*, Pearson Education (Addison-Wesley and ACM Press), Harlow, 2000.
2. IEEE, IEEE Architecture Working Group. Recommended practice for architectural description. Draft IEEE Standard P1471/D4.1, IEEE, (1998).
3. R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, NY, 1992.
4. T. K. Landauer, *The Trouble with Computers: Usefulness, Usability and Productivity.*, MIT Press., Cambridge, 1995.
5. L. Bass, J. Kates, and B. E. John, Achieving Usability through software architecture, *Technical Report* CMU/SEI-2001-TR-005, (2001).
6. E. Folmer, J. v. Gurp, and J. Bosch, 'Investigating the Relationship between Usability and Software Architecture ', *Software process improvement and practice* 0-0.(2003)
7. E. B. Swanson, The dimensions of maintenance, *proceedings of the 2nd international conference on software engineering*, 1976.
8. E. Folmer and J. Bosch, 'Architecting for usability; a survey', *Journal of systems and software* 61-78.(2002)
9. N. Lassing, P. O. Bengtsson, H. van Vliet, and J. Bosch, 'Experiences with ALMA: Architecture-Level Modifiability Analysis', *Journal of systems and software* 47-57.(2002)
10. E. Folmer, J. v. Gurp, and J. Bosch, 'Architecture level usability assessment', *Journal of information & software technology* 0-0.(2004)
11. R. Kazman, G. Abowd, and M. Webb, SAAM: A Method for Analyzing the Properties of Software Architectures, *Proceedings of the 16th International Conference on Software Engineering*, 1994.
12. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Son Ltd, New York, 1996.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns elements of reusable object-orientated software.*, Addison -Wesley, 1995.
14. J. v. Gurp and J. Bosch, 'Design Erosion: Problems and Causes', *Journal of systems and software* 105-119.(2002)
15. J. Nielsen, *Usability Engineering*, Academic Press, Inc, Boston, MA., 1993.
16. L. L. Constantine and L. A. D. Lockwood, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*, Addison-Wesley, New York NY, 1999.
17. D. Hix and H. R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process.*, John Wiley and Sons, 1993.
18. J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Holland, and T. Carey, *Human-Computer Interaction*, Addison Wesley, 1994.

19. B. Shackel, Usability - context, framework, design and evaluation, in *Human Factors for Informatics Usability*, Shackel, B. and Richardson, S., Cambridge University Press, 1991.
20. B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, Reading, MA, 1998.
21. D. Wixon and C. Wilson, The usability Engineering Framework for Product Design and Evaluation., in *In Handbook of Human-Computer Interaction*, Helander, M. G., Elsevier North-Holland, 1997, 1997.
22. S. J. Ravden and G. I. Johnson, *Evaluation usability of human-computer interfaces: A practical method.*, Ellis Horwood Limited, New York, 1989.
23. D. A. Norman, *The design of everyday things*, Basic Books , 1988.
24. P. G. Polson and C. H. Lewis, Theory-based design for easily learned interfaces, 1990.
25. ISO, ISO 9241-11 Ergonomic requirements for office work with visual display terminals (VDTs) -- Part 11: Guidance on usability., (1994).
26. R. Holcomb and A. L. Tharp, What users say about software usability., *International Journal of Human-Computer Interaction, vol. 3 no. 1*, 1991.
27. R. Rubinstein and Hersh.H., *The Human Factor: Designing Computer Systems for People.*, Digital Press, Bedford, MA, 1984.
28. J. Tidwell, Interaction Design Patterns, *Conference on Pattern Languages of Programming 1998*, 1998.
29. Brighton, *The Brighton Usability Pattern Collection.* http://www.cmis.brighton.ac.uk/research/patterns/home.html
30. M. Welie and H. Trætteberg, Interaction Patterns in User Interfaces, *Conference on Pattern Languages of Programming (PloP) 7th*, 2000.
31. Pointer, *PoInter: Patterns of INTERaction collection*, http://www.comp.lancs.ac.uk/computing/research/cseg/projects/pointer/patterns.html
32. P. O. Bengtsson, *Architecture-Level Modifiability Analysis*, Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Sweden, 2002.
33. S. Lauesen and H. Younessi, Six styles for usability requirements, *Proceedings of REFSQ'98*, 1998.
34. J. T. Hackos and J. C. Redish, *User and Task Analysis for Interface Design*, John Wiley and Sons, Inc. New York, 1998.
35. P. B. Kruchten, The 4+1 View Model of Architecture, IEEE Software, (1995).
36. C. Hofmeister, R. L. Nord, and D. Soni, *Applied Software Architecture*, Addison Wesley Longman, Reading, MA, 1999.
37. C. Argyris, R. Putnam, and D. Smith, *Action Science: Concepts, methods and skills for research and intervention*, Jossey-Bass, San Francisco, 1985.
38. E. Folmer, J. v. Gurp, and J. Bosch, Software architecture analysis of Usability , *The 9th IFIP Working Conference on Engineering for Human-Computer Interaction*, 2004.
39. E. Folmer, J. v. Gurp, and J. Bosch, SALUTA: Scenario Based Architecture Level Usability Analysis., in *Human-Centered Software Engineering (Volume 1): Bridging HCI, Usability and Software Engineering*, Seffah, A., Desmarais, M., and Gulliksen, J., Unknown, Submitted to, 2004.
40. S. Berkun, *The list of fourteen reasons ease of use doesn't happen on engineering projects*, http://www.uiweb.com/issues/issue22.htm
41. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison Wesley Longman, Reading MA, 1998.
42. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, The Architecture Tradeoff Analysis Method, *Proceedings of ICECCS'98*, 8-1-1998.
43. D. L. Cuomo and C. D. Bowen, 'Understanding usability issues addressed by three user-system interface evaluation techniques', *Interacting with Computers* 86-108.(1994)
44. ISO 9126-1 Software engineering - Product quality - Part 1: Quality Model, (2000).
45. J. Nielsen, Heuristic Evaluation., in *Usability Inspection Methods.*, Nielsen, J. and Mack, R. L., John Wiley and Sons, New York, NY., 1994.
46. L. Nigay and J. Coutaz, Bridging Two Worlds using Ergonomics and Software properties., in *Formal Methods in Human-Computer Interaction,* Palanque & Paterno', Springer-Verlag., London, 1997.
47. K. Perzel and D. Kane, Usability Patterns for Applications on the World Wide Web, 1999.
48. R. Kazman, M. Klein, and P. Clements, ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004), (2000).
49. A. Alonso, M. Garcia-Valls, and J. A. de la Puente, Assessment Timing Properties of Family Products, *Proceedings of Second International ESPRIT ARES Workshop* , 1998.
50. P. O. Bengtsson and J. Bosch, Architecture Level Prediction of Software Maintaince, *EuroMicro Conference on Software Engineering*, 1999.