Parameter History Logging as an Architecturally Sensitive Usability Scenario in Automation Systems

Daniel Neeteson  
(s0921815)

Master thesis for Systems & Software Engineering

June, 2007

Supervisors:

Kai Koskimies  
Jarmo Palviainen  
Department of Software Systems  
Tampere University of Technology  
Finland

Paris Avgeriou  
Department of Software Engineering  
University of Groningen  
The Netherlands
# Table of Contents

1. Introduction ........................................................................................................................ 4
2. Software architecture and usability .................................................................................... 6
   2.1 Importance of software architecture ............................................................................ 6
   2.2 What is usability? ......................................................................................................... 7
   2.3 Gaps between software engineering and usability engineering ................................... 8
   2.4 Separating the User Interface from application functionality is not enough ............. 12
3. Previous work .................................................................................................................. 13
   3.1 The Usability & Software Architecture Group .......................................................... 13
      3.1.2 Example: Canceling Commands ......................................................................... 14
      3.1.2.1 Situation description .................................................................................... 15
      3.1.2.2 Conditions on the situation .......................................................................... 15
      3.1.2.3 Potential usability benefits ........................................................................... 15
      3.1.2.4 Formulating the forces that cause the problem and their solutions ............. 15
   3.2 A theoretical approach: the usability framework ....................................................... 16
      3.2.1 Usability properties ............................................................................................. 17
      3.2.2 Usability patterns ................................................................................................ 18
      3.2.3 Scenario based Architecture Level Usability Analysis (SALUTA) ................... 20
         3.2.3.1 Determine the goal of the assessment .......................................................... 20
         3.2.3.2 Create usage profile ..................................................................................... 20
         3.2.3.3 Describe the provided usability ................................................................... 21
         3.2.3.4 Evaluate scenarios ........................................................................................ 21
         3.2.3.5 Interpret the results ...................................................................................... 21
   3.3 A view on the approaches of U&SA and STATUS ................................................... 21
4. Usability in automation systems ...................................................................................... 23
   4.1 Characteristics of automation systems ....................................................................... 23
   4.2 Primary usability scenarios for automation systems .................................................. 24
   4.3 Secondary scenarios for automation systems ............................................................ 25
   4.4 Other usability issues ................................................................................................ 25
      4.4.1 Parameter history logging: a new scenario ......................................................... 26
5. The Parameter History Logging usability scenario ......................................................... 27
   5.1 Relationship with other usability scenarios ............................................................... 27
   5.2 Parameter history logging as a usability scenario architecture pattern (USAP) ........ 28
      5.2.1 Situation description ........................................................................................... 28
      5.2.2 Conditions on the situation ............................................................................... 28
      5.2.3 Allocating the scenario to the Benefit and Mechanism matrix ......................... 29
      5.2.4 Allocation to the Usability Benefits Hierarchy ................................................... 29
      5.2.5 Allocation to the Software Architectural Tactics Hierarchy .............................. 29
   5.3 Formulating Responsibilities ..................................................................................... 30
   5.4 Parameter history logging as a usability pattern in the light of STATUS .................. 32
   5.5 Parameter history logging example ........................................................................... 32
6. Applying parameter history logging to an existing system ............................................. 34
   6.1 System description ..................................................................................................... 34
   6.2 Usability Assessment .................................................................................................. 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.1 Mapping the scenarios</td>
<td>35</td>
</tr>
<tr>
<td>6.2.2 Conclusions</td>
<td>38</td>
</tr>
<tr>
<td>6.3 Implementing parameter history logging</td>
<td>38</td>
</tr>
<tr>
<td>6.3.1 Changes to the database</td>
<td>39</td>
</tr>
<tr>
<td>6.3.2 Changes to the GUI</td>
<td>39</td>
</tr>
<tr>
<td>6.3.3 Changes to SQL statements</td>
<td>40</td>
</tr>
<tr>
<td>6.4 Options to implementation</td>
<td>40</td>
</tr>
<tr>
<td>6.4.1 Parameter hierarchy</td>
<td>40</td>
</tr>
<tr>
<td>6.4.2 Managing the number of configurations</td>
<td>40</td>
</tr>
<tr>
<td>6.5 Effects of implementing Parameter History Logging to the results of a Usability Assessment</td>
<td>41</td>
</tr>
<tr>
<td>6.6 Relationships between architecturally-sensitive usability scenarios</td>
<td>43</td>
</tr>
<tr>
<td>7. Evaluation &amp; discussion</td>
<td>45</td>
</tr>
<tr>
<td>7.1 Summary and research questions</td>
<td>45</td>
</tr>
<tr>
<td>7.2 Future work</td>
<td>46</td>
</tr>
<tr>
<td>7.2.1 Extending the list of architecture-sensitive usability scenarios</td>
<td>46</td>
</tr>
<tr>
<td>7.2.2 Harmonizing software engineering and usability engineering processes</td>
<td>47</td>
</tr>
<tr>
<td>7.2.3 Refining and extending the Usability Framework</td>
<td>47</td>
</tr>
<tr>
<td>7.2.4 Reasoning Frameworks &amp; Usability</td>
<td>48</td>
</tr>
<tr>
<td>7.2.5 The relationships between scenarios or patterns</td>
<td>48</td>
</tr>
<tr>
<td>7.2.6 Tool support</td>
<td>48</td>
</tr>
<tr>
<td>7.2.7 Evidence for reduced maintenance costs</td>
<td>49</td>
</tr>
<tr>
<td>7.3 Conclusion</td>
<td>49</td>
</tr>
<tr>
<td>References</td>
<td>50</td>
</tr>
<tr>
<td>Appendix: Parameter History Logging example in DHTML</td>
<td>53</td>
</tr>
</tbody>
</table>
1. Introduction

It is a well-known problem for any software developer: once you have combined the necessary technologies to satisfy all functional requirements, and come up with that great solution architecture, the moment you start implementing it you encounter more and more problems concerning the usability of the resulting product. The later you discover these problems, the more difficult it becomes to fix it, especially when it affects the overall design of that system. Somehow it seems that current software engineering practice does not take usability into account until the product has been implemented.

The high cost of fixing usability issues in late stages of the software life cycle has been an incentive to investigate whether it is possible to include usability engineering in earlier stages of software development. In order to achieve this, differences between software engineering and usability engineering practices must be overcome. So far, software engineering has been focused more on technology, and usability engineering on human aspects. Communication between members of both communities has therefore proven to be difficult. Experts from either side have only a vague idea how the other one works and thinks. Some terms have different definitions, and similar concepts have different names. There is still a lot of room for improvement concerning usability in software engineering.

Software architectures are generally accepted as an important artifact in software design, implementation and maintenance. It allows for early stage analysis of quality attributes of a system. Because of this role of software architecture in software engineering, difficulties concerning usability might be solved by analyzing in which way the architecture of a system affects or limits the usability of that resulting system. Two main groups have investigated this relation extensively: the Usability & Software Architecture Group from the Software Engineering Institute of Carnegie Mellon University (U&SA), and a project financed by the European Union called Software Architecture that supports Usability (STATUS). Their work will be discussed in the Previous Work section.

This thesis is focused on the field of software architecture and usability. Several usability scenarios have been identified that affect the software architecture of a system. This thesis proposes a new usability scenario, that of Parameter History Logging. In any application which includes configurations with multiple parameters, a user can feel the need for knowing which values have been used before, and when they have been used. The need for this scenario is especially present in automation systems, as they often control machines that can be configured according to a large set of parameters.

The Usability & Software Architecture Group from the Software Engineering Institute of Carnegie Mellon University (U&SA) has developed a methodology which is the basis for creating usability supporting architectural patterns (USAP) for different usability scenarios. These patterns assist in modifying or adapting the design of software architectures to support a specific usability concern (Golden et al., 2005). USAP’s have very much in common with the concepts of design and architectural patterns that are well known in computing science. They all provide solutions in telling why, when and how it should be done (Avgeriou and Zdun, 2005). There are, however, differences. USAP’s only serve as a solution to deal with a specific usability concern. Architectural patterns are broader than that, and represent some rationale for the design of a whole system. Design patterns are indeed aimed at a single concern, but they can be described in a modeling
language like UML, whereas USAP’s cannot as it depends on the design of the system it is meant to be implemented in.

This thesis elaborates a new usability concern, parameter history logging, into a usability supporting architectural pattern (USAP), and shows how this USAP can be implemented in an existing system as a case study. It also evaluates and discusses the relation between different usability scenarios, as some of them obviously have some dependencies towards one and other.

The problem statement therefore is: how can the usability scenario of parameter history logging be worked out to a USAP? This should be answered by investigating the following questions:

- Before we can work to our USAP, we first need to know what we exactly want to work out. Thus, what exactly is parameter history logging and in what situations and under what conditions would a user need this usability-related functionality?
- In what way does parameter history logging support usability? More specifically, what aspects of usability does it relate to?
- How can we compose a generic set of requirements that should be addressed when implementing parameter history logging?
- More in general: are there usability scenarios that are in some way related to parameter history logging, and does it influence the necessity of implementing those?

The thesis is organized in the following chapters: Chapter 2 will introduce the reader into the fields of software architecture and usability engineering and show why the combination of these is important. Chapter 3 presents previous work and comparison between different views. Chapter 4 discusses usability issues in automation systems, showing the necessity for those systems to include parameter history logging. Chapter 5 presents parameter history logging as a usability scenario, and worked out according to the methods discussed in Chapter 3. Chapter 6 presents a case study, which contains a specific solution for retrofitting parameter history logging to that system, and assesses the effect of implementing Parameter History Logging to other usability scenarios. Chapter 7 discusses and evaluates the results, and presents possible future work in these particular fields.
2. Software architecture and usability

This chapter introduces the reader into the research areas of software architecture and that of usability engineering. It will clarify why software engineering practices can benefit from combining these fields.

2.1 Importance of software architecture

Anyone who has some programming experience knows that a growing piece of code soon becomes unmanageable if there is no proper design. You will quickly lose all track of what’s happening, by solving little bugs with ugly hacks and so on. The more changes are made, the more difficult it becomes to implement a new change, because it might effect other changes that have been made before. Especially when the software is written by more than one individual, some model or shared notion of the system that can be communicated among different developers is needed. It was already recognized in the 1960’s that designing large-scale software systems has its own unique problems. The following aphorism goes for anything that needs planning: Failing to prepare is preparing to fail. These developments made software design became an activity separate from implementation. Software architecture as a research topic was first mentioned in (Perry and Wolf, 1992). They introduced this term to distinguish it from software design, because it would not only include the process of designing, but also the ways how to represent a system and the fact that people could be trained to become software architects.

During the 1990’s, software architecture has become an important artifact in software design. About half a century ago, designing software was mainly concerned with translating a small set of requirements into associated executable functions in an assembly language. It soon became clear that some sequences of instructions could be reused for other functions or programs, and quickly developers adopted this design approach. Later, language support for this could be provided by functions and procedures of programming languages, hereby achieving a higher level of abstraction in describing software systems (Garlan and Shaw, 1994). As the size and cost of software systems increased beyond the scope of a human individual, the need for a high-level representation of a system early in the design phase became clearer. The top-level organization of computational elements and interactions between those elements is a critical aspect of the design for any large software system (Garlan, 1995). This top-level is called the software architecture.

There are many definitions for software architecture, but a commonly accepted one is the following: “The software architecture is 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)”. Software architectures serve different purposes. Of course, they provide a set of blueprints for implementation, which is important for the development team. Furthermore, software architectures are used for communication among stakeholders. Software architectures are also an artifact for analysis, because they embed early design decisions. Various methods, mostly scenario-based, for
analyzing software architectures have been developed, such as SAAM (Kazman et al., 1994) and ATAM (Kazman et al., 1998).

Over the years, software architecture gained more recognition as an important tool to analyze systems in an early design stage. It became clear that certain qualities such as modifiability and performance were partly related to the architecture of a system. Consequently, software architecture design was not only associated with achieving the required functionality, but also with achieving a certain quality level (Bengtsson and Bosch, 2000). Tools and techniques that allow for design for quality attributes such as performance or modifiability have been developed in recent years, but architecting for usability remains to be very difficult. Until recently, achieving usability was mainly addressed as a problem in modifiability, as one of the problems is that most usability issues are noticed only after initial design and implementation. The later in the life cycle of software systems a problem is detected, the more expensive it is to fix (Bass et al., 2001). Usability is a lot more difficult to define than modifiability or performance. When is a system more usable than another? Before any answer can be provided, a definition must be agreed.

2.2 What is usability?

Software systems that offer all functionality but are not usable, will not sell. Building a usable system is thus one of the goals of software engineering. In order to be able to architect for usability, some sort of definition of the term is necessary. Usability is such an abstract term that it has many definitions. The term was originally derived from “user friendly”. The latter term was later replaced by usability because it had acquired a lot of vague and subjective connotations (Bevan et al., 1991). There are different authors who have proposed different definitions dependent on the way they wanted to measure usability (Nielsen, 1993), (Shackel, 1991), (ISO 9126), (ISO 9241-11). According to Nielsen, usability is a quality attribute that assesses how easy user interfaces are to use. Usability is indeed closely related to user interaction, but adding for example the possibility to undo is not making a user interface easier to use. The definition according to ISO 9241-11 is more specific: the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. ISO 9126’s definition is similar: the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions. Shackel names four attributes necessary for usability: effectiveness, learnability, flexibility and attitude, which has in this context merely the same meaning as satisfaction. Most of the definitions specify attributes of which usability is composed. In (Folmer & Bosch, 2003), four of those attributes are distilled from the different approaches. These attributes are:

1. Learnability
2. Efficiency of use
3. Reliability in use
4. Satisfaction
Learnability is affected by both the time needed for a new user of the system to be able to do productive work, and the ease of remembering how the system should be used. Efficiency of use is measured by the number of tasks a user can accomplish while using the system during a specific amount of time. Reliability in use deals with the error rate in system use and the time it takes to recover from errors. Satisfaction of a system is a subjective attribute and can be measured by observing and interviewing system users.

The Usability Benefits Hierarchy (Table 1) as proposed by (Bass et al. 2001) is more detailed in assessing usability. Here, contrary to the attributes of (Folmer & Bosch, 2003), a distinction is being made between routine and non-routine performance, both as part of user effectiveness. Another distinction is being made between user and system errors, and user errors are caused either by slips or by lack of knowledge. All four attributes named by (Folmer & Bosch, 2003) are represented in this hierarchy. Learnability and efficiency of use are both part of “Increases individual user effectiveness”. Reliability in use is covered in both “Increases individual user effectiveness” and “Reduces the impact of system errors”. Satisfaction can easily be compared with “Increases user confidence and comfort”.

Concluding, the Usability Benefits Hierarchy offers a useful systematic framework of usability attributes, which is not too detailed to use when evaluating usability only. It captures nearly all aspects of other definitions and is therefore most useful within the scope of this thesis.

### Table 1: Usability Benefits Hierarchy

<table>
<thead>
<tr>
<th>Increases individual user effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expedites routine performance</td>
</tr>
<tr>
<td>Accelerates error-free portion of routine performance</td>
</tr>
<tr>
<td>Reduces the impact of routine user errors (slips)</td>
</tr>
<tr>
<td>Improve non-routine performance</td>
</tr>
<tr>
<td>Supports problem-solving</td>
</tr>
<tr>
<td>Facilitates learning</td>
</tr>
<tr>
<td>Reduces the impact of user errors by lack of knowledge (mistakes)</td>
</tr>
<tr>
<td>Prevents mistakes</td>
</tr>
<tr>
<td>Accommodates mistakes</td>
</tr>
<tr>
<td>Reduces the impact of system errors</td>
</tr>
<tr>
<td>Prevents system errors</td>
</tr>
<tr>
<td>Tolerates system errors</td>
</tr>
<tr>
<td>Increases user confidence and comfort</td>
</tr>
</tbody>
</table>

### 2.3 Gaps between software engineering and usability engineering

Usability has its own experts: usability engineers. Usability engineering focuses specifically on designing a system that people find usable and will use (Ovaska, 1991). The Association for Computing Machinery (ACM) has defined usability in the following way: “Usability engineering, also known as human-computer interaction engineering, is a discipline concerned with the design, evaluation and implementation of interactive computer systems for human use and the study of major phenomena surrounding them”.
Over the years, usability engineering has become an important field within software engineering, but as both disciplines have developed separately, cooperation between the two communities has proven to be difficult. There are many reasons why this cooperation is hard, but two factors are the most important. At first, there is a different view on where the development focus lies. Software engineers tend to use a technology-driven approach, as they try to solve the technical difficulties faced when implementing a given functionality, where usability engineers use a more user-centered approach in solving the problem of which functionality should be provided in what way to different groups or users. Secondly, communication between the two fields can be hindered because different terminology is used for the same concepts, or even worse, the same terms are used for different entities (Campos, 2003). Table illustrates the different interpretations or definitions of several concepts between usability and software engineers:

Table 2: Different views on concepts used in both usability and software engineering

<table>
<thead>
<tr>
<th>Concept</th>
<th>Usability Engineers</th>
<th>Software Engineers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions (ISO 9126). Usability engineers are interested in any user-related aspect of a software product. In fact, these aspects give the field reason for existence.</td>
<td>Usability is one of the quality attributes that determines the overall quality of a software product. All those quality attributes need to be addressed and evaluated during software design and maintenance. For software engineers technological issues are the first to be coped with before any product can work. Usability is a quality attribute that can be addressed later during the design stage.</td>
</tr>
<tr>
<td>Scenario</td>
<td>A description of the situation in which a certain appliance or – in the case of computing – application can be used. The emphasis lies on user behavior when interacting with an application. The technology resulting in the desired application behavior is regarded as a black box.</td>
<td>A narrative describing a set of user activities, from which defined input and output software states and the instruction needed for the transition can be derived, resulting in. The instructions needed have limited negative effects on: - Performance (use of memory and processing power) - Security (user activities should not compromise data integrity) - Cost of development effort.</td>
</tr>
<tr>
<td>Requirement</td>
<td>A singular documented need of what a particular product or service should be or do. As a usability engineer's focus is primarily on the user, the requirements he will identify are strictly functional.</td>
<td>SE engineers use the same definition for requirement as their HCI fellows. However, software engineers do also need to take non-functional requirements into account, such as system performance and memory use. On the other hand, software engineers lack knowledge of techniques that allows for more complete elicitation of functional requirements.</td>
</tr>
<tr>
<td>Architecture</td>
<td>The fundamental organization of a system embodied in its components, their relationships to each other and their interaction to the user; and the principles guiding its functional design and evolution (derived from IEEE definition in order to illustrate the primary focus of each discipline).</td>
<td>The fundamental organization of a system embodied in its components, the interfaces to each other and the output they generate; and the principles guiding development, testing and maintenance (derived from IEEE definition).</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Pattern</td>
<td>An interaction design (ID) pattern is a general repeatable solution to a commonly-occurring usability or accessibility problem in interface design or interaction design. HCI engineers only focus on interaction, whereas software engineers focus on software design.</td>
<td>A design pattern is a general repeatable solution to a commonly occurring problem in software design. An architectural pattern expresses a fundamental structural organization schema for a software system (Avgeriou, 2005).</td>
</tr>
<tr>
<td>Module</td>
<td>An entity within the application that can be added to or removed from that application and offers specific functionality to the user.</td>
<td>An entity within the software architecture that can be added or removed and has interfaces with other entities within the architecture.</td>
</tr>
</tbody>
</table>

 Somehow it seems that usability engineers come from Venus, and software engineers from Mars. Usability engineers tend to regard software engineers as cold technophiles with no interest in human factors, and software engineers tend to regard usability engineers as informal hippies with fuzzy methods and whose ideas cannot be captured in clear code. Software architects can complain about HCI experts ‘dumping’ all kinds of usability requirements at the architect, and tell them to just build it. On the other hand, HCI experts often have the idea they are not understood by the software architect. All in all, the usability engineer will end up thinking “why can’t we?” instead of the famous “because we can!”, and the software engineer will end up shrugging “why should we?” Bringing those groups together is a complex but necessary task.

Filling the gap between the usability engineering and software engineering communities has several benefits (Folmer, 2004). First, the quality of software is to a large extent dependent on usability (ISO 9126-1). Software products that are usable, and are thus of higher quality, will lead to more productive and satisfied customers. Secondly, although usability engineering may not be directly beneficial to the developer’s organization, usable software leads to more productive customers and hence improves the reputation of the software developer, which in turn will have positive effect on sales. A third benefit is minimizing engineering costs. Most software projects do not meet their deadlines because of issues concerning usability engineering (Nielsen, 1993). After using testing, there are often requests for interface changes and overlooked tasks come to light, causing software projects to overrun in costs. The minimizing of costs works in two ways: architecting for usability is not only aimed at decreasing the number of change requests, but also on diminishing the cost of individual changes. Since it is practically impossible to elicit all
change scenarios, minimizing the effort needed to make changes concerned with usability is the biggest challenge in filling this gap between software engineering and usability engineering.
2.4 Separating the User Interface from application functionality is not enough

In order to minimize the cost of changes concerned with usability, software architects have dealt with usability primarily as a problem in modifiability (John et al., 2004). In the 80’s and 90’s, it was assumed that the best solution was to simply separate the presentation from the dialog and the application. One of the patterns that implements this is the Model-View-Controller pattern (MVC). The MVC-pattern simplifies modifying the user interface (UI) by separating it from the functionality (which is covered in the controller and model components) of the application. This facilitates iterative UI design. Of course, accommodating iterative design is desirable as most changes in UI are made only after user testing. The problem is however, a good UI is not the only factor contributing to usability. For example, adding the ability to cancel a command that takes more than 1 second affects the UI only by having to add a cancel button at the right place, but the functionality of the application needs to be changed dramatically to support this. This example will be discussed extensively in the Previous Work section.
3. Previous work

It has only been for a few years now that the relationship between usability and software architecture has received considerable attention from research groups. There are two main groups active on this field: The Usability & Software Architecture (U&SA) Group of the Software Engineering Institute (SEI) at Carnegie Mellon University, and Software Architecture that supports Usability (STATUS), a European project which is part of the EU’s Information Society Technologies Program.

3.1 The Usability & Software Architecture Group

The U&SA Group have used a bottom-up approach in investigating the relationship between usability & software architecture. In (Bass et al., 2001) they have gathered a number of usability scenarios that are architecturally sensitive, and they provided them with architectural patterns that can be used to implement those aspects of usability. In an attempt to gain insight in the relationship between usability and software architecture, they have assigned each scenario to what they call the usability benefit hierarchy (shown in Table 1) and the software engineering hierarchy. Combining these hierarchies, they have built a Benefit and Mechanism Matrix (Bass et al., 2001). The purpose of that matrix is to be able to provide some help evaluating architectures for usability, or to see how much architectural impact an additional usability scenario has on an existing system.

<table>
<thead>
<tr>
<th>Table 3: The 26 usability scenarios named in (Bass et al., 2001)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregating Data</td>
</tr>
<tr>
<td>Aggregating Commands</td>
</tr>
<tr>
<td>Canceling Commands</td>
</tr>
<tr>
<td>Using Applications Concurrently</td>
</tr>
<tr>
<td>Checking for Correctness</td>
</tr>
<tr>
<td>Maintaining Device Independence</td>
</tr>
<tr>
<td>Evaluating the System</td>
</tr>
<tr>
<td>Recovering from Failure</td>
</tr>
<tr>
<td>Retrieving Forgotten Passwords</td>
</tr>
<tr>
<td>Providing Good Help</td>
</tr>
<tr>
<td>Reusing Information</td>
</tr>
<tr>
<td>Supporting International Use</td>
</tr>
<tr>
<td>Leveraging Human Knowledge</td>
</tr>
</tbody>
</table>

After working with these architecture design patterns in practice, it became clear that these solutions were too general to fit in existing architectures (Adams et al., 2005). It proved too hard to see where the pattern might fit in, or the pattern was not suitable for the specific architecture. A new approach to solve this problem is the use of a template (Table 3), which contains the context, the problem, a general and a specific solution to a usability scenario (John et al., 2004). At first, we must take a look at the forces that influence the software architecture of a system. Systems, especially in an industrial or commercial
setting, are designed to provide benefits to the organization that aims for efficiency, absence of error, creativity and job satisfaction. It is the organization that forces its users to perform tasks in order to realize these benefits. These tasks define the functionality of the system, and therefore they impact also the general responsibilities of the system concerning usability. The user is of course the one who actually interacts with the system, and his desires and capabilities also have impact on these general responsibilities. For example, determining a comfortable scrolling speed is largely affected by the desires and capabilities of the user. At last, the state of the software itself influences the general responsibilities. Software may crash, or network connections broken, so recovering to a previous state is an important usability responsibility caused by the state of the software. Bringing these three forces together, it will produce a general usability problem and a set of responsibilities. Combining this with previous design decisions, a specific solution can be produced by assigning the responsibilities to existing components.

### Table 4: Usability supporting architecture pattern (USAP)

<table>
<thead>
<tr>
<th>Name: the name of the pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability Context</strong></td>
</tr>
<tr>
<td><strong>Situation:</strong> A brief description of the situation from the user’s perspective that makes this pattern useful</td>
</tr>
<tr>
<td><strong>Conditions on the situation:</strong> Any conditions on the situation constraining when the pattern is useful</td>
</tr>
<tr>
<td><strong>Potential usability benefits:</strong> A brief description of the benefits to the user if the solution is implemented. These benefits are expressed by using the Usability Benefits Hierarchy described in table 1.</td>
</tr>
<tr>
<td><strong>Problem</strong></td>
</tr>
<tr>
<td>Forces exerted by the environment and the task: Each row contains a different force.</td>
</tr>
<tr>
<td>Forces exerted by human desires and capabilities: Each row contains a different force.</td>
</tr>
<tr>
<td>Forces exerted by the state of the software: Each row contains a different force.</td>
</tr>
<tr>
<td>Responsibilities of the general solution that resolve the forces in the row.</td>
</tr>
<tr>
<td><strong>General solution</strong></td>
</tr>
<tr>
<td>Forces that come from prior design decisions</td>
</tr>
<tr>
<td>Allocation of responsibilities to specific components</td>
</tr>
<tr>
<td>Rationale justifying how this assignment of responsibilities to specific modules satisfy the problem</td>
</tr>
<tr>
<td><strong>Specific solution</strong></td>
</tr>
<tr>
<td>Component diagram of specific solution</td>
</tr>
<tr>
<td>Sequence diagram of specific solution</td>
</tr>
<tr>
<td>Deployment diagram of specific solution (if necessary)</td>
</tr>
</tbody>
</table>

### 3.1.2 Example: Canceling Commands

To demonstrate how this template is used, an example will be presented. Consider the usability scenario Canceling Commands:
3.1.2.1 Situation description

A user invokes an operation, and then no longer wants the operation to be performed. The user now wants to stop the operation rather than wait for it to complete. It does not matter why the user launched the operation. The mouse could have slipped. The user could have mistaken one command for another. The user could have decided to invoke another operation. For these reasons (and many more), systems should allow users to cancel operations.

3.1.2.2 Conditions on the situation

A user will only want to cancel a command if a command takes longer than a certain time (e.g. 10 seconds). With only short-running commands it is enough to support only undo. If the decision is made to support canceling commands, the architect may also consider adding a progress indicator and supporting the prediction of the duration of that task, which are part of the Observing System State and Predicting Task Duration scenarios.

3.1.2.3 Potential usability benefits

Supporting the ability to cancel commands has several usability benefits. A user can correct an erroneous click faster if he doesn’t have to wait until the command has finished, so it will increase routine performance. A user will also be less reluctant to start a long-running command if he knows he can always cancel it, so it also improves problem-solving. All these things increase user effectiveness. The impact of system errors is reduced by allowing the user to cancel a command that was jammed because it was not working properly. Hence, a user will be more confident when using the system, because the fear of choosing the wrong command will be less.

3.1.2.4 Formulating the forces that cause the problem and their solutions

For this scenario, we try to find a general solution by formulating the forces that call for the need of the scenario itself:

*Forces exerted by:*
- The environment of the task:
  - Networks are sometimes unresponsive
- Human desires and capability:
  - Users slip or change their mind
- The state of the software
  - Software is sometimes unresponsive

*General responsibility (R1):*
- Must provide a means to cancel a command
Now that we have formulated the general responsibility, we must continue to the next responsibility to be formulated:

*Forces exerted by:*
- Human desires and capability:
  - Users have to communicate their intentions to the system
- The state of the software:
  - Software must receive an action from the user to do something

**Responsibility R2:**
- Provide a button, etc., to cancel the command

The next step:

*Forces exerted by:*
- The environment of the task:
  - No one can predict when the environment will change
- Human desires and capability:
  - No one can predict when the user will want to cancel a command

**Responsibility R3:**
- Must always listen to cancel command or environmental changes

**Responsibility R4:**
- Must always gather information that allows for recovery to the state before the execution of the command

An overview of all the responsibilities can be found in (John et al., 2004). Some of the responsibilities are conditional, which means they are only necessary in certain cases. As the list shows, it is important not to overlook details. Making sure that enough resources are free to perform the action is an example of this. It is also required to inform the user if the action (in this case: cancel) taken can or cannot be completed. Furthermore, collaborators must be informed of any change of state in the system.

The work by the U&SA Group is promising, considering the fact that it is very straightforward to put into practice. Its foundation is perhaps disputable on a theoretical basis, but it does offer a fairly easy and comprehensible way to address usability in both design and maintenance of software systems. The SEI have developed tactics for quality attributes such as modifiability and performance (Bachman et al., March 2003), which also makes use of the principle of responsibilities. Developing tactics for usability based on USAP’s is still in a preliminary phase (Bass et al., 2005). For further reading about these developments, please refer to future work in paragraph 7.2.4.

### 3.2 A theoretical approach: the usability framework

A European project on the relationship between usability and software architecture is Software Architecture that supports Usability (STATUS). Their aim is to investigate the relationship between software architecture and the usability of the resultant software
system. Their approach is not bottom-up, but top-down, which means that they first tried to capture theoretically the different layers between usability, its attributes, and the design decisions made at architecture level. The result is a framework, which is shown in the figure below.

Figure 1: Relationships between elements in the framework (Folmer and Bosch, 2004)

In this framework, specific design decisions are the implemented solution of an architecturally sensitive usability pattern. These patterns are solutions to fulfill usability properties. Examples of usability properties elicited by STATUS are error management, consistency and user control. For example, the undo pattern is a solution which contributes to user control. Usability properties are the guidelines to improve usability indicators. For example, consistency is one of the properties that will improve the time needed for a new user to learn how to use the system. Usability indicators have the advantage that they can be actually measured, like, for example, time to learn or the number of errors in per time unit. Each indicator contributes to one or more usability attributes. The four usability attributes identified by STATUS are learnability, efficiency, reliability and satisfaction.

3.2.1 Usability properties

The usability properties elicited by STATUS are (STATUS D.2, 2002):

- Providing feedback
- Error management
  - Error prevention
  - Error correction
- Consistency
  - UI Consistency
  - Functional Consistency
- Guidance
- Minimize cognitive load
- Explicit user control
- Natural mapping
  - Predictability
  - Semiotic significance
  - Ease of navigation
3.2.2 Usability patterns

As stated before, these properties are related to usability patterns. Like the U&SA Group, STATUS have elicited a number of usability patterns, comparable to architecturally-sensitive usability scenarios. The relationship between each pattern and usability properties is documented in the STATUS documentation (STATUS D2). The list is shown below:

- Different languages
- Alert
- Status indication
- Shortcuts
- Form/Field validation
- Undo
- Context-sensitive help
- Wizard
- Standard help
- Tour
- Workflow model
- History logging
- Provision of views
- User profile
- Cancel
- Multi-tasking
- Commands aggregation
- Actions for multiple objects
- Reuse information
- Different access methods

How can we compare this list to the scenarios collected by U&SA? The following things can be noted:

- Different Languages (Supporting International Use), Form/Field Validation (Checking For Correctness), Undo, Cancel, Commands Aggregation, and Reuse Information are patterns that are similar to U&SA scenarios on a one-to-one basis.
- Status Indication is in fact the same as U&SA’s observing system state, but STATUS also considers Predicting Task Duration as part of Status Indication.
- Shortcuts are not covered by U&SA according to STATUS, but it can be seen as generalization of Navigating Within A Single View.
- Context-Sensitive Help, Wizard, Standard Help and Tour are all instantiations of U&SA’s Provide Good Help. STATUS regard the latter as a usability property called Guidance.
- The workflow model, which means providing different only those tools and tasks that meet the needs of specific users, is a generality of Working in an Unfamiliar Context.
- History logging for users is not named by U&SA. U&SA’s Evaluating The System provides information about usage for usability engineers. However, both scenarios need logging, so they are in fact related. History logging is especially important when the user has to set a whole collection of variables. The user then can always rely on sets that have been used in the past.
- Provision of views is related to Making Views Accessible and similar to Supporting Visualization.
- Multi-tasking is a generality of Using Applications Concurrently and Supporting Multiple Activities in such a way that if the latter are supported, Multi-Tasking is.
- Different Access Methods is a generality of Maintain Device Independence.

Some scenarios of U&SA are not named as a pattern by STATUS:

- Recovery From Failure lies outside the scope of usability, as, according to the scenario, the failure is caused by system error and not user error.
- Retrieving Forgotten Passwords is regarded by STATUS as a security issue.
- Leveraging Human Knowledge is not seen as a pattern, but shows strong resemblance with the usability property UI Consistency and to a lesser extent Natural Mapping.
- Working At A User’s Pace can be seen as part of the usability property accessibility.
- Supporting Comprehensive Searching is regarded by STATUS as a functional requirement for specific applications, and therefore not as a usability pattern. However, it does influence the usability property Explicit User Control, and through that the usability attribute Efficiency.
- Operating Consistently Across Views and Modifying Interfaces are part of the overall developing process and are therefore not regarded as usability patterns according to STATUS.

There is one usability pattern which is not named by U&SA: User Profile. User profiles can serve to make the system adaptable to the needs of various users. This may concern screen preferences, specific tasks and options that are shown, etc. Its architectural impact lies in the fact that there needs to be a component that stores the user information.

The term ‘pattern’ in ‘usability pattern’ is perhaps a bit misleading. It suggests that it contains a context, problem and a solution (Avgeriou and Zdun, 2005). However, the usability patterns as described by STATUS do not offer any solution. Instead, STATUS presents a lower abstraction layer with ‘design solutions’ that could support on or more of the ‘usability patterns’ (see also Figure 1). Therefore, it is important to view ‘usability patterns’ not as patterns but more as a ‘concern’.
3.2.3 Scenario based Architecture Level Usability Analysis (SALUTA)

STATUS have developed a technique for assessing software architectures for usability, called SALUTA (Folmer et al., 2004). The method used in SALUTA is very similar to other assessment techniques for software architectures, like e.g. ALMA (Architecture Level Modifiability Analysis) (Bengtsson and Bosch, 1999). It consists of five steps:

1. Determine the goal of the assessment
2. Create usage profile
3. Describe the provided usability
4. Evaluate each scenario in the profile
5. Interpret the results

3.2.3.1 Determine the goal of the assessment

The first step concerns the goal of the assessment. It can either be to:

- Predict the level of usability support for a given architecture
- Assess risks, which means detecting usability issues for which the architecture is inflexible
- Compare two or more candidate software architectures and select the optimal candidate which has the best support for usability

3.2.3.2 Create usage profile

A usage profile describes usability requirements in terms of a set of usage scenarios. A usage profile is created using the following steps:

1. Identify the users
2. Identify the tasks
3. Identify the context of operation
4. Create attribute preference table
5. Scenario selection
6. Scenario prioritization

As users perform tasks in a specific context, the first three steps are rather obvious. The attribute preference table serves to create a ranking in importance of the usability attributes of each scenario. An example is show in Table 4.

Table 4: Scenario profiles

<table>
<thead>
<tr>
<th>No</th>
<th>Users</th>
<th>Tasks</th>
<th>Context</th>
<th>Satisfaction</th>
<th>Learnability</th>
<th>Efficiency</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Content Administrator</td>
<td>Edit object</td>
<td>Helpdesk</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
Scenario selection is done to create a set of those scenarios that are important to the assessment, depending on the goal of the assessment. Additionally, weights can be assigned to the selected scenarios. For example, if the goal is to predict the level of usability, only the scenarios that occur most frequently are selected and prioritized as such.

3.2.3.3 Describe the provided usability

In this step, information about the software architecture needed to perform the analysis is gathered. Software architectures can be described in different ways, but for this assessment, only information that is related to usability is needed. The usability framework can be used to describe the usability the architecture should provide.

3.2.3.4 Evaluate scenarios

The scenarios can be evaluated in different ways. The patterns defined by STATUS can be used for heuristic evaluation. If design decisions are documented during architecture design the assessment can be more detailed. Use case maps (UCM) are even more useful because they capture behavioral and structural aspects of systems at a high level of abstraction.

3.2.3.5 Interpret the results

As a last step, the results are interpreted depending on the goal of the assessment.

3.3 A view on the approaches of U&SA and STATUS

Both U&SA and STATUS have made considerable contributions in investigating the relationship between usability and software architecture. The approach of U&SA is top-down: through the elicitation of architecturally-sensitive usability scenarios they have tried to link the scenarios with both usability and software architecture. For this, a benefit hierarchy framework has been created for both fields, and these have been combined in a matrix where each scenario has been filled in. This matrix serves to see if architectures do support the usability requirements and to see what it takes to implement a usability requirement later on. Furthermore, U&SA have developed a pattern approach to implement architectural changes for usability requirements that have to be added to an existing system. The key of this pattern is to formulate responsibilities needed to fulfill the particular scenario. Forces that influence these responsibilities are the state of the software, the environment of the task and human desires and capabilities. A set of responsibilities leads to a general solution, and by assigning these responsibilities to existing or new components, a specific solution is created.
The STATUS group have followed a different approach, which is bottom-up. They have developed a framework, which connects a definition of usability through different layers to design decisions taken in architecture development. This usability framework serves as a basis for their later work. By studying existing literature and using their own experience they have elicited usability attributes, indicators, properties and patterns, which form consecutive layers in the usability framework. Using the framework, a scenario-based analysis method for evaluating usability at architecture level (SALUTA) has been developed, using a methodology similar to other analysis techniques based on scenarios, such as for modifiability.

Comparing the approaches of both groups, it can roughly be concluded that U&SA are practically oriented and STATUS more theoretically. Software developers probably need less time to see the benefits they could have from the U&SA material than from STATUS. They can use the list of scenarios simply as a checklist, to see whether some commonly known usability issues have been overlooked. The worked out solution patterns for some specific scenarios provide an excellent way to retrofit these scenarios after deployment of the system. However, their list of scenarios is not complete, and furthermore it is diverse in level of abstraction. For example, some scenarios are instances of others, like for instance Predicting Task Duration and Observing System State. Others scenarios are of a higher level of abstraction, and maybe should not be called a scenario, like for example Modifying Interfaces. This difference in abstraction levels is also seen with the Parameter History Logging USAP presented in chapters 5 and 6 in relation to different other USAP’s. Other relationships between various USAP’s are discussed in paragraph 6.6. This may hamper the development of a framework that captures different USAP’s in order to measure or make some sort of qualitative assessment of usability. Such a framework is being developed by US&A and elements of this are discussed in the future work chapter in paragraph 7.2.4.

The work of STATUS has a better theoretical foundation, and can therefore be promising. The usability framework is an interesting concept, solving at least the issue of different abstraction levels in usability issues, but it still needs to be developed in a more detailed way, for example by substantiating the relationships between the different layers within the framework. Their usability evaluation method, SALUTA, is very much based on other methods that have been developed in recent years, revealing that it has been done by roughly the same team of researchers in the software architecture field. So far, SALUTA is very much dependent on the available documentation and the experience of the software architect and the results of such an analysis is more of less rather speculative. Because of this limited practicality so far, chapters 5 and 6 presenting the Parameter History Logging usability scenario and its implementation discuss the work of STATUS only to a small extent. During the coming years, SALUTA will probably undergo more refining, perhaps by making the usability framework more detailed. Future work concerning the usability framework is discussed in paragraph 7.2.3.
4. Usability in automation systems

Automation systems are electronically computerized systems used to control machines, processes and procedures. Because automation systems are used in a different context than desktop applications, they might also have different usability concerns. A big difference between the two is that desktop applications do not control objects that exist for real. Automation systems therefore have their own properties concerning safety, which has its effect on usability. In order to get some ideas of usability in automation system, a brainstorming session involving professionals from forestry and mining industries together with usability experts from the Technical University of Tampere was held. The goal was to identify the most important usability scenarios taken from the list of U&SA within the domain of automation systems. Apart from identifying those, other usability scenarios that are important issues in their field have been identified.

4.1 Characteristics of automation systems

A lot of automation systems are embedded systems. Embedded systems are specialized computer systems that are part or a larger system or machine. Typically, an embedded system is housed on a single microprocessor board with the programs stored in ROM. An important characteristic of automation systems is that they are designed for specific functionalities only.

Usually, the tasks assigned to an embedded system are not processor-intensive. Reproducing embedded systems is, of course, more expensive than desktop software. Designers try to save money and energy consumption, which is especially important in portable devices, by not only using a slower processing unit, but also by simplifying the whole architecture of such a system.

The characteristics discussed above influence the way these systems should be designed for usability. Automation systems communicate with their users not only through mouse and keyboard, but also through buttons, light indicators, audible feedback, and user interfaces (either text or graphic), depending on the context of the system. It is important to realize that automation systems have very specialized functionalities, so user interfaces design should be simple, serving only specific tasks. This is different from user interfaces designed for desktop applications, which are used in a broader context and therefore need more complex interfaces. Also, some automation systems are used in a different environment than a quiet office, so the environmental context in which the system is used, should drive the design. For example, sound feedback in a noisy industrial environment is of course pointless.

Desktop computer applications are often rich in modes. The application can be in many different states, and these states all require a different kind of interaction. Think for example of beginner and advanced modes in some applications. In automation systems, it is best to limit the number of modes, if possible. Screens are often limited in size, making
navigation harder. When having to browse through submenus, a user is easily lost, so it’s important to always let the user know how he can return to the opening screen. If multiple modes are needed, it should always be clear how to return to the default mode.

### 4.2 Primary usability scenarios for automation systems

The brainstorming session was organized in the following way: the participants were divided into groups, so that each group consisted of at least one individual from the fields of mining, forestry and one usability expert. After the session, the results of each individual group were presented and compared. The following usability scenarios from U&SA were found to be the most important:

- **Checking for Correctness:**
  If user input is checked before an operation starts, this scenario is fulfilled. In desktop applications this often means checking data consistency before it is sent to a database. In automation systems, it can also be to check the state of any machine part before something starts operating.

- **Evaluating the System:**
  This scenario means including evaluation points in the software in order to track the use of it, so it will bring usability issues to light and improve future versions.

- **Recovering from Failure:**
  Systems should be designed in such a way that the amount of work that is lost in case of failure is minimized. The system should also automatically restore itself to the most recent state it can return to.

- **Observing System State:**
  This scenario is most likely the most important usability issue in automation systems. The user must at any time be informed in a very straightforward way about the state of the machine. Communicating by means of alert messages is part of this scenario.

- **Supporting Visualization:**
  There should be different ways to visualize the data that is available. In automation systems, this is often needed when there is a lot of measurement data that can be monitored.

These five scenarios were considered to be the most important ones in the domains of forestry and mining. Other scenarios that were also regarded to have at least some specific importance to automation systems are listed in the next subsection.
4.3 Secondary scenarios for automation systems

- **Aggregating Data & Commands:**
  In automation systems, large amounts of data are often collected from sensors. In other situations, the same operation must be performed on multiple objects, or a specific sequence of operations must be performed in a particular order. All these needs ask for supporting this scenario.

- **Maintaining Device Independence:**
  Data is available using different devices. Changing a part of the machine will not affect the software controlling other devices.

- **Providing Good Help:**
  Help should be available at all times and should be context-sensitive.

- **Leveraging Human Knowledge:**
  New versions should be designed in such a way that users of older versions feel acquainted with it. This becomes particularly important when users have novice skills and do not use the system very often.

- **Predicting Task Duration:**
  Users want to be informed about the possible duration of long-running tasks, so they do other things at the same time. Without this, users have to rely on their experiences and guess the time needed for e.g. warming up a machine.

- **Verifying Resources:**
  Before starting an operation, it should be verified that every condition that is needed for that operation is met. An example of this in automation systems is for instance that starting and stopping of a machine.

- **Reusing Information:**
  Data that comes from measurements should transferable to other applications.

- **Modifying Interfaces:**
  Different classes of users might need different or even personalized user interfaces. It might even increase safety when some users cannot access all parts of a user interface, for example when lots of advanced-level parameters have to be set.

4.4 Other usability issues

During the brainstorming sessions, other usability issues came to light. Most of these issues were specific cases related to existing scenarios, such as start and stop sequences of machines. Starting and stopping machines should be preceded by checks if the machine can
be started or stopped at all, and if more machine parts are involved, some sequence may have to be defined. This scenario resembles Verifying Resources, and will not be considered further.

4.4.1 Parameter history logging: a new scenario

Another issue that came up deals with changing parameters. In both mining and forestry machines are used that are configured by a large number of parameters. These parameters need to be changed because of various reasons. For instance, seasonal differences influence the way a tree should be grasped by a boom. If a new part is installed to a machine, calibrating is needed to get the right parameters. In other words, getting the parameters right is an important issue in automation systems. One of the ways to guide the user when setting parameters is to offer him information about previously used settings. This is a new usability scenario, which has impact on both the user interface and the database structure of a system, and therefore it will be presented as a usability supporting architecture pattern in the next chapter.
5. The Parameter History Logging usability scenario

One of the patterns defined by STATUS is that of history logging (STATUS D.2, 2002). According to them, history logging is the recording of a log of actions that the user (and possibly the system) takes that allows the user (or the system) to look back over what was done previously. This pattern is not named by the U&SA Group, although it has some similarities with the Evaluating the System scenario. In that scenario, the idea of recording user actions is aimed at improving the system by analyzing data about system usage. According to STATUS, Evaluating the System cannot be seen as a usability pattern, because it is part of the general development process and has less to do with architectural design itself. STATUS’ opinion on this is plausible. Recording user actions only as a means to identify usability issues does not improve the usability of that system directly. It can only contribute to improve the usability of future versions of the same system.

Recorded user actions can give the user useful information about what he or other users did before, and can therefore serve as a piece of advice in some situation. When using an application involves a lot of navigation, keeping a history of visited pages will greatly increase usability. Examples of such applications are internet browsers, online dictionaries, encyclopedias or other sorts of look-up programs, and large manuals or help documents. Most internet browsers store their history somewhere so it can be viewed later, but a lot of other applications only keep a history log until the program is terminated by the user.

There are also other instances of history logging. In many industrial applications, devices or items can be configured according to a number of settings. When the user wants to change these settings, he might want to view some information about previously used settings. Recalling this information is harder than to recognize them, so offering the possibility of showing previously used settings will reduce the user’s cognitive load. This form of history logging can be called parameter history logging. It is based solely on recording values entered by the user. Other user actions, such as mouse and keyboard gestures, do not need to be recorded for this.

Remembering filled-in values, which is a feature of a lot of internet browsers, is a well-known example of parameter history logging. It helps personalizing web pages and the user can save time when filling out forms on the web. It also remembers a number of passwords that a user would otherwise long forget because he is subscribed to so many services that it is difficult to remember all log-in information. Of course, there is a security issue here, but all browsers ask the user whether to store this data or not.

5.1 Relationship with other usability scenarios

Showing previously used settings is not named by the U&SA Group as an architecturally-sensitive usability scenario. It is however a usability scenario and it also affects the overall architecture of the system by creating the need of a repository to store data about these settings. As mentioned before, parameter history logging has a relationship with the Evaluating the System usability scenario. It also has a relationship with Undo, because the user might want to use previously used settings in order to restore an old situation when a
set of parameters cause errors. Another related scenario is Checking for Correctness, because both scenarios deal with input fields. Using the same pattern as the U&SA Group, the scenario is elaborated to a usability scenario architecture pattern below.

5.2 Parameter history logging as a usability scenario architecture pattern (USAP)

5.2.1 Situation description

A user has to set a number of parameters for e.g. a configuration. The user wants more information about previous settings that have been used, default values etc. These values have to be stored somewhere. A system administrator should be able to store and edit the parameter settings. Settings are automatically stored when saved, or only kept there when the applied settings cause no error, so the system can always return to that state after reboot if it doesn’t run properly. The user should be able to see the default values next to the edit boxes. The user can also choose to view different sets by choosing them from some menu.

5.2.2 Conditions on the situation

Parameter history logging is useful when the application has configurable items with a reasonable amount of parameters. Users can often not recall settings that have been used in the past, and it might well be possible that these settings have been put into effect by other users. It does not really matter how often these parameters are being changed. If changed often, the user will save time when copying values from previously used parameters sets. If changes are rarely made, it becomes more and more a guidance tool because users may have forgotten all about things they have done long ago, or those actions have been done by other users in the past.

There are different options when implementing this scenario. When something can be configured according a large number of parameters, parameters have to be grouped and organized in different screens. This is the responsibility of the interface designer. However, when loading historical data it does raise the complexity of the situation. Parameters can have different relations to each other, and also groups of parameters can have hierarchical relationships. For example, a set of parameters might only have values when another parameter has been set. When loading previously used values, extra complications may arise because of this.
5.2.3 Allocating the scenario to the Benefit and Mechanism matrix

In order to allocate this scenario to the Benefit and Mechanism matrix, it should first be allocated to the Usability Benefits Hierarchy and the Software Architectural Tactics Hierarchy as described by (Bass et al., 2001). Allocating it to the Usability Benefits Hierarchy defines the way in which this scenario supports usability. Allocation to the Software Architectural Tactics Hierarchy defines the architectural mechanisms needed to implement the scenario.

5.2.4 Allocation to the Usability Benefits Hierarchy

- Increases individual user effectiveness
  - Expedites routine performance:
    Parameter history logging accelerates the error-free portion of routine performance by providing guidance to the user through knowledge from history. The probability that the user will provide a non-valid value for a parameter will be reduced.
  - Improves non-routine performance:
    Parameter history logging supports problem-solving by giving the user more clues about what to do. It also facilitates learning by letting the users understand the possible effects of various settings.
- Increases user confidence and comfort
  - The user will feel more confident when it receives some clues about the possible effects of previously used settings when having to set a large number of parameters, which is a task that requires a lot of thinking, and therefore error-prone.

5.2.5 Allocation to the Software Architectural Tactics Hierarchy

- Separation
  - Data from the view of that data:
    Every set of parameters needs to be stored somewhere, and be accessible in different ways.
- Indirection
  - Data:
    Because it is advisable to store the settings in a database, the communication with that database will very likely be done by making use of some form of middleware.
  - Recording:
    Parameter history logging involves recording settings.
- Models
5.3 Formulating Responsibilities

As described in the Previous Work section, the U&SA Group have developed a method to retro-fit a usability scenario to an existing system. At first, the forces that create the need of this scenario are identified. These forces result in the identification of responsibilities that need to be fulfilled when implementing the scenario. The first step expresses the need for the scenario itself:

- **Forces exerted by the environment of the task:**
  The software controls a device or item that is configured through a sizable number of parameters.
- **Forces exerted by human desires and capabilities:**
  Users cannot remember all details of previously used settings that involve a lot of parameters.
- **Forces exerted by the state of the software:**
  The previously used settings were unsuccessful. The user wants to return to settings that caused no errors.

- **Responsibility (R1):**
  The user must have the possibility to be informed of the parameter values of previously used settings.

Now that the first responsibility is formulated, new problems arise before the whole scenario can work:

- **Forces exerted by human desires and capabilities:**
  Users have to communicate their intentions to the system.
- **Forces exerted by the state of the software:**
  The system must be informed by the user that it has to show previously used parameter values.

- **Responsibility (R2):**
  A viewing option must be provided where the user can choose if and which settings must be shown.
- **Responsibility (R3):**
  Screen real estate must be or be made available for showing the parameter values.

The command issued by the user can either successful or unsuccessful. Both situations need to be communicated to the user:

- **Responsibility (R4):**
  The system loads the data from the database, and shows the values on the screen.

Retrieving the requested data is unsuccessful:
Forces exerted by the state of the software:
Database or middleware may be unresponsive.

Responsibility (R5):
Report error to user if data is not available.

Once the user is shown previously used values, he should be offered the possibility of copying individual or all values of these settings to the input boxes:

Forces exerted by human desires and capabilities:
In order to save time, the user wants to copy individual or all values to the input boxes by a single gesture. These gestures should also be undoable, but this lies outside the scope of this scenario.

Responsibility (R6):
Buttons that allow for copying individual or all values when pressed should be shown on the screen.

Without any storing of settings, no previously used settings can be shown. Responsibilities R7 to R11 deal with saving those settings in some repository.

Forces exerted by the environment of the task:
The user wants to retrieve settings that are used in the past.

Responsibility (R7):
Settings that take effect successfully must be automatically stored, not only as active settings, but also in a database so that they can be retrieved later.

If storage is unsuccessful:

Forces exerted by human desires and capabilities:
User tries to save invalid data.

Forces exerted by the state of the software:
Database or middleware is, for some reason, unresponsive.

Responsibility (R8):
The system should check for correctness before trying to save the data into the database. This checking for correctness is actually an architecturally-sensitive usability scenario named by U&SA, so it will not be further elaborated here.

Responsibility (R9):
The system should report to the user that storage was unsuccessful.

If storage is successful:

Responsibility (R10):
The system should report to the user that storage was successful and settings are put into effect.

In order to prevent the storing of far too many parameter sets, a user with special rights should be able to alter the contents of the repository that holds all the parameter sets:
• **Forces exerted by the environment of the task:**
  Because of frequent altering of settings, the number of sets becomes rather large. Some sets may be even duplicates.

• **Responsibility (R11):**
  A user with special rights should be able to alter the contents of the repository that holds all the parameter sets.

By formulating these responsibilities, a general solution for this usability scenario has been proposed. If we want to retrofit this scenario into an existing system, a specific solution must be developed. To apply this scenario, a case study has been performed to use these general responsibilities in order to create a specific solution for an existing system.

### 5.4 Parameter history logging as a usability pattern in the light of STATUS

The parameter history logging scenario can also be named as an instance of the History Logging usability pattern named by STATUS. A first question already arises then, because can an instance of a usability pattern also be named a usability pattern? Parameter history logging is something that can be implemented because of a design decision, and it has an impact on the design of the architecture. Then what usability properties are related to parameter history logging? At first, it minimizes cognitive load by showing data that a user is not expected to remember. It also improves error management by giving the user advice about what to do. When an option is added that makes it possible to copy all historic values directly to the input boxes, explicit user control is improved. In other words, there are several usability properties that can be associated with parameter history logging, depending on the exact implementation of this scenario.

### 5.5 Parameter history logging example

In order to offer some visual support in presenting this usability scenario, a little implementation is presented here using DHTML (Dynamic Hypertext Markup Language). The code can be found in Appendix A. Figure 2 shows a screenshot of the demo. On the left, three different kinds of HTML input fields (three text fields, a drop down menu and two radio buttons) with the values of Configuration A filled in. To the right display fields show settings belonging to Configuration B. The complete set of Configuration B can be copied from right to left by clicking the ‘Copy All’ button. The result of this is shown in Figure 3. Individual values can be copied by using the ‘Copy’ buttons between the input and display fields. In order to relieve the user’s memory load during the occasionally complex task of adjusting a parameter, an undo button has been added for convenience.

This example is made using DHTML with JavaScript. JavaScript code is generally well interpreted by most browsers (e.g. Internet Explorer, Mozilla Firefox, Opera) and it offers an easy way to present functionality to users without having to install specific client
software. Older client-server solutions had the main drawback that changes in server software often required an upgrade in client software as well, adding to the support cost and decreasing productivity.

Figure 2: The input fields show settings from Configuration A; to the right the loaded settings of Configuration B are shown.

Figure 3: All the values of Configuration B have been copied into the input fields after pressing the 'Copy All' button. Clicking 'Undo' will revert those changes.
6. Applying parameter history logging to an existing system

In order to see how the general solution of parameter history logging can be turned into a specific solution, a case study is presented. Due to a non-disclosure agreement, no explicit details of the system’s architecture can be revealed. Before the specific solution is presented, a usability assessment will be done according to the scenarios of U&SA. The assessment will assist in investigating the relation between various usability scenarios and parameter history logging. Usability assessment with the SALUTA method has been done before (Folmer et al., 2004) for other systems and is considered unnecessary here.

6.1 System description

The system used for this case study is a web application used in the naval transport industry. Its main tasks are to monitor ship cargo, perform measurements, collect information on central computer, and send alarm messages (e-mail or SMS) in certain situations, which can be defined by special users. Information for the measurements comes from a number of devices that are connected to the main computer on board of the ship. Logs are being generated that allow for analysis by office personnel, after this information has been sent from the ship to the main offices. Four different types of users have been defined for this system:

- ship personnel
- service personnel
- office personnel
- system maintenance personnel

Ship personnel can use the devices from a web browser and watch some reports and logs. They do not have the rights to change anything. Servicemen may watch all reports and logs, and they can also configure devices in alarm situations. Office personnel have all the rights of the previous groups, and furthermore the right to change software settings. System maintenance personnel have all the rights, including updating the software.

The main architecture of the system is a three-tiered architecture, which is used often in business applications. The bottom layer is called the platform layer, and holds the operation system and the database server. The middle layer is the abstraction layer, and serves as middleware between the top and bottom layers. At the top lies the application layer. The application itself is also divided into three layers: a GUI, a proxy that handles all the communication between the GUI and the rest of the system, and a number of components which are responsible for communication with the connected devices and the database.
6.2 Usability Assessment

The system will be mapped with the architecturally-sensitive usability scenarios from the U&SA Group. Each scenario that is applicable to the system the support for it is discussed in the following sections. In paragraph 6.3, an implementation of the Parameter History Logging scenario is presented, according to the pattern developed by U&SA. The effects that such an implementation has on other usability scenarios is assessed in paragraph 6.5.

6.2.1 Mapping the scenarios

A number of scenarios named by U&SA are, because of limited necessity, not applicable to this system. These scenarios that will not be discussed with respect to this system are:

- Aggregating Commands
- Canceling Commands
- Using Applications Concurrently
- Leveraging Human Knowledge
- Supporting Multiple Activities
- Working At The User’s Pace
- Predicting Task Duration
- Supporting Comprehensive Searching
- Working In An Unfamiliar Context
- Operating Consistently Across Views
- Making Views Accessible

The other scenarios will be discussed one by one. For each scenario, the scope of the scenario will be shortly explained. For more information, look at the scenario descriptions in Appendix A. The extent to which the system supports each scenario is discussed, and quantified using ‘--’ for no support at all, ‘-’ for limited support, ‘+’ for reasonable support and ‘++’ for full support. Some of the usability issues are supported by the operating system or browser, and could be outside the scope of the system itself. These are marked with an ‘O’.

Aggregating Data (++)

Aggregating data means allowing the user to perform one task on multiple objects at the same time. In this system, there are two ways in which this is supported. First, when updating the software for the devices connected to the system, the user can choose multiple devices to be updated at the same time. Second, when exporting logs to other formats, it is possible to export all logs at the same time. Exporting logs is also discussed in the Reusing Information scenario.

Checking for Correctness (++)
Checking for correctness means validating input values before sending them to the database, in order to prevent useless or corrupt data from entering the database. The GUI component of the system takes care of this, by defining which values are allowed for each input box.

**Maintaining Device Independence (++)**

Maintaining device independence means that the system should be able to work on different platforms and devices, for example mobile devices. Because the system is also accessible with WAP devices, this scenario is supported. The alarms that can be generated can be sent through different media, such as e-mail or SMS.

**Evaluating the System (-)**

Evaluating the system means adding evaluation points in the system by recording user gestures in order to provide usage information to the system developer. There is indeed some logging in the system. All changes made to device configurations are logged. However, it does not really provide clues about possible usability issues.

**Recovering from Failure (+)**

Recovering from failure means limiting the amount of lost work for the user when the system stops running. Information from the devices is collected within a minute buffer, so the data loss in case of failure is limited. Within the web application, changes will be lost if they have not been saved to the database (or file if data is exported), but this is the responsibility of the operating system and browser itself.

**Retrieving Forgotten Passwords (--)**

Retrieving forgotten passwords means offering an alternative way for a user to log on in case he has forgotten his password. These ways include asking a secret question or sending a new password to a verified e-mail address of that user. In this system, there is a hierarchy of user groups which defines the number of rights a user has. Retrieving forgotten passwords is not supported here, but it is probably not even needed here. The users work in the same company, and very likely passwords can be retrieved by just sending an e-mail or making a phone call to the administrator.

**Providing Good Help (-)**

Providing good help means providing user guidance throughout usage. Context-sensitive help is part of this. This scenario is particularly important for applications that are built for unknown future users or a very wide variety of users. This system is however designed for use within a small domain, so the importance of it is much smaller. Its support is only limited.

**Reusing Information (++)**
Reusing information means offering ways to transport data from one application to another. In this system, this is supported by making it possible to export measurement data in a format that is supported by other applications, like for example MS Excel. This scenario is therefore fully supported.

**Supporting International Use (++)**

Supporting international use means being able to use different languages. This scenario is more often necessary than people might think at first. Adding a new language to the GUI is much easier if the architects already anticipated to it. In this system, it is fully supported in the GUI itself, although Finnish is still the main language used.

**Modifying Interfaces (+)**

Modifying interfaces means that the system adapts its user interface to the tasks of that specific user. This system has defined different user groups which have the rights to perform tasks at different levels. This is in fact the only way in which this scenario is supported. Further adaptability of the interfaces is also not needed in this system.

**Navigating within a Single View (-)**

Navigating within a single view means offering various navigation options on the screen when navigating. This especially includes linking screens that have some relation to each other. Windows XP offers such functionality by showing a list of icons of frequently used tasks when navigating. In this system, links could be added to link the screens that show the measurement reports of the various devices and those that can configure the devices. The scenario is not supported, but it might be useful for the system.

**Observing System State (O)**

Observing system state means that the user should always be able to see the state the software is in, as unexpected behavior of software leads to annoyances and discomfort. This is particularly important within systems that have long-running threads. Within this system, system state is reflected either by the screen the user is in, or by process indicators which are part of the browser which is used. Support for this scenario is therefore largely supported by the browser, not by the system itself.

**Supporting Undo (O)**

Supporting undo means that the user is able to undo previous actions. This can be single-level or multi-level undo. The extent to which this scenario is supported is completely dependent on the browser. Both Mozilla en Internet Explorer only support single-level undo.

**Verifying Resources (++)**
Verifying resources means that the system checks if all resources that are needed to perform a specific task are available. This is indeed supported, because any action that involves connected devices first checks if the device is connected properly before running.

**Supporting Visualization (++)**

Supporting visualization means supporting different ways of showing the same data. This scenario is supported when looking at the measurement reports. There are many viewing options which can be specified by the user.

**6.2.2 Conclusions**

Most scenarios are taken care of in this system, especially those who have any relation to parameter history logging. The most important scenario that deserves some more attention is that of Navigating within a Single View. Parameter history logging is not supported. The need for this might be less than for example in forest harvester machines, because the configurations of the attached devices do not change often, but just because they do not change often, parameter history logging can help the user remembering what has been used before. Implementing this scenario into this system is explained in the next section.

**6.3 Implementing parameter history logging**

Now that we have formulated the responsibilities needed to fulfill this scenario, we can map those responsibilities onto the system itself. Parameter history logging needs changes to GUI and database. These components are connected though a GUI Proxy component. Changes to the GUI include the responsibilities R2, R3, R5, R6, R8, R9, R10 and R11. The GUI Proxy is responsible for storing data to the database and informing all subsystems about changes made to the database.

![Figure 2: the relationship between GUI, GUIProxy and DB.](image-url)
6.3.1 Changes to the database

Adding parameter history logging to this system involves only the GUI, the database and the accompanying SQL code. A table should be added containing a number of configurations with date and name. One attribute should be added to the table that holds the configuration values. This attribute holds the key of the configuration the value belongs to. In this particular case, an attribute called ConfigurationID should be added to the DeviceParameterValue table, with a functional key to a new table called Configuration. This table should hold attributes for a name and a date of creation of the configuration.

6.3.2 Changes to the GUI

Now that we can actually store different configurations, the GUI still needs to be adapted to this scenario. A drop-down menu is a good way to present all available configurations when trying to view one. This is an easy thing to do in HTML, and the items should reveal name and date of the configurations that are available in the database. Furthermore, the screen will need an extra column to show the loaded parameter values. This can be achieved by changing the code of the resulting HTML as well. In order to make it easier to quickly copy values from a stored configuration to an active configuration, buttons should
be shown that support this. They can show an arrow which either points left to indicate the value will be copied to that direction, or right so the user can undo his action and the old value will reappear. These buttons can be made functional by using JavaScript. Of course, there are different types of values that can be copied, e.g. plain text or cardinal numbers, but also items from drop-down menus, which involves slightly more coding.

6.3.3 Changes to SQL statements

In order to work with the new table, SQL code must be adapted to the new database scheme. Except the SQL lines that will be needed to load configuration data, the existing SQL lines storing a configuration should now also include a configuration ID for the new configuration.

6.4 Options to implementation

The suggestions made above are just basic for implementing parameter history logging. There are, of course a few options that, depending on the situation, might be convenient to consider.

6.4.1 Parameter hierarchy

Parameters can sometimes be dependent from each other. For example, a certain value for a parameter may only be used if another parameter value is within a certain range. Usually, these properties are hard-coded within a GUI, but if we take modifiability into account, it would be more elegant to design the database in such a way that information about dependent parameters is stored there. For this, an attribute can be added in the parameter value table, containing some logical formula that will evaluate to true if the parameter is actually used. An example of a language that can be used for those formulas is OCL.

Parameter hierarchy can influence the way parameter history logging should be implemented. If dependent parameters are only shown when used, and those parameters are used within a loaded configuration, the system should show them. Also, when copying a ‘mother’ value to an active configuration, the user should be prompted if he wants to copy the dependent values as well. Of course, copy buttons of dependent values should only be active if the mother value in the active configuration has an appropriate value.

6.4.2 Managing the number of configurations

After a certain period of use, the number of configurations stored in the database may become numerous. A decent way of managing those configurations is to offer the possibility to delete and edit certain configurations. A developer might opt to only allow
this for a specific user group. Next to manual management, automatic deletion of configurations after a certain time or reaching a certain limit in number should be considered.

6.5 Effects of implementing Parameter History Logging to the results of a Usability Assessment

In paragraph 6.2, a Usability Assessment was conducted for the naval transport monitoring system presented in the preceding paragraph. As the name USAP already suggests, it should be usability supporting. The benefits of Parameter History Logging have already been presented in paragraph 5.2.4. However, we should reassess how the usability scenarios applicable for the naval transport monitoring system are affected if it would be including support for Parameter History Logging. Below, the usability scenarios used for the pre-implementation assessment are reassessed. A negative effect is denoted with ‘-‘, a positive effect with a ‘+‘. If Parameter History Logging does not affect the usability scenario at all, it is denoted with a ‘O’.

Aggregating Data (-)

The Parameter History Logging scenario may affect support for Aggregating Data if implemented only in such a way that only individual values can be copied into the input fields. The presented implementation only includes the functionality of copying an entire data set to the input fields, but not to copy only a selected amount of input values to their respective fields. In order to support this as well, a selection box should be added to each loaded input value.

Checking for Correctness (+)

Parameter History Logging means saving input data for future use. This means that the loaded input values will probably not harm data integrity as they have been successfully saved before. The probability that a user will enter invalid data himself is smaller as parameter history offers the user guidance in what to fill in.

Maintaining Device Independence (-)

Parameter History Logging may hamper applicability for different devices, especially those with limited screen real estate.

Evaluating the System (+)

Parameter History Logging has a lot in common with the Evaluating the System scenario. Both scenarios involve saving a certain state of the system. Parameter History Logging supports Evaluating the System by offering extra points of evaluation, namely by
investigating how many input sets were saved within a certain time span before one was found that was satisfactory to the user.

**Recovering from Failure (+)**

Parameter History Logging facilitates recovering from failure as it offers the user all previously used data sets.

**Retrieving Forgotten Passwords (O)**

Parameter History Logging has no effect on offering alternative ways to retrieve a password.

**Providing Good Help (+)**

Both providing good help and Parameter History Logging aims at offering the user guidance throughout usage, therefore, implementing Parameter History Logging supports the usability scenario of providing context-sensitive help.

**Reusing Information (+)**

Parameter History Logging involves saving data sets in a particular format. This facilitates reusing this data in other applications, depending on the chosen format for saving this data.

**Supporting International Use (-)**

Parameter History Logging can make support for international use more difficult if some input fields require data in natural language. In case of Boolean parameters, the English ‘yes’ or ‘no’ can be quite something different in other languages. For example, in Finnish, ‘kyllä’ means ‘yes’ and ‘ei’ means ‘no’. The software architect should therefore be aware that some parameters might need to be coded into numbers which can be translated into any natural language when the parameter set is loaded.

**Modifying Interfaces (O)**

Parameter History Logging has no effect on this usability scenario.

**Navigating within a Single View (O)**

An option to use previous parameter sets should always be shown in the same screen, therefore this usability scenario will not be affected by implementing Parameter History Logging.

**Observing System State (+)**
When a user is entering or changing input values, the risk exists that a user copies some loaded values into input fields, goes away from the system for lunch, returns and is not sure whether the values in the input fields have been saved or not. Therefore, the current state should be shown separately from the input fields in order to always show the state the system is in. Therefore, it is necessary to include this view when implementing Parameter History Logging.

**Supporting Undo (+)**

Supporting undo means that the user is able to undo previous actions. Parameter History Logging allows the user to go back to previous states that even date from previous sessions. Therefore, support for undo is increased when implementing Parameter History Logging.

**Verifying Resources (O)**

Parameter History Logging has no effect on this usability scenario.

**Supporting Visualization (O)**

Different ways to showing the same data can also be applied to data coming from previously saved parameter sets. Therefore, Parameter History Logging has no effect on this usability scenario.

Implementing Parameter History Logging does indeed have an effect on other usability scenarios. The benefits that some usability scenarios (for instance: Supporting Undo) offer are increased by implementing Parameter History Logging, but on the other hand, the activities involved when working with previously saved parameter sets can raise usability concerns, such as the possibility to select multiple input fields when copying data. This should be taken into account by the software architect when making a decision whether or not to implement Parameter History Logging.

### 6.6 Relationships between architecturally-sensitive usability scenarios

In paragraph 5.1, it was discussed that the implementation of parameter history logging makes use of other architecturally-sensitive usability scenarios. An undoing option is preferable for those copy buttons, and when storing a new configuration, checking for correction may prevent errors. This raises the question whether there are more of those links between scenarios. These links are not based on using the same software tactics (Bass et al., 2001) to implement certain scenarios, but on whether a scenario cannot be implemented without the presence of another, or that a scenario facilitates the use of another scenario. For example, Aggregating Commands and Undo are both implemented by designing some command structure, but they can exist independently from each other. Of course, an undo method facilitates a lot of usability scenarios. This is also the reason why most operating systems already support some form (usually only single-level) of undo.
For example, in Microsoft Windows, all text fields that are part of a standard GUI have a single-level undo option by pressing CTRL-Z.

There are also scenarios that cannot exist, or whose existence becomes illogical without the presence of another scenario. For example, predicting the duration of a task is in fact already making sure that the system state can be observed by the user. Also, providing a cancel button but not having an undo option would be strange in many situations, as a user would only be able to revert an action if the task is not finished. There are exceptions to this, of course. Writing a CD, for example, is something that cannot be undone afterwards, but the process of burning can be canceled.

There are more scenarios that are related. Using Applications Concurrently and Supporting Multiple Activities seem to be exactly the same, but their scope is different. The latter scenario deals with plain multi-tasking support, while the other scenario aims to prevent interference between concurrent applications. This means that multi-tasking is already included in that scenario, as we can conclude that all possibly interfering applications interact with the user, otherwise this would not have been a usability scenario in the first place. Taking this into account, Using Applications Concurrently is just a restriction of Supporting Multiple Activities, and cannot exist without it.

A similar relation can be found between the scenarios Operating Consistently Across Views and Supporting Visualization, where the latter is the more general one. Supporting Visualization only includes the possibility to view data in more than one way, while Operating Consistently Across Views implies multiple views already. Therefore, this scenario should be seen as a way to further elaborate its prerequisite.

All these relationships between scenarios are important if these scenarios are implemented within a developing tool. Scenarios that have a prerequisite should only exist as an option within their prerequisite, and implementation of other related scenarios should be recommended by the tool, if applicable. Tool support for usability scenarios as a possible future research field is further discussed in the next chapter, paragraph 7.2.6.
7. Evaluation & discussion

7.1 Summary and research questions

The relationship between software architecture and usability has been the focus of this thesis. This topic has attracted the attention of several research groups, and the work of two main groups (The European STATUS project on the one hand, The Usability & Software Architecture group of the Software Engineering Institute at Carnegie Mellon University on the other) has been presented in chapter 3. Usability scenarios play a considerable role in identifying the relationship between design decisions and the resulting usability. STATUS have approached their research from a top-down perspective, which means that they have started identifying attributes that support usability, subsequently properties and indicators for each attribute and eventually the usability properties that are affected by design decisions. U&SA have approached their research from a bottom-up perspective by first identifying a number of architecturally-sensitive usability scenarios and subsequently analyze how those can improve usability. As the thesis was focused on elaborating a single usability scenario, the latter approach has been followed.

Each usability scenario can be elaborated to a usability supporting architecture pattern (USAP) following a specific methodology. The list of scenarios that has been elicited so far is based on experiences with applications running in desktop environments. Automation systems are electronically computerized systems used to control machines, processes and procedures. As they are used in different ways than desktop applications, the ways to achieve usability in such systems is also different. Users of systems used in Finnish forestry and mining have stressed the need for storing a history of parameter values used in configurations. This usability scenario has been presented here as a USAP, including various options in implementation. The following research questions were presented in Chapter 1:

- Before we can work to our USAP, we first need to know what we exactly want to work out. Thus, what exactly is parameter history logging and in what situations and under what conditions would a user need this usability-related functionality?
- In what way does parameter history logging support usability? More specifically, what aspects of usability does it relate to?
- How can we compose a generic set of requirements that should be addressed when implementing parameter history logging?
- More in general: are there usability scenarios that are in some way related to parameter history logging, and does it influence the necessity of implementing those?

Parameter history logging is a functionality that supports the user in making choices when entering a set of user inputs for some form of configuration by offering the user saved sets of user inputs that have been used before. It is especially needed for products that require a lot of user input.
Offering the user sets of previously used inputs prevents the user from making errors by not forcing the user to rely on his memory. It will also assist the user in problem solving (especially when it involves trial-and-error), and giving information about previously successful sets of parameters increases user confidence and satisfaction. This is summarized in the Usability Benefit Matrix in paragraph 5.2.3.

Paragraph 5.3 elaborates a hierarchical set of responsibilities involving parameter history logging which result in a generic set of requirements that should be addressed during implementation of this scenario. The set starts with addressing the main responsibility – the scenario itself – and iteratively addresses all concerns that may arise from responsibilities that are added to the set until all concerns are taken care of. Ideally this methodology results in a set of detailed requirements which can be mapped to individual components, which makes actual implementation easier.

The assessments performed both before (paragraph 6.2) and after (paragraph 6.5) implementing Parameter History Logging, indicate that Parameter History Logging increases the benefits of some usability scenarios, such as Supporting Undo and Recovering from Failure, but that the nature of the activities involved when using previously saved parameters as guidance for new inputs requires other usability scenarios to be reassessed or even more required. One example of this is Aggregating Data, as the user might want to do copy actions for more than one parameter at once when working with multiple parameters.

### 7.2 Future work

The relationship between software architecture and usability has only been investigated for the last couple of years and will continue to receive considerable attention from research groups. There are various issues that are likely to be addressed:

#### 7.2.1 Extending the list of architecture-sensitive usability scenarios

The list of scenarios or patterns gathered by U&SA and STATUS is not complete. This thesis addresses a new scenario, but there will still be others, and existing scenarios may have different instances depending on the context in which the system is used. It is also possible that domain-specific lists will evolve out of the current one. Different devices and different environments during use (e.g. outside instead of at the office) influence the extent to which a particular scenario can be useful.
7.2.2 Harmonizing software engineering and usability engineering processes

The processes used in software engineering differ a lot from those that are used by usability engineers. The latter community regards usability mainly as part of functionality, while usability is usually regarded by software engineers as a quality. Software engineering processes are more formal in nature, where on the other hand usability engineering practices are fuzzier in nature. This continues to lead to misunderstandings between the two communities. Software architects usually develop their first design by gathering the technologies needed to fulfill the functional requirements, without taking usability into account. The patterns and scenarios will help them in dealing with these issues earlier, but these do not cover all potential usability risks. HCI experts on the other hand find it hard to come up with only those usability requirements that affect the design of an architecture, and usually overload the software architect with far too many requirements that are irrelevant to the architecture. Usability engineers with a basic understanding of software architecture might be better able to filter those usability aspects that are relevant to the architect, in order to harmonize the development phase. On the other hand, software architects should be able to somehow foresee the effects of certain design decisions on usability. The handbook that is planned by U&SA will be a great help for software architects, although it will never be complete. Every domain has its own specific usability issues, so the ideal of a software architect just having to check a number of scenarios doesn't seem to be possible. Therefore, in order to reduce time and cost of software development, integrating the processes of both disciplines will continue to receive attention in the coming years.

7.2.3 Refining and extending the Usability Framework

The Usability Framework postulated by STATUS is a nice basis to capture the mechanisms in which software architecture is related to usability. However, it would be better if the relations in this framework could somehow be quantified in order to improve that validity of architecture assessments for usability, rather than the qualitative nature that it has now. Quantifying these relationships will not be easy, and maybe even impossible, because the relationships between the different entities in the framework are probably very much dependent on the context of the system itself. The same scenario will have different impact on usability properties if it is implemented in different systems. Some things just cannot be formalized and captured into meaningful metrics.

Another possible research focus is to extend the framework by adding relations to other software qualities, like for example safety and security (Folmer, 2005). Some patterns that have a positive effect on one quality can have a negative effect on another quality. Expanding the framework can give the architect more clues in making design decisions by finding the best tradeoff between the different qualities.
7.2.4 Reasoning Frameworks & Usability

The Software Engineering Institute (SEI) at Carnegie Mellon University has developed the concept of reasoning frameworks in architecture design (Bass et al., 2005). This approach takes into account that software architects cannot be an expert in every quality attribute that is relevant for his software design. Therefore, Quality Attributes Experts should be responsible for creating reasoning frameworks for a quality attribute based on their knowledge of this attribute. Several quality attribute frameworks then act as input for the software architect who designs the system based on a set of specific requirements. This makes quality attribute experts a sort of generalists that are responsible for a single quality attribute of many systems, and a software architect responsible for the specific system he designs. This can be seen as segregation of duties between quality and system design. In the context of a USAP, the quality attribute expert is responsible for defining the general responsibilities, after which the software architect can make the solution specific for his system (see also Table 3). The U&SA group is currently in the process of developing a reasoning framework for usability (Lee et al., 2005).

Reasoning frameworks also relate to the following two fields of interest in future research: the relationship between usability scenarios and tool support for implementing these scenarios.

7.2.5 The relationships between scenarios or patterns

The patterns named by STATUS and the scenarios gathered by U&SA are not just loose entities that have no relationship with one and other. For example, implementing canceling and undoing a command both need some command pattern that keeps a previous state into memory. Aggregating commands also influences the undo functionality. Further research could be directed towards tying related scenarios together, especially in such a way that it becomes clear which scenarios become more necessary when other scenarios have been implemented. The US&A group are addressing this problem in their development of a reasoning framework for usability. In their technical note ‘Elements of a Usability Reasoning Framework’ (Lee et al., 2005), they propose to combine several shared responsibilities that are part of the USAP’s ‘Progress Feedback’ and ‘Cancel Command’. For other USAP’s, such an approach has not yet been worked out.

7.2.6 Tool support

Tool support can automate certain processes that are still done manually. Some work has been done to incorporate USAP’s into a modeling tool named ArchE (Architecture Expert) that is being developed by the Software Engineering Institute (SEI) at Carnegie Mellon University (Bachmann et al., September 2003; Bachmann et al., 2005). This tool allows a software architect to include responsibilities derived from reasoning frameworks (see paragraph 7.2.4) into his architecture design. An ArchE Reasoning Language (ARL) is
being developed in order to be able to specify a reasoning framework or responsibility in code instead of natural language. Experimental work has been done to capture USAP’s ‘Progress Feedback’ and ‘Cancel Command’ in ARL (Bass et al., 2005), but the language was still not fully functional. The concept of intersecting or preceding responsibilities, which indicates relationships between responsibilities, has been included in the methodology (Bachmann et al., September 2003; Bass et al., 2005).

Another thing that can be automated is architecture usability assessment. Following the SALUTA method by hand is time consuming, and tools could be built to automate some of the steps. Looking for usability patterns in a software architecture can also be automated, by applying pattern matching, which is a research topic on its own and has received considerable attention by software engineering research groups lately.

7.2.7 Evidence for reduced maintenance costs

So far there is no empirical evidence that architecting for usability decreases the costs of maintenance. From a manager’s viewpoint, investing more into usability must pay off, and if there’s no evidence for cost reduction, there will be no incentive for extra investments. It will not be easy to find a suitable set of data which proves that including usability concerns into architecture design really brings the business benefits hoped for. For this, one needs to prevent biasing by excluding other environmental factors that accompany a software project, and there should be enough historical data to compare with.

7.3 Conclusion

Architecting for usability will remain the subject of research in the coming years. The biggest gains can probably be made by improving development tools with usability supporting features. Although new insights will lead to practices that reduce development & maintenance cost, it is unlikely that there will be no overlooked usability requirements in the future, but as far as they are less in both number and size it can still be said that it has improved the software development processes.
References


GARLAN, D. **An introduction to software architecture.** *Advances in Software Engineering and Knowledge Engineering, Volume I*

GARLAN, D. **Research directions in software architecture.** *ACM Computing Surveys (CSUR)*, 1995(27), Issue 2: p257-261


OVASKA, S. **Usability as a goal for the design of computer systems.** *Scandinavian Journal of Information Systems*. 1991(3).

STATUS DELIVERABLE NO. 2: http://is.ls.fi.upm.es/status/results/STATUS_D2_v1.1.pdf
Appendix: Parameter History Logging example in DHTML

```html
<html>
<title>Demo Parameter History Logging</title>
<body bgcolor=#DFFFFF>
<center>
<script>
function Config(a,b,c,option,check)
{
    /* object definition of a configuration */
    this.a = a;
    this.b = b;
    this.c = c;
    this.option = option;
    this.check = check;
}

var i;
var boolArray = new Array("no", "yes");
var config = new Array();

/* initial values */
cfg[0] = new Config(25,10,"header",3,1);
cfg[1] = new Config(43,58,"footer",2,0);
cfg[2] = new Config(18,83,"left",1,1);
cfg[3] = new Config(98,12,"right",0,0);

var selectedBackup = new Config("", "", "", 0, null);
var selectedConfig = cfg[3];
var undoConfig = new Config("", "", "", 0, null);
var undoAction = "";

function copyAll()
/* copies all config data to input fields */
{
    undoConfig.a = window.document.form.a.value;
    undoConfig.b = window.document.form.b.value;
    undoConfig.c = window.document.form.c.value;
    for(i=0;i<4;i++) {
        if (window.document.form.option.options[i].selected) {
            undoConfig.option = i;
        }
    }
    if (window.document.form.yes.checked) {
        undoConfig.check = 1;
    }
}
</script>
</center>
</body>
</html>
```
undoConfig.check = 0;
}
undoAction = "All";

window.document.form.a.value = selectedConfig.a;
window.document.form.b.value = selectedConfig.b;
window.document.form.c.value = selectedConfig.c;
window.document.form.option.options[selectedConfig.option].selected = true;
if (selectedConfig.check) {
    window.document.form.yes.checked = true;
    window.document.form.no.checked = false;
} else {
    window.document.form.no.checked = true;
    window.document.form.yes.checked = false;
}
window.document.form.undoButton.disabled = false;
}

function undo()
/* undo copy action */
{
    selectedBackup = clone(selectedConfig);
    selectedConfig = clone(undoConfig);
    if (undoAction=="All") {
        copyAll();
    } else {
        copyValue(undoAction);
    }
    selectedConfig = clone(selectedBackup);
    window.document.form.undoButton.disabled = true;
}

function copyValue(index)
/* copies single value to corresponding input field */
{
    switch(index) {
        case 1: undoConfig.a = window.document.form.a.value; break;
        case 2: undoConfig.b = window.document.form.b.value; break;
        case 3: undoConfig.c = window.document.form.c.value; break;
        case 4:
            for(i=0;i<4;i++) {
                if (window.document.form.option.options[i].selected) {

undoConfig.option = i;
}
}
window.document.form.option.options[selectedConfig.option].selected = true;
break;
case 5:
if (window.document.form.yes.checked) {
  undoConfig.check = 1;
} else {
  undoConfig.check = 0;
}
if (selectedConfig.check) {
  window.document.form.yes.checked = true;
  window.document.form.no.checked = false;
} else {
  window.document.form.no.checked = true;
  window.document.form.yes.checked = false;
}
break;
default: break;
}
undoAction = index;
window.document.form.undoButton.disabled = false;
}

function showAll() {
  /* shows the config data */
  fieldA.innerText = selectedConfig.a;
  fieldB.innerText = selectedConfig.b;
  fieldC.innerText = selectedConfig.c;
  fieldOption.innerText =
  window.document.form.option.options[selectedConfig.option].innerText;
  fieldCheck.innerText = boolArray[selectedConfig.check];
}

function loadConfig() {
  /* loads config (dummy) data */
  {
    for(i=0;i<4;i++) {
      if (window.document.form.config.options[i].selected) {
        selectedConfig = config[i];
      }
    }
    showAll();
  }

  function clone(myObj)
{
    /* copies object itself instead of reference only */
    if(typeof(myObj) != 'object') return myObj;
    if(myObj == null) return myObj;

    var myNewObj = new Object();

    for(var i in myObj)
        myNewObj[i] = clone(myObj[i]);

    return myNewObj;
}

</script>
<form name=form>
    <input name=undoButton disabled type=button value="Undo" onClick="undo()">
<table>
    <tr>
        <td></td>
        <td><input name=buttonAll type=button value="Copy All" onClick="copyAll()"></td>
        <td><select name=config onChange="loadConfig()">
                <option>Configuration A</option>
                <option>Configuration B</option>
                <option selected>Configuration C</option>
                <option>Configuration D</option>
        </select></td>
    </tr>
    <tr><td></td><table>
        <tr>
            <td align=right>Input A</td>
            <td><input name=a size=10></td>
            <td><input type="button" value="Copy" onClick="copyValue(1)" width=80 align=right id="fieldA"></td>
        </tr>
        <tr>
            <td align=right>Input B</td>
            <td><input name=b size=10></td>
            <td><input type="button" value="Copy" onClick="copyValue(2)" width=80 align=right id="fieldB"></td>
        </tr>
    </table>
</form>
<table>
  <tr>
    <td align=right id="fieldB"></td>
  </tr>
  <tr>
    <td align=right>Input C</td>
    <td><input name=c size=10></td>
    <td><input type="button" value="Copy" onClick="copyValue(3)"</td>
    <td align=right id="fieldC"></td>
  </tr>
  <tr>
    <td>Select</td>
    <td>
      <select name=option>
        <option>option A</option>
        <option selected>option B</option>
        <option>option C</option>
        <option>option D</option>
      </select>
    </td>
    <td><input type="button" value="Copy" onClick="copyValue(4)"</td>
    <td align=right id="fieldOption"></td>
  </tr>
  <tr>
    <td>Check</td>
    <td><input type=radio name=yes>no<input type=radio name=no></td>
    <td><input type="button" value="Copy" onClick="copyValue(5)"</td>
    <td align=right id="fieldCheck"></td>
  </tr>
</table>
</form>
</body>
</html>