Design and Evaluation of Software Architecture

PerOlof Bengtsson

University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science

ISSN 1103-1581
ISRN HK-R-RES--99/10--SE

Printed in Sweden
Kaserntryckeriet AB
Karlskrona 1999

to Hans and Kristina for everlasting support

This thesis is submitted to the Research Board at University of Karlskrona/Ronneby, in partial fulfillment of the requirements for the degree of Licentiate of Engineering.

Contact Information:

PerOlof Bengtsson
Department of Software Engineering
and Computer Science
University of Karlskrona/Ronneby
Soft Center
SE-372 25 RONNEBY
SWEDEN

Tel.: +46 457 787 41
Fax.: +46 457 271 25
email: PerOlof.Bengtsson@ipd.hk-r.se
URL: http://www.ipd.hk-r.se/pob

Abstract

The challenge in software development is to develop software with the right quality levels. The main problem is not to know if a project is technically feasible concerning functionality, but if a solution exists that meet the software quality requirements. It is therefore desired to get an early indication of the qualities of the resulting software.

Software architecture is concerned with what modules are used to compose a system and how these modules are related to each other, i.e. the structure of system. The software architecture of a system sets the boundaries for these qualities. Hence, to design the software architecture to meet the quality requirements is to reduce the risks of not achieving the required quality levels.

In this thesis we present the experiences from a software architecture design project with two industry partners. Based on these we propose a method for reengineering architectures and exemplify by an applying the method on a real world example. The method is based on architecture transformations and software quality evaluation of the architecture. Further, we present a method for predicting software maintenance effort from the software architecture, for use in the design and reengineering method. The method uses change scenario profiles to describe future changes to the system and architecture impact analysis provide knowledge of the modification volume required for the realization of each scenario. The results from a quasi experiment supports that groups consisting of prepared members create better scenario profiles. Also, the results suggest that there is a large room for variation between scenario profiles created by individual persons.

Acknowledgements

I would like express my gratitude and appreciation to my advisor *Professor Jan Bosch*, for offering me this opportunity, for his guidance, and for being a good friend. I would also like to thank my colleague, *Michael Mattsson*, who always have time for me and my research. I would also like to thank my colleagues at the department, especially the members of the RISE research group.

The results presented in this thesis would not have been possible if it were not for; the people at the companies I have worked with, *Lars-Olof Sandberg* at Althin Medical, *Anders Kambrin* and *Mogens Lundholm* at EC-gruppen, the students volunteers in the experiments, and the staff at Infocenter in Ronneby. From the library at Campus Gräsvik I would like to thank *Peter Linde* who assisted me swiftly when time was scarce before printing.

I would also like to express my gratitude to all my friends, especially *Magnus C. Ohlsson* for his friendship and lots of fun. An important person in my life is my friend and teacher, *Olle Tull*, who taught me lots more than playing the trumpet.

I am greatly indebted to my family; to my father *Hans* and my mother *Kristina* for their never failing support and wisdom, to my brother *Jonas* and my sister *Elisabeth* for. I give my greatest love to *Kristina* for sharing her life with me.

Contents

Introduction

The challenge in software development is to develop software with the right quality levels. The problem is not so much to know if a project is technically feasible concerning functions required, but instead if a solution exists that meets the software quality requirements, such as throughput and maintainability. Traditionally the qualities of the developed software have, at best, been evaluated on the finished system before delivering to the customer. The obvious risks of having spent much effort on developing a system that eventually did not meet the quality requirements have been hard to manage. Changing the design of the system would likely mean rebuilding the system from scratch to the same cost.

In software development it is therefore desired to get an early indication of the qualities of the resulting software. Such an indication would allow the software engineers to make changes to the design of the system before expensive resources have been used for programming parts that is removed later in the development process.

In 1972 Parnas [22] argued that the way a software system is decomposed into modules affects its abilities to meet levels of efficiency in certain aspects, e.g. flexibility, performance. Software architecture is concerned with what modules are used to compose a system and how these modules are related to each other, i.e. the structure of system. The software architecture of a system sets the boundaries for these qualities [2, 9, 11, 22, 28]. Hence, to design the software architecture to meet the quality requirements is to reduce the risks of not achieving the required quality levels.

The result from the software architecture design activity is a software architecture. But, the description of that software architecture is far from trivial. A reason is that it is hard to decide what information is needed to describe a software architecture, and hence, it is very hard to find an optimal description technique.

The problems of not having a notation with clear semantics become apparent in the software architecture assessment part of the software architecture design process. In software architecture assessment, the goal is to learn if the software qualities of the future system will meet the quality requirements, and if not, what qualities that are lacking. The assessment of a software architecture is done very early in the development process and in the software architecture design it is primarily used to evaluate alternatives and changes.

In this thesis, we give a brief introduction to software architecture in section 1, followed by an overview of software architecture design in section 2. In section 3 we survey description of software architecture and continue with a section about analysis of software architecture. We conclude the first part of this thesis with a discussion about the contributions of this thesis and further research. Part two of the thesis is a compilation of selected papers with results from the research.

1. Software Architecture

The expression *software architecture* was used, perhaps the first time, in a scientific article as early as in 1981 in [27] and the concept of dealing with systems by decomposing the software into modules is not new. Even earlier David L. Parnas reported on the problem of increasing software size in his article, "On the Criteria To Be Used in Decomposing Systems into Modules" [22] in 1972. In that article, he identified the need to divide systems into modules by other criteria than the tasks identified in the flow chart of the system. A reason for this is, according to Parnas, that "The flow chart was a useful abstraction for systems with in the order of 5,000-10,000 instructions, but as we move beyond that it does not appear to be sufficient; something additional is needed". Further Parnas identifies information hiding as a criterion for module decomposition, i.e. every module in the decomposition is characterized by its knowledge of a design decision which it hides from all modules [22].

Thirteen years later Parnas together with P. Clements and D. Weiss brings the subject to light again, in the article "The Modular Structure of Complex Systems" [23]. In the article it is shown how development of an inherently complex system can be supplemented by a hierarchically structured *module guide*. The module guide allows the software engineers to know what the interfacing modules are, and help the software engineer to decide which modules to study.

In [28] the authors also identified that the size and complexity of systems increases and the design problem have gone beyond algorithms and data structures of the computation. In addition, we now have structural issues of the organization of the system in large, the control structures, communication protocols, physical distribution, and selection among design alternatives. These issues are part of software architecture design.

In the beginning of the 1990:s software architecture got wider attention in the software engineering community and later also in industry. Today, software architecture has become an accepted concept, most evident, perhaps, by the new role, software architect, appearing in the software developing organizations. Other evidence includes the growing number of software architecture courses on the software engineering curricula and attempts to provide certification of software architects.

1.1 Elements, form and rationale

In the paper by Perry and Wolf [24] the foundations for the study of software architecture define software architecture as follows:

Software Architecture = {Elements, Form, Rationale}

Thus, a software architecture is a triplet of (1) the elements present in the construction of the software system, (2) the form of these elements as rules for how the elements may be related, and (3) the rationale for why elements and the form were chosen. This definition has been the basis for other researchers, but it has also received some critique for the third item in the triplet. In [2] the authors acknowledge that the rationale is indeed important, but is in no way part of the software architecture. The basis for their objection is that when we accept that all software systems have an inherent software architecture, even though it has not been explicitly designed to have one, the architecture can be recovered. However, the rationale is the line of reasoning and motivations for the design decisions made by the design, and to recover the rationale we would have to seek information not coded into software. The objection implies that software architecture is an artifact and that it could be coded, although scattered, into source code.

1.2 Components & connectors

In a paper about software architecture by Garlan & Shaw [28] we find the following definition:

> Software architecture is the *computational components*, or simply *components*, together with a description of the *interactions* between these components, the *connectors*.

The definition is probably the most widely used, but has also received some critique for the *connectors*. The definition may be interpreted that components are those entities conerned with computing tasks in the domain or supprt tasks, e.g.persistence via a data base management system. Connectors are entities that are used to model an implement interactions between components. Connectors take care of interface adaptation and other properties specific to the interaction. This view is supported in, for example, the Unicon architecture description language [29].

1.3 Architecture business cycle

Software architecture is the result from technical, social and business influences. Software architecture distills away details and focuses only on the interaction and behavior between the black box components. It is the first artifact in the life cycle that allow analysis of priorities between competing concerns. The concerns stem from one or more of the stakeholders concerned with the development effort and must be prioritized to find an optimal balance in requirement fulfillment. Stakeholders are persons concerned with the development effort in some way, e.g. the customer, the end-user, etc.

The factors that influenced the software architecture are in turn influenced by the software architecture and form a cycle, the *architecture business cycle (ABC)* [2] (figure 1).

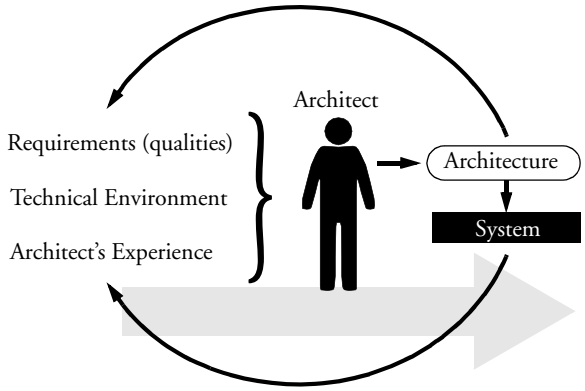

Figure 1. *Architecture Business Cycle [2]*

In the architecture business cycle the following factors have been identified:

- The software architecture of the built system affects the structure of the organization. The software architecture describes units of the system and their relations. The units serve as a basis for planning and work assignments. Developers are divided into teams based on the architecture.
- Enterprise goals affect and may be affected by the software architecture. Having control over the software architecture that dominates a market means a powerful advantage [21].
- Customer requirements affects and are affected by the software architecture. Opportunities in having robust and reliable software architecture might encourage the customer to relax some requirements for architectural improvements.
- The architect's experience affects and is affected by the software architecture, in particular by deploying a system from it. The architect will most likely use architectures that have proven to be good in the architect's own experience, and architectures that failed will be refrained from.
- Systems are affected by the architecture and some few systems will affect the software engineering community as a whole.

2. Software Architecture Design

Software system design consists of the activities needed to specify a solution to one or more problems, such that a balance in fulfillment of the requirements is achieved. A *software architecture design method* implies the definition of two things. First, a process or procedure for going about the included tasks. Second, a description of the results or type of results to be reached when employing the method. From the software architecture point-of-view, the first of the aforementioned two, includes the activities of specifying the components and their interfaces, the relationships between components, and making design decisions and document the results to be used in detail design and implementation. The second is concerned with the definition of the results, i.e. what is a component and how is it described, etc.

The traditional object-oriented design methods, e.g. (OMT [26], Booch [5], Objectory [14]) has been successful in their adoption by companies worldwide. Over the past few years the three aforementioned have jointly produced a *unified modeling language* (UML) [6] that has been adopted as de facto standard for documenting object-oriented designs. Object-oriented methods describe an iterative design process to follow and their results. There is no guarantee that you will reach the desired results from following the prescribed process. The reason is that the processes prescribes no technique or activity for evaluation of the halting criterion for the iterative process, i.e. the software engineer is left for himself to decide when the design is finished. This is both from a method perspective and from a design perspective, insufficient since the stopping criterion relates to whether or not the requirements on the design result will be achieved.

Software architecture is the highest abstraction level [2] at which we construct and design software systems. The software architecture sets the boundaries for the quality levels resulting systems can achieve. Consequently, software architecture represents an early opportunity to design for software quality requirements, e.g. reusability, performance, safety, and reliability.

The design method must in its process have an activity to determine if the design result, in this case the software architecture, has fulfilled the requirements. We only consider design methods with such an activity as considered complete.

The enabling technology for the design phase is neither technological nor physical, but it is the human creative capability. It is the task of the human mind to find the suitable abstractions, define relations, etc. to ensure that the solution fulfills its requirements. Even though parts of these activities can be supported by detailed methods, every design method will depend on the creative skill of the designer, i.e. the skill of the individual human's mind. Differences in methods will present themselves as more or less efficient handling of the input and the output. Or more or less suitable description metaphors for the specification of the input and output. This does not prohibit design methods from distinguishing themselves as better or worse for some aspects. It is important to remember that results from methods are *very dependent* on the skill of the persons involved and can never make up for lack of experience.

The following sections present different approaches to designing software architectures.

2.1 Architecture patterns & styles

Experienced software engineers have a tendency to repeat their successful designs in new projects and avoid using the less successful designs again. In fact, these different styles of designing software systems could be common for several different unrelated software engineers. This has been observed in [13] where a number of systems were studied and common solutions to similar design problems were documented as *design patterns*. The concept has been successful and today most software engineers in are aware of design patterns.

The concept has been used for software architecture as well. First by describing *software architecture styles* [28] and then by describing *software architecture patterns* [11] in a form similar to the design patterns. The difference between software architecture styles and software architecture patterns have been extensively debated. Two major view points are; styles and patterns are equivalent, i.e. either could easily be written as the other, and the other view point is, they are significantly different since styles are a categorization of systems and patterns are general solutions to common problems. Either way styles/patterns make up a common vocabulary. It also gives software engineers support in finding a well-proven solution in certain design situations.

Software architecture patterns impact the system in large, by definition. Applying software architecture patterns late in the development

cycle or in software maintenance can by prohibitly costly. Hence, it is worth noting that software architecture patterns should be used to find a proper software architecture in the first place.

2.2 Schlaer & Mellor - Recursive design

The authors of the recursive design method [30] intends to change the view of software development from five general assumptions:

- Analysis treats only the application.
- Analysis must be represented in terms of the conceptual entities in the design.
- Because software architecture provides a view of the entire system, many details must be omitted.
- Patterns are small units with few objects.
- Patterns are advisory in nature into another alternate view:

into five alternative views:

- Analysis *can be performed on any domain.*
- Object oriented analysis (OOA) method does not imply anything about the fundamental design of the system.
- Architecture domain, like any domain, can be modeled in complete detail by OOA.
- OOA models of software architecture provides a comprehensive set of large-scale interlocking patterns.
- Use of patterns is *required.*

Domain analysis

Fundamental for the recursive design method is the domain analysis. A domain is a separate real or hypothetical world inhabited by a distinct set of conceptual entities that behave according to rules and policies characteristic of the domain. Analysis consists of work products that identify the conceptual entities of a single domain and explain, in detail, the relationships and interactions between these entities. Hence,

domain analysis is the precise and detailed documentation of a domain. Consequently, the OOA method must be detailed and complete, i.e. the method must specify the conceptual entities of the methods and the relationships between these entities. The elements must have fully defined semantics and the dynamic aspects of the formalism must be well defined (the virtual machine that describes the operation must be defined).

Application Independent Architecture

The recursive design method regards everything as its own domain. An application independent architecture is a separate domain and deals in complete detail with; organization of data, control strategies, structural units, and time. The architecture does not specify the allocation of application entities to be used in the application independent architecture domain. This is what gives the architecture the property of application independence.

Patterns and Code Generation

The recursive design method includes the automatic generation of the source code of the system and design patterns play a key role in the generation process. Design patterns can be rendered as archetypes, which is conceptually equivalent to defining macros for each element of the patterns. The code generation relies heavily on that the architecture is specified using patterns. Therefore use of patterns is absolutely required.

Process

The recursive design process defines a linear series of seven operations; each described in more detail in following sections. The operations are:

1. Characterize the system.
2. Define conceptual entities.
3. Define theory of operation.
4. Collect instance data.
5. Populate the architecture.
6. Build archetypes.
7. Generate code.

Activities

Start with eliciting the characteristics that should shape the architecture. Attached to the method is a questionnaire with heuristic questions that will serve as help in the characterization. The questionnaire brings up fundamental design considerations regarding size, memory usage etc. The information source is the application domain and other domains, but the information is described in the semantics of the system. The results from this operation are the system characterization report, often containing numerous tables and drawings.

The conceptual entities and the relationships should be precisely described. The architect selects the conceptual entities based on the system characterization and their own expertise and experience, and document the results in an object information model. Each object is defined by its attributes, which in turn is an abstraction of a characteristic.

The next step in the process is to precisely specify the theory of operation. The authors of the method have found that an informal, *but* comprehensive document works well to define the theory of operation, later described in a set of state models.

In the application domain, a set of entities is considered always present or pre-existing. Collecting instance data for populating the instance database means finding those entities, typically only a few items, e.g. processor names, port numbers etc.

The *populator* populates the architecture by extracting elements from the repository containing the application model and then uses the elements to create additional instances in the architecture instance data-

base. The architecture instance database contains all the information about the system to be built.

The building of archetypes is the part where all the elements in the architecture have to be precisely and completely specified. To completely define an archetype we use text written in the target programming language and placeholders to represent the information from the architecture instance database.

The last operation, that of generating the code requires the implementation of a script, called the system construction engine. This script will generate the code from the analysis models, archetypes and the architecture instance database.

2.3 4+1 View model design method

The 4+1 View Model presented in [16] was developed to rid the problem of software architecture representation. Five concurrent views (Figure 2) are used, each view addresses concerns of interest to different stakeholders. On each view, the Perry/Wolf definition [24] (discussed in section 1.1) is applied independently. Each view is described using its own representation, a so called *blueprint.* The fifth view (+1) is a list of scenarios that drives the design method.

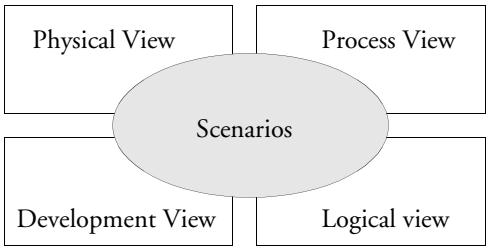

Figure 2. *Each view address specific concerns*

Logical View

The logical view denotes the partitions of the functional requirements onto the logical entities in the architecture. The logical view contains a set of key abstractions, taken mainly from the problem domain, expressed as objects and object classes

If an object's internal behavior must be defined, use state-transition diagrams or state charts.

The object-oriented style is recommended for the logical view. The notation used in the logical view is the Booch notation [5]. However, the numerous adornments are not very useful at this level of design.

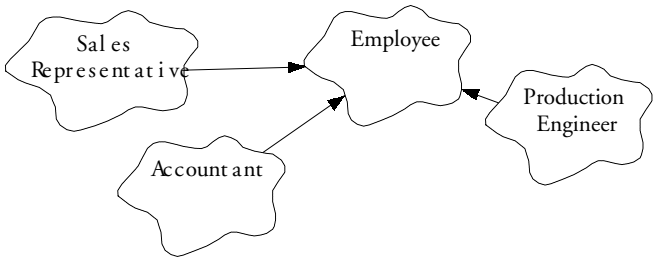

Figure 3. *Booch notation example in Logical View*

Process view

The process view specifies the concurrency model used in the architecture. In this view, for example, performance, system availability, concurrency, distribution system integrity and fault-tolerance can be analyzed. The process view is described at several levels of abstractions, each addressing an individual concern.

In the process view, the concept of a process is defined as a group of tasks that form an executable unit. Two kinds of tasks exist; major and minor. Major tasks are architectural elements, individually and uniquely addressable. Minor tasks, are locally introduced for implementation reasons, e.g. time-outs, buffering, etc. Processes represent the tactical level of architecture control. Processes can be replicated to deal with performance and availability requirements, etc.

For the process view use an expanded version of the Booch process view. Several styles are useful in the process view, e.g. pipes & filters [11,28], client/server [28].

Physical View

The elements of the physical view are easily identified in the logical, process and development views and are concerned with the mapping of these elements onto hardware, e.g. networks, processes, tasks and objects. In the physical view, quality requirements like availability, reliability (fault-tolerance), performance (throughput) and scalability can be addressed.

Development View

The development view takes into account internal, or, intrinsic properties/requirements like reusability, ease of development, testability, and commonality. This view is the organization of the actual software modules in the software development environment. It is made up of program libraries or subsystems. The subsystems are organized in a hierarchy of layers. It is recommended to define 4-6 layers of subsystems in the development view. A subsystem may only depend on subsystems in the same or lower layers, to minimize dependencies.

The development view supports allocation of requirements and work division among teams, cost evaluation, planning, progress monitoring, reasoning about reuse, portability and security.

The notation used is taken from the Booch method, i.e. modules/subsystems graphs. Module and subsystems diagrams that show import and export relations represent the architecture.

The development view is completely describable *only* after all the other views have been completed, i.e. all the software elements have been identified. However, rules for governing the development view can be stated early.

Scenarios

The fifth view (the +1) is the list of scenarios. Scenarios serve as abstractions of the most important requirements on the system. Scenarios play two critical roles, i.e. design driver, and validation/illustration. Scenarios are used to find key abstractions and conceptual entities for the different views, or to validate the architecture against the predicted usage.

The scenario view should be made up of a small subset of *important* scenarios. The scenarios should be selected based on criticality and risk.

Each scenario has an associated *script*, i.e. sequence of interactions between objects and between processes [25]. Scripts are used for the validation of the other views and failure to define a script for a scenario discloses an insufficient architecture.

Scenarios are described using a notation similar to the logical view, with the modification of using connectors from the process view to show interactions and dependencies between elements.

Design Process

The 4+1 View Model consists of ten semi-iterative activities, i.e. all activities are not repeated in the iteration. These are the activities:

1. Select a few scenarios based on risk and criticality.
2. Create a straw man architecture.
3. Script the scenarios.
4. Decompose them into sequences of pairs (object operation pairs, message trace diagram).
5. Organize the elements into the four views.
6. Implement the architecture.
7. Test it.
8. Measure it/evaluate it.
9. Capture lessons learned and iterate by reassessing the risk and extending/revising the scenarios.
10. Try to script the new scenarios in the preliminary architecture, and discover additional architectural elements or changes.

Activities

The activities are not specified in more detail by the author [16]. But some comments are given.

- Synthesize the scenarios by abstracting several user requirements.
- After two or three iterations the architecture should become stable.
- Test the architecture by measurement under load, i.e. the implemented prototype or system is executed.
- The architecture evolves into the final version, and even though it can be used as a prototype before the final version, it is not a throw away.

The results from the architectural design are captured in two documents; software architecture as the 4+1 views, and a software design guidelines. (Compare to the rationale in the Perry and Wolf definition [24])

2.4 Iterative software architecture design method

The scenario-based software architecture design method (ARCS) [9] exploits the benefits of using scenarios for making software quality requirements more concrete. Abstract quality requirements, like for example reusability, can be described as scenarios in the context of this system and its expected lifetime.

Also, the method puts emphasis on evaluation of the architecture to ensure that the quality requirements can be fulfilled in the final implementation of the system. Four categories of evaluation techniques are described in the method, i.e. scenario-based evaluation, simulation, mathematical modeling and experience-based reasoning (heuristics).

Process

The process is iterative and meant to be iterated in close cycles. The process' activities and the activities' relationships are shown in figure 4.

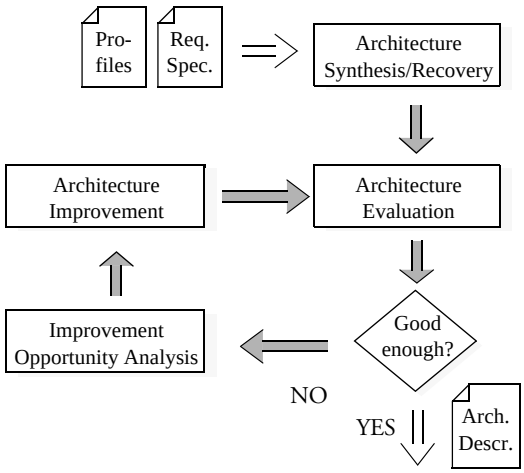

Figure 4. *The ARCS design method process*

Activities

The software architect starts with synthesizing a software architecture design based only on functional requirements. The requirement specification serves as input to this activity. Essentially the functionality-based architecture is the first partitioning of the functions into subsystems. At this stage in the process, it is also recommended that the scenarios for evaluating the quality requirements be specified. No particular attention is given to the quality requirements, as of yet.

The next step is the evaluation step. Using one of the four types of evaluation techniques the software architect decides if the architecture is good enough to be implemented, or not. Most likely, several points of improvement will reveal themselves during the evaluation and the architecture has to be improved.

Architecture transformation is the operation where the system architect modifies the architecture using one or more of the five transformation types to improve the architecture. The idea behind the transformation is that the architecture has the exact same functions after the transformation as before the transformation. The only difference is that the quality properties of the architecture have changed.

Transformations will affect more then one quality attribute, e.g. reusability *and* performance, and perhaps in opposite directions, i.e. improving one and degrading the other. The result is a trade-off between software qualities [3, 18]. After the transformation has been completed, the software engineer repeats the evaluation operation and obtains a new results. Based on these either the architecture is fulfills the requirements or the software engineer makes new transformations.

Evaluation Techniques

The first of the four types of evaluation techniques is the scenario-based evaluation, which is a central part in the method. Scenarios make quality requirements more concrete and meaningful in the context of the future system by describing events relevant to the quality attribute. Executing the scenario on the architecture, similar to scripting in 4+1 View Model, and analyzing the result does the evaluation. Provided that the scenarios defined are representative, this kind of analysis will lead to relevant results.

Second, simulation is suggested as a type of evaluation of the architecture. The components in architecture implements the interfaces, using for example the architecture description language Rapide [17], and then typical execution situations, e.g. scenarios, are simulated and the results are analyzed.

Thirdly, mathematical modeling is a type of evaluation. In various computer science research domains, a number of task specific mathematical models exist for determining software attributes, for example process schedulability. Software metrics also fall into this category. The main difference is that metrics are based on statistical evidence, more than actual understanding of cause and causality.

Finally, an important evaluation type is the experience-based reasoning. This is often used informally and as a guide to the software engineer in selecting a more formal evaluation technique where the architecture seem suspect.

Transformation Techniques

The first of the transformation categories is transformation by imposing an architectural style. This means that the fundamental organization of the architecture changes.

Second, the architecture can be transformed by imposing an architectural pattern. The difference from imposing a style is that a pattern is not changing the fundamentals of the architecture, but impose a rule on all elements of the architecture. For example, adding a concurrency mechanism to all elements using the Periodic objects pattern [20].

Thirdly, the architecture can be transformed using a design pattern [11,13]. The result is a less dramatic change of the architecture.

Fourthly, the architecture can be transformed by converting the quality requirements into functionality. For example, increasing the fault-tolerance by introducing exception handling.

Finally, the quality requirements can be distributed. For example, instead of putting availability requirements on the complete system, the availability of the server part in a client/server, could have higher requirements than the clients.

2.5 Method comparison

The 4+1 View Model has its three major strengths in its tools support, experience with the application of the method and its solution to the problem of too many aspects in the same document. However, the method as presented in [16] give to little information to really benefit to the reader interested in using the method.

In [30] the authors fail in proving their case for the recursive design of an application independent architecture. The method suffers from several unclarities and limitations. For example, the system generation script seem to be the key to the whole automatic code generation and the developing organization have to implement it themselves. That is not what does supporting automatic code generation in general mean.

The scenario-based software architecture design method has its major strengths in the way evaluation is addressed. In the previous method the evaluation of the design results is left to the architect to deal with in whatever fashion seems appropriate. In the 4+1 View Model the evaluation supports is basically the scripting and what ever conclusions the architect can make of it. At the same time as it also is the strong point of the scenario-based design method, it is a drawback. Since the method does not attach a list of concrete techniques for the architect to choose from it is a drawback for the method.

The transformation part of the method also suffers from the problem that no list of concrete transformation with additional information of its application is part of the method.

Of these three methods [9,16,30], two have activities for determining if the requirements will be fulfilled, i.e. the halt criterion. However, in the 4+1 View Model this is done late, i.e. after the implementation using traditional testing.

3. Software Architecture Description

The description of software architectures is an important issue, since very many persons are dependent on the software architecture for their work, e.g. project managers use it for estimating effort and planning resources, software engineers use it as the input to the software design, etc. The inherent problem in most of software development is the abstract nature of a computer program. Unlike products in other engineering fields, e.g. cars, houses, or airplanes, software is non-tangible and has no natural visualization. Like the remainder of the software industry there is no perfect solution to the description problem. Currently, the most accurate description of the system is its source code, or rather the compiled code since most compilers have their own ways of compiling, structuring and optimizing the source code. Hence the problem with software architecture description is to find a description technique that suites the purposes in software development, i.e. communication, evaluation and implementation. In this section some description techniques will be presented, starting with the most commonly used, boxes and lines.

3.1 Boxes and lines

Most architectures see the first light on a white-board in form of an informal figure consisting of a number of boxes with some text or names and lines to connect the related boxes. The boxes and lines notation is very fast to draw and use on paper and on white-boards. During a work meeting the participants have enough context by following the discussion and modifications to the figure to understand what it means and how it should be interpreted. Without the context given in such a meeting, the description consisting of only boxes with names and lines to connect them could be interpreted in many ways and give but a very coarse picture of the architecture. The modules have important properties that are not described at all by simple boxes, e.g. the public interface of the module, control behavior, and the rationale. This is why the boxes and lines description techniques are needed, but not sufficient. After the first descriptions using boxes and lines, the architecture elements need to be specified in more detail, and for this the description technique needs to be more expressive. In [2] an extended boxes and

lines notation for describing software architecture is presented with a key to the semantic interpretation (see figure 5 for partial description of the notation).

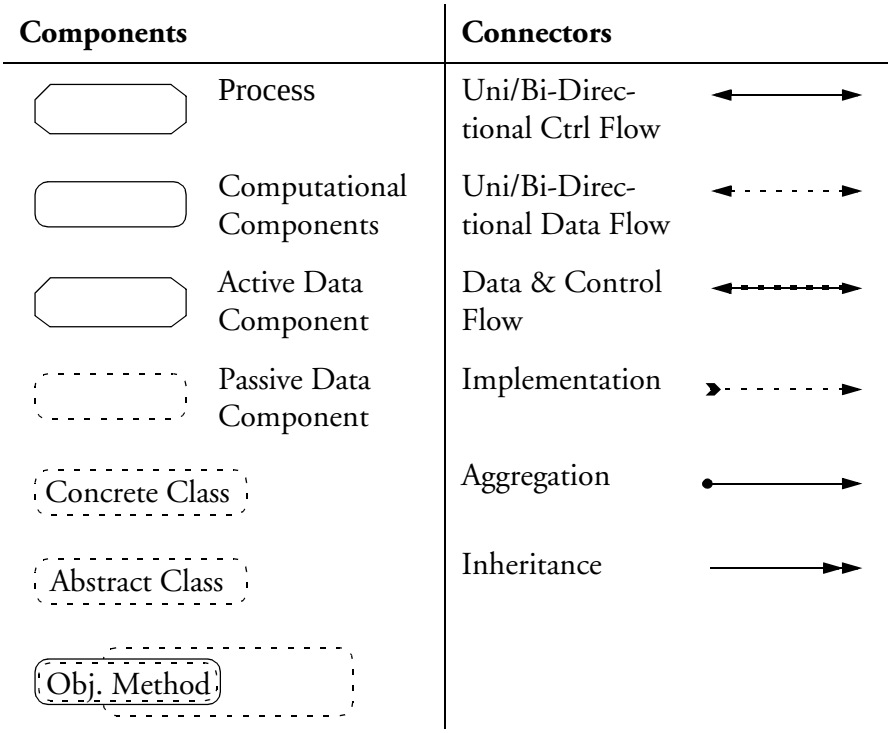

Figure 5. *Software Architecture Notation (from [2])*

The notation includes the concepts of aggregation and inheritance, without a semantic definition. There is a risk of mistaking the inheritance and aggregation concepts proposed in this notation for the semantics of the same words in object-orientation.

Inheritance in software architecture could be interpreted that a module that inherits from another module has the same interface as the super module. In object orientation this is also the case, but the implementation would also be inherited. However, even in object oriented languages the interpretations differ, for example Java and C++ have different inheritance constructs. Java uses the inheritance mechanism for code reuse, whereas the interface construct is used to denote type compliance.

Aggregation also has no definition of its semantic interpretation in software architecture. In object orientation aggregation would be interpreted as an object being composed of other objects, i.e. nested objects. In software architecture, that interpretation would imply nested modules. This, however, seems less probable since the module/component concept, as it is used in industry [8], is that a module could be fairly large and consists of several classes.

Parameterization together with interfaces play an important role in software architecture. When a service of a module is called by another module, the caller also supplies a reference to one of the modules that should be used to carry out the service request. For example, a database request is sent to the database manager module, with references to the database connection to be used for the query and a reference to the container component where the query results are to be stored or processed. This parameterization could be seen as a kind of module relation, but the notation does not allow unambiguous specification.

3.2 Multiple views

Software architecture could be viewed from many perspectives and an approach to describe a software architecture is to describe every relevant aspect of the software architecture.

In section 2.3 the 4 + 1 View Model [16] is presented as a design method, but it is also a way to describe the software architecture from five different perspectives. Every view has an associated description method, and in the 4 +1 View Model three of the are subsets of UML [6], the fourth is the structure of the code and the fifth (+1) is a list of scenarios specified in text.

In [2] the views of the architecture are called architectural structures and every stakeholder are concerned with, at least, one structure. A few of the examples of potential structures are listed here:

1. Module structure, as the basis for work assignments and products. It is mainly used for project planning and resource allocation.
2. Conceptual, or logical structure, as the description of partitioning and abstraction of the system's functional requirements.
3. Process structure, or coordination structure, describes the control flow of the system, mainly in terms of processes and threads.
4. Physical structure describes the hardware entities that are relevant to the system.
5. Uses structure, to show dependencies between modules.
6. Calls structure, to show the utilization of other modules' services or procedures.
7. Data flow, describes the data flow between modules, i.e. what modules send or accept data from other modules.
8. Control flow

According to the authors [2] the views, or structures, are dependent on elements from each other. However, traceability between the views is not obvious. In a small system the similarities between the view are more than the differences, but as the system grows the more significant differences between the views becomes.

3.3 Unified modeling language

The *unified modeling language (UML)* [6] has gained an important role in the design of software today. By unifying the design method and notations [5, 14, 26] the software industry have gained a well thought through design method and notation with a corresponding market of CASE-tools. In UML we find support for classes, abstract classes, relationships, behavior by interaction charts and state machines, grouping in packages, nodes for physical deployment, etc. All this is supported in nine (9) different types of diagrams; class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state chart diagram, activity diagram, component diagram and deployment diagram. For further information about UML in general we refer to [6].

In UML we find support for some software architecture concepts, i.e. components, packages, libraries and collaboration. First, the UML allow description of the components in the software architecture on two main levels, either specifying only the name of the component or specifying the interfaces or the classes implementing the components. The notation for a component in UML is shown in figure 6 and a component diagram in figure 7.

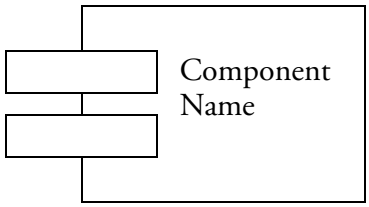

Figure 6. *Component denoted using UML*

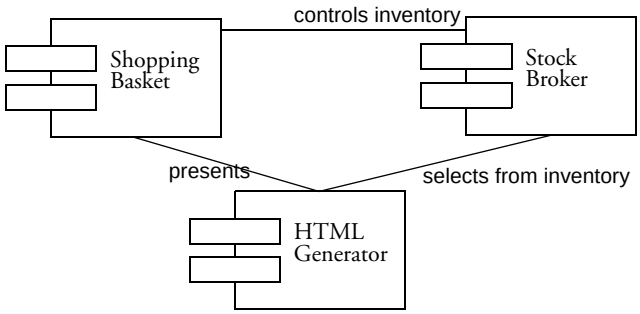

Figure 7. *An example component diagram in UML*

UML provides notational support for describing the deployment of the software components onto physical hardware, i.e. nodes. This corresponds to the physical view in the 4+1 View Model in section 2.3. Deployment of the software allows the software engineers to make fewer assumptions when assessing the qualities of the deployed system. Fewer assumptions help in finding a better suited solution for the specific system. The deployment notation, as shown in figure 8, can be extended to show more details about the components deployed on the specific

nodes. Nodes can be connected to other nodes using the UML notation, see example of a deployment diagram in figure 9.

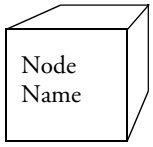

Figure 8. *Node denoted using UML*

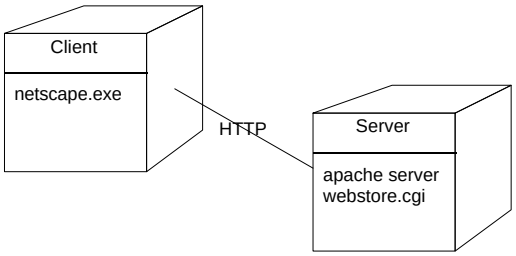

Figure 9. *An example deployment diagram in UML*

Collaborations are sets or societies of classes, interfaces and other elements that collaborate to provide services that beyond the capability of the individual parts. The collaboration has a structural aspect, i.e. the class diagram of the elements involved in the collaboration, and a behavior diagram, i.e. interaction diagrams describing different behavior in different situations. Collaborations also have relationships to other collaborations.

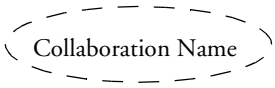

Figure 10. *Notations for collaboration in UML*

Patterns and frameworks are supported in UML by combined usage of packages, components, collaborations and stereotypes. For a more detailed description of stereotypes and the notations for frameworks, design patterns, and architectural patterns, we refer to [6].

3.4 Architecture description languages

More formal approaches to describing software architectures have emerged in form of *architecture description languages* (ADL). In comparison to requirement specification languages that are more in the problem domain, software architecture description languages are more in the solution domain. Most architecture description languages have both a formal textual syntax and a graphical representation that maps to the textual representation. ADLs should, according to [29], have; ability to represent components and connectors, abstraction and encapsulation, types and type checking, and an open interface for analysis tools. And in [17], architecture description languages shall have component and communication abstraction, communication integrity, model support for dynamic architectures, causality and time support, and relativity or comparison.

At this point, over ten architecture description languages have been presented, e.g. Rapide [17], and Unicon [29]. In [12] eight ADLs are surveyed and compared on different features, and in [19] the author proposes a framework for comparison of software architecture description languages and comparison of number of existing architecture description languages.

4. Software Architecture Analysis

The goal with software architecture analysis is to learn about the system to be built with the software architecture. This kind of analysis requires mappings between the software architecture and the system to be built. The accuracy of the results from such analyses are very dependent on how ambiguous these mappings are. The mappings, or semantics, of the elements of the software architecture descriptions are today very unclear. The current state of the research and practice make use of what is available and the semantics of the software architecture description are shared by stories, written English text, and usage of other related description techniques, e.g. UML, OMT, or state charts.

The analysis of software architecture for the purpose of learning about the system that is going to be implemented would benefit from having a clear and universally defined semantics of a software architecture description technique.

Software architecture has much impact on the quality of a software system and it is important to be able to make informed decisions concerning the software architecture in a number of situations. Decision-making regarding software architecture includes:

- compare two alternatives relatively,
- compare the original and the modified software architecture relatively,
- compare one software architecture with the requirements,
- compare a software architecture to a theoretically viable software architecture, or
- grading the software architecture on an interval or absolute scale.

An important source of information is the software architecture itself, and by analyzing the software architecture using different techniques we gather information that allows the stakeholders make more informed decisions about the situation.

The analysis take into account that when the software architecture is designed; detailed design is done on every module in the architecture and the implementation. This is a source of variation in what could be expected from the implemented system. For example, a brilliant team of software engineers may still be able to do a good job with a poor soft-

ware architecture. Or a perfect software architecture may lead to unacceptable results in the hand of a team of inexperienced software engineers that fails to understand the rationale behind the software architecture. The following sections presents some existing software architecture assessment methods.

4.1 Scenario-based architecture assessment method

The *scenario-based architecture assessment method (SAAM)* [2] has been used to assess software architectures before the detailed design and implementation starts. It involves all stakeholders of the architecture and requires a few days to carry out. The goal with the assessment is to make sure that all stakeholders' interests will be accounted for in the architecture.
The steps in SAAM are:

1. *Scenario development* is the activity where each stakeholder lists a number of situations, usage situations, or changes, that are relevant for him/her concerning the system.

2. The software *architecture description* serves as the input together with the scenarios for the subsequent steps of the method. The description should be in a form that is easily understood by all stakeholders.

3. The leader of the SAAM session directs the *classification of scenarios* into direct or indirect scenarios. Direct scenarios means that it is clear that this scenario is no problem to incorporate or execute in the implemented system. Indirect scenarios mean that it is not clear whether a scenario can or cannot be directly incorporated in the architecture. The purpose of the classification is to reduce the number of scenarios that is used as input for the next step in the method.

4. The indirect scenarios are now *evaluated individually* to remove any doubts as to whether or not the scenarios are direct or indirect.

5. The specified scenarios are mostly related to some extent and sometimes require changes to the same components. This is ok when the scenarios' semantics are related closely, but when semantically very different scenarios require changes to the same components it may be an indication of problems. *Scenario interaction assessment* exposes these components in the architecture.

6. In the case when architectures are compared the *overall evaluation* plays an important role. The overall evaluation is the activity of quantifying the results from the assessment. Assigning all scenarios weights of the relative importance does this.

4.2 Architecture trade-off analysis method

A younger cousin to the SAAM method is the *architecture trade-off analysis Method (ATAM)* [15]. In difference to SAAM, focus on finding trade-off points in the architecture from the perspective of the requirements on the product. The method is a spiral model of *design* and has both similarities and differences with the original spiral model [4]. Similar since each iteration takes one to a complete understanding of the system. Different since no implementation is involved. Further, the method prescribes exact analytic methods for assessing the quality attributes of the system, but relies on the existence of such techniques for the quality attributes relevant to each case. All in all it is a method for using the available analysis methods to learn more about the architecture. These are the steps of the method along with a brief description of each:

1. Collect Scenarios. The situations that are to be analyses are described using scenarios. Scenarios are elicited from system stakeholders and serve the same purpose as in SAAM [2].
2. Collect Requirements / Constraints / Environment. In this step the *attribute-based* requirements, i.e. quality requirements, are identified, characterized and made measurable.
3. Describe Architectural Views. The system architecture is described using the views relevant to the requirements from step 2. Competing alternatives are all specified here as well.
4. Attribute-Specific Analysis. In this step the quality specific analysis are applied, in any order and results in statements about the qualities tied to a specific alternative.
5. Identify Sensitivities. In the architecture, changes certain points will affect the results of the analysis significantly and these spots are considered sensitivity points.
6. Identify Trade-offs. Identifying the trade-off points means finding the elements in the architecture that are sensitivities for multiple quality requirements.

4.3 Architecture discoveries and reviews

At AT&T architecture assessment have been divided into two kinds, *architecture discovery* and *architecture review* [1]. Architecture discovery is used very early in the project and helps the development group make decisions and balance benefits and risks. The motivation for doing architecture discovery is to find and evaluate alternatives and associate risks. The second assessment type, architecture review, is done before any coding begins (compare to SAAM). The goal is to assure that the architecture is complete and identify potential problems. The best technical people not belonging to project perform architecture assessments. The review team chairperson and the project's lead architect assign reviewers in cooperation to ensure that the review team incorporates the top competence in *all* areas relevant to the project. The strategy of architecture discoveries and reviews have been evaluated empirically with promising results.

5. Contributions in this Thesis

The challenge facing the software architect is to find an optimal balance in software qualities to make the implemented application fulfil its quality requirements.

We have learned from our projects with industry [7, 10, 20] and presented in [Paper I] that a system is never a pure real-time system, a pure fault-tolerant system, or a pure reusable system. Instead, systems should provide several of these properties. However, software quality requirements are often in conflict [3, 18], e.g., real-time versus reusability, flexibility versus efficiency, reliability versus flexibility, etc. The difficulty of system design originates from the need for carefully balancing of the various software qualities. It is far too common with applications that do not have the required balance of, for example, performance, scalability, maintainability, etc. Software development very often take the form of design, implementation and test, where test is the activity of verifying all requirements, including the quality requirements as well. However, it is not satisfying, for neither the developer, nor the customer, to find that performance is threatening the usage of the system. At that time, fixing means doing most of the work all over again.

In [Paper II], we propose a method for reengineering software architectures to meet quality requirements, and give a real-world example. The method does not change the fact that much resources and effort have been spent, partly, in vain and that it will be costly to reengineer to meet the requirements. But, the method proposed will reduce the risk of repeating the failure a second time by using a cycle of *transformation*, i.e. modification, and *evaluation*. Every modification proposed is evaluated with respect to the quality requirements. Hence, we get much earlier indications if and how well the requirements will be met and what requirements were affected by the modification. The quality attributes will not, most likely, be affected in the same direction by each modification, i.e. some trade-offs may exist that makes, for example, reliability decrease, in return for an increase in performance. It is the goal with the presented method to clearly visualize these trade-offs and allow the designer to make informed design decisions. The contribution is a practical method for reengineering software architectures, an example of its application to a real-world example.

The software architecture reengineering method [Paper II] and the ARCS design method [9] depend mainly on two things. First, the trans-

formations needed to be better understood in terms of how a transformation affects quality attributes. Transformations are represented mostly by design patterns and architecture patterns, where only a general idea of the impact on quality attributes exists. Secondly, it is assumed that evaluation techniques for 'all' quality attributes are available for architecture level evaluation, which is not the case. The problem is that the assessment, or analysis, methods generally are concerned with source code and do not work with the information available at the architectural level. Or the methods require to much details and effort to use in short design cycles as described in the reengineering method. The latter is often true for the method from research communities specializing on one single quality attribute, e.g. performance, real-timeliness etc. This problem is addressed in [Paper III], where we propose a method for assessing software maintenance using change scenario profiles.

The empirical validation of the prediction methods has one obvious solution, i.e. find a set of cases, make predictions, and study their maintenance. The problem is that the life cycle of a system spans over years, and without any preliminary results it is very hard to find companies that will allow their projects to be studied. Another way is to state the underlying hypotheses and challenge them individually. The method for assessing software maintenance from software architecture presented in [Paper III] builds on a set of hypotheses.

1. Scenario profiles must be able to represent future changes of a system.

2. Scenario profiles may not vary too much between individuals or groups, when specified for the same system.

3. Each scenario is used as input for doing impact analysis on the architecture, and impact analysis must be reasonably accurate.

In [Paper IV] hypothesis number three have been studied in a quasi experiment and the results shows that an individually prepared group creates the best change scenario profiles. It also shows that the variation between change profiles by individual persons are rather big, which is an argument for using a group to specify change scenario profilesd. Hypothesis one and two remain to be studied and are discussed in section 6.

To better understand the line of reasoning behind the contribution of this thesis, the work is depicted as an upside down triangle in figure 11. Starting from the top with the experiences gained from research projects, [Paper I]. The understanding of the need for engineering approaches to software architecture design lead to the proposal of the reengineering method in [Paper II]. To better support the presented method we started studying the assessment of quality attributes. It soon became apparent that the sheer number of different quality attributes and their variants would make the task unmanageable for the time period available. The decision was made to focus on the software maintainability quality attribute, and the result was a method proposal in [Paper III]. An experiment, in [Paper IV], addresses the empirical validation of the method.

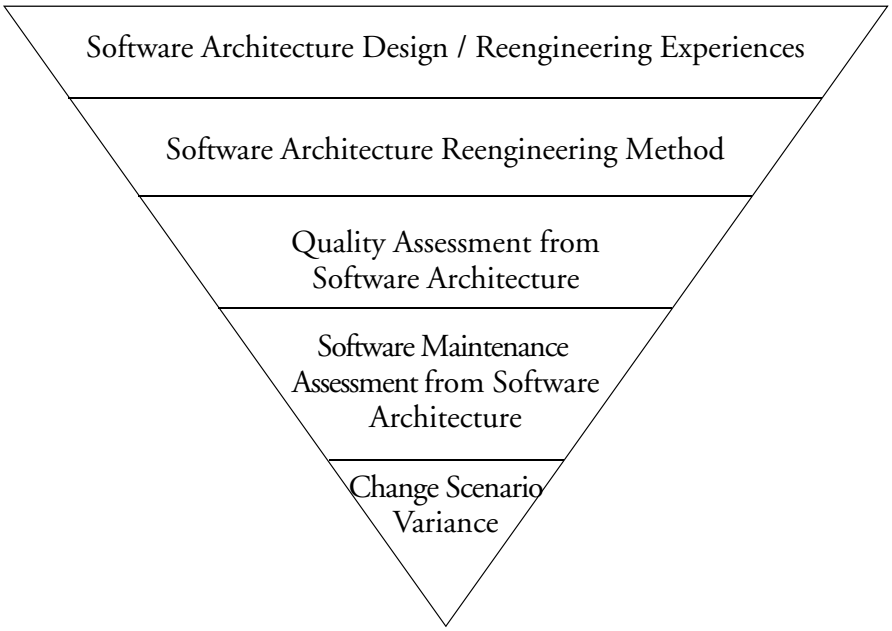

Figure 11. *My Research Path Through Software Architecture Design*

5.1 List of papers

Below is a list of the papers included in this thesis, along with their publication status:

[Paper I] Haemo Dialysis Software Architecture Design Experiences
PerOlof Bengtsson & Jan Bosch
Proceedings of ICSE'99, International Conference on Software Engineering, Los Angeles, USA, 1999.

[Paper II] Scenario-based Software Architecture Reengineering
PerOlof Bengtsson & Jan Bosch
Proceedings of ICSR'5, International Conference on Software Reuse, Victoria, Canada, June 1998

[Paper III] Architecture Level Prediction of Software Maintenance
PerOlof Bengtsson & Jan Bosch
Proceedings of CSMR'3, 3rd European Conference on Software Maintenance and Reengineering, Amsterdam, March 1999.

[Paper IV] An Experiment on Creating Scenario Profiles for Software Change
PerOlof Bengtsson & Jan Bosch
Research Report 99/2, ISSN 1103-1581, ISRN HK-R-RES--99/6--SE, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, 1999. (Submitted)

6. Further Research

The goal of my future research is to further challenge the underlying hypotheses of the software maintenance prediction method. Two hypotheses remain to be studied, i.e. if the accuracy of architecture change impact analysis is sufficient and if scenario profiles represent future changes well enough. Plans for their validation will be described in the remainder of this section.

When predicting software maintenance using change scenario profiles it is important the change profile accurately represents the future changes of the system. The problem is that the current state of the art does not provide any techniques for such validation. The intuitive way to study this suffers from the same calendar time problem that the validation of the method as a whole (see previous section). It would require a number of change profiles be created, for a number of projects, by a number of persons, studies of the maintenance of these projects and finally a comparison of the change profiles and the actual changes done to the system. Instead the approach we will take is to try and state underlying hypotheses for the main hypothesis and study them first.

The first hypothesis in the method proposal that remains to be validated is that impact analysis on the architecture level is accurate enough for use in the prediction method. Two steps in the validation of this hypothesis are required. First, the accuracy required of the impact analysis for prediction purposes need to be established. Assuming the need for certain accuracy in the prediction, e.g. +/-10%, sensitivity calculations on the variables of prediction model can show the accuracy required from the impact analysis. Second, an experiment will establish the accuracy of the impact analysis. A preliminary design of the experiment is to take a sample of software engineers, let them estimate the impact of change for a controlled set of change scenarios. Then another sample of software engineers implement the changes described in the scenarios. Finally, the modification of the implemented changes and the predicted modification are compared. One of two possible control groups may be that the change impact are estimated without the support of an software architecture, and the other control group may be that the impact analysis are done with support from design and/or source code.

The second hypothesis in the method proposal that remains to be validated is that scenario profiles are good representations of the changes

of a future system or system release. We can think of two ways of studying this. First, a case study approach could be used and have a number of persons create change profiles for a set of real projects that are under development. Then we carefully study the systems during their life time and gather the modification data and compare it to the change profiles created earlier. This approach will also have to be carried out, but if suffers from the life-cycle time problem and could take years to produce results. Second, a historical case study could be carried out. Let a number of persons unacquainted with an older system created change profiles for the system based on the original specifications. Then the modification history is collected and compared to the scenario profiles. A major threat to this approach is that the persons creating the scenario profiles have some knowledge of the evolution of the domain in which the system belongs and hence create the scenario profile based on this. The threat could be compared to betting on horses after the races have been run.

The empirical validation of the method as a whole is finally addressed. We cannot work around the fact that empirical validation of the method will have to take time, i.e. more than the life-cycle time of the software system studied. At this point in the studies of the prediction method, the method is much better supported by partial results and assessing the chance being able to validate the method much higher. Previous studies have given enough knowledge to allow a good design of a multiple case study for studying the predictive accuracy of the method. The multiple case study approach will be used very similar to the design and use of experiments, with literal and theoretical replication, and not, to gain statistical strength by using larger a sample. A preliminary design would be to have 3-6 projects that should be predicted reasonably well, 2-3 projects that use other techniques on the software architecture level, and finally, 2-3 projects that use the state-of-the-art at any level, e.g. make use of source code. This design would help characterize the prediction method for its accuracy, and bench marks it towards the existing alternatives. After successfully establishing the accuracy of the method, the real challenge of disseminating the results and the method to practitioners, remain.

References

[1] A. Avritzer, E.J. Weyuker, "Investigating Metrics for Architectural Assessment", *5th. International Symposium on Software Metrics*, IEEE, Nov 1998.

[2] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison Wesley, 1998

[3] B. Boehm, 'Aids for Identifying Conflicts Among Quality Requirements,' *International Conference on Requirements Engineering (ICRE96),* Colorado, April 1996, and IEEE Software, March 1996.

[4] B. Boehm, 'A Spiral Model of Software Development and Enhancement', *ACM Software Engineering Notes*, 11(4), pp. 22-42, 1986.

[5] G. Booch, *Object-Oriented Analysis and Design with Applications* (2nd edition), Benjamin/Cummings Publishing Company, 1994.

[6] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Object Technology Series, Addison-Wesley, October 1998.

[7] J. Bosch, 'Design of an Object-Oriented Measurement System Framework', in *Object-Oriented Application Frameworks*, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, 1999.

[8] J. Bosch, 'Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study', in proceedings of the *First Working IFIP Conference on Software Architecture*, October 1998.

[9] J. Bosch, P. Molin, 'Software Architecture Design: Evaluation and Transformation', in proceedings of *1999 IEEE Engineering of Computer Based Systems Symposium* (ECBS99), Nashville, USA, March 1999

[10] J. Bosch, P. Molin, M. Mattsson, PO. Bengtsson, 'Object-oriented Frameworks: Problems and Experiences,' in *Object-Oriented Application Frameworks*, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, 1999.

[11] F. Buschmann, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996.

[12] P. C. Clements, "A Survey of Architecture Description Languages", *Eighth International Workshop on Software Specification and Design*, Germany, March 1996.

[13] R. Gamma et. al., *Design Patterns Elements of Reusable Design*, Addison.Wesley, 1995.

[14] I. Jacobson, et. al., *Object-oriented software engineering. A use case approach*, Addison-Wesley, 1992.

[15] R. Kazman, et. al., "The Architecture Tradeoff Analysis Method", in proceedings of *4th Int'l Conference on Engineering of Complex Computer Systems* (ICECCS98), Aug. 98.

[16] P.B. Kruchten, 'The 4+1 View Model of Architecture,' *IEEE Software*, pp. 42-50, November 1995.

[17] D. C. Luckham, et. al., 'Specification and Analysis of System Architecture Using Rapide', *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, 21(4):336-355, April 1995

[18] J.A. McCall, 'Quality Factors', *Software Engineering Encyclopedia*, Vol 2, J.J. Marciniak (ed.), Wiley, 1994, pp. 958 - 971

[19] N. Medvedovic, R.N. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages", In Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, September 1997.

[20] P. Molin, L. Ohlsson, 'Points & Deviations - A pattern language for fire alarm systems,' Martin, Riehle, Buschmann (eds.), *Pattern Languages of Program Design 3*, Addison-Wesley, 1998

[21] C. R. Morris, C. H. Ferguson, 'How Architecture Wins Technology Wars', Harward Business Review, March-April 1993, pp. 86-96.

[22] Parnas, D.L, "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058.

[23] Parnas, Clements and Weiss , 'On the Modular Structure of Complex Systems', IEEE Transactions on Software Engineering, Vol SE-11, No. 3, March 1985.

[24] D.E. Perry, A.L.Wolf, 'Foundations for the Study of Software Architecture', *Software Engineering Notes*, Vol. 17, No. 4, pp. 40-52, October 1992.

[25] K. Rubin, A. Goldberg, "Object Behaviour Analysis", *Communications of ACM*, September 1992, pp. 48-62

[26] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.

[27] E. Sandewall, C. Strömberg, and H. Sörensen, "Software Architecture Based on Communicating Residential Environments", *Fifth International Conference on Software Engineering (ICSE'81)*, San Diego, CA, IEEE Computer Society Press, March. 1981, pp. 144-152.

[28] M. Shaw, D. Garlan, *Software Architecture - Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[29] M.Shaw, et. al., 'Abstractions for Software Architecture and Tools to Support Them', *IEEE Transactions on Software Engineering*, 21, 4, April 1995

[30] S. Shlaer, S.J. Mellor, 'Recursive Design of an Application-Independentt Architecture', *IEEE Software,* pp. 61-72, January/February 1997.

I

PAPER I

Haemo Dialysis Software Architecture Design Experiences

PerOlof Bengtsson & Jan Bosch

Published in the proceedings of ICSE'99, International Conference on Software Engineering, Los Angeles, USA, 1999.

Abstract

In this paper we present the experiences and architecture from a research project conducted in cooperation with two industry partners. The goal of the project was to reengineer an existing system for haemo dialysis machines into a domain specific software architecture [23]. Our main experiences are (1) architecture design is an iterative and incremental process, (2) software qualities require a context, (3) quality attribute assessment methods are too detailed for use during architectural design, (4) application domain concepts are not the best abstractions, (5) aesthetics guides the architect in finding potential weaknesses in the architecture, (6) it is extremely hard to decide when an architecture design is ready, and (7) documenting software architectures is a important problem. We also present the architecture and the design rational to give a basis for our experiences. We evaluated the resulting architecture by implementing a prototype application.

1. Introduction

Software architecture design is an art. Today only a few, sketchy methods exist for designing software architecture [3,14,15,16]. The challenge facing the software architect is to find an optimal balance in software

qualities to make the resulting application able to fulfil its quality requirements. The tools and techniques available for the software architect are scarce, i.e. design patterns [11], software architecture patterns [7], and various ADLs with accompanying analysis tools [9, 17]. In this list of tools and techniques we are missing time-proven methods for evaluation and assessment of architecture and software architecture design methods. Proposals exist, but none has been proven by time. In our work towards better and more efficient methods for design and assessment of software architecture we have participated in research and design projects with a number of industry partners [3, 6, 20]. These projects have given us some hard-earned hands on experience of what really makes the design of software architecture difficult.

In the next section we present the case studied in this paper, i.e., haemo dialysis machines. In section 3, we present and discuss our experiences. Further motivation for our experiences is given in section 4 were we present the archetypes, the architecture and the design rationale. Finally we present our conclusions of the paper in section 5.

2. Case: Haemo Dialysis Machines

Haemo dialysis systems present an area in the domain of medical equipment where competition has been increasing drastically during recent years. The aim of a dialysis system is to remove water and certain natural waste products from the patient's blood. Patients that have, generally serious, kidney problems and consequently produce little or no urine use this type of system. The dialysis system replaces this natural process with an artificial one.

The research project aimed at designing a new software architecture for the dialysis machines produced by Althin Medical. The software of the existing generation products was exceedingly hard to maintain and certify. The partners involved in the project were Althin Medical, EC-Gruppen and the University of Karlskrona/Ronneby. The goal for EC-Gruppen was to study novel ways of constructing embedded systems, whereas our goal was to study the process of designing software architecture and to collect experiences. As a research method, we used *Action Research* [2], i.e. researchers actively participated in the design process and reflected on the process and the results.

An overview of a dialysis system is presented in figure 1. The system is physically separated into two parts by the dialysis membrane. On the left side the dialysis fluid circuit takes the water from a supply of a certain purity (not necessarily sterile), dialysis concentrate is added using a pump. A sensor monitors the concentration of the dialysis fluid and the measured value is used to control the pump. A second pump maintains the flow of dialysis fluid, whereas a third pump increases the flow and thus reduces the pressure at the dialysis fluid side. This is needed to pull the waste products from the patient's blood through the membrane into the dialysis fluid. A constant flow of dialysis fluid is maintained by the hydro mechanic devices that ensure exact and steady flow on each side (rectangle with a curl).

On the right side of figure 1, the extra corporal circuit, i.e. the blood-part, has a pump for maintaining a specified blood flow on its side of the membrane. The patient is connected to this part through two needles usually located in the arm that take blood to and from the patient. The extra corporal circuit uses a number of sensors, e.g. for identifying air bubbles, and actuators, e.g. a heparin pump to avoid cluttering of the patients blood while it is outside the body. However, these details are omitted since they are not needed for the discussion in the paper.

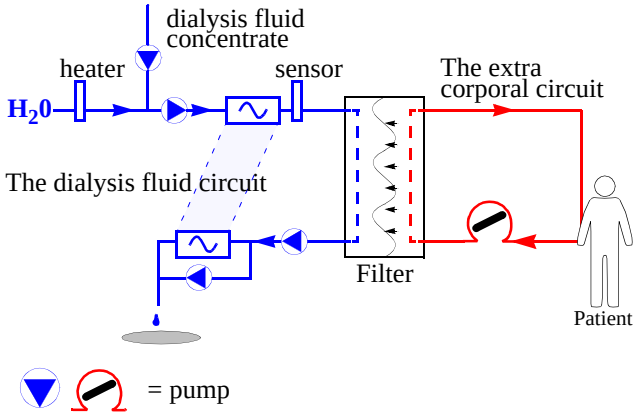

Figure 1. *Schematic of Haemo Dialysis Machine*

The dialysis process, or *treatment*, is by no means a standard process. A fair collection of treatments exists including, for example, Haemo Dialysis Filtration (HDF) and Ultra Filtration (UF) and other variations, such as single needle/single pump, double needle/single pump. Treatments are changed due to new research results but also since the effectiveness of a particular treatment decreases when it is used too long

for a patient. Although the abstract function of a dialysis system is constant, a considerable set of variations exists already. Based on experience the involved company anticipates several additional changes to the software, hardware and mechanical parts of the system that will be necessary in response to developments in medical research.

2.1 Legacy Architecture

As an input to the project, the original application architecture was used. This architecture had evolved from being only a couple of thousand lines of code very close to the hardware to close to a hundred thousands lines mostly on a higher level then the hardware API. The system runs on a PC-board equivalent using a real-time kernel/operating system. It has a graphical user interface and displays data using different kinds of widgets. It is a quite complex piece of software and because of its unintended evolution, the structure that was once present has deterioated substantially. The three major software subsystems are the Man Machine Interface (MMI), the Control System, and the Protective system (see figure 2).

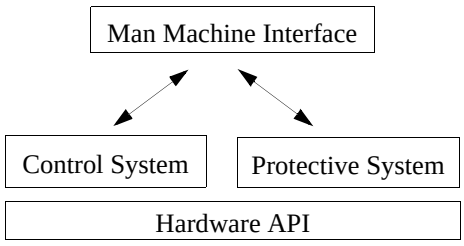

Figure 2. *Legacy system decomposition*

The *MMI* has the responsibilities of presenting data and alarms the user, i.e. a nurse, and getting input, i.e., commands or treatment data, from the user and setting the protective and control system in the correct modes.

The *control system* is responsible for maintaining the values set by the user and adjusting the values according to the treatment selected for the time being. The control system is not a tight-loop process control system, only a few such loops exists, most of them low-level and implemented in hardware.

The *protective system* is responsible for detecting any hazard situation where the patient might be hurt. It is supposed to be as separate from the other parts of the system as possible and usually runs on a own task or process. When detecting a hazard, the protective system raises an alarm and engages a process of returning the system to a safe-state. Usually, the safe-state is stopping the blood flow or dialysis-fluid flow.

The documented structure of the system is no more fine-grained than this and to do any change impact analysis, extensive knowledge of the source code is required.

2.2 Requirements

The aim during architectural design is to optimize the potential of the architecture (and the system built based on it) to fulfil the software quality requirements. For dialysis systems, the driving software quality requirements are *maintainability*, *reusability*, *safety*, *real-timeliness* and *demonstrability*. Below, these quality requirements are described in the context of dialysis systems.

Maintainability

Past haemo dialysis machines produced by our partner company have proven to be hard to maintain. Each release of software with bug corrections and function extensions have made the software harder and harder to comprehend and maintain. One of the major requirements for the software architecture for the new dialysis system family is that maintainability should be considerably better than the existing systems, with respect to *corrective* but especially *adaptive* maintenance:

1. *Corrective maintenance* has been hard in the existing systems since dependencies between different parts of the software have been hard to identify and visualize.

2. *Adaptive maintenance* is initiated by a constant stream of new and changing requirements. Examples include new mechanical components as pumps, heaters and AD/DA converters, but also new treatments, control algorithms and safety regulations. All these new requirements need to be introduced in the system as easily as possible. Changes to the mechanics or hardware of the system

almost always require changes to the software as well. In the existing system, all these extensions have deteriorated the structure, and consequently the maintainability, of the software and subsequent changes are harder to implement. Adaptive maintainability was perhaps the most important requirement on the system.

Reusability

The software developed for the dialysis machine should be reusable. Already today there are different models of haemo dialysis machines and market requirements for customization will most probably require a larger number of haemo dialysis models. Of course, the reuse level between different haemo dialysis machine models should be high.

Safety

Haemo dialysis machines operate as an extension of the patients blood flow and numerous situations could appear that are harmful and possibly even lethal to the patient. Since the safety of the patient has very high priority, the system has extremely strict safety requirements. The haemo dialysis system may not expose the dialysis patient to any hazard, but should detect the rise of such conditions and return the dialysis machine and the patient to a state which present no danger to the patient, i.e. a safe-state. Actions, like stopping the dialysis fluid if concentrations are out of range and stopping the blood flow if air bubbles are detected in the extra corporal system, are such protective measures to achieve a safe state. This requirement have to some extent already been transformed into functional requirements by the safety requirements standard for haemo dialysis machines [8], but only as far as to define a number of hazard situations, corresponding thresh-hold values and the method to use for achieving the safe-state. However, a number of other criteria affecting safety are not dealt with. For example, if the communication with a pump fails, the system should be able to determine the risk and deal with it as necessary, i.e. achieving safe state and notify the nurse that a service technician is required.

Real-timeliness

The process of haemo dialysis is, by nature, not a very time critical process, in the sense that actions must be taken within a few milli- or microseconds during normal operation. During a typical treatment, once the flows, concentrations and temperatures are set, the process only requires monitoring. However, response time becomes important when a hazard or fault condition arises. In the case of a detected hazard, e.g. air is detected in the extra corporal unit, the haemo dialysis machine must react very quickly to immediately return the system to a safe state. Timings for these situation are presented in the safety standard for haemo dialysis machines [8].

Demonstrability

As previously stated, the patient safety is very important. To ensure that haemo dialysis machines that are sold adhere to the regulations for safety, an independent certification institute must certify each construction. The certification process is repeated for every (major) new release of the software which substantially increases the cost for developing and maintaining the haemo dialysis machines. One way to reduce the cost for certification is to make it easy to demonstrate that the software performs the required safety functions as required. This requirement we denote as *demonstrability*.

2.3 Design Method

In the project we used the design method of our research group ARchitecture and Composition of Software (ARCS) for designing the architecture [3], presented in figure 3. The ARCS method starts with the requirement specification. From this input data, the architect synthesizes an architecture primarily based on the functional requirements. This first version of the architecture contains the initial archetypes. Our definition and usage of the term 'archetype' differs from [22]. We define the archetype as a basic abstraction, which is used to model the application architecture. The archetypes generally evolve through the design iterations.

The architecture is evaluated through the use of different evaluation techniques. The ARCS method uses four evaluation approaches:

- *Scenario-based evaluation* is techniques were the software qualities are expressed as typical or likely scenarios. For example, maintainability could be expressed as *change scenarios* defining likely changes and the implementation of the changes should require minimal modification to the architecture.
- *Mathematical modeling* (including metrics & statistics) is a technique were product and process data are used to make predictions about the potential qualities of a resulting product or task.
- *Simulation* is a technique similar to scenarios, but more suitable to dynamic properties, such as performance and reliability. The architecture is modeled in a simulation environment and its behavior is used to predict the software quality attribute. For example, safety could be evaluated by simulating the execution of the haemo dialysis architecture in different hazard situations.
- *Experience-based reasoning* is the technique that is most widely used and serves as a suitable complement to other techniques. Experience designers often intuitively identifies designs that are not addressing certain quality requirements adequately. Based on the initial identification, further investigation may be performed using the other more objective techniques.

If the results show that the potential for the software qualities is sufficient, the architecture design is finished. Generally, the evaluation of the initial architecture reveals a number of deficiencies. To address these, the designer transforms the architecture into a new version, using a set of available transformations. In the ARCS method, five categories of transformations are identified:

- *Applying an architecture style* result in changes to the overall structure.
- *Applying and architecture pattern* add certain behavioral rules to the architecture, e.g. Periodic Objects [20].
- *Applying design patterns* impact only a few elements of the architecture.
- *Converting quality requirement to functionality*, e.g., handling robustness by introducing exception handling.

- *Distributing Requirements*. For example, response time requirements on the whole system may be decomposed into response time requirements for individual elements.

These transformations only reorganize the domain functionality and affect only the software quality attributes of an architecture. After a set of transformations, architecture evaluation is repeated and the process is iterated until the quality requirements are fulfilled. The method may appear similar to the spiral model presented in [4], but some important differences in focus and scope exist.

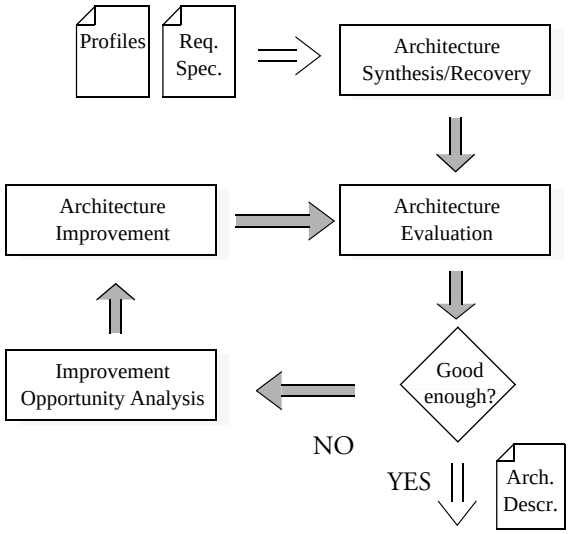

Figure 3. *Repeated evaluation for control of the design*

3. Lessons Learned

During the architecture design project, we gathered a number of experiences that, we believe, have validity in a more general context than the project itself. In this section, we present the lessons that we learned. In the next section, the architecture design process leading to them and the foundations for our experiences are presented.

3.1 Quality requirements without context

Different from functional requirements, quality requirements are often rather hard to specify. For instance, one of the driving quality requirements in this project was maintainability. The requirement from Althin Medical, however, was that maintainability should be "as good as possible" and "considerably better than the current system". In other projects, we have seen formulations such as "high maintainability". Even in the case where the IEEE standard definitions [12] are used for specifying the requirements, such formulations are useless from a design and evaluation perspective. For example, maintainability mean different things for different applications, i.e. haemo dialysis machine software and a word processor have totally different maintenance. The concrete semantics of a quality attribute, like maintainability, is dependent on its context. The functional requirements play an important role in providing this context, but are not enough for the designer to comprehend what actual maintenance tasks can be expected.

Based on our experience, we are convinced that quality requirements should be accompanied with some context that facilitates assessment. The nature of the context depends on the quality requirement. For instance, to assess maintainability, one may use a *maintenance profile*, i.e. a set of possible maintenance scenarios with an associated likelihood. To assess performance, one requires data on the underlying hardware and a *usage profile*. Based on these profiles, one is able to perform an objective analysis of the quality attributes. Every quality requirement requires its own context, although some profiles, e.g., the usage profile, can be shared by multiple contexts.

Since the customer had specified the quality requirements rather vaguely, we were forced to define them in more detail. We felt that the time needed to specify the profiles was well worth the effort. It serves as a mental tool for thinking what the real effects on the system and its usage are. Also it helps to separate different qualities from each other, as they are influencing each other in different ways. Finally, the profiles can be used for most forms of assessment, including simulation.

3.2 Too large assessment efforts

For each of the driving quality requirements of the dialysis system architecture, research communities exist that have developed detailed assess-

ment and evaluation methods for their quality attribute. In our experience, these techniques suffer from three major problems in the context of architecture assessment. First, they focus on a single quality attribute and ignore other, equally important, attributes. Second, they tend to be very detailed and elaborate in the analysis, requiring, sometimes, excessive amounts of time to perform a complete analysis. And finally the techniques are generally intended for the later design phases and often require detailed information not yet available during architecture design.

Since software architects generally have to balance a set of quality requirements, lack the data required by the aforementioned techniques and work under time pressure, the result is that, during architectural design, assessment is performed in an ad-hoc, intuition-based manner, without support from more formal techniques. Although some work e.g., [15], is performed in this area, there still is a considerable need for easy to use architecture assessment techniques for the various quality attributes, preferably with (integrated) tool support.

3.3 Architecture abstractions outside application domain

Traditional object oriented design methods, like [5, 13, 21, 24], provide hints and guidelines for finding the appropriate abstractions for the object oriented design. A common guideline is to take the significant concepts from the problem domain and objectify them. However, in this project as well as in a number of other projects, we observed that some or several of the architectural abstractions, or archetypes, used in the final version did not exist (directly) in the application domain. Instead, these archetypes emerged during the design iterations and represented abstract domain functionality organized to optimize the driving quality attributes.

We found that when a true understanding of the concept and its relations emerges, we found the most suitable abstraction. For example, during the first design iteration, we used the domain concepts we had learned from studying the documentation and talking to domain experts. As we came to know the requirements and expected behavior of the system, we iterated the design and the abstractions used in the architecture design were changed from domain concepts to archetypes that

incorporate the quality requirements. During the design iterations, we became more and more aware of how the quality requirements would have to work in *cooperation.* For example, even though using design patterns might help with flexibility in some cases, the demonstrability and real-timeliness became hard to ensure and thus other abstractions had to be found.

3.4 Architecting is iterative

After the design of the dialysis system architecture, but also based on earlier design experiences, we have come to the conclusion that designing architectures is necessarily an iterative activity and that it is impossible to get it completely right the first time. We designed the software architecture in two types of activities, i.e. individual design and group meeting design. We learned that group meetings and design teams meeting for two-three hours were extremely efficient compared to merging single individuals designs. Although one or two were responsible for putting things on paper and dealing with the details, virtually all creative design and redesign work was performed during these meetings.

In the case where one individual would work alone on the architecture it was very easy to get stuck with one or more problems. The problems were, in almost every case, resolved the next design meeting. We believe that the major reason for this phenomenon is that the architecture design activity requires the architect to have almost all requirements, functional and quality, in mind at the same time. The design group have a better chance in managing the volume and still come up with creative solutions. In the group, it is possible for a person to disregard certain aspects to find new solutions. The other group members will ensure that those aspects are not forgotten.

Another problems we quickly discovered was that design decisions were forgotten or hard to remember between the meetings. We started early with writing design notes. The notes were very short, a few lines, with sketches where helpful. First, it helped us to understand why changes were made from previous design meetings. Secondly, it also made it easier to put together the rationale section of the architecture documentation. At some points, undocumented design decisions were questioned at a later stage and it took the quite some time to reconstruct the original decision process.

The design notes we used were not exposed to inspections, configuration management or other formalizing activities. As such an informal document it was easy to write during the meeting. In fact, the designers soon learned to stop and have things written down for later reference. Since the sole purpose is to support the memory of the designers, often a date and numbering of the notes is enough.

3.5 Design aesthetics

The design activity is equally much a search for an *aesthetically appealing design* as it is searching and evaluating the balance of software qualities. The *feeling* of a good design worked as a good indicator when alternatives were compared and design decisions needed to be made. In addition, the feeling of disliking an architecture design often sparked a more thorough analysis and evaluation to find what was wrong. Most often, the notion proved correct and the analysis showed weaknesses.

According to our experience, the sense of a aesthetic design was often shared within the group. When differences in opinions existed, the problem or strength could be explained using a formal framework and we reached consensus. It is our belief that a software designer with roughly the same amount of experience outside the project would experience the same feeling of aesthetic design. That is, although the "feeling" is not objective, it is at least intersubjective.

Since this intuitive and creative aspect of architecture design triggered much of the formal activities and analyses, we recognize it as very important. However, design methods, techniques and processes do not mention nor provide this as a part. It is not a secret but nor is it articulated very often how important this gut feeling is to software design.

3.6 Are we done?

We found it hard to decide when the design of the software architecture had reached its end criteria. One important reason is that software engineers are generally interested in technically perfect solutions and that each design is approaching perfectness asymptotically, but never reaches it completely. Architecture design requires balancing requirements and, in the compromise, requirements generally need to be weakened. Even if a first solution is found, one continues to search for solutions that

require less weakening of requirements. Also, we found it very hard to decide when the architecture design was not architecture design anymore but had turned into detailed design.

A second important reason making it hard to decide whether a design is finished is that a detailed evaluation giving sufficient insight in the attributes of an architecture design is expensive, consuming considerable time and resources. Engineers tend to delay the detailed evaluation until it is rather certain that the architecture fulfils its requirements. Often, the design has passed that point considerably earlier. As we identified in section 1, there is a considerable need for inexpensive evaluation and assessment techniques, preferably with tool support.

3.7 Documenting the essence of a software architecture

During the architecture design only rudimentary documentation was done, i.e. sketchy class diagrams and design notes. When we delivered the architecture to detailed design it had to be more complete. We tried to use the 4+1 View Model [16], but found it hard to capture the essence of the architecture. Project members that had not participated in the design of the new architecture had to read the documentation and try to reconstruct this essence themselves. We have not yet been able to understand what the essence of a software architecture are, but we feel that its not equivalent with design rationale.

However, since we were able to communicate with the designers and implementers, we could overcome the problems with the documentation. The problem was put on its edge, when we started writing this paper. This time, we would not get a chance to communicate the architecture and its essence with any other means than this document. It is our opinion that although many of the aspects of this architecture are presented in this paper, the essence still remain undocumented.

4. Architecture

The haemo dialysis architecture project started out with a very informal description of the legacy architecture, conveyed both in text and figures and via several discussions during our design meetings. For describing the resulting architecture we use two of the four views from the 4+1 View Model [16], i.e. Logical View and Dynamic View. The develop-

ment view we omit since it do not contribute to the understanding of the design decisions, trade offs and experiences. We also omit the physical view since the hardware is basically a single processor system. However, we feel that it is appropriate to add another subsection of our architecture description, i.e. archetypes. During the design iterations we focused on finding suitable archetypes which allowed us to easily model the haemo dialysis architecture and its variants. The archetypes are very central to the design and important for understanding the haemo dialysis application architecture.

4.1 Logic Archetypes

When we started the re-design of the software architecture for the haemo dialysis machine, we were very much concerned with two things; the maintainability and the demonstrability.

We knew that the system had to be maintained by others than us, which meant that the archetypes we used, would have to make sense to them and that the form rules were not limiting and easy to comprehend. Also, we figured, that if we could choose archetypes such that the system was easy to comprehend the effort to show what the system does becomes smaller. We realized that much of the changes would come from the MMI and new treatments and we needed the specification and implementation of a treatment to be easy and direct. Our aim was to make the implementation of the treatments look comparable to the description of a treatment written by a domain expert using his or hers own terminology on a piece of paper.

After three major iterations we decided on the Device/Control abstraction, which contained the following archetypes and their relations (figure 4):

Device

The system is modeled as a device hierarchy, starting with the entities close to the hardware up to the complete system. For every device, there are zero or more sub-devices and a controlling algorithm. The device is either a leaf device or a logical device. A leaf *device* is parameterized with a *controlling algorithm* and a *normalizer*. A logical device is, in addition to the *controlling algorithm* and the *normalizer*, parameterized with one or more sub devices.

ControllingAlgorithm

In the device archetype, information about relations and configuration is stored. Computation is done in a separate archetype, which is used to parameterize Device components. The ControllingAlgorithm performs calculations for setting the values of sub output devices based on the values it gets from input sub devices and the control it receives from the encapsulating device. When the device is a leaf node the calculation is normally void.

Normaliser

To deal with different units of measurement a normalization archetype is used. The normalizer is used to parameterize the device components and is invoked when normalizing from and to the units used by up-hierarchy devices and the controlling algorithm of the device.

AlarmDetectorDevice

Is a specialization of the Device archetype. Components of the AlarmDetectorDevice archetype is responsible for monitoring the sub devices and make sure the value read from the sensors are within the alarm threshold value set to the AlarmDetectorDevice. When threshold limits are crossed an AlarmHandler component is invoked.

AlarmHandler

The AlarmHandler is the archetype responsible for responding to alarms by returning the haemo dialysis machine to a safe-state or by addressing the cause of the alarm. Components are used to parameterize the AlarmDetectorDevice components.

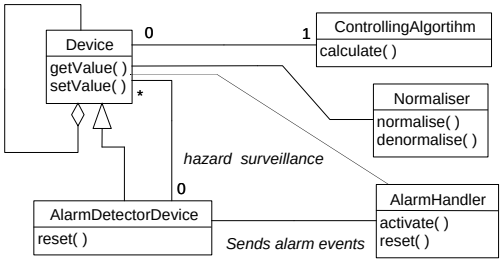

Figure 4. *The relations of the archetypes*

4.2 Scheduling Archetypes

Haemo dialysis machines are required to operate in real time. However, haemo dialysis is a slow process that makes the deadline requirements on the system less tough to adhere to. A treatment typically takes a few hours and during that time the system is normally stable. The tough requirements in response time appear in hazard situations where the system is supposed to detect and eliminate any hazard swiftly. The actual timings are presented in medical equipment standards with special demands for haemo dialysis machines [8]. Since the timing requirements are not that tight we designed the concurrency using the *Periodic Object pattern* [20]. It has been used successfully in earlier embedded software projects.

Scheduler

The scheduler archetype is responsible for scheduling and invoking the periodic objects. Only one scheduler element in the application may exist and it handles all periodic objects of the architecture. The scheduler accepts registrations from periodic objects and then distributes the execution between all the registered periodic objects. This kind of scheduling is not pre-emptive and requires usage of non-blocking I/O.

Periodic object

A periodic object is responsible for implementing its task using non-blocking I/O and using only the established time quanta. The *tick()* method will run to its completion and invoke the necessary methods to complete its task. Several periodic objects may exist in the application architecture and the periodic object is responsible for registering itself with the scheduler (figure 5).

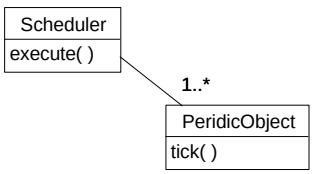

Figure 5. *Basic Concurrency with Periodic Objects*

4.3 Connector Archetypes

The communication between the architecture elements is done by using *causal connections* [18]. The principle is similar to the *Observer pattern* [11] and the *Publisher-Subscriber pattern* [7]. An observer observers a target but the difference is that a master, i.e. the entity registering and controlling the dependency, establishes the connection. Two different ways of communication exist, the push connection and the pull connection. In the first case, the target is responsible for the notifying the observer by sending the notify message. In the second case it is the observer that request data from the target. The usage of the connection allows for dynamic reconfiguration of the connection, i.e. push or pull. (figure 6)

Target

holds data other entities are dependent on. The target is responsible for notifying the link when its state changes.

Observer

depends on the data or change of data in the target. Is either updated by a change or by own request.

Link

Maintains the dependencies between the target and its observers. Also holds the information about the type of connection, i.e. push or pull. It would be possible to extend the connection model with periodic updates.

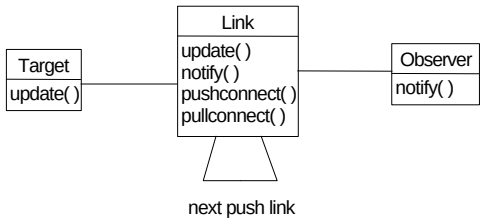

Figure 6. *Push/Pull Update Connection*

4.4 Application Architecture

The archetypes represent the building blocks that we may use to model the application architecture of a haemo dialysis machine. In figure 7 the application architecture is presented. The archetypes allow for the application architecture to be specified in a hierarchical way, with the alarm devices being orthogonal to the control systems device hierarchy.

This also allows for a layered view of the system. For example, to specify a treatment we only have to interface the closest layer of devices to the HaemoDialysisMachine device (figure 7). There would be no need to understand or interfacing the lowest layer. The specification of a treatment would look something like this in source code:

```
conductivity.set(0.2); // in milliMol
temperature.set(37.5); // in Celsius
weightloss.set(2000); // in milliLitre
dialysisFluidFlow.set(200);//in milliLitre/minute
overHeatAlarm.set(37.5,5); // ideal value in
// Celsius and maximum deviation in percent
wait(180); // in minutes
```

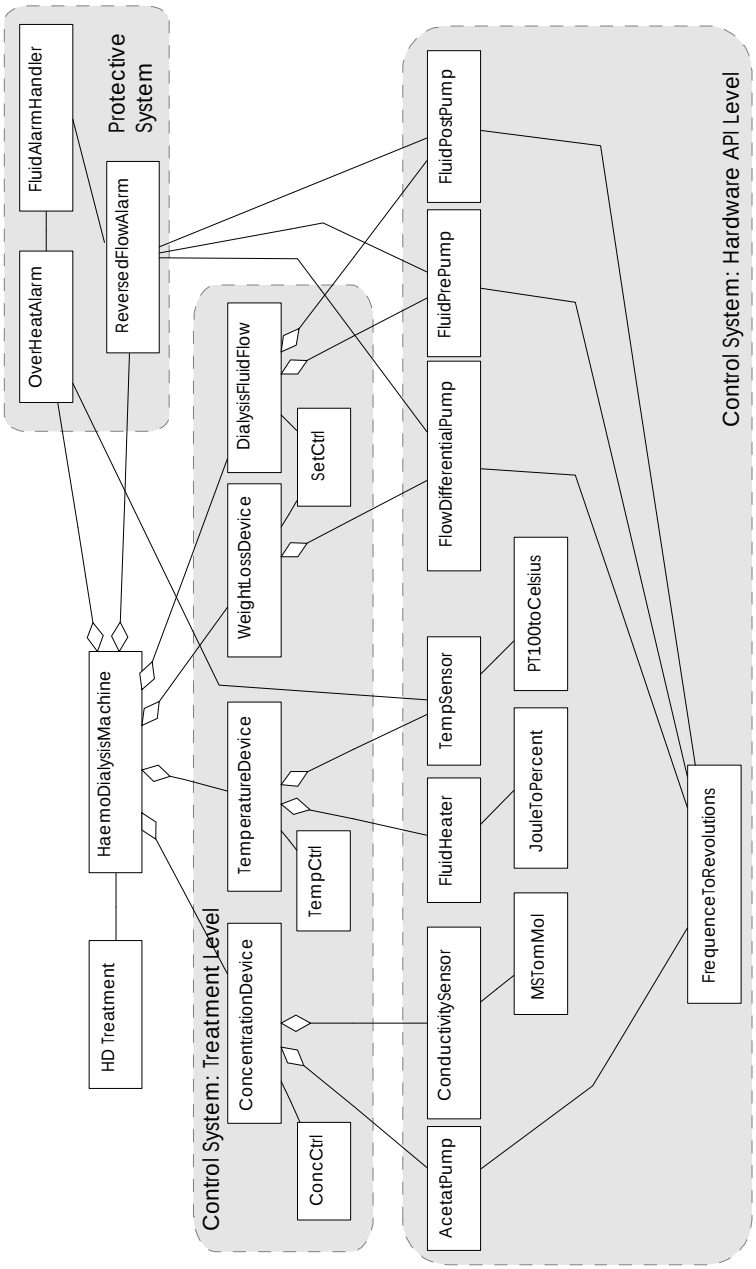

Figure 7. *Example haemo dialysis Application Architecture*

4.5 Dynamic View

The Control System

The application architecture will be executed in pseudo parallel, using the periodic object pattern. In figure 8, the message sequence of the execution of one *tick()* on a device is presented. First, the Device collects the data, normalizes it using the normalizer parameter and then calculates the new set values using the control algorithm parameter.

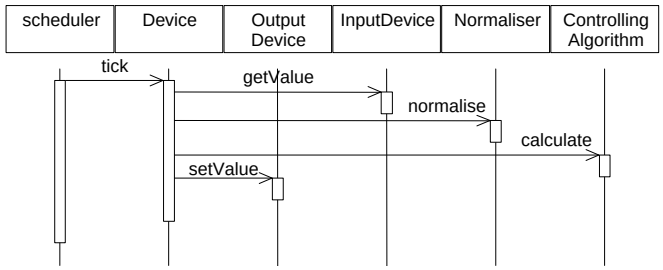

Figure 8. *The message sequence of a control tick()*

Alarm Monitoring

The control system may utilize AlarmDevices to detect problem situations and the protective system will consist of a more complex configuration of different types of AlarmDevices. These will also be run periodically and in pseudo parallel. The message sequence of one *tick()* for alarm monitoring is shown in figure 9.

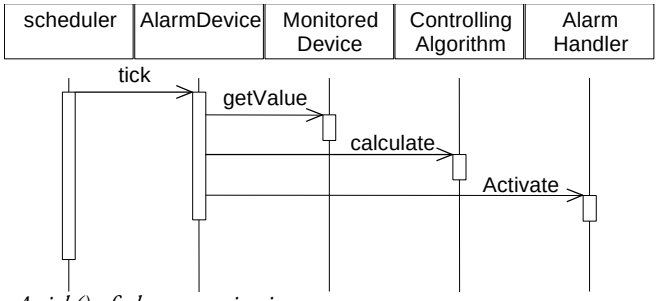

Figure 9. *A tick() of alarm monitoring*

Treatment Process

The treatment process is central to the haemo dialysis machine and its software. The general process of a treatment consists of the following steps; (1) preparation, (2) self test, (3) priming, (4) connect patient blood flow, (5) set treatment parameters, (6) start dialysis, (7) treatment done (8) indication, (9) nurse feeds back blood to patient, (10) disconnect patient from machine, (11) treatment records saved, (12) disinfecting, (13) power down.

The process generally takes several hours and the major part of the time the treatment process are involved in a monitoring and controlling cycle. The more detailed specification of this sub process is here.

1. Start fluid control
2. Start blood control
3. Start measuring treatment parameters, e.g. duration, weight loss, trans-membrane pressure, etc.
4. Start protective system
5. Control and monitor blood and fluid systems until time-out or accumulated weight loss reached desired values
6. Treatment shutdown

4.6 Rationale

During the design of the haemo dialysis architecture we had to make a number of design decisions. In this section the major design decisions and their rationale are presented.

The starting point

From the start we had a few documents describing the domain and the typical domain requirements. In the documents, the subsystems of the old system were described. Also, we had earlier experiences from designing architectures for embedded systems, i.e. Fire Alarm Systems [20] and Measurement Systems [3]. We started out using the main archetypes from these experiences which were *sensors* and *actuators*.

Initially, we wanted to address the demonstrability issues by ensuring that an architecture was easy to comprehend, consequently improving maintainability. The intention was to design an architecture that facilitated visualization and control of the functions implemented in the final application. Especially important is demonstration of the patient safety during treatments.

Our goal was to make the specification and implementation of the treatments very concise and to as high extent as possible look like the specification of a treatment that a domain expert would give, i.e. using the same concepts, units, and style.

The result was that our initially chosen abstraction, *sensor* and *actuators* did not suit our purpose adequately. The reason is that the abstraction gives no possibility of shielding the hardware and low-level specifics from the higher-level treatment specifications.

The iterations

The architecture design was iterated and evaluated some three times more, each addressing the requirements of the previous design and incorporating more of the full requirement specification.

In the first iteration, we used the Facade design pattern [11] to remedy the problem of hiding details from the treatment specifications. Spurred by the wonderful pattern we introduced several facades in the architecture. The result was unnecessary complexity and did not give the simple specification of a treatment that we desired.

In the second iteration, we reduced the number of facades and adjusted the abstraction, into a *device hierarchy*. This allowed us to use sub-devices that were communicating with the hardware and dealt with the low-level problems such as normalization and hardware APIs. These low-level devices were connected as logical inputs and outputs to other logical devices. The logical devices handle logical entities, e.g. a heater device and a thermometer device are connected to the logical device Temperature (figure 7). This allows for specification of treatments using the vocabulary of the logical devices, adapted from the low level hardware parameters to the domain vocabulary.

In the third major iteration, the architecture was improved for flexibility and reuse by introducing parameterization for normalization and control algorithms. Also the alarm detection device was introduced for detecting anomalies and hazards situations.

Concurrency

The control system involves constantly executing control loops that evaluate the current state of the process and calculates new set values to keep the process at its optimal parameters. This is supposed to be done simultaneously, i.e. in parallel. However, the system is in its basic version only equipped with a signal processor reducing parallelism to pseudo parallel. On a single processor system we have the options of (1) choose to use a third party real-time kernel supporting multi-threads and real-time scheduling. And (2) we can design and implement the system to be semi-concurrent using the periodic objects approach [20] and make sure that the alarm functions are given the due priority for achieving swift detection and elimination of hazards. Finally (3) we may choose the optimistic approach, i.e., design a sequentially executing system and make it fast enough to achieve the tightest response time requirements.

The first one is undesirable because of two reasons, i.e. resource consumption and price. The resources, i.e. memory and processor capacity, consumed by such a real-time kernel are substantial especially since we most likely will have to sacrifice resources, e.g. processor capacity and memory, for service we will not use. In addition, the price for a certified real-time kernel is high and the production and maintenance departments become dependent on third-party software.

The third option is perhaps the most straightforward option and could probably be completed. The profound problem is that it becomes un-deterministic. This is affecting the demonstrability negatively. Because of the software certification, it is unrealistic to believe that such an implementation would be allowed in a dialysis machines.

The second option, pose limitations in the implementation and design of the system, i.e. all objects must implement their methods using non-blocking I/O. However, it still is the most promising solution. Periodic objects visualize the parallel behavior clearly, using the scheduler and its list of periodic objects especially since it has been used successfully in other systems.

Communication

The traditional component communication semantics are that a sender sends a message to a known receiver. However, this simple message send

may represent many different relations between components. In the design of the dialysis system architecture, we ran into a problem related to message passing in a number of places. The problem was that, in the situation where two components had some relation, it was not clear which of the two components would call the other component. For example, one can use a *pushing* approach, i.e. the data generating component pushing it to the interested parties, or a *pulling* approach, where the interested components inquire at the data generating component, and each approach requires a considerable different implementation in the involved components.

As a solution, the notion of *causal connections* [18] was introduced that factors out the responsibility of abstracting the calling direction between the two components such that neither of the components needs to be concerned with this.

The advantage of using a causal connection is that the involved components can be focused on their domain functionality and need not concern about how to inform or inquire the outside world, thus improving the reusability of the components. In addition, it allows one to replace the type of causal connection at run-time, which allows a system to adapt itself to a changing context during operation.

4.7 Evaluation

In this section an analysis of the architecture design is presented with respect to the quality requirements. As stated in section 3.2, the traditional assessment methods are inadequate for the architecture level and therefore our evaluation was strongest on maintainability (prototype) and more subjective for the other quality requirements.

Maintainability

To evaluate the maintainability and feasibility of the architecture the industrial partner EC-Gruppen developed a prototype of the fluid-system. The prototype included controlling fluid pumps and the conductivity sensors. In total the source code volume for the prototype was 5,5 kLOC.

The maintainability was validated by an extension of the prototype. Changing the pump process control algorithms, a typically common maintenance task. The change required about seven (7) lines of code to

change in two (2) classes. And the prototype was operational again after less than two days work from one person. Although this is not scientifically valid evidence, it indicates that the architecture easily incorporates planned changes.

Reusability

The reusability of components and applications developed using this architecture has not been measured, for obvious reasons. But our preliminary assessment shows that the sub quality factors of reusability [19], i.e. generality, modularity, software system independence, machine independence and self-descriptiveness, all are reasonably accounted for in this architecture. First, the architecture supports generality. The device archetype allow for separation between devices and most of the application architecture will be made of devices of different forms. Second, the modularity is high. The archetypes allows for clear and distinguishable separation of features into their own device entity. Third, the architecture has no excessive dependencies to any other software system, e.g. multi processing kernel. Fourth, the hardware dependencies have been separated into their own device entities and can easily by substituted for other brands or models. Finally, the archetypes provide comprehensible abstraction for modeling a haemo dialysis machine. Locating, understanding and modifying existing behavior is, due to the architecture an easy and comprehendible task.

Safety

The alarm devices ensure the safety of the patient in a structured and comprehensible way. Every hazard condition is monitored has its own AlarmDetectorDevice. This makes it easier to demonstrate what safety precautions have been implemented from the standard.

Real-timeliness

This requirement was not explicitly evaluated during the project. Instead our assumption was that the data processing performance would equal that of a Pentium processor. Given that the prototype would work on a PC running NT, it would be able to run fast enough with a less resource consuming operating system in the haemo dialysis machine.

Demonstrability

Our goal when concerned with the demonstrability was to achieve a design that made the specification of a treatment and its complex sub-processes very similar to how domain experts would express the treatment their own vocabulary. The source code example in section 4 for specifying a treatment in the application is very intuitive compared to specifying the parameters of the complex sub processes of the treatments. Hence, we consider that design goal achieved.

5. Conclusions

In this paper, the architectural design of a haemo dialysis system and the lessons learned from the process leading to the architecture have been presented. The main experiences from the project are the following. First, quality requirements are often specified without any context and this complicates the evaluation of the architecture for these attributes and the balancing of quality attributes. Second, assessment techniques developed by the various research communities studying a single quality attribute, e.g. performance or reusability, are generally intended for later phases in development and require sometimes excessive effort and data not available during architecture design. Third, the archetypes use as the foundation of a software architecture cannot be deduced from the application domain through domain analysis. Instead, the archetypes represent chunks of domain functionality optimized for the driving quality requirements. Fourth, during the design process we learned that design is inherently iterative, that group design meetings are far more effective than individual architects and that documenting design decisions is very important in order to capture the design rationale. Fifth, architecture designs have an associated aesthetics that, at least, is perceived inter-subjectively and an intuitively appealing design proved to be an excellent indicator, as well as the lack of appeal. Sixth, it proved to be hard to decide when one was done with the architectural design due to the natural tendency of software engineers to perfect solutions and to the required effort of architecture assessment. Finally, it is very hard to document all relevant aspects of a software architecture. The architecture design presented in the previous section provides some background to our experiences.

Acknowledgements

We would like to thank our partners in the research project, Althin Medical AB, Ronneby, and Elektronik Gruppen AB, Helsingborg, especially, Lars-Olof Sandberg, Anders Kambrin, and Mogens Lundholm, and our collegues, Michael Mattsson and Peter Molin, who participated in the design of the architecture.

References

[1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, A. Moormann Zaremski, *Recommend Best Industrial Practice for Software Architecture Evaluation,* CMU/SEI-96-TR-025, 1997.

[2] C. Argyris, R. Putnam, D. Smith, *Action Science: Concepts, methods, and skills for research and intervention*, Jossey-Bass, San Fransisco, 1985

[3] P. Bengtsson, J. Bosch, "Scenario-based Software Architecture Reengineering", *Proceedings of the 5th International Conference on Software Reuse* (ICSR5), IEEE, 2-5 june, 1998

[4] B. W. Boehm, “A Spiral Model of Software Development and Enhancement”, *IEEE Computer*, 61-72, May 1988

[5] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/ Cummings Publishing Company, 1994.

[6] J. Bosch, ‘Design of an Object-Oriented Measurement System Framework’, *Object-Oriented Application Frameworks*, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, (coming)

[7] F. Buschmann, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996.

[8] CEI/IEC 601-2, Safety requirements std.for dialysis machines

[9] P. C. Clements, *A Survey of Architecture Description Languages*, Eight International Workshop on Software Specification and Design, Germany, March 1996

[10] N.E. Fenton, S.L. Pfleeger, *Software Metrics - A Rigorous & Practical Approach* (2nd edition), International Thomson Computer Press, 1996.

[11] Gamma et. al., *Design Patterns Elements of Reusable Design*, Addison.Wesley, 1995.

[12] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12-1990.

[13] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-oriented software engineering. A use case approach*, Addison-Wesley, 1992.

[14] R. Kazman, L. Bass, G. Abowd, M. Webb, 'SAAM: A Method for Analyzing the Properties of Software Architectures,' *Proceedings of the 16th International Conference on Software Engineering*, pp. 81-90, 1994.

[15] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, *The Architecture Tradeoff Analysis Method*, Proceedings of ICECCS, (Monterey, CA), August 1998

[16] P.B. Kruchten, 'The 4+1 View Model of Architecture,' *IEEE Software*, pp. 42-50, November 1995.

[17] D. C. Luckham, et. al., Specification and Analysis of System Architecture Using Rapide, *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, 21(4):336-355, April 1995

[18] C. Lundberg, J. Bosch, "Modelling Causal Connections Between Objects", Journal of Programming Languages, 1997.

[19] J.A. McCall, Quality Factors, *Software Engineering Encyclopedia*, Vol 2, J.J. Marciniak ed., Wiley, 1994, pp. 958 - 971

[20] P. Molin, L. Ohlsson, 'Points & Deviations - A pattern language for fire alarm systems,' to be published in *Pattern Languages of Program Design 3*, Addison-Wesley.

[21] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.

[22] S. Shlaer, S.J. Mellor, 'Recursive Design of an Application-Independent Architecture', *IEEE Software,* pp. 61-72, Jan/Feb 1997.

[23] W. Tracz, 'DSSA (Domain-Specific Software Architecture) Pedagogical Example,' *ACM Software Engineering Notes*, Vol. 20, No. 3, pp. 49-62, July 1995.

[24] Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

PAPER II

Scenario-based Software Architecture Reengineering

II

PerOlof Bengtsson & Jan Bosch

Published in the proceedings of ICSR'5, International Conference on Software Reuse, Victoria, Canada, June 1998.

Abstract

This paper presents a method for reengineering software architectures. The method explicitly addresses the quality attributes of the software architecture. Assessment of quality attributes is performed primarily using scenarios. Design transformations are done to improve quality attributes that do not satisfy the requirements. Assessment and design transformation can be performed for several iterations until all requirements are met. To illustrate the method, we use the reengineering of a prototypical measurement system into a domain-specific software architecture as an example.

1. Introduction

Reengineering of a software system is generally initiated by major changes in the requirements the system should fulfil. These changes are often concerned with the software qualities rather than the functional requirements. For example, due to architecture erosion [20], the maintainability of the software system may have deteriorated. To improve this, the system is reengineered.

To the best of our knowledge, few architecture reengineering methods have been defined. Traditional system design methods tend to focus

on the functionality that is to be provided by the system. They spend much less effort on the software quality requirements that are to be fulfilled by the system. Object-oriented methods, e.g., [2,12,24], address reusability but, generally, no assessment of the achieved result is done. Other research communities, e.g., real-time [16] and fault-tolerant [28], have proposed design methods that incorporate design steps for supporting their respective software. However, these approaches generally focus on a single software quality.

We have learned from our projects with industry [4,5,19] that a system never is a pure real-time system, or a fault-tolerant system, or a reusable system. Instead, systems should provide all these properties and perhaps more. However, as identified in [3,18], software quality requirements often conflict, e.g., real-time versus reusability, flexibility versus efficiency, reliability versus flexibility, etc. The difficulty of system design originates from the need for carefully balancing of the various software qualities. For the discussion in this paper, we distinguish between *development* related software qualities, e.g., reusability and maintainability, and *operational* related software qualities, e.g., reliability and performance.

The software architecture determines, to a considerable extent, the ability of a system to fulfil its software quality requirements. Once the application architecture is finalised, the boundaries for most quality attributes have been set. On the other hand, architectural design and reengineering are the steps in software development that are least understood and supported by traditional means.

The contribution of this paper, we believe, is that we present a practical method for reengineering software architectures and illustrated it using a real-world example.

The remainder of this paper is organised as follows. In the next section, the example system that will be reengineered is presented. We discuss our architecture reengineering method in section 3. Section 4 provides an introduction to the measurement systems domain, the software quality requirements that should be fulfilled and the scenarios used to assess the requirements. In section 5, we illustrate the use of the method by applying it to the example application. We discusses related work in section 6 and conclude the paper in section 7.

2. Example: Beer Can Inspection

To illustrate the architecture reengineering method, we use a beer can inspection system as a basis for the discussion in the remainder of the paper. The inspection system is located at the beginning of a beer can filling process and its goal is to remove dirty beer cans from the input stream. Clean cans should just pass the system without any further action.

The system consists of a triggering sensor, a camera and an actuator that can remove cans from the conveyer belt. When the hardware trigger detects a can, it send an trigger event to the software system. After a predefined amount of time, the camera takes a number of image samples of the can. Subsequently, the measured values, i.e., images, are compared to the ideal images and a decision about removing or not removing the can is made. If the can should be removed, the system invokes the actuator at a predefined time point relative to the trigger event. Figure 1 presents the process graphically.

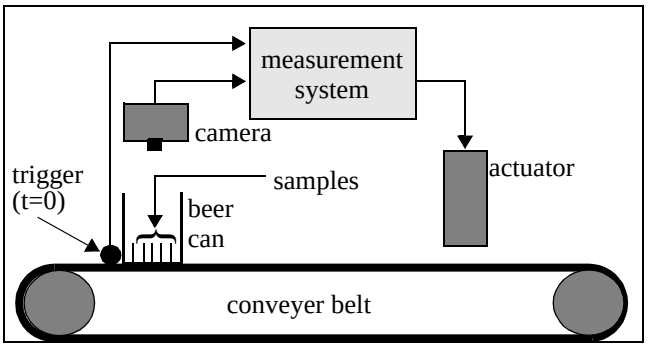

Figure 1. *Example beer cans measurement system*

Figure 2 presents the application architecture of the beer can system. The architecture was designed based on the functional requirements of the system without any explicit design effort with respect to software quality requirements such as reusability and performance.

We use the architecture of the presented system as the basis for creating a domain-specific software architecture (DSSA) [11,29] that allows the software engineer to instantiate applications in the domain of measurement systems.

3. Architecture Reengineering Method

According to our experience, software engineers generally handle software quality requirements by a rather informal process during architecture design and reengineering. Once the system is implemented, it is tested to determine whether the software quality requirements have been met. If not, parts of the system are redesigned. We consider this approach unsatisfactory since the iteration over system development is generally very costly and, secondly, the redesign cannot be planned and budgeted.

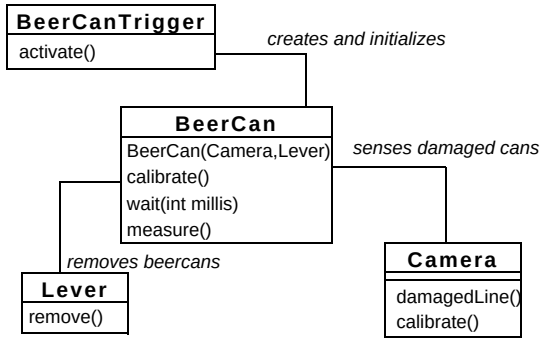

Figure 2. *Object model of the beer can application*

Conventional design methods, e.g., [2,12,24], tend to focus on achieving the required system functionality, rather than software qualities. The various software quality-based research communities identified this as unsatisfactory, and have proposed their own design methods for developing real-time [16], high-performance [28] and reusable systems [13], respectively. However, all these methods focus on a single quality attribute and treat all others as having secondary importance, if at all. We consider these approaches unsatisfactory since the software engineer needs to balance the various quality attributes for any realistic system. However, lacking a supporting method, the software engineer designs and reengineers system architectures in an ad-hoc and intuition-based manner, with the associated disadvantages.

To address this, we have defined an architecture reengineering method that provides a more objective and still practical approach. In the remainder of this section, we present an overview of the method and refer to [6] for a more extensive overview of the method.

3.1 Overview

The input for the architecture reengineering method consists of the updated requirements specification and the existing software architecture. As output, an improved architectural design is generated. In figure 3, the steps in the method are presented graphically.

1. *Incorporate new functional requirements in the architecture*. Although software engineers generally will not design a system less reliable or reusable, the software qualities are not explicitly addressed at this stage. The result from this is a first version of the application architecture design.

2. *Software quality assessment*. Each quality attribute (QA) is estimated, using primarily scenario-based analysis for assessment technique. If all estimations are as good or better than required, the architectural design process is finished. Otherwise, the next step is entered.

3. *Architecture transformation*. During this stage, QA-optimizing transformations are used to improve the architecture. Each set of transformations (one or more) results in a new version of the architectural design.

4. *Software quality assessment*. The design is again evaluated and the process is repeated from 3 until all software quality requirements are met or until the software engineer decides that no feasible solution exists.

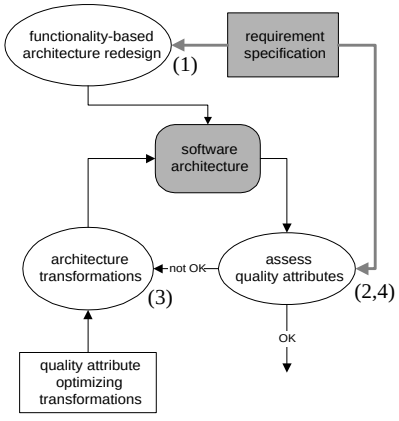

Figure 3. *Outline of the method*

The architecture reengineering method has evolved through its application, in three projects, i.e., for fire-alarm systems [19], measurement systems [4] and dialysis systems. In the following sections, the individual stages of the method are described in more detail.

3.2 Functionality-based architecture redesign

Based on the updated requirement specification, a redesign of the top-level decomposition of the system into its main components is performed. The main issue during this phase is to find and evaluate the core abstractions in the system. Although these abstractions are modelled as objects, our experience, see e.g., [19], is that these objects are not found immediately in the application domain. Instead, they are the result of a creative process that, after analysing the various domain entities, abstracts the most relevant properties and models them as architecture entities. Once the abstractions are identified, the interactions between them are defined in more detail.

The process of identifying the entities that make up the architecture is different from, for instance, traditional object-oriented design methods [2,12,24]. Those methods start by modelling the entities present in the domain and organise these in inheritance hierarchies, i.e., a bottom-up approach. Our experience is that during architectural design and reengineering it is not feasible to start bottom-up since that would require dealing with the details of the system. Instead its better to use a top-down approach.

3.3 Assessing software quality requirements

One of the core features of the architecture reengineering method is that the software qualities of a system or application architecture are explicitly evaluated. For assessing the architecture of the existing system, the system itself can be used. After the first transformation, however, no concrete system is available for evaluation. It is *not* possible to measure the quality attributes for the final system based on the architecture design. That would imply that the detailed design and implementation are a strict projection of the architecture. Instead, the goal is to evaluate the *potential* of the designed architecture to reach the software quality requirements. For example, some architectural styles, e.g., layered archi-

tectures, are less suitable for systems where performance is a major issue, although the flexibility of this style is high.

Four different approaches for assessing quality attributes have been identified, i.e., scenarios, simulation, mathematical modelling and experience based reasoning. For each quality attribute, the engineer can select the most suitable approach for evaluation. In the subsequent sections, each approach is described in more detail.

Scenario-based evaluation

The assessment of a software quality using scenarios is done in these steps:

1. *Define a representative set of scenarios.* A set of scenarios is developed that concretises the actual meaning of the attribute. For instance, the maintainability quality attribute may be specified by scenarios that capture typical changes in requirements, underlying hardware, etc.

2. *Analyse the architecture*. Each individual scenario defines a context for the architecture. The performance of the architecture in that context for this quality attribute is assessed by analysis. Posing typical question[18] for the quality attributes can be helpful (section 3.3).

3. *Summarise the results*. The results from each analysis of the architecture and scenario are then summarised into an overall results, e.g., the number of accepted scenarios versus the number not accepted.

The usage of scenarios is motivated by the consensus it brings to the understanding of what a particular software quality really means. Scenarios are a good way of synthesising individual interpretations of a software quality into a common view. This view is both more concrete than the general software quality definition[10], and it is also incorporating the uniqueness of the system to be developed, i.e., it is more context sensitive.

In our experience, scenario-based assessment is particularly useful for development related software qualities. Software qualities such as maintainability can be expressed very naturally through change scenarios. In

[14] the use of scenarios for evaluating architectures is also identified. The software architecture analysis method (SAAM) however, uses only scenarios and only evaluates the architecture in cooperation with stakeholders prior to detailed design.

Simulation

Simulation of the architecture[30] using an implementation of the application architecture provides a second approach for estimating quality attributes. The main components of the architecture are implemented and other components are simulated resulting in an executable system. The context, in which the system is supposed to execute, could also be simulated at a suitable abstraction level. This implementation can then be used for simulating application behaviour under various circumstances.

Simulation complements the scenario-based approach in that simulation is particularly useful for evaluating operational software qualities, such as performance or fault-tolerance.

Mathematical modelling

Various research communities, e.g., high-performance computing [28], reliability [25], real-time systems [16], etc., have developed mathematical models, or metrics, to evaluate especially operation related software qualities. Different from the other approaches, the mathematical models allow for static evaluation of architectural design models.

Mathematical modelling is an alternative to simulation since both approaches are primarily suitable for assessing operational software qualities.

Experience-based reasoning

A fourth approach to assessing software qualities is through reasoning based on experience and logical reasoning based on that experience. Experienced software engineers often have valuable insights that may prove extremely helpful in avoiding bad design decisions and finding issues that need further evaluation. Although these experiences generally are based on anecdotal evidence, most can often be justified by a logical line of reasoning. This approach is different from the other approaches.

First, in that the evaluation process is less explicit and more based on subjective factors as intuition and experience. Secondly, this technique makes use of the tacit knowledge of the involved persons.

3.4 Architecture transformation

Once the architecture properties have been assessed, the estimated values are compared to the requirements specification. If one or more of the software qualities are not met, the architecture has to be changed to achieve those. In the architectural reengineering method discussed in this paper, changes to the architecture are performed as *architecture transformations*. Each transformation leads to a new version of the architecture that has the same functionality, but different values for its quality attributes. Five categories of architecture transformations have been identified. In the sections below, each category is discussed in more detail.

Impose architectural style

Shaw and Garlan [26] and Buschmann et al. [7] present several architectural styles that improve certain quality attributes for the system the style is imposed upon and impair other software qualities. For example, the layered architectural style, increase the flexibility of the system by defining several levels of abstraction, but generally decrease the performance. With each architectural style, a fitness for each system property is associated. The most appropriate style for a system depends primarily on its software quality requirements. Transforming architecture by imposing an architectural style results in a major reorganisation of the architecture.

Impose architectural pattern

A second category of transformations is the use of *architectural patterns* [7]. These are different from architectural styles in that they are not predominant in the architecture. They are also different from design patterns since they affect the larger part of the architecture. Architectural patterns generally impose a *rule* [23] on the architecture that specifies how the system will deal with one aspect of its functionality, e.g., concurrency or persistence.

Apply design pattern

A less dramatic transformation is the application of a design pattern [9,7] on a part of the architecture. For instance, an Abstract Factory pattern might be introduced to abstract the instantiation process for its clients. The Abstract Factory increases maintainability, flexibility and extensibility of the system since it encapsulates the actual component types(s) that are instantiated. Nevertheless, it decreases the efficiency of creating new instances due to the additional computation, thereby reducing performance and predictability. Different from imposing an architectural style or pattern, causing the complete architecture to be reorganised, the application of a design pattern generally affects only a limited number of components in the architecture. In addition, a component can be involved in multiple design patterns without creating inconsistencies.

Convert quality requirements to functionality

Another type of transformation is the conversion of a software quality requirement into a functional solution. This solution consequently extends the architecture with functionality not related to the problem domain but is used to fulfil a software quality requirement. Exception handling is a well-known example that adds functionality to a component to increase the fault-tolerance of the component.

Distribute requirements

The final type of transformation deals with software quality requirements using the *divide-and-conquer* principle: a software quality requirement at the system level is distributed to the subsystems or components that make up the system. Thus, a software quality requirement X is distributed over the n components that make up the system by assigning a software quality requirement x_i to each component c_i such that $X=x_1+ \ldots +x_n$. A second approach to distribute requirements is by dividing the software quality requirement into two or more software quality requirements. For example, in a distributed system, fault-tolerance can be divided into fault-tolerant computation and fault-tolerant communication.

4. Measurement Systems

The domain of measurement systems denotes a class of systems used to measure the relevant values of a process or product. These systems are different from the, better known, process control systems in that the measured values are not directly, i.e., as part of the same system, used to control the production process that creates the measured product or process. Industry uses measurement systems for quality control on produced products, e.g., to separate acceptable from unacceptable products or to categorise the products in quality grades.

The goal of the reengineering project was to define a DSSA that provides a reusable and flexible basis for instantiating measurement systems. Although the software architecture of the beer can inspection system introduced in section 2 is a rather prototypical instance of a measurement system, we used it as a starting point. This application architecture, obviously, does not fulfil the software quality requirements of a DSSA. Consequently, it needs to be re-engineered and transformed to match the requirements.

The challenge of reengineering projects is to decide when one has achieved the point where the reengineered architecture fulfils its requirements. The functional requirements generally can be evaluated relatively easy by tracing the requirements in the design. Software quality requirements such as reusability and robustness, on the other hand, are much harder to assess. In the next section, we describe our approach to evaluating some of the software quality requirements put on the measurement system DSSA.

4.1 Software quality requirements

The DSSA for measurement systems should fulfil a number of requirements. The most relevant software quality requirements in the context of this paper are reusability and maintainability. Measurement systems also have to fulfil real-time and robustness requirements, but we leave these out of the discussion in this paper.

As was described in section 3, development related software qualities are generally easiest assessed using scenarios. The assessment process consists of defining a set of scenarios for each software quality, manually executing the scenarios for the architecture and subsequently interpret-

ing the result. The assessment can be performed in a complete or a statistical manner. In the first approach, a set of scenarios is defined that combined cover the concrete instances of the software quality. For instance, for reusability, all relevant ways of reusing the architecture or parts of it are represented by a scenario. If all scenarios are executed without problems, the reusability of the architecture is optimal. The second approach is to define a set of scenarios that is a representative sample without covering all possible cases. The ratio between scenarios that the architecture can handle and scenarios not handled well by the architecture provides an indication of how well the architecture fulfils the software quality requirements. Both approaches, obviously, have their disadvantages. A disadvantage of the first approach is that it is generally impossible to define a complete set of scenarios. The definition of a *representative* set of scenarios is the weak point in the second approach since it is unclear how does one decide that a scenario set is representative.

Despite these disadvantages, scenarios are a useful technique for evaluating development related software qualities. As Poulin [21] concluded for reusability, no predominant approach to assessing the quality attribute exists. Although scenario-based evaluation depends on the objectivity and creativity of the software engineers that define and execute them. We have not experienced this to be a major problem in our projects. In the next two sections, the scenarios used for evaluating reusability and maintainability are defined.

Reusability

The reusability quality attribute provides a balance between the two properties; generality and specifics. First, the architecture and its components should be general because they should be applied in other similar situations. For example, a weight sensor component can be sufficiently generic to be used in several applications. Secondly, the architecture should provide concrete functionality that provides considerable benefit when it is reused.

To evaluate the existing application architecture we use scenarios. However, reusability is a difficult software property to assess. We evaluate by analysing the architecture with respect to each scenario and assess the ratio of components reused *as-is* and total number of components.

Note that the scenarios are presented as *vignettes* [14] for reasons of space.

R1 Product packaging quality control. For example, sugar packages that are both measured with respected to intact packaging and weight.

R2 Surface finish quality control where multiple algorithms may be used to derive a quality figure to form a basis for decisions.

R3 Quality testing of microprocessors where each processor is either rejected or given a serial number and test data logged in a quality history database in another system.

R4 Product sorting and labelling, e.g., parts are sorted after tolerance levels and labelled in several tolerance categories and are sorted in different storage bins.

R5 Intelligent quality assurance system, e.g., printing quality assurance. The system detects problems with printed results and rejects occasional misprints, but several misprints in a sequence might cause rejection and raising an alarm.

All presented scenarios require behaviour not present in the initial software architecture. However, the scenarios are realistic and measurement systems exist that require functionality defined by the scenarios.

Maintainability

In software-intensive systems, maintainability is generally considered important. A measurement system is an embedded software system and its function is very dependent on its context and environment. Changes to that environment often inflict changes to the software system. The goal for maintainability in this context is that the most likely changes in requirements are incorporated in the software system against minimal effort.

In addition, maintainability of the DSSA is assessed using scenarios. For the discussion in this paper, the following scenarios are applied on the DSSA. Again, the scenarios are presented as *vignettes* for reasons of space.

M1 The types of input or output devices used in the system is excluded from the suppliers assortment and need to be changed. The correspond-

ing software needs to be updated.

M2 Advances in technology allows a more accurate or faster calculation to be used. The software needs to be modified to implement new calculation algorithms.

M3 The method for calibration is modified, e.g., from user activation to automated intervals.

M4 The external systems interface for data exchange change. The interfacing system is updated and requires change.

M5 The hardware platform is updated, with new processor and I/O interface.

These are the scenarios we have found to be representative for the maintenance of *existing* measurement systems. Of course, other changes may possibly be required, but are less likely to appear.

5. Applying the Method

In this section, we transform the application architecture using an iterative process in order to satisfy the software quality requirements put on the DSSA. First, we evaluate the software qualities of the application architecture. Subsequently, we identify the most prominent deficiency and transform the architecture to remove the deficiency. we repeat this process until all quality attributes have satisfactory levels for the defined software quality requirements.

For reasons of space, we do not present a complete analysis of the resulting architecture after every transformation, but summarise the results in table 1.

5.1 Application evaluation

We analyse the application architecture by asking a typical question for reusability, i.e., *How much will I be able to reuse of the software*[1] in the context of each reuse scenario. The results from this analysis and the transformations are shown in table 1. Every scenario is assigned a quote number of affected components in the scenario divided by the total

1. Modified from the original, 'Will I be able to reuse some of the software?' in [18].

number of components in the current architecture. For reusability this should be as close to one as possible, i.e., as many of the components as possible should be reusable as-is. For maintainability, this should be as low as possible, i.e., as few components as possible should need to be modified. These are the results:

> Analysing the initial application using R1, we find that we can probably reuse the camera and the lever components. These are not necessarily dependent on the rest of the components to be useful. We get similar results from R2.
>
> In R3, we find that the sensing device will have to be more complex and include sophisticated algorithms to work. The same goes for the actuating device that now also needs to be fed the data to imprint on the product. Therefore, it is most likely that none of the components can be reused. Similar for R4 we find increasingly complex actuation schemes. Hence, we cannot expect any reuse in this scenario either.
>
> In R5 we find that the actuation device possibly could be reused. Because of using previous results, i.e., the actuation history, we need more sophisticated measurement items.

Then we follow the same procedure with the maintenance scenario. The question replaced, by *How easy is it to fix*[1] in the context of each maintenance scenario. These are the results:

> In M1, we see that changing hardware interface for the lever, the trigger or the camera is possible to do without modification to any other components. It is concise enough. For one change only one component has to be modified.
>
> It is worse for the result of M2. Modifying the calculation algorithm in the initial architecture is impossible for us to exclude changes to any of the components. Possibly, all components will have to be updated. Similar is the situation for M3. We cannot exclude any component that requires no modification.
>
> In the case of M4, we may if the detailed design was done well enough get by with modification to only one component. Again, we cannot exclude any of the other components.

1. Modified from the original, 'Can I fix it?' in [18].

The last maintenance scenario, M5, will most likely require modification to all the components. No component can be excluded as not using any hardware specific services from the platform.

Based on the results from the analysis we make the following conclusions about the application architecture.

- Reusability is not to good in the initial architecture. The classes are tightly coupled and the reuse scenarios show limited possibilities to reuse as-is.
- Maintainability could also be better. One of the scenarios (M1) is satisfied. The other scenarios however, are not supported satisfactory. The main problem seems to be that a change is not very concise and not at all simple.

The results of the analysis indicate that the reusability and maintainability attributes of the architecture are not satisfying. In the remainder of this section, we present transformations for improving these attributes.

5.2 Transformations

Component level transformations

Problem. Although the beer cans application is a small set of classes, the task of changing or introducing a new type of items require the source code of most components to be changed. In all the specified reuse scenarios, the use of new types of measurement item is involved.

Alternatives. From the set of transformations categories described in section 5.2 first look for an architectural style that might fit our needs. An alternative could be to organise the system into a pipe&filter architecture with sensing, data interpreting and actuation filters. A second alternative, and perhaps more natural for object oriented designers, is to capture the real-world correspondents of these classes and define relevant abstractions, i.e., to define them as components with an interface and behaviour. Support for this can be found in the scenarios.

Transformation. The pipes & filters alternative requires changes to more than one filter for several scenarios, i.e., R2-R5 and M3-M5. Instead, we choose to 'componentify' the classes into the generic component definitions. The DSSA should be able to deal with the different kinds of items in a uniform way, and therefore the abstraction MeasurmentItem is introduced.

Subsequently, component types Actuator, for the RemoveLever, and Sensor, for the LineCamera, are defined. Different types of actuators can be defined as components of the type Actuator. The Trigger is defined as a specialised sensor component, with interface additions. The redesigned architecture is presented in figure 4. In addition, the affected components are marked with (1) in figure 5.

Problem resolved. The components introduced reduced the coupling between the components by use of the principle programming towards an interface and not an implementation. General software engineering experience tell us that this principle of programming give us benefits both in situations of reuse and maintenance.

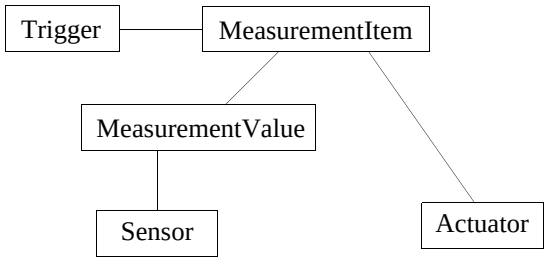

Figure 4. *Model after first transformation*

Separation of component creation from trigger

Problem. The introduction of the component types facilitates the introduction of several concrete components but does not remove the problem of type dependence at the time of component creation. Every time a client creates a component it needs to know the actual sensor, actuator and measurement item types. From a maintainability perspective, this is not at all concise or simple to modify.

Alternatives. One can identify two alternatives to address this problem. The first is to introduce pre-processor macros in the source code to easily change all instances of component creation of the affected types. The second alternative is to use the Abstract Factory [9] to centralise the information about concrete types.

Transformation. There are a number of drawbacks with the macro alternative; for example, it is a static solution that when changed, it must be recompiled. The ItemFactory could have an interface for changing the instantiation scheme. Therefore the Abstract Factory pattern is selected and a factory component is introduced to handle the instantiation of actuators, sensors and measurement items. See (2) in figure 5.

Problem resolved. The trigger need no longer to know the actual type of measurement item to be created, but instead requests a new measurement item from the ItemFactory. The use of the Abstract Factory pattern did eliminate that problem.

Changing to strategies

Problem. Different components in the measurement system perform similar tasks. Changes to these tasks require updating the source code of every component containing the same implementation for the task. For example, the measurement of a measurement item aspect could be performed by different methods. A change to the similar parts is not concise enough in the current design and inhibits the maintainability. The measurement item may pull data from sensor, the sensor may push data to measurement item or the sensors may pass on data whenever they are changed. These alternatives are common to all sensors, independent of their types, and a way of reusing this aspect of the sensor is desired.

Alternatives. The Strategy pattern [9] allows more distinction between the method for getting input data and the method of deriving the actual value for the actuation decision.

Transformation. This increases the conciseness and the modularity of the architecture and should improve the maintainability. Since the reuser selects one out of many strategies will be somewhat reduced since we will have reuse smaller number of components as is. However, the

benefits for maintainability outweigh the liabilities with loss in reusability. The Strategy pattern is applied to all components where we can identify a similar need

- *Sensor update strategy*. The three variations (push, pull, on change) apply to how sensors pass their data. The strategies are OnChangeUpdate, ClientUpdate, and PeriodicUpdate.
- *Calculation strategy*: Calculation of derived data and decisions show that there are similar methods to perform different tasks of different objects. Different calculation strategies can be defined. These strategies can be used by sensors, measurement items and actuators.

The newly introduced components are marked with (3) in figure 5.

Problem resolved. Maintainability improved greatly from this transformation, to the cost of some reusability. The gain was substantial enough to motivate the trade-off.

Unification of actuation and calibration

Problem. Calibration is performed by letting the system measure a manually inspected item and the values are stored as ideal. In normal service, deviations from the ideal values measured are considered faults. The calibration is supposed to be carried out at different times in runtime. Calibration of the measurement system is considered a special service not implemented using the concepts of the current architecture. This makes the system more tightly coupled, in the sense that the factory, the measurement item and the sensors need to be aware of the calibration state. This makes several components dependent of a common

state and changes to modify calibration becomes complex. This is not good for maintainability, or for reusability.

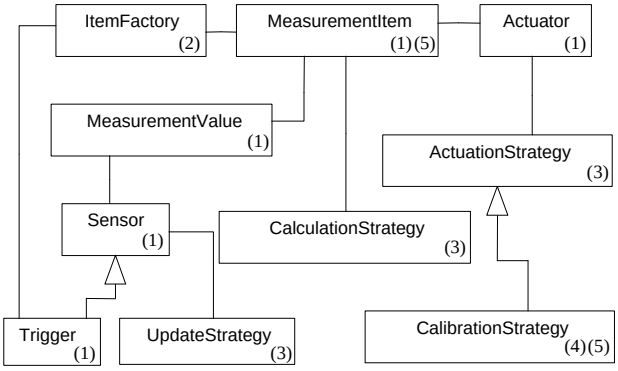

Figure 5. *The Measurement Systems DSSA*[1]

Alternatives. Two alternatives to decreasing the coupling due to dependency of the common state exist. First, the calibration is defined as a separate service that all sensors and measurement items must implement. Each component checks if its current state is calibration and then call its calibration services instead of the normal services. The second alternative is to use the strategy design pattern and identify that most of the calibration is really the same as for the normal measuring procedure. The difference is that the ideal measures have to be stored for later referencing.

Transformation. The first alternative is not desirable since this introduces behaviour that is not really the components responsibility. Instead we favour the second alternative and introduce a special case of an actuation strategy, i.e., the CalibrationStrategy. The result is that the calibration is performed with the calibration actuation strategy, and when invoked stores the ideal values where desired. See (4) in figure 5.

Problem resolved. The use of the calibration strategy as a special type of actuation removes the dependence on a global state. This reduces the need to modify several components when modifying the calibration behaviour. Consequently, we have improved the maintainability.

1. The UML notation for inheritance is used to show the equivalence in interfaces

Adding prototype to factory

Problem. The introduction of the calibration strategy addressed most of the identified problems. However, there is a problem remaining, i.e., the calibration strategy is dependent on the implementation type the measurement item component. This couples these two, so that the calibration strategy cannot be reused separate from the measurement item.

Alternatives. One alternative solution is to decrease the coupling by introducing intermediate data storage, e.g., a configuration file, where the ideal values are stored. The second alternative is to apply the Prototype design pattern [9]. The ideal measurement item is stored in the ItemFactory and used as a template for subsequent measurement item instances.

Transformation. We decide to apply a variant of the prototype pattern. The calibration of the system is performed by setting the ItemFactory into calibration mode. When a real-world item triggers the ItemFactory, it creates a new measurement item with the normal sensors associated. The measurement item collects the data by reading the sensors, but when invoking the actuation strategy, instead of invoking the actuators, the calibration strategy causes the measurement item to store itself as a prototype entity at the item factory. The affected relation is marked (5) in figure 5.

Software Quality		**Iteration no.**					
	Scenario	**0**	**1**	**2**	**3**	**4**	**5**
Reusability	R1	2/4	3/5	4/6	3/9	3/9	**4/10**
	R2	2/4	3/5	4/6	3/9	3/9	**4/10**
	R3	0/4	0/5	1/6	2/9	2/9	**3/10**
	R4	0/4	0/5	1/6	2/9	2/9	**3/10**
	R5	0/4	0/5	1/6	2/9	2/9	**3/10**

Table 1. Analysis of architecture

Software Quality		Iteration no.					
	Scenario	0	1	2	3	4	5
Maintainability	M1	1/4	1/5	1/6	2/9	3/9	**2/10**
	M2	4/4	4/5	3/6	2/9	3/9	**2/10**
	M3	4/4	5/5	6/6	9/9	2/9	**2/10**
	M4	4/4	3/5	3/6	3/9	3/9	**3/10**
	M5	4/4	5/5	6/6	9/9	9/9	**10/10**

Table 1. Analysis of architecture

Problem resolved. The Prototype pattern was applied to the ItemFactory. Consequently, the calibration strategy no longer needs to know the concrete implementation of the measurement item. The only component in the system with knowledge about concrete component types is the ItemFactory. The information has been localised in a single entity.

5.3 Final assessment

In table 1, the results from the final assessment of the domain specific software architecture for measurement systems are presented. Figure 5 contains the most relevant classes of the DSSA. As shown in the evaluation, the scenario M5 is not supported by the DSSA. However, since we estimate the likelihood for scenario M5 rather low, we accept the bad score for that scenario. The results of the second transformation are especially relevant, since it illustrates the trade-off between reusability and maintainability (see figure 6). Overall, we find the result from the

transformations satisfying and the analysis of the scenarios shows substantial improvement.

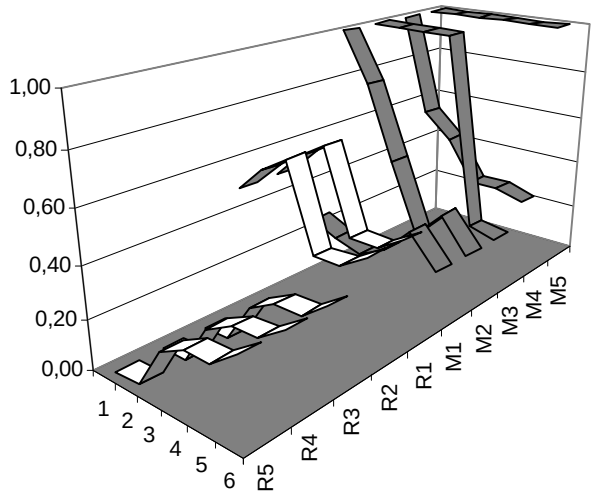

Figure 6. *The effect of each transformation*

6. Related Work

To the best of our knowledge, no architecture reengineering methods exists to date. Traditional design methods often focus on the system functionality [2,12,22,24,30] or on a single software quality, e.g., high-performance [28], real-time [16] and reusable systems [13]. Architecture design methods have been proposed by Shlaer & Mellor [27], Kruchten [15] and Bosch & Molin [6]. Different from architecture reengineering, architecture design methods start from the requirement specification only. Boehm [3] discusses conflicts between, what he calls, quality requirements, but focuses on identifying these conflicts and solving them during requirement specification rather than during architectural design or reengineering.

Architecture evaluation is discussed by Kazman *et al.* [14]. Their SAAM method also uses scenarios, but does not discuss other techniques for architecture evaluation. In addition, no relation to architecture (re)design is made. Further, the work described in [1] primarily uses scenarios for architecture evaluation.

Architecture transformation uses the notions of architectural styles [26], architectural patterns [7] and design patterns [9]. However, rather than viewing these concepts as ready designs, we treat styles and pat-

terns as active entities transforming an architecture from one version to another.

7. Conclusions

This paper presented method for reengineering software architectures that provides a practical approach to evaluating and redesigning architectures. The method uses four techniques for architecture evaluation, i.e., scenarios, simulation, mathematical modelling and experience based reasoning. To improve the architecture, five types of architecture transformations are available; to impose architectural style, to apply architectural pattern, to use design pattern, convert quality requirements to functionality and distribute quality requirements.

We illustrated the method using a concrete system from the measurement systems domain, i.e., a beer can inspection system. This system was reengineered into a domain-specific software architecture for measurement systems. The focus in this paper was on the reusability and maintainability requirements on the DSSA. Scenarios were defined for assessing each requirement. Although no predominant approach to assessing quality attributes of systems exists, this paper shows that scenarios provide a powerful means for practitioners. Architecture transformations provide an objective way to redesign since associated with each transformation, a problem is explicitly addressed, alternative transformations are investigated and the rationale for the design decisions is captured.

Future work includes the extensions of the reengineering method for more quality requirements and the application of the method in more industry case studies and projects.

Acknowledgements

The authors wish to thank Will Tracz for his constructive and detailed comments.

References

[1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, A. Moormann Zaremski, *Recommend Best Industrial Practice for Software Architecture Evaluation,* CMU/SEI-96-TR-025, January 1997.

[2] G. Booch, *Object-Oriented Analysis and Design with Applications* (2nd edition), Benjamin/Cummings Publishing Company, 1994.

[3] B. Boehm, 'Aids for Identifying Conflicts Among Quality Requirements,' *International Conference on Requirements Engineering (ICRE96),* Colorado, April 1996, and IEEE Software, March 1996.

[4] J. Bosch, 'Design of an Object-Oriented Measurement System Framework,' *submitted*, 1997.

[5] J. Bosch, P. Molin, M. Mattsson, PO Bengtsson, 'Object-oriented Frameworks: Problems and Experiences,' *submitted*, 1997.

[6] J. Bosch, P. Molin, 'Software Architecture Design: Evaluation and Transformation,' *submitted,* 1997.

[7] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996.

[8] N.E. Fenton, S.L. Pfleeger, *Software Metrics - A Rigorous & Practical Approach* (2nd edition), International Thomson Computer Press, 1996.

[9] Gamma et. al., *Design Patterns Elements of Reusable Design*, Addison.Wesley, 1995.

[10] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12-1990.

[11] R.S. D'Ippolito, Proceedings of the Workshop on Domain-Specific Software Architectures, *CMU/SEI-88-TR-30*, Software Engineering Institute, July 1990.

[12] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-oriented software engineering. A use case approach*, Addison-Wesley, 1992.

[13] E. Karlsson ed., *'Software Reuse A Holistic Approach'*, Wiley, 1995.

[14] R. Kazman, L. Bass, G. Abowd, M. Webb, 'SAAM: A Method for Analyzing the Properties of Software Architectures,' *Proceedings of the 16th International Conference on Software Engineering*, pp. 81-90, 1994.

[15] P.B. Krutchen, 'The 4+1 View Model of Architecture,' *IEEE Software*, pp. 42-50, November 1995.

[16] J.W.S. Liu, R. Ha, 'Efficient Methods of Validating Timing Constraints,' in *Advanced in Real-Time Systems*, S.H. Son (ed.), Prentice Hall, pp. 199-223, 1995.

[17] D. C. Luckham, et. al., Specification and Analysis of System Architecture Using Rapide, *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, 21(4):336-355, April 1995

[18] J.A. McCall, Quality Factors, *Software Engineering Encyclopedia*, Vol 2, J.J. Marciniak ed., Wiley, 1994, pp. 958 - 971

[19] P. Molin, L. Ohlsson, 'Points & Deviations - A pattern language for fire alarm systems,' to be published in *Pattern Languages of Program Design 3*, Addison-Wesley.

[20] D.E. Perry, A.L.Wolf, 'Foundations for the Study of Software Architecture,' *Software Engineering Notes*, Vol. 17, No. 4, pp. 40-52, October 1992.

[21] J.S. Poulin, 'Measuring Software Reusability', Proceedings of the *Third Conference on Software Reuse*, Rio de Janeiro, Brazil, November 1994.

[22] *The RAISE Development Method*, The RAISE Method Group, Prentice Hall, 1995.

[23] D.J. Richardson, A.L. Wolf, 'Software Testing at the Architectural Level,' *Proceedings of the Second International Software Architecture Workshop,* pp. 68-71, San Francisco, USA, October 1996.

[24] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.

[25] P. Runeson, C. Wohlin, 'Statistical Usage Testing for Software Reliability Control', *Informatica,* Vol. 19, No. 2, pp. 195-207, 1995.

[26] M. Shaw, D. Garlan, *Software Architecture - Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[27] S. Shlaer, S.J. Mellor, 'Recursive Design of an Application-Independentt Architecture', *IEEE Software,* pp. 61-72, January/February 1997.

[28] C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[29] W. Tracz, 'DSSA (Domain-Specific Software Architecture) Pedagogical Example,' *ACM Software Engineering Notes*, Vol. 20, No. 3, pp. 49-62, July 1995.

[30] Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

PAPER III

Architecture Level Prediction of Software Maintenance

PerOlof Bengtsson & Jan Bosch

Published in the proceedings of CSMR'3, 3rd European Conference on Software Maintenance and Reengineering, Amsterdam, March 1999.

III

Abstract

A method for the prediction of software maintainability during software architecture design is presented. The method takes (1) the requirement specification, (2) the design of the architecture (3) expertise from software engineers and, possibly, (4) historical data as input and generates a prediction of the average effort for a maintenance task. Scenarios are used by the method to concretize the maintainability requirements and to analyze the architecture for the prediction of the maintainability. The method is formulated based on extensive experience in software architecture design and detailed design and exemplified using the design of software architecture for a haemo dialysis machine. Experiments for evaluation and validation of the method are ongoing and future work.

1. Introduction

One of the major issues in software development today is the software quality. Rather than designing and implementing the correct functionality in products, the main challenge is to satisfy the software quality requirements, e.g. performance, reliability, maintainability and flexibility. The notion of software architecture has emerged during the recent years as the appropriate level to deal with software qualities. This

because, it has been recognized [1,2] that the software architecture sets the boundaries for the software qualities of the resulting system.

Traditional object-oriented software design methods, e.g. [5,14,21] focus primarily on the software functionality and give no support for software quality attribute-oriented design, with the exception of reusability and flexibility. Other research communities focus on a single quality attribute, e.g. performance, fault-tolerance or real-time. However, real-world software systems are never just a real-time system or a fault-tolerant system, but generally require a balance of different software qualities. For instance, a real-time system that is impossible to maintain or a high-performance computing system with no reliability is of little use.

To address these issues, our ongoing research efforts aim on developing a method for designing software architectures, i.e. the ARCS method [6]. In short, the method starts with an initial architecture where little or no attention has been given to the required software qualities. This architecture is evaluated using available techniques and the result is compared to the requirements. Unless the requirements are met, the architect transforms the architecture in order to improve the software quality that was not met. Then the architecture is again evaluated and this process is repeated until all the software quality requirements have been met or until it is clear that no economically or technically feasible solution exists.

The evaluation of software architectures plays a central role in architectural design. However, software architecture evaluation is not well understood and few methods and techniques exist. Notable exceptions are the SAAM method discussed in [15] and the approach described in [10]. In this paper, we propose a method for predicting maintainability of a software system based on its architecture. The method defines a *maintenance profile*, i.e. a set of change scenarios representing perfective and adaptive maintenance tasks. Using the maintenance profile, the architecture is evaluated using so-called scenario scripting and the expected maintenance effort for each change scenario is evaluated. Based on this data, the required maintenance effort for a software system can be estimated. The method is based on our experience in architectural design and its empirical validation is part of ongoing and future work. The remainder of this paper is organized as follows. In the next section, the maintenance prediction method is presented in more detail. The architecture used as an example is discussed in section 3 and the

application of the method in section 4. Related work is discussed in section 5 and the paper is concluded in section 6.

2. Maintenance Prediction Method

The maintenance prediction method presented in this paper estimate the required maintenance effort for a software system during architectural design. The estimated effort can be used to compare two architecture alternatives or to balance maintainability against other quality attributes.

The method has a number of inputs: (1) the requirement specification, (2) the design of the architecture (3) expertise from software engineers and, possibly, (4) historical maintenance data. The main output of the method is, obviously, an estimation of the required maintenance effort of the system built based on the software architecture. The *maintenance profile* is a second output from the method. The profile contains a set of scenario categories and a set of scenarios for each category with associated weighting and analysis (scripting) results. The maintenance prediction method consists of six steps:

1. Identify categories of maintenance tasks
2. Synthesize scenarios
3. Assign each scenario a weight
4. Estimate the size of all elements.
5. Script the scenarios
6. Calculate the predicted maintenance effort.

The steps are discussed in more detail in the following sections.

2.1 Identify categories of maintenance tasks

Software maintainability is defined by IEEE [13] as:

> *The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.*

This definition renders three categories of maintenance, i.e. corrective, perfective and adaptive. The prediction method focuses only on perfective and adaptive maintenance and does not predict efforts required for corrective maintenance. Nevertheless, the remaining categories are too abstract to be relevant in this process step. Instead, the categories are defined based on the application or domain description. For example, a haemo dialysis machine might have maintenance scenarios concerned with treatment maintenance, hardware changes, safety regulation changes, etc. These categories reflect the meaning of the maintainability requirement in the context of this application or domain and give the designer a better understanding of the requirements posed on the architecture.

2.2 Synthesize scenarios

For each of the maintenance categories, a representative set of concrete scenarios is defined. The software architect, or the domain expert, is responsible for selecting the scenarios such that the set is representative for the maintenance category. The number of scenarios in the set is dependent on the application and the domain, but in our experience we define about ten scenarios for each category. Scenarios should define very concrete situations. Scenarios that specify types of maintenance cases or sub categories is to be avoided. For example, a scenario could be, “Due to changed safety regulations, a temperature alarm must be added to the hydraulic module in the dialysis machine”. Another example is, “Due to a new type of pump, the pump interface must be changed from duty cycle into a digital interface, with a set value in kP (kilo Pascal)”.

It is important to note that our use of the term ‘scenarios’ is different from Object-Oriented design methods where the term generally refers to use-case scenarios, i.e. scenarios describing system behavior. Instead, our scenarios describes an action, or sequence of actions that might occur related to the system. Hence, a change scenario, describes a certain maintenance task. In addition, due to reasons of space, in this paper we present scenarios as *vignettes* [15] rather than in their full size representation.

2.3 Assign each scenario a weight

Change scenarios have different likelihood of actually occurring during the lifetime of the system. In order to generate an accurate measure for maintainability, the prediction method requires probability estimates, i.e. weights, for each scenario. These probabilities are used for balancing the impact on the prediction of more occurring and less occurring maintenance tasks. We define the weight measure as the relative probability of this scenario resulting in a maintenance task during a particular time interval, e.g. a year, or between two releases. Consequently, scenarios that describe often-recurring maintenance tasks will get higher probabilities and therefore impact the predicted value more and the architecture will generally be optimized for incorporating those maintenance tasks with minimal effort.

The weight of scenarios is produced in two ways. If no historical maintenance data is available from similar applications or earlier releases, the software architect, or the domain expert, estimates the scenario weights. If empirical data about maintenance of the product exists in the organization, the probability data of earlier maintenance efforts should be used as basis for weighting. Based on the probability data for the individual scenarios, it is possible to calculate a probability figure for each category as well. The exact calculation of probabilities is illustrated in section 4.

2.4 Estimate the size of all elements

To estimate the maintenance effort, the size of the architecture needs to be known and the sizes of the affected components need to be known. The component size influences the effort required to implement a change in the component. At least three techniques can be used for estimating the size of components. First, the size of every component can be estimated using the estimation technique of choice. Secondly, an adaptation of an Object-Oriented metric (SIZE2 [8]) metric may be used (SIZE2') [2]. Finally, when historical data from similar applications or earlier releases is available, existing size data can be used and extrapolated to new components.

2.5 Script the scenarios

Based on the selected scenarios from each maintenance category, the weights defined for each scenario and the categories and the size data for the components, we estimate the maintainability of the architecture by *scripting* [16] the scenarios. For each scenario, the impact of the realization of that scenario in the architecture and its components is evaluated. Thus, find what components are affected and to what extent will they be changed.

For example, implementing the earlier described scenario of adding a temperature alarm in the dialysis machine would require changes to the hydraulic module component and addition of three new components of type device and controlling algorithm. In addition, the components for system definition and the protective system need to be changed.

2.6 Calculate the predicted maintenance effort

The prediction value is a weighted average for the effort (expressed as size of modification) for each maintenance scenario. Based on that, one can calculate an average effort per maintenance task. To predict the required maintenance effort for a period of time, an estimation or calculation of the number of maintenance tasks has to be done. That figure is then multiplied with the average effort per maintenance task. Note that the above is only necessary when predicting maintenance effort for a period of time. When comparing two alternative architectures, it is sufficient to compare the weighted average effort per maintenance task.

$$M_{tot} = \sum_{n=1}^{k_s} \left(P(S_n) \cdot \sum_{m=1}^{k_c} V(S_n, C_m) \right)$$

$P(S_n)$ the probability weight of scenario n
$V(S_n,C_m)$ the affected volume of component m in scenario n
k_s = number of scenarios
k_c = number of components in architecture

Figure 1. *Assessment Calculation Equation*

3. Example Application Architecture

Haemo dialysis systems present an area in the domain of medical equipment where competition has been increasing drastically during recent years. The aim of a dialysis system is to remove water and certain natural waste products from the patient's blood. Patients that have, generally serious, kidney problems and consequently produce little or no urine use this type of system. The dialysis system replaces this natural process with an artificial one.

An overview of a dialysis system is presented in figure 2. The system is physically separated into two parts by the dialysis membrane. On the left side the dialysis fluid circuit takes the water from a supply of a certain purity (not necessarily sterile), dialysis concentrate is added using a pump. A sensor monitors the concentration of the dialysis fluid and the measured value is used to control the pump. A second pump maintains the flow of dialysis fluid, whereas a third pump increases the flow and thus reduces the pressure at the dialysis fluid side. This is needed to pull the waste products from the patient's blood through the membrane into the dialysis fluid. A constant flow of dialysis fluid is maintained by the hydro mechanic devices denoted in the figure with rectangles with curls.

On the right side of figure 2, the extra corporal circuit, i.e. the blood-part, has a pump for maintaining a specified blood flow on its side of the membrane. The patient is connected to this part through two needles usually located in the arm that take blood to and from the patient. The extra corporal circuit uses a number of sensors, e.g. for identifying air bubbles, and actuators, e.g. a heparin pump to avoid cluttering of

the patients blood while it is outside the body. However, these details are omitted since they are not needed for the discussion in this paper.

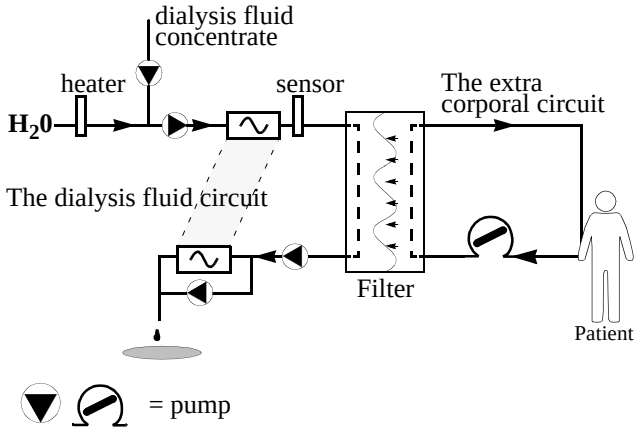

Figure 2. *Schematic of Haemo Dialysis Machine*

The dialysis process, or *treatment*, is by no means a standard process. A fair collection of treatments exists including, for example, Haemo Dialysis Filtration (HDF), Ultra Filtration (UF) and other variations, such as single needle/single pump, double needle/single pump. Treatments are changed due to new research results but also since the effectiveness of a particular treatment decreases when it is used too long for a patient. Although the abstract function of a dialysis system is constant, a considerable set of variations exists already. Based on experience, the involved company anticipates several additional changes to the software, hardware and mechanical parts of the system that will be necessary in response to developments in medical research.

3.1 Requirements

The aim during architectural design is to optimize the potential of the architecture (and the system built based on it) to fulfil the software quality requirements. For dialysis systems, the driving software quality requirements are *maintainability*, *reusability*, *safety*, *real-timeliness* and *demonstrability*. Below, we elaborate on the maintainability requirement.

Maintainability. Past haemo dialysis machines produced by our partner company have proven to be hard to maintain. Each release of software with bug corrections and function extensions have made the software

harder and harder to comprehend and maintain. One of the major requirements for the software architecture for the new dialysis system family is that maintainability should be considerably better than the existing systems, with respect to *corrective* but especially *adaptive* maintenance:

- *Corrective maintenance* has been hard in the existing systems since dependencies between different parts of the software have been hard to identify and visualize.
- *Adaptive maintenance* is initiated by a constant stream of new and changing requirements. Examples include new mechanical components as pumps, heaters and AD/DA converters, but also new treatments, control algorithms and safety regulations. All these new requirements need to be introduced in the system as easily as possible. Changes to the mechanics or hardware of the system almost always require changes to the software as well. In the existing system, all these extensions have deteriorated the structure, and consequently the maintainability, of the software and subsequent changes are harder to implement. Adaptive maintainability was perhaps the most important requirement on the system.

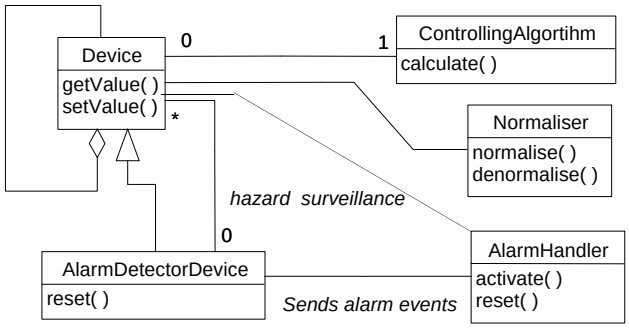

Figure 3. *The relations of the logical archetypes*

3.2 Logic Archetypes

One of our main concerns when we designed the software architecture for the haemo dialysis machine was maintainability. The logical archetypes are based on device hierarchy (figure 3). The archetypes are central to the design and important for understanding the haemo dialysis appli-

cation architecture when doing the scripting, i.e. change impact analysis.

Device

The system is modeled as a device hierarchy, starting with the entities close to the hardware as leaves, ending with the complete system as the root. For every device, there are zero or more sub-devices and a controlling algorithm. The device is either a leaf device or a logical device.

ControllingAlgorithm

In the device archetype, information about relations and configuration is stored. Computation is done in a separate archetype, the ControllingAlgorithm, which is used to parameterize Device components.

Normaliser

To convert from and to different units of measurement the normalization archetype is used.

AlarmDetectorDevice

Is a specialization of the Device archetype. Components of the AlarmDetectorDevice archetype are responsible for monitoring the sub devices. When threshold limits are crossed an AlarmHandler component is invoked.

AlarmHandler

The AlarmHandler is the archetype responsible for responding to alarms by returning the haemo dialysis machine to a safe-state or by addressing the cause of the alarm.

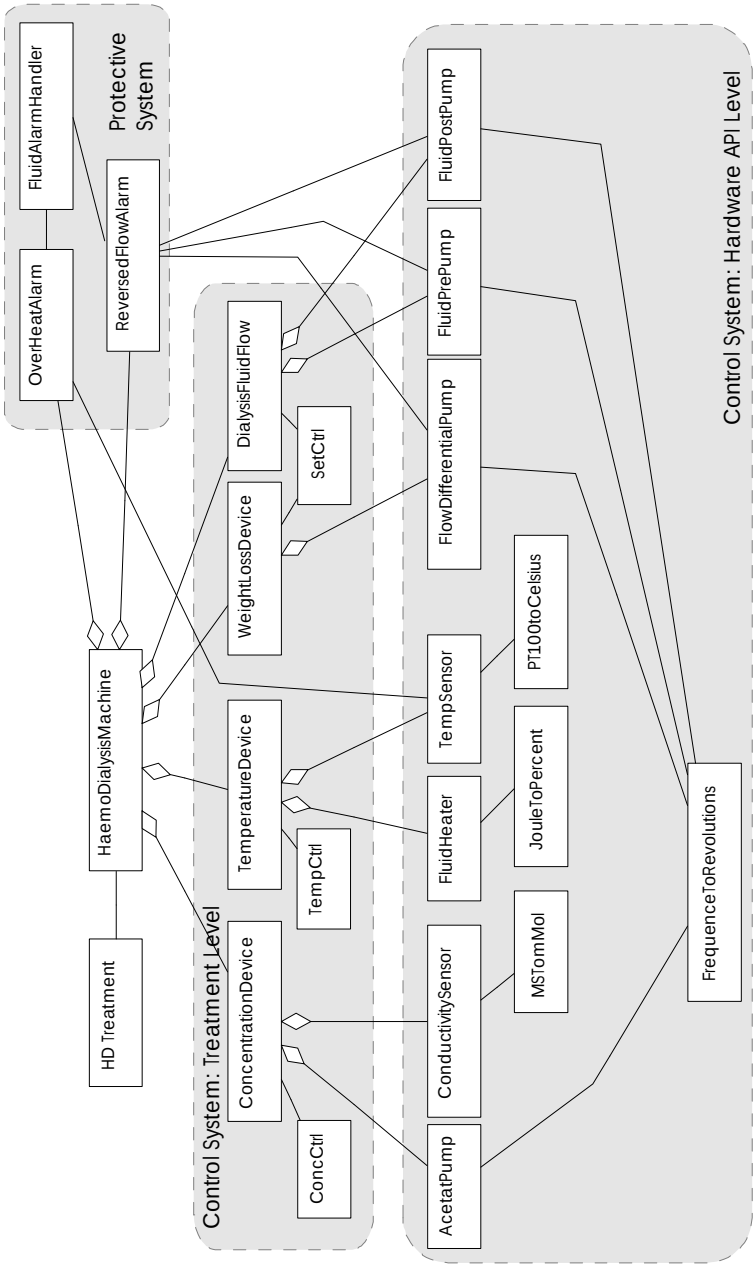

Figure 4. *Example Haemo Dialysis Application Architecture*

3.3 Scheduling Archetypes

Haemo dialysis machines are required to operate in real time. However, haemo dialysis is a slow process that makes the deadline requirements on the system less tough to adhere to. A treatment typically takes a few hours and during that time the system is normally stable. Since the timing requirements are not that tight we designed the concurrency using the *Periodic Object pattern* [19]. It has been used successfully in earlier embedded software projects.

Scheduler

The scheduler archetype is responsible for scheduling and invoking the periodic objects. Only one scheduler element in the application may exist and it handles all periodic objects of the architecture. The scheduler accepts registrations from periodic objects and then distributes the execution between all the registered periodic objects.

Periodic object

A periodic object is responsible for implementing its task using non-blocking I/O and using only the established time quanta. The tick() method will run to its completion and invoke the necessary methods to complete its task.

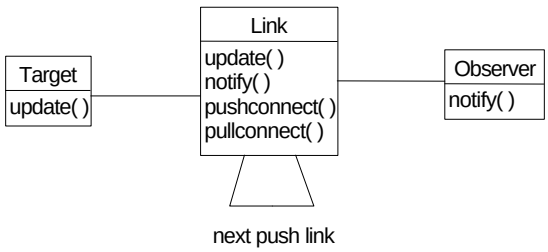

Figure 5. *Push/Pull Update Connection*

3.4 Connector Archetypes

Causal connections [18] implements the communication between the architecture elements. The principle is similar to the *Observer pattern* [11] and the *Publisher-Subscriber pattern* [7]. The usage of the connec-

tion allows for dynamic reconfiguration of the connection, i.e. push or pull. (Figure 5)

Target

Maintains information that other entities may be dependent on. The target is responsible for notifying the link when its state changes.

Observer

Depends on the data or change of data in the target. Is either updated by a change or by own request.

Link

Maintains the dependencies between the target and its observers. Also holds the information about the type of connection, i.e. push or pull. It would be possible to extend the connection model with periodic updates.

3.5 Application Architecture

The archetypes represent the building blocks that we may use to model the application architecture of a haemo dialysis machine. In figure 4, the application architecture is presented. The archetypes allow for the application architecture to be specified in a hierarchical way, with the alarm devices being orthogonal to the control systems device hierarchy. The description serves as input for scenario scripting, which is architecture level impact analysis in the maintainability case.

This also allows for a layered view of the system, not meaning that the architecture is layered. For example, to specify a treatment we only have to interface the closest layer of devices to the HaemoDialysisMachine device (figure 4). There would be no need to understand or interfacing the lowest layer. The specification of a treatment would look something like this in source code:

```
conductivity.set(0.2); // in milliMol
temperature.set(37.5); // in Celsius
weightloss.set(2000); // in milliLitre
```

```
dialysisFluidFlow.set(200);//in milliLitre/minute
overHeatAlarm.set(37.5,5); // ideal value in
// Celsius and maximum deviation in percent
wait(180); // in minutes
```

4. Prediction Example

In this section, we will present an example prediction for the architecture presented in section 3. It is presented to illustrate the practical usage of the method, rather than to give a perfect prediction of this particular case.

4.1 Scenario Categories

In the example domain we identify the following categories of maintenance tasks;

1. Hardware changes, i.e. additions and replacements of hardware require changes to software.
2. Algorithm changes, i.e. algorithms become obsolete and is replaced by new improved ones.
3. Safety changes, i.e. safety standards changes and sets new requirements on the system.
4. Medical advances requires changes, i.e. new treatments and parameters are introduced.
5. Communication and I/O change.
6. Market driven changes. Different markets or countries require certain functionality.

We use these broad categories of maintenance task in the next step of the method to ensure that we include all the important aspects in the broad sense.

4.2 Change Scenarios

When we have the categories, we list a number of scenarios for each category that describe concrete maintenance tasks that may occur during the next maintenance phase.

Scenarios describe a possible situation and change scenarios, in particular, describe possible change situations that will cause the maintenance organization to perform changes in the software and/or hardware. For reasons of space, the scenarios are presented very brief. In our real world application of the method, the scenarios are generally more verbose. This list presented in table 1 represents a maintenance profile, i.e. it profiles the relevant interpretation of software maintenance for the resulting system.

Table 1. Maintenance Profile

Category	**Scenario Description**	**Weight**
Market Driven	C1 Change measurement units from Celsius to Fahrenheit for temperature in a treatment.	0.043
Hardware	C2 Add second concentrate pump and conductivity sensor.	0.043
Safety	C3 Add alarm for reversed flow through membrane.	0.087
Hardware	C4 Replace duty-cycle controlled heater with digitally interfaced heater using percent of full effect.	0.174
Medical Advances	C5 Modify treatment from linear weight loss curve over time to inverse logarithmic.	0.217
Medical Advances	C6 Change alarm from fixed flow limits to follow treatment.	0.087
Medical Advances	C7 Add sensor and alarm for patient blood pressure	0.087
Hardware	C8 Replace blood pumps using revolutions per minute with pumps using actual flow rate (ml/s).	0.087
Com. and I/O	C9 Add function for uploading treatment data to patient's digital journal.	0.043
Algorithm Change	C10 Change controlling algorithm for concentration of dialysis fluid from PI to PID.	0.132
Sum		**1.0**

In section 2.2, a total number of ten scenarios per category were suggested. Both for reasons of space and illustrativeness, we will however only use a total of ten scenarios in this example.

4.3 Assign Weights

Each scenario has a certain likelihood of appearing during the next phase of maintenance. Each scenario is therefore assigned a value for the probability of which any arbitrary maintenance task from the maintenance phase will be like this (see table 1, column 3). The sum of all the weights must be exactly 1.

For assigning each weight we can use two approaches. First, we can make qualified guesses that some changes are more likely than others. Domain experts or software engineers can support the estimation with experiences from the earlier maintenance phases. Second, we can collect and categorize historical data from other similar development projects.

Table 2. Estimated Component Size

Component	**Size (LOC)**
HDFTreatment	200
HaemoDialysisMachine	500
ConcentrationDevice	100
TemperatureDevice	100
WeightlossDevice	150
DialysisFluidFlowDevice	150
ConcCtrl	175
TempCtrl	30
SetCtrl	30
AcetatPump	100
ConductivitySensor	100
FluidHeater	100
Sum	**2805**

Table 2. Estimated Component Size

Component	**Size (LOC)**
TempSensor	100
FlowdifferentialPump	100
FluidPrePump	100
FluidPostPump	100
mSTomMol	20
JouleToPercent	20
PT100toCelsius	40
FrequenceToRevolutions	40
OverHeatAlarm	50
ReversedFlowAlarm	300
FluidAlarmHandler	200
Sum	**2805**

4.4 Component Size Estimates

There are two ways of estimating the size of components. First, the component sizes are estimated using the estimation technique of choice. In most organizations, some estimation technique is used and could also be used for the method presented in this paper. In many cases the project planning already use and require size estimates of the system for work division. These estimates are either equivalent to those or, the estimates for the architecture are one level more fine grained.

Second, a prototype implementation or a previous release may be available which can be used as basis for the estimation. The size esti-

mates presented in table 2 are synthesized using the size data from an early prototype implementation.

Table 3. Impact Analysis per Scenario

Scenario	Dirty Components	Volume
C1	HDFTreatment (20% change) + new Normaliser type component	,2*200+ 20 = 60
C2	ConcentrationDevice (20% change) + ConcCtrl (50% change) + reuse with 10% modification of AcetatPump and ConductivitySensor	,2*100+ ,5*175+ ,1*100+ ,1*100 = 127,5
C3	HaemoDialysisMachine (10% change) + new AlarmHandler + new AlarmDevice	,1*500+ 200+100 =350
C4	Fluidheater (10% change), remove DutyCycleControl and replace with reused SetCtrl	,1*100 = 10
C5	HDFTreatment (50% change)	,5*200 = 100
C6	AlarmDetectorDevice (50% change) + HDFTreatment (20% change) + HaemoDialysisMachine (20% change)	,5*100+ ,2*200+ ,2*500 = 190
C7	see C3	= 350
C8	new ControllingAlgorithm + new Normaliser	100+20 = 120
C9	HDFTreatment (20% changes) + HaemoDialysisMachines (50% changes)	,2*200+ ,5*500 = 290
C10	Replacement with new ControllingAlgorithm	= 100

4.5 Script the Scenarios

The scenario scripting, or change impact analysis, is done by investigating the required changes to the components of the application architecture and the severity of the change in percent. To this stage we have not

investigated if any particular method for scripting is to prefer over others. An introduction to change impact analysis can be found in [4]. The result of scripting the scenarios in our example is shown in table 3.

4.6 Calculation

The prediction is calculated using the formula presented in figure 1:

0.043*60 + 0.043*127.5 + 0.087*350 + 0.174*10 + 0.217*100 + 0.087*190 + 0.087*350 + 0.087*120 + 0.043*290 + 0.132*100 = 145 LOC / Change

Given that we estimate around 20 maintenance task for the predicted period of time, either from first to second release or for the coming year. Assuming that we also have an estimated or historical data of maintenance productivity we are able to extrapolate the estimate from this method to a total maintenance effort estimate. We assume that we have a perfective maintenance productivity that are similar to the median reported in [12], i.e. 1.7 LOC/day, which amounts to about 0.2 LOC/hour. Then we get the following estimate:

20 changes per 145 LOC = 2900 LOC

2900 / 0.2 = 14 500 hours of effort

This would represent a medium project of about 6-7 persons working around 2300 hours per year.

5. Related work

Architecture assessment is important for achieving the required software quality attributes. A well-known method is the scenario-based architecture assessment method (SAAM) [15]. The SAAM method of assessing software architecture is primarily intended for assessing the final version of the software architecture and involves all stakeholders in the project. The method we propose differs in that it does not involve all stake holders, and thus requires less resources and time, but instead provides an instrument to the software architects that allows them to repeatedly evaluate architecture during design. We recognize the need for stakeholder commitment and believe that these two methods should be used in combination.

In addition, a method based on an ISO standard has been proposed in [10], which suggests a rigorous metrics approach to the problem of

software quality evaluation of software architectures. The method make a clear distinction on internal and external views, where the external view is the view important to or seen by the clients of the resulting products. The rigorous ambition makes it hard to believe that the method will be suitable for usage in every cycle in an iterative and incremental software architecture design process.

Within the software maintenance community efforts have been made to predict maintainability. A set of object oriented metrics was validated in [17] to be good predictors of the software maintenance effort for each module in a software system. However, the metrics suite used requires data that can only be collected from the source code and thus cannot be used for software architecture when no or only prototype source exist.

Software change impact analysis is an established research area within the software maintenance community [4]. A variety of models and techniques exist. However, the techniques are often based on having the software available and its source code and this prohibits their application to software architectures. To the best of our knowledge, no impact analysis method exists that is specific to software architecture.

6. Conclusions

We have presented a method for prediction of maintainability from software architecture. The method provides a number of benefits: First, it is practical and has been used during architectural design. Second, its use provides benefits for more than just the prediction, e.g. improved requirements understanding. Third, it combines the usage of design expertise and historical data for validation of scenario profiles. This way the method more efficiently incorporates the uniqueness of the changes for the predicted period of time. Fourth, the method is very slim in terms of effort and produced artifacts. Finally, it is suitable for design processes that iterate frequently with evaluation in every iteration, e.g. as in the ARCS method [3].

Weaknesses of the method include its dependency on a representative maintenance profile and the problem of validating that a profile is representative. In our future work we aim to address this in a number of ways. First, we are planning a study investigating how individual knowledge and expertise affects the representativeness of a maintenance pro-

file and thus how the activities concerned with generating maintenance profiles should be staffed. Second, we will continue to study industrial maintenance practice and intend to incorporate that knowledge can be incorporated into the method. Finally, we intend to study the sensitivity of the method for variation of the input variables, e.g. if the method is more or less sensitive to the representativeness of the maintenance scenario profile than we currently think, or if the size estimates are more significant for the results.

References

[1] L. Bass, P. Clements, R. Kazman, *'Software Architecture In Practise'*, Addison Wesley, 1998.

[2] P. Bengtsson, '*Towards Maintainability Metrics on Software Architecture: An Adaptation of Object-Oriented Metrics*', First Nordic Workshop on Software Architecture (NOSA'98), Ronneby, August 20-21, 1998.

[3] P. Bengtsson, J. Bosch, 'Scenario Based Software Architecture Reengineering', *Proceedings of International Conference of Software Reuse 5 (ICSR5)*, 1998.

[4] Bohner, S. A, Arnold, R.S., *Software Change Impact Analysis*, IEEE Computer Society Press, 1996.

[5] G. Booch, *Object-Oriented Analysis and Design with Applications*, (2nd edition), Benjamin/Cummings Publishing Company, 1994.

[6] J. Bosch, P. Molin, 'Software Architecture Design: Evaluation and Transformation', *submitted,* 1997.

[7] F. Buschmann, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996.

[8] S.R. Chidamber and C.F. Kemerer, 'Towards a metrics suite for object-oriented design,' *in proceedings: OOPSLA'91*, pp.197-211, 1991.

[9] CEI/IEC 601-2 Safety requirements standard for dialysis machines.

[10] J.C. Dueñas, W.L. de Oliveira, J.A. de la Puente, 'A Software Architecture Evaluation Method,' *Proceedings of the Second International ESPRIT ARES Workshop,* Las Palmas, LNCS 1429, Springer Verlag, pp. 148-157, February 1998.

[11] E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, *Design Patterns Elements of Reusable Design*, Addison.Wesley, 1995.

[12] Henry, J. E., Cain, J. P., "A Quantitative Comparison of Perfective and Corrective Software Maintenance", Journal of Software Maintenance: Research and Practice, John Wiley & Sons, Vol 9, pp. 281-297, 1997

[13] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12-1990.

[14] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *'Object-oriented software engineering. A use case approach'*, Addison-Wesley, 1992.

[15] R. Kazman, L. Bass, G. Abowd, M. Webb, 'SAAM: A Method for Analyzing the Properties of Software Architectures,' *Proceedings of the 16th International Conference on Software Engineering*, pp. 81-90, 1994.

[16] P.B. Krutchen, 'The 4+1 View Model of Architecture', *IEEE Software*, pp. 42-50, November 1995.

[17] W. Li, S. Henry, 'Object-Oriented Metrics that Predict Maintainability', *Journal of Systems and Software,* vol. 23, no. 2, pp. 111-122, November 1993.

[18] C. Lundberg, J. Bosch, "Modelling Causal Connections Between Objects", *Journal of Programming Languages,* 1997.

[19] P. Molin, L. Ohlsson, 'Points & Deviations - A pattern language for fire alarm systems,' to be published in *Pattern Languages of Program Design 3*, Addison-Wesley.

[20] L. H. Putnam, 'Example of and Early Sizing, Cost and Scehdule Estimate for an Application Software System', *Proceedings of COMPSAC'78*, IEEE , pp. 827-832, Nov 1978.

[21] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.

PAPER IV

An Experiment on Creating Scenario Profiles for Software Change

PerOlof Bengtsson & Jan Bosch

Research Report 99/2, ISSN 1103-1581, ISRN HK-R-RES--99/6--SE, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, 1999. (Submitted)

IV

Abstract

Scenario profiles are used increasingly often for the assessment of quality attributes during the architectural design of software systems. However, the definition of scenario profiles is subjective and no data is available on the effects of individuals on scenario profiles. In this paper we present the design, analysis and results of a controlled experiment on the effect of individuals on scenario profiles, so that others can replicate the experiments on other projects and people. Both scenario profiles created by individuals and by groups are studied. The findings from the experiment showed that groups with prepared members proved to be the best method for creating scenario profiles. Unprepared groups did not perform better than individuals when creating scenario profiles.

1. Introduction

During recent years, the importance of explicit design of the architecture of software systems is recognized [2,5,8,13]. The software architecture constrains the quality attributes and the architecture should support the quality attributes significant for the system. This is important since changing the architecture of a system after it has been developed is generally prohibitively expensive, potentially resulting in a

system that provides the correct functionality, but has unacceptable performance or is very hard to maintain.

Architecture assessment is important to decide the level at which the software architecture supports various quality attributes. The need for evaluation and assessment methods have been indicated by [1,2, 9,10, 11]. Architecture assessment is not just important to the software architect, but is relevant for all stakeholders, including the users, the customer, project management, external certification institutes, etc.

One can identify three categories of architecture assessment techniques, i.e. scenario-based, simulation and static model-based assessment. However, these techniques all make use of scenario profiles, i.e. a set of scenarios. For assessing maintainability, for example, a maintenance profile is used, containing a set of change scenarios.

Although some scenarios profiles can be defined as 'complete', i.e. covering all scenarios that can possibly occur, most scenario profiles are 'selected'. *Selected scenario profiles* contain a representative subset of the population of all possible scenarios. To use the aforementioned maintenance profile as an example, it is, for most systems, impossible to define all possible change scenarios, which requires one to define a selection that should represent the complete population of change scenarios.

Scenario profiles are generally defined by the software architect as part of architecture assessment. However, defining a selected scenario profile is subjective and we have no means to completely verfiy the representativeness of the profile. Also, to the best of our knowledge, no studies have been reported about the effects of individuals on the creation of scenario profiles, i.e. what is the deviation between profiles created by different individuals. The same is the case for groups defining scenario profiles.

Above, the general justification of the work reported in this paper is presented. A second reason for conducting this study is that in [5] we proposed a method for (re)engineering software architectures and architecture assessment is a key activity. As part of that method, we have developed a technique for scenario-based assessment of maintainability [6]. An important part of the technique is the definition of a maintenance scenario profile. Since the accuracy of the assessment technique is largely dependent on the representativeness of the scenario profile, we conducted an experiment to determine what the effect of individuals and groups is on the definition of scenario profiles. Therefore, we will

primarily use maintenance scenario profiles as examples, even through the results are valid for other quality attributes as well.

The intention of the experiment is threefold:

1. testing three different methods of synthesizing scenario profiles.
2. test the hypothesis that there is a difference between the methods.
3. find out which one of the three methods that are the best.

To conduct the experiment, we used volunteering students from the Software Engineering study program at the University of Karlskrona/ Ronneby, all currently on their Masters year.

The remainder of the paper is organized as follows. In the next section we describe the concept of scenario profiles in more detail. The design of the experiment is presented in section 3, followed by the analysis and interpretation of the results in section 4. Related work is discussed in section 5 and the paper is concluded in section 6.

2. Scenario Profiles

Scenario profiles describe the semantics of software quality factors, e.g. maintainability or safety, for a particular system. The description is done in the terms of a set of scenarios. Scenarios may be assigned an associated weight or probability of occurrence within in a certain time, but we do not address that in this paper. To describe, for example, the maintainability requirement for a system, we list a number of scenarios that each describe a possible and, preferably, likely change to the system. The set of scenarios is called a scenario profile. An example of a software change scenario profile for the software of a haemo dialysis machine is presented in figure 1.

Category	**Scenario Description**
Market Driven	**S1** Change measurement units from Celsius to Fahrenheit for temperature in a treatment.
Hardware	**S2** Add second concentrate pump and conductivity sensor.
Safety	**S3** Add alarm for reversed flow through membrane.
Hardware	**S4** Replace duty-cycle controlled heater with digitally interfaced heater using percent of full effect.

Category	Scenario Description
Medical Advances	**S5** Modify treatment from linear weight loss curve over time to inverse logarithmic.
Medical Advances	**S6** Change alarm from fixed flow limits to follow treatment.
Medical Advances	**S7** Add sensor and alarm for patient blood pressure
Hardware	**S8** Replace blood pumps using revolutions per minute with pumps using actual flow rate (ml/s).
Com. and I/O	**S9** Add function for uploading treatment data to patient's digital journal.
Algorithm Change	**S10** Change controlling algorithm for concentration of dialysis fluid from PI to PID.

Figure 1. *Maintenance Scenario Profile Example*

Scenario profiles represent a way to document and leverage from the experts knowledge about the system. It also provides its users with a way to determine where they lack knowledge about the system.

2.1 Scenario Profile Usage

A scenario profile can, basically, be defined in one of two contexts, i.e. the 'greenfield' and the experienced context. If a scenario profile is defined in an organization using the technique for the first time, for a new system and no historical data is available about similar systems, we are fully dependent on the experience, skill and creativeness of the individuals defining the profile. The resulting scenario profile is the only input to the architecture assessment. The lack of alternative data sources in this case and the lack of knowledge about the representativeness of scenario profiles defined by individuals and groups, indicates that there is a need to increase our understanding of profiles in this situation.

In the second situation, there is either an earlier release of the system or historical data of similar systems available. Since, in this case, empirical data can be collected, we can use this data as an additional input for the next prediction and thus get an more accurate result. However, even when historical data is available to be used as a reference point, it is important that the persons synthesizing the profile also incorporate the difference from similar systems or with the previous release of the system. The problem might otherwise be that, using only the historical data, one predicts the history. Consequently, important future scenarios,

that the domain experts are aware of, may be over looked. The latter, however, remains to be empirically validated and will not directly be addressed in this experiment.

In the experiment reported in this paper, we address the first situation, i.e. defining a scenario profile without historical data, since few prediction methods are available for this situation. Once the source code of a (similar) system is available, traditional assessment methods exists, e.g. Li & Henry [14].

2.2 Methods of Profile Creation

Scenario profiles can be created in at least three different ways. First, an individual could be assigned the task of independently creating a scenario profile for a software quality attribute of a system. Second, a group of people could be assigned the same task. Third, a group of people could be assigned the same task, but are required to prepare themselves individually before meeting with the group.

In the case of an individual creating a scenario profile, the advantage is, obviously, the relatively low resource cost for creating the profile. However, the disadvantage is that there is a, hard to assess, risk that the scenario profile is less representative due to the individual's lack of experience in the domain, or misconceptions about the system.

The second alternative, i.e. a group that jointly prepares a scenario profile, has as an associated disadvantage that the cost of preparing the profile is multiplied by the number of members of the group, meaning maybe three to five times more expensive. However, the risk of the forecast being influenced by individual differences is reduced since the group has to agree on a profile. Nevertheless, a risk with this method is that the resulting scenario profile is influenced by the most dominant rather than the most knowledgeable person, and thus affecting the scenario profile negatively. Finally, the productivity might be very low when in group session, since obtaining group consensus is a potentially tedious process.

The third alternative, in which the group members prepare an individual profile prior to the group meeting, has as an advantage that the individual productivity and creativity is incorporated when preparing the profiles, and then the unwanted variation of individuals are reduced by having the group agreeing on a merged scenario profile. A disadvan-

tage is the increased cost, at least when compared to the individual case, but possibly also when compared to the unprepared group alternative.

The experiment reported in this paper studies the difference in the produced results from these three methods and compares the methods.

3. The Experiment

3.1 Goal and purpose

The purpose of this experiment is to gain understanding of the characteristics of scenario profiles and the influence and sensitivity of individuals participating in the specification of the scenario profiles. The questions we are asking and would like to answer are:

- How much do profiles created by independent persons vary for a particular system?
- How does a profile, created by an independent persons, differentiate from a profile created by a group?
- What are the difference in the results from scenario profile created by a group, if the individual members have prepared their own profiles first, compared to profiles created groups with unprepared members.
- How does these variances impact the predicted values? Are they absolutely critical to the method?

In the next section these questions have been formulated as more specific hypotheses and corresponding null-hypotheses.

3.2 Hypotheses

We state the following null-hypotheses:

> **H_{01}** = *No significant difference in score between scenario profiles created by individual persons, or groups with unprepared members.*

H_{02} = *No significant difference in score between scenario profiles created by individual persons, or groups with prepared members.*

H_{03} = *No significant difference in score between scenario profiles created by groups with unprepared members, or groups with prepared members.*

In addition we state our six main hypotheses that allow us to rank the methods, even partially if the experiment does not produce significant results to support all stated hypotheses:

H_1 = *Scenario profiles created by groups with unprepared members, generally get better scores than scenario profiles created by an individual person.*

And the counter hypothesis to **H_1**, denoted **H_{10}** to more clearly show its relation to **H_1**.

H_{10} = *Scenario profiles created by individuals generally get better score than profiles created by groups with unprepared members.*

H_2 = *Scenario profiles created by groups with prepared members, generally get better scores than scenario profiles created by an individual person.*

H_{20} = *Scenario profiles created by individuals generally get better score than profiles created by groups with prepared members.*

H_3 = *Scenario profiles created by groups with prepared members generally get better scores than group profiles with unprepared members.*

H_{30} = *Scenario profiles created by groups with unprepared individuals generally get better score than profiles created by groups with prepared members.*

These hypothesis will allow us to make some conclusions about the ranking between the methods, even though the data does not allow us to dismiss all null-hypotheses or support all the main hypotheses.

3.3 Experiment Design

To test these hypotheses using an experiment, we decided to employ a blocked project design with two project requirement specifications and twelve persons divided into four groups with three persons in each group.

From the hypotheses we get three types of methods for creating change scenario profiles, i.e. treatments. Since scenario profiles need to be concrete, we decided to use the definition of change scenario profiles. However, the design of the experiment is such the results are applicable to selected scenario profiles for other quality attributes as well. The three 'treatments' that we use in the experiment are the following:

1. One independent person create a change scenario profile.
2. A group, with unprepared members, create a change scenario profiles.
3. A group, with members prepared by creating personal profiles before meeting in the group, creates the change scenario profile.

One of the problems in executing software development experiments is the number of persons required to test different treatments in robust experiment designs. In our previous experimentation experience, our main problem has been to find sufficient numbers of voluntary participants. Because of this we have taken great care to factor the block design to allow us to test all three treatments, with a minimum amount of experiment participants. To do this we identify that the scenario profile created by an individual, i.e. treatment 1, may also be regarded as a preparation for a group meeting, i.e. treatment 3. This is exploited in the design of the experiment by having the group members prepare their own scenario profile that is collected and distributed before the group meeting is held and the group creates the group scenario profile. Thus, data for treatment 1 is collected as part of treatment 3. This way we reduce the required number of subjects by half.

3.4 Analysis of Results

In order to confirm or reject any of the hypotheses, we need to rank the scenario profiles. The ranking between two scenario profiles must, at least, allow for deciding whether one scenario profile is better, equivalent, or worse than another scenario profile. The problem is that the profile is supposed to represent the future maintenance of the system and hence, the best profile is the one that is the best approximation of that. In the case where historical maintenance data is available, we can easily rank the scenario profiles by comparing each profile with the actual maintenance activities. However, for the project requirement specifications used in the experiment, no such data is available.

Instead we assume that the consensusof all scenario profiles generated during the experiment, i.e. a synthetic reference profile, can be assumed to be reasonably close to a scenario profile based on historical data. Consequently, the reference profile can be used for ranking the scenario profiles.

When conduction the experiment we will get 20 scenario profiles divided on two projects, 12 individually created, and 8 created by groups. These scenario profiles will share some scenarios and contain some scenarios that are unique. If we construct a reference profile containing all unique scenarios using the scenario profiles generated during the experiment, we are able to collect the frequency for each unique scenario. Each scenario in the reference profile would have a frequency between 1 and 10. Using the reference profile, we are able to calculate a score for each of the scenario profile generated by the experiment by summarizing the frequency of each scenario in the scenario profile. This is based on the assumption that the importance of a scenario is indicated by the number of persons who believed it to be relevant. Consequently, the most important scenario will have the highest frequency. Consequently, the most relevant scenario profile must be composed by the most relevant scenarios and thus render the highest score. By comparing each scenario profile to the reference profile, we can rank the scenario profiles and find out which one is better than the other.

To formalize the above, we define the set of all scenario profiles generated by the experiment $Q = \{P_1, ..., P_{20}\}$, where $P_i = \{s_1, ..., s_n\}$. The reference profile R is defined as $R = \{u_1, ..., u_m\}$ where u_i is a unique scenario existing in one or more scenario profiles P. The function $f(u_i)$ returns the number of occurrences of the unique scenario in Q, whereas

the function $m(s_i)$ maps a scenario from a scenario profile to a unique scenario in the reference profile. The score of a scenario profile can then be defined as follows:

$$score(P_i) = \sum_{x=1}^{n_{P_i}} f(m(s_x))$$

3.5 The Selected Projects

Two requirements specifications have been selected from two different projects. Project Alpha is the requirements specification of the prototype for a successor system of an library system called BTJ 2000. This system is widely used in public libraries in Sweden. The system is becoming old-fashioned needs to be re-newed to enter the market of university libraries. The old system is built on a Unix server and connected text-based terminals. The new system must have a graphical user interface to increase the user friendliness. In addition the new system want to increase the possibility for library customers to self service. The new system is to make use of new technologies such as java and the world-wide-web.

The requirements specification of project Beta defines a support and service application for haemo dialysis machines. The new application shall aid service technicians in error tracing when the system behaves in erroneous ways or for doing diagnostics on the system for fault prevention.

Both projects have been performed by teams of between 10 and 15 students as part of their software engineering education with commercial companies as customers and represent commercial software applications. In fact, one of the projects resulted in a ready product that has been included in the product portfolio of the customer.

3.6 Operation

The experiment is executed according to the following steps: (schedule in figure 2)

1. A selection of individuals with varying programming and design experience are appointed.
2. All individuals receive a presentation/tutorial of the method. This is done as a part of the experiment briefing. A document describing the method is also available for all individuals to study during the experiment.
3. Each person fills in a form with some data about his or her experience and knowledge.
4. Individuals are assigned to groups of three using the matched pairs principle (see section 3.8). We planned to involve 12 subjects divided into 4 groups, i.e. group A through D.
5. The groups are assigned a 'treatment' and the requirement specification for the first project is handed out. The group A and B that are assigned to the prepared group profile method, start on individual basis which is part of both treatment 1 and treatment 3.
6. When 1.5 hours of time have passed the profiles of the individuals are collected during a short break. During the break the individual profiles are photocopied and handed back to the respective authors. Great care must be taken in that the profile returns to the correct person without any other person getting a glimpse.
7. After the break groups A and B continue in plenum and each group prepares a group scenario profile.
8. At noon, all the group profiles are collected and groups proceed to lunch.
9. After lunch, the process is repeated from step 5, but groups A and B now produce a group profile from start, and groups C and D begin with preparing an individual scenario profile before proceeding in plenum to produce their respective group scenario profile.

Time	Group A	Group B	Group C	Group D	Project
08.00	Introduction and experiment instructions				
09.00	Individual Profile Preparation	Individual Profile Preparation	Unprepared Group	Unprepared Group	Alpha
10.30	Prepared Group	Prepared Group			
12.00	LUNCH BREAK				
13.00	Unprepared Group	Unprepared Group	Individual Profile Preparation	Individual Profile Preparation	Beta
14.30			Prepared Group	Prepared Group	
16.00	De-briefing				

Figure 2. *Experiment One Day Schedule*

All information collected during the experiment is tracked by an identification code also present on the personal information form. Consequently, the data is not anonymous, but this is, in our judgement, not an imminent problem since the data collected is not, in any clear way, directly related to individual performance. Instead being able to identify persons that had part in interesting data points is more important than the risk of getting tampered data because of lack of anonymity.

3.7 Data Collection

The data collection in this experiment is primarily to collect the results of the work performed by the participants, i.e. the scenario profiles. However, some additional data is required, for example, a form to probe the experience level of the participants. The following forms are used:

- personal information form (Appendix I)
- individual scenario-profile form (Appendix II)
- group scenario-profile form (Appendix III)

The forms have been designed and reviewed with respect to gathering the correct data and ease of understanding, since misunderstanding the forms pose threats on the validity of the data. The personal information form is filled in by the participants after the introduction and collected immediately The others forms are collected during the experiment. During the experiment briefing all forms are presented and explained.

The Personal Scenario Profile form (Appendix II) is handed out to the experiment subjects at the beginning of the Individual Profile Preparation activity. It will be collected at the end of the activity and photocopies will be made for archives.

The Group Scenario Profile form (Appendix III) is handed out to the groups, along with the respective individuals completed profile form, at the start of the Group Synthesis Consensus activity.

3.8 External Threats

Some external threats can be identified in the experiment design, e.g. differences in experience and learning effects. For the most part of the identified threats measures have been taken to eliminate these by adapting the design. By using the blocked project design we eliminate, for example, the risk of learning effects, and in the case of differences in participants experience we use the matched pairs technique when composing the groups, to ensure that all groups have a similar experience profile.

Although precautions has been taken in selecting a system from a large student project which is the result of a 'real' customers demands. This cannot absolutely exclude that the system is irrelevant. The industry customer often use this kind of student projects as proof of concept implementations. However, there are no reasons for the scenario profile prediction method not to be applicable in this situation like the above mentioned. And for the purpose of the experiment we feel that it is more crucial to the results that the individuals in the project have no experience with the particular system's successors.

3.9 Internal Threats

The internal validity of the experiment is very much dependent on the way we analyze and interpret the data. During the design, preparation, execution and analysis of the experiment and the experiment results, we have found some internal threats or arguments for possible internal validity problems. We discuss them are their impact in the following subsections.

Ranking Scheme

Some problems exists with this method of ranking. First, the reference profile will be relative to the profiles since it is based on them. Second, there might be one single brilliant person that has realized a unique scenario that is really important, but since only one profile included it, its impact will be strongly reduced.

The first problem might not be a problem, if we accept the assumption that the number of profiles containing a particular scenario is an acceptable indicator of its relevance. In the case of having significant differences between the individually created profiles and the group profiles, the differences will be normalized in the reference profile. Given that the individually prepared profiles are more diverse than the profiles prepared by groups, those profiles will render on average lower rank scores, while the group profiles will render on average higher rank scores. In case the results of the experiment is in favor to the null-hypothesis, we will not be able to make any distinction between the group prepared profiles or the individually prepared profiles ranking scores.

The second problem can be dealt with in two ways. First we can make use of the delphi method or the wide band delphi [7]. In that case we would simply synthesize the reference profile, distribute it and have another go at the profiles and get more refined versions of the profile. The second approach is to make use of the weightings of each scenario and make the assumption that the relevance of a scenario is not only indicated by the number of profiles that include it, but also the weight it is assigned in these profiles. The implication of this is that a scenario that is included in all 20 of the profiles but has a low average weighting, is relevant but not significant for the outcome. However, a scenario included in only one or a few profiles is deemed less relevant by the general opinion. If it has a high average weighting, those few consider it very important for the outcome. Now, we can incorporate the average weighting in the ranking method by defining the ranking score for a profile as the sum of rank products (the frequency times the average weighting) of its scenarios. This would decrease the impact of a commonly occurring scenario with little impact and strengthen the less frequent scenarios with higher impact.

Our conclusion, however, is that the ranking scheme used in this paper does not suffer from any major threats to the validity of the conclusions that we base on it.

Technique itself based on hypothesis

The ranking of profiles is based on the assumption that frequent scenarios are more important and, thus, lead to higher scores. A possible threat to internal validity could be that this would be beneficial for, especially prepared, group profiles since many of the scenarios in the group profile will also be present in one or more of the individual profiles of the group members. One could suspect that the ranking technique is biased towards prepared groups and consequently implicitly favors the hypotheses we hope to confirm.

A closer analysis shows that scenarios that are included in the profiles defined by prepared groups indeed have higher frequencies. However, this does not just benefit the score for the prepared group profile, but also the individual profiles of the group members. Since both profile types benefit, this does not influence the outcome of the experiment.

The analysis technique biased for quantity instead of quality.

It could be the case that a profile reaches a high score by using many, unimportant scenarios. Some scenarios may not even be related to the project. This profile, that intuitively should obtain a low score, scores higher than a profile with fewer, but more important scenarios.

When we examine the example closer, we find that the first profile in the example could render a maximum score of 60, because of the limitation on six categories and ten scenarios in each category. A profile with only ten scenarios would have to score on average more than six per profile to out rank the long profile. In the first profile we would get a ratio of the number of scenarios in the profile and the score for that profile of exactly one. In the other profile example, the ratio would be more then one. In figures 8 and 9, the ratios are presented for the projects.

Concluding, although this may, theoretically, be an internal validity threat, it did not occur in the experiment reported in this paper.

The coding of the scenarios to produce the reference profile is biased towards one of the proposed hypotheses.

To create the reference profile all scenarios are put together in a table, i.e. the union of all profiles. Since scenarios may be equivalent in semantics in spite of being lexically different, the experimenter needs to

establish what scenarios are equivalent, i.e. coding the data. The coding is done by taking the list of scenarios and for every scenario check if there was a previous scenario describing a semantically equivalent situation. This is done using the database table and we establish a reference profile using a frequency for each unique scenario.

The possible threat is that the reference profile reflects the knowledge of the person coding the scenarios of all the profiles, instead of the consensus among the different profiles. To reduce the impact of this threat, the coded list has been inspected by an additional person. Any deviating interpretations have been discussed and the coding have been updated according to the consensus after that discussion.

4. Analysis & Interpretation

In the previous section, the design of the experiment was discussed. In this section, we report on the results of conducting the experiment. We analyze the data by preparing the reference profiles, calculating the scores for all profiles and determining average and standard deviations for each type of treatment. Finally, the hypotheses stated in section 3.2 are evaluated.

4.1 Mortality

The design of the experiment requires the participation of 12 persons for a full day. For the experiment we had managed to gather in excess of 12 voluntary students, with the promise of a free lunch during the experiment and a nice á la carte-dinner after participating in the experiment. Unfortunately, some students did not show up for the experiment without prior notification. As a result, the experiment participants were only nine persons. Instead of aborting the experiment, we chose to keep the groups of three and to proceed with only three groups, instead of the planned four. As a consequence, the data from the experiment is less complete as intended (see figures 5 and 6). But nevertheless, we feel that the collected data is useful and allow us to validate our hypotheses and make some interesting observations.

Once the experiment had started, we had no mortality problems, i.e. all the participants completed their tasks and we collected the data according to plan.

4.2 Reference profiles

During the experiment we collected 142 scenarios from 9 profiles for project alpha, 85 scenarios from 6 profiles for project beta, totalling 227 scenarios from 15 profiles. The scenarios were coded with references to the first occurring equivalent scenario using a relational database to later generate one reference profile per project. The reference profile for project alpha included 72 scenarios and project beta included 39. The top 10 and top 8 scenarios of the reference profiles are presented in figures 3 and 4. In the alpha case, we note that one scenario has been included in all nine profiles, i.e. has the score nine. In the beta project, we note that the top scenario is included seven times in six profiles. This could be an anomaly, but when investigated, we recognized that in one of the profiles from the beta project two scenarios have been coded as equivalent. This is probably not the intention by the profile creator, an individual person in this case, but we argue that the two scenarios is only slightly different and should correctly be coded as equivalent to the same scenario in another profile.

Description	**frequency**
new DBMS	9
new operating system on server	7
new version of TOR	7
introduction of smart card hardware	5
additional search capabilities	5
pureWeb (cgi) clients	4
support for serials	4
new communication protocol	4
user interface overhaul	4
new java technology	4

Figure 3. *Alpha Reference Profile Top 10*

Description	frequency
remote administration	7
upgrade of database	6
upgrade of OS	6
real-time presentation of values	4
change of System 1000 physical components (3-4 pcs.)	4
rule-based problem-learning system	3
change from metric system to american standard	3
new user levels	3

Figure 4. *Beta Reference Profile Top 8*

Another interesting observation to make is that among the top three scenarios in both projects we find changes of the database management system and changes of the operating systems, either new version or upgrade and we find it in just about all the profiles. This suggests that these two changes to a system are among the first scenarios that come to mind when thinking about future changes to a system.

Finally, it is worth noting that the major part of the top scenarios are related to interfacing systems or hardware. Only a few of the scenarios in the top 10 or 8 are related to functionality specific to the application domain, e.g. “support for serials” in figure 3 or “new user levels” in figure 4.

4.3 Ranking & Scores

In this section, we presented the coded and summarized data collected from the experiment. In the table presented in figure 5 the score for each of the profiles generated for the Alpha project are presented. It is interesting that group A and B, that both are prepared groups, score strikingly high scores, compared to the other profiles in project Alpha. Further, we notice little difference between the profiles created by the individual persons and the unprepared groups, in neither project.

Identity	Members	Profile Score	Remarks
group A	C,D,E	72	individual preparation
group B	H,I,F	77	individual preparation
group C	A,B,G	49	no individual preparation
Arthur			only participated in a group
Bertram			only participated in a group
Charlie		39	
David		38	
Ernie		45	
Frank		19	
Gordon			only participated in a group
Harald		66	
Ivan		55	

Figure 5. *Project Alpha*

In the table in figure 6, the profile scores for project Beta are presented. The prepared group, C in this case, scores a very high score, but the unprepared groups, A and B, score much less. This is interesting since the groups members are the same for both projects.

Identity	Members	Profile Score	Remarks
group A	C,D,E	44	no individual preparation
group B	H,I,F	37	no individual preparation
group C	A,B,G	72	individual preparation
Arthur		36	
Bertram		43	
Charlie			only participated in a group
David			only participated in a group
Ernie			only participated in a group
Frank			only participated in a group
Gordon		33	
Harald			only participated in a group
Ivan			only participated in a group

Figure 6. *Project Beta*

In figure 7 the average scores for each type of treatment is presented for the Alpha, Beta project and in total. In addition, the standard devia-

tion over the scores and the number of cases is presented. The average score for prepared groups is substantially higher than the score for unprepared groups or individuals. Secondly, the standard deviation is the largest for individuals, i.e. 13, but only 6 for unprepared groups and 3 for prepared groups. Finally, it is interesting to note that the standard deviation for all profiles is larger than for any of the treatments, which indicates that the profiles for each type of treatment are more related to each other than to profiles for other treatment types.

Treatment	**Alpha**	**Beta**	**Total**	**Std. Dev.**	**#cases**
Individual	43	37	41	13	9
Unprepared group	39	40	43	6	3
Prepared group	75	72	74	3	3
Total	51	44	48	17	15

Figure 7. *Average and Standard Deviation Data*

In section 3.9, we discussed various threats to the internal validity of the experiment. One of the discussed threats is the risk that a profile with many unimportant scenarios scores higher than a profile with fewer, but more important scenarios, while this is counter intuitive. Based on the data in figure 8 and 9, we can conclude that although a theoretical threat was present, it did not occur in the experiment.

Identity	**Profile Score**
group B	77
group A	72
Harald	66
Ivan	55
group C	49
Ernie	45
Charlie	39
David	38
Frank	19

Identity	**Ratio**
Ernie	4,5
Harald	3,9
group A	3,8
Charlie	3,5
group B	3,3
David	3,2
Ivan	2,9
group C	2,7
Frank	1,5

Identity	**Profile Length**
group B	23
group A	19
Ivan	19
group C	18
Harald	17
Frank	13
David	12
Charlie	11
Ernie	10

Figure 8. *Project Alpha*

Identity	Profile Score
group C	72
group A	44
Bertram	43
group B	37
Arthur	36
Harald	33

Identity	Ratio
Arthur	3,6
group A	3,4
Bertram	3,3
group C	3
Harald	2,8
group B	2,8

Identity	Profile Length
group C	24
group A	13
group B	13
Bertram	13
Harald	12
Arthur	10

Figure 9. *Project Beta*

4.4 Evaluating the Hypotheses

The experiment data does not allow us to identify any significant difference in ranking between profiles created by an independent person or profiles created by a group with unprepared members. Hence we *cannot dismiss* the null hypothesis, $\mathbf{H_{01}}$.

The first null hypothesis, $\mathbf{H_{01}}$, counters the two hypotheses $\mathbf{H_1}$ and $\mathbf{H_{10}}$. Since the experiment data does not allow us to dismiss the null hypothesis $\mathbf{H_{01}}$, we cannot expect to validate those two hypotheses and therefore, we can *dismiss* $\mathbf{H_1}$ and $\mathbf{H_{10}}$. We can, however, make an *interesting observation* on the variation in the ranking scores between the profiles of the individuals and the unprepared groups. The scores of the profiles created by independent persons range from 19 to 62 over both projects, while the scores of the profiles created by the unprepared groups only ranges from 32 - 49 over both projects. The observation is also supported by the standard deviation values presented in figure 7. This suggests that using unprepared groups does not lead to higher scores on the average, but provides more stable profiles and reduces the risk for extreme results, i.e. outliers.

With respect to the second null hypothesis, $\mathbf{H_{02}}$, we find compelling evidence in the analyzed data for a significant difference between the profiles created by individuals, with an score average of 43, and profiles created by a group with prepared members, with an score average of 74 (see figure 7). We also observe that no profile created by an independent person has scored a higher score than any profile created by a group with prepared members. Hence, we can *dismiss* the second null hypothesis, $\mathbf{H_{02}}$.

Because we were able to dismiss the second null hypothesis, it is worthwhile to examine the two related hypotheses, $\mathbf{H_2}$ and $\mathbf{H_{20}}$. The scores clearly show that the group with prepared members in all cases have scored higher than the profiles created by independent persons. This allows us to *confirm* of the hypothesis, $\mathbf{H_2}$ and allow us to *dismiss* the counter hypothesis, $\mathbf{H_{20}}$.

The last null-hypothesis is $\mathbf{H_{03}}$. With respect to this hypothesis, we find evidence that a significant difference exists between the average scores for unprepared and prepared groups. Profiles created by groups with prepared members score 74 on average, as opposed to profiles from groups with unprepared members, that score 41 on average. Hence, we *can dismiss* the null hypothesis, $\mathbf{H_{03}}$, and evaluate the related hypotheses $\mathbf{H_3}$ and $\mathbf{H_{30}}$. The average score for prepared groups is 74, which is considerably higher than the average score for unprepared groups, i.e. 41. Based on this, we are able to *confirm* hypothesis $\mathbf{H_3}$ and, consequently, *dismiss* the counter hypothesis $\mathbf{H_{30}}$.

5. Related Work

Architecture assessment is important for achieving the required software quality attributes. Several authors propose and advocate scenario based techniques for architecture assessment. A well-known method is the scenario-based architecture assessment method (SAAM) [12]. SAAM assesses the architecture after the architecture design and incorporates all stakeholders of the system. Other methods include the architectural trade-off analysis method (ATA) [11] that uses scenarios to analyze and bring out trade off points in the architecture. The 4+1 View method [13] uses scenarios in its fifth view to verify the resulting architecture. To this point, no studies have been reported on the creation of scenario profiles for architecture assessment.

In [3] a framework for experimentation in software engineering is presented along with a survey of experiments conducted up to 1986. In our work with the experiment design we have used this framework to ensure an as robust design as possible.

6. Conclusions

During recent years, the importance of explicit design of the architecture of software systems is recognized. This is because the software architecture constrains the quality attributes of the system. Consequently, architecture assessment is important to decide how well the software architecture supports various quality attributes. One can identify three categories of architecture assessment techniques, i.e. scenario-, simulation- and static model-based assessment. However, these techniques make all use of scenario profiles. Although some scenarios profiles can be defined as 'complete', i.e. covering all scenarios that can possibly occur, most scenario profiles are 'selected'. *Selected scenario profiles* contain a representative subset of the population of all possible scenarios.

Scenario profiles are generally defined as a first step during architecture assessment. However, defining a selected scenario profile is subjective and we have no means to decide upon the representativeness of the profile. Also, to the best of our knowledge, no studies are available about the effects of individuals on the definition of scenario profiles, i.e. what is the deviation between profiles defined by different individuals. The same is the case for groups defining scenario profiles.

In this paper we have presented the design and results of an experiment on three methods for creating scenario profiles. The methods, or treatments, for creating scenario profiles that were examined are (1) an individual prepares a profile, (2) a group with unprepared members prepares a profile and (3) a group with members that, in advance, created their individual profiles as preparation.

We also have stated a number of hypotheses, with the corresponding null-hypotheses and, although, the results of the experiment data do not allow us dismiss each of our null-hypotheses, we find support for the following hypotheses:

> **H_2** = *Scenario profiles created by groups with prepared members, generally get better scores than scenario profiles created by an individual person.*

> **H_3** = *Scenario profiles created by groups with prepared members generally get better scores than group profiles with unprepared members.*

Thus, based on the experiment data, we are able to conclude that using groups with prepared members is the preferable method for preparing scenario profiles.

In addition we have also made a number of observations during the experiment and during the analysis of the data. These are as follows:

1. Two change scenarios occurring in just about all the profiles were new version or upgrade of the database management system and the operating system.

2. Few scenarios among the top 10 or 8 are related to the application, instead most of the scenarios in the top are related to interfacing systems or hardware.

3. The standard deviation in score is lower for profiles created by unprepared groups, than for individuals, although the average of the profiles scores cannot not be said to differ significantly between the two. A plausible interpretation is that the group reduces the variation by filtering out the scenarios that are questionable, in contrast to the individually created profiles.

Acknowledgments

We would like to thank the students who participated in the experiment.

References

[1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, A. Moormann Zaremski, *Recommend Best Industrial Practice for Software Architecture Evaluation,* CMU/SEI-96-TR-025, 1997.

[2] Basili, V.R., Selby, R.W., Hutchens, D.H., *"Experimentation in Software Engineering"*, IEEE Transactions on Software Engineering, vol. se-12, no. 7, July, 1986

[3] L. Bass, P. Clements, R. Kazman, *'Software Architecture In Practise'*, Addison Wesley, 1998.

[4] P. Bengtsson, '*Towards Maintainability Metrics on Software Architecture: An Adaptation of Object-Oriented Metrics*', First Nordic Workshop on Software Architecture (NOSA'98), Ronneby, August 20-21, 1998.

[5] P. Bengtsson, J. Bosch, 'Scenario Based Software Architecture Reengineering', *Proceedings of International Conference of Software Reuse 5 (ICSR5)*, Victoria, Canada, June 1998.

[6] P. Bengtsson, J. Bosch, 'Architecture Level Prediction of Software Maintenance', *Proceedings of Third European Conference on Software Maintenance and Reengineering*, pp. 139-147, March 1999.

[7] Boehm, B.W, *Software Engineering Economics*, Prentice Hall, 1981.

[8] J. Bosch, P. Molin, 'Software Architecture Design: Evaluation and Transformation', in proceedings of *1999 IEEE Engineering of Computer Based Systems Symposium* (ECBS99), Nashville, USA, March 1999

[9] J. Carrière, R. Kazman, S. Woods, "Assessing and Maintaining Architectural Quality", in proceedings of The Third European Conference on Software Maintenance and Reengineering (CSMR'99), IEEE Computer Society, pp. 22-30, 1999

[10] J.C. Dueñas, W.L. de Oliveira, J.A. de la Puente, 'A Software Architecture Evaluation Method,' *Proceedings of the Second International ESPRIT ARES Workshop,* Las Palmas, LNCS 1429, Springer Verlag, pp. 148-157, February 1998.

[11] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The Architecture Tradeoff Analysis Method, Proceedings of ICECCS, (Monterey, CA), August 1998

[12] R. Kazman, L. Bass, G. Abowd, M. Webb, 'SAAM: A Method for Analyzing the Properties of Software Architectures,' *Proceedings of the 16th International Conference on Software Engineering*, pp. 81-90, 1994.

[13] P.B. Krutchen, 'The 4+1 View Model of Architecture', *IEEE Software*, pp. 42-50, November 1995.

[14] W. Li, S. Henry, 'Object-Oriented Metrics that Predict Maintainability', *Journal of Systems and Software,* vol. 23, no. 2, pp. 111-122, November 1993.

Appendix I. Individual Information Form

Identification: _____

Started SE Curriculum: ❑ 1994 ❑ 1995 ❑ 1996 ❑ 1997

Working Since: _____ (Year)

Number of Study Points: _____

Maintenance Experience: _____ (Years/Months)

Software Development Experience: _____ (Years/Months)

Knowledge in:

C	❑ None	❑ Novice	❑ Skilled	❑ Expert
C++	❑ None	❑ Novice	❑ Skilled	❑ Expert
Java	❑ None	❑ Novice	❑ Skilled	❑ Expert
Eiffel	❑ None	❑ Novice	❑ Skilled	❑ Expert
Pascal/Delphi	❑ None	❑ Novice	❑ Skilled	❑ Expert
Visual Basic	❑ None	❑ Novice	❑ Skilled	❑ Expert
Assembly language	❑ None	❑ Novice	❑ Skilled	❑ Expert

Software Modelling Experiences:

Booch	❑ None	❑ Novice	❑ Skilled	❑ Expert
OMT	❑ None	❑ Novice	❑ Skilled	❑ Expert
Objectory	❑ None	❑ Novice	❑ Skilled	❑ Expert

Training in Software Architecture:_____ (Days or Credits)

Requirements Experience: _____ (Years / Months)

Appendix II. Individual Scenario Profile

Identification: _____

Project: ❑ Alpha ❑ Beta

Previous Domain experience: _____ Years

No.	Cat.	Scenario Description	Weight
S1			
S2			
S3			
S4			
...			
S80			

ID	Category
C1	
C2	
...	
C10	

Appendix III. Group Scenario Profile

Project: ❑ Alpha ❑ Beta

Group Identification: _____

Identification: _____ & _____& _____

Previous Domain Experience: _____ & _____& _____ Years

No.	Cat.	Scenario Description	Weight
S1			
S2			
S3			
S4			
...			
S80			

ID	Category
C1	
C2	
...	
C10	