Haemo Dialysis Software Architecture Design Experiences

PerOlof Bengtsson & Jan Bosch
Department of Computer Science and
Business Administration
University of Karlskrona Ronneby
S-372 25 Ronneby, Sweden, +46 457 787 41
[PerOlof.Bengtsson | Jan.Bosch]@ ide.hk-r.se
http://www.ide.hk-r.se/ [~pob | ~bosch]

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 [22]. 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.

Keywords

Software Architecture, Software Architecture Design

1 INTRODUCTION

Software architecture design is an art. Today only a few, sketchy methods exist for designing software architecture [3,13,14,15]. 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 [10], software architecture patterns [6], and various ADLs with accompanying analysis tools [8, 16]. 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, 5, 19]. These projects have given us some hard-earned hands on experience of what really makes the design of software architecture difficult.

The remainder of this paper is organized as follows. 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.

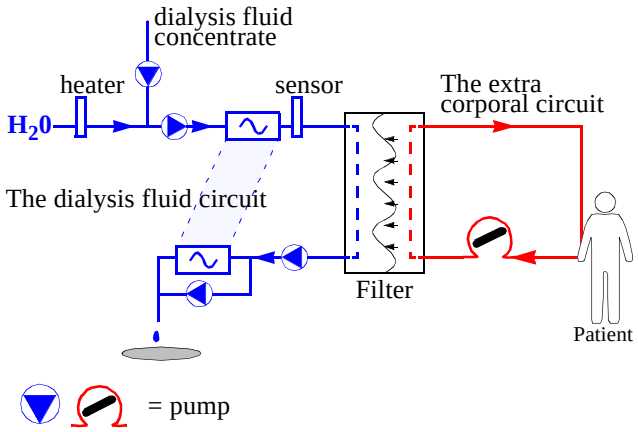

Figure 1: Schematic of Haemo Dialysis Machine

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).

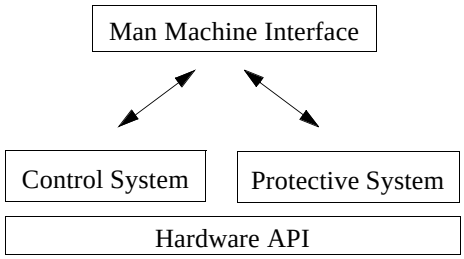

Figure 2: Legacy system decomposition

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:

- *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.

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 [7], 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 [7].

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 our ARCS design method for designing the architecture [3], presented graphically 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 [21]. 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 intutively 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 behavioural rules to the architecture, e.g. Periodic Objects [19].
- *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.

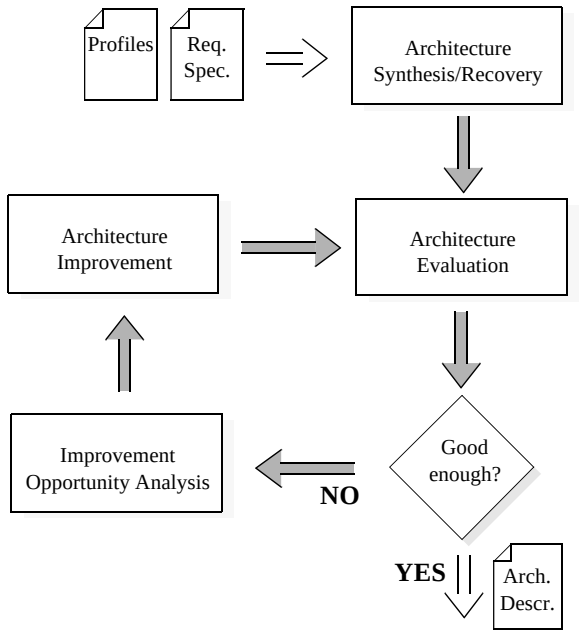

Figure 3: Repeated evaluation for control of the design

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 [11] 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 assessment 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., [14], 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 [4, 12, 20, 23], 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 behaviour 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 phenomenom 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. In cases where differences in opinions existed, the problem or strength could be explained using our more 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. To make this even harder, in our experience, 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 often 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 3.2, 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 [15], but found it hard to capture the essence of the architecture. Projet 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 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 written 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 [15], i.e. Logical View and Dynamic View. The development 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.

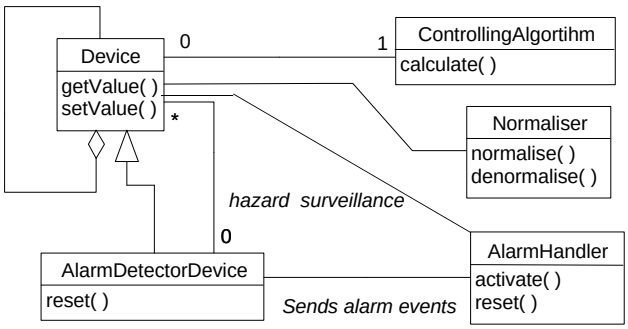

Figure 4: The relations of the archetypes

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 [7]. 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. 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 entities of this archetype may exists in the application architecture and the periodic object is responsible for registering itself with the scheduler. (see figure 5)

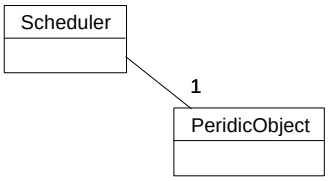

Figure 5: Basic Concurrency with Periodic Objects

4.3 Connector Archetypes

The communication between the architecture elements is done by using *causal connections* [17]. The principle is similar to the *Observer pattern* [10] and the *Publisher-Subscriber pattern* [6]. 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. 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.

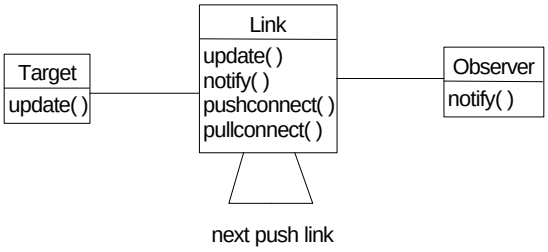

Figure 6: Push/Pull Update Connection

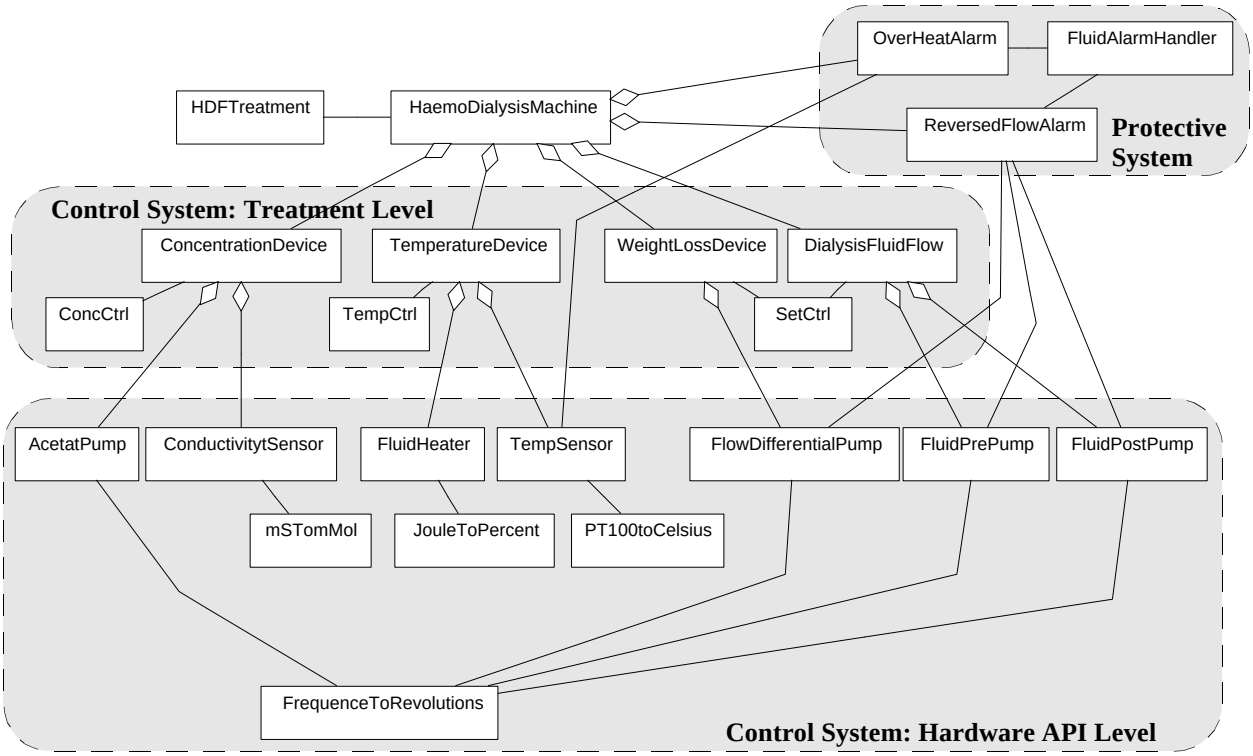

Figure 7: Example haemo dialysis Application Architecture

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
```

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.

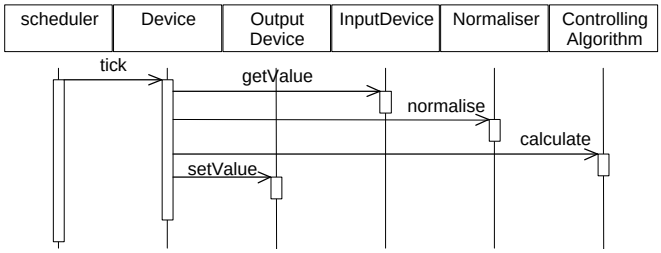

Figure 8: The message sequence of a control *tick()*

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.

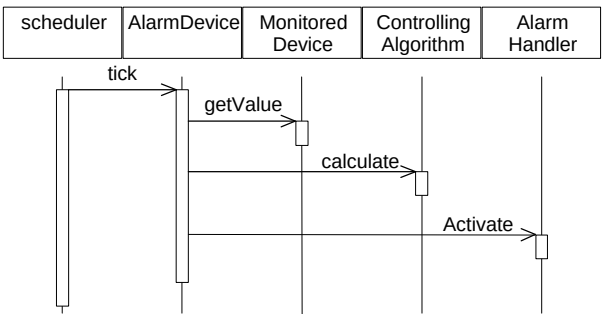

Figure 9: A *tick()* of alarm monitoring

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) nurse disconnects patient from machine, (11) patient and treatment records saved, (12) disinfecting, (13) power down/ stand by.

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 [19] 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 it to demonstrate the patient safety for the time which the patient is connected with the machine, i.e. 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 [10] 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 that made the architecture more complex 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*. The new abstraction allowed us to use sub-devices that were communicating with the hardware and dealt with the low-level problems such as normalization and hardware API's. These low-level devices were connected as logical inputs and outputs to other logical devices. These logical devices handle logical entities that are less hardware specific, 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 that can be 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 [19] 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 supplier.

The third option is perhaps the most straightforward option and could probably be done in such a way that it works. The profound problem is that it becomes un-deterministic. This is affecting the demonstrability negatively. Having software certification in mind, it is unrealistic to believe that such an implementation would be allowed in a haemo dialysis machine.

The second option give some 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 more clearly, using the scheduler and its list of periodic objects especially since it has been used successfully in earlier 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* [17] 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. (None of the project members volunteered to act as a patient).

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 [18], 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 that is monitored for has its own AlarmDetectorDevice. This makes it easier to visualize the lack of monitoring for certain hazards and it makes it easier to keep the relationships between hazard conditions visible.

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 sophisticated operating system in the haemo dialysis machine.

Demonstrability. Our goal with the architecture design when concerned with demonstrability was to achieve a design that made the specification of a treatment very similar to how a

domain expert would express the treatment in his or her own words. The example with the source code in section 4.4 for specifying a treatment in the application is very intuitive and similar to the information we have about 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.

ACKNOWLEDGMENTS

We would like to thank our partners in the research project, Althin Medical AB, Ronneby, and Elektronik Gruppen AB, Helsingborg, especially, Anders Kambrin, Mogens Lundholm, and Lars-Olof Sandberg 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, January 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, pp. 308-317, 2-5 june, 1998
4. G. Booch, *Object-Oriented Analysis and Design with Applications* (2nd edition), Benjamin/Cummings Publishing Company, 1994.
5. J. Bosch, 'Design of an Object-Oriented Measurement System Framework', Accepted for publication in *Object-Oriented Application Frameworks*, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, (forthcoming)
6. F. Buschmann, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996.
7. CEI/IEC 601-2 Safety requirements standard for dialysis machines
8. P. C. Clements, *A Survey of Architecture Description Languages*, Eight International Workshop on Software Specification and Design, Germany, March 1996
9. N.E. Fenton, S.L. Pfleeger, *Software Metrics - A Rigorous & Practical Approach* (2nd edition), International Thomson Computer Press, 1996.
10. Gamma et. al., *Design Patterns Elements of Reusable Design*, Addison.Wesley, 1995.
11. IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12-1990.
12. I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-oriented software engineering. A use case approach*, Addison-Wesley, 1992.
13. 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.
14. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, *The Architecture Tradeoff Analysis Method*, Proceedings of ICECCS, (Monterey, CA), August 1998
15. P.B. Kruchten, 'The 4+1 View Model of Architecture,' *IEEE Software*, pp. 42-50, November 1995.
16. 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
17. C. Lundberg, J. Bosch, "Modelling Causal Connections Between Objects", Journal of Programming Languages, 1997.
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. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.
21. S. Shlaer, S.J. Mellor, 'Recursive Design of an Application-Independentt Architecture', *IEEE Software*, pp. 61-72, January/ February 1997.
22. W. Tracz, 'DSSA (Domain-Specific Software Architecture) Pedagogical Example,' *ACM Software Engineering Notes*, Vol. 20, No. 3, pp. 49-62, July 1995.
23. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.