Quality Attributes in Software Architecture Design

Lars Lundberg, Jan Bosch, Daniel Häggander and Per-Olof Bengtsson
Department of Software Engineering and Computer Science
University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden

Abstract

Quality attributes of large software systems are to a large extent determined the system's software architecture, i.e. qualities such as performance and modifiability depend at least as much on the overall architecture as on the code level implementation. Our experience shows that there are conflicts between modifiability and performance. The largest conflicts occur when there is a requirement that it should be possible to modify the system by run-time reconfigurations. Consequently, there is a need for providing tradeoffs between modifiability and performance when designing the system's architecture.

Based on experiences from five industrial projects we define eight design guidelines and a small taxonomy for some performance related quality attributes as well as for attributes related to the modifiability of the system. We also incorporate the guidelines in a design method, thus making it clear how and when the guidelines should be used.

1 Introduction

Perhaps the most complex activity during application development is the transformation of a requirement specification into an application architecture. The other phases also are challenging activities, but they are better understood and more methodological and technological support is available. The process of architectural design is less formalized and often more like intuitive craftsmanship than rational engineering.

The domain of software architecture has received considerable attention during recent years. This is, at least to some extent, because especially quality requirements (QRs) are heavily influenced by the architecture of the system. Some QR are conflicting, thus making it necessary to find an architecture that provides an appropriate compromise. Architectural design is a typical multiple objective design activity where the software engineer has to balance the various requirements during architectural design. Although there are methods for analyzing specific quality attributes, these analyses have typically been done in isolation [12][14][18].

In this paper we present our experiences from a number of architecture design projects, ranging from telecommunication applications to embedded systems. Based on the accumulated experience from these projects, we define a set of guidelines and outline a design method that make it possible to obtain a reasonable balance between different QR. At this point we provide guidelines for performance related quality attributes, i.e. throughput and response time, as well as guidelines related to modifiability, i.e. the cost for modifying and reconfiguring the system after initial deployment, in the maintenance phase.

2 Experiences

In this section describe five industrial systems that we have studied and summarize some of the experiences from these projects. A more detailed description of the systems can be found in [2][3][6][8][9].

2.1 Billing Gateway (BGw)

2.1.1 System description

BGw collects billing information about calls from mobile phones [8]. The system has been developed by Ericsson. BGw is written in C++ (approximately 100,000 lines of code) using object-oriented design, and the parallel execution has been implemented using Solaris threads.

BGw transfers, filters and translates raw billing information from Network Elements (NE), such as switching centers and voice mail centers, in the telecommunication network to billing systems and other Post Processing Systems (PPS). Customer bills are then issued from the billing systems. The raw billing information consists of Call Data Records (CDRs). The CDRs are continuously stored in files in the network elements. With certain time intervals or when the files have reached a certain size, these files are sent to the BGw.

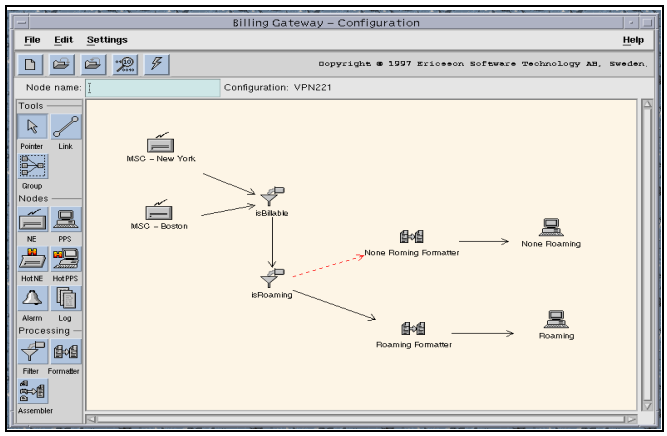

Figure 1: BGw configuration window.

There is a graphical user interface connected to BGw. In this interface the different streams of information going through the gateway are visualized as a directed graph. Figure 1 shows an application where there are two network elements producing billing information (the two leftmost nodes). These are called "MSC - New York" and "MSC - Boston" (MSC = Mobile Switching Center). The CDRs from these two MSCs are sent to a filter called "isBillable". There is a function associated with each filter, and in this case the filter function evaluates to true

for CDRs which contain proper information about billable services. CDRs which do not contain information about billable services are simply filtered out. The other CDRs are sent to another filter called “isRoaming”. In this case, there are two streams going out from the filter.

The function associated with “isRoaming” evaluates to true if the CDR contains information about a roaming call (a roaming call occurs when a customer is using a network operator other than his own, e.g when travelling in another country). In this case, the record is forwarded to a formatter, and then to a billing system for roaming calls, otherwise the record is sent to a formatter and billing system for non-roaming calls.

The record format used by the billing systems differs from the format produced by the MSCs. This is why the CDRs coming out of the last filter have to be reformatted before they can be sent to the billing systems. The graph in Figure 1 is only one example of how billing applications can be configured.

Figure 2 shows the major threads for the application in Figure 1. When there is no flow of data through the BGw, the system contains a number of static threads. When there is a flow of information going through the system, additional threads are created dynamically. When a NE sends a billing file to the BGw a data collection thread is created. This thread reads the file from the network and stores it on disk. When the complete file has been stored the data collection thread terminates.

Data processing, i.e. the part of BGw that does the actual filtering and formatting, is implemented in a different way, i.e. there is one data processing thread for each NE node in the configuration (see Figure 2).

2.1.2 Experiences

The system architecture is parallel and we expected good multiprocessor speedup. However, the speedup was very disappointing; in the first version the performance dropped when the number of processors increased. The reason for this was that the dynamic memory management was a major bottleneck.

The designers of BGw wanted a flexible system where new CDR formats could be handled without changing the system. One major component in BGw was a very flexible parser that could handle data formats specified using ASN.1. This parser uses a lot of dynamic memory.

In order to increase the flexibility even more, a new language which makes is possible to define more complex filters and formatters was defined. The new language makes it easier to adapt BGw to new environments and configurations. However, the introduction of the new language led to a very intensive use of dynamic memory, even for small configurations. Consequently, the excessive use of dynamic memory stemmed from the efforts of building a flexible and configurable system.

It turned out that the performance problems due to dynamic memory management could be removed relatively easy. By replacing the standard memory management routines in Solaris with a multiprocessor implementation called ptmalloc, the performance was improved significantly. By investing in a redesign effort of 1-2 weeks the performance was improved with approximately a factor of eight for the sequential case and a factor of more than 100 when using a multiprocessor. The major part of the redesign was to introduce memory pools for commonly used objects.

2.2 Fraud Control Centers (FCC)

2.2.1 System description

When cellular operators first introduce cellular telephony in an area, their primary concern is to establish capacity, coverage and signing up customers. However, as their network matures financial issues become more important, e.g lost revenues due to fraud.

Software in the switching centers, provides real-time surveillance of suspicious activities, e.g. irregular events associated with a call. The idea is to identify potential fraud calls and have them terminated. However, one single indication is not

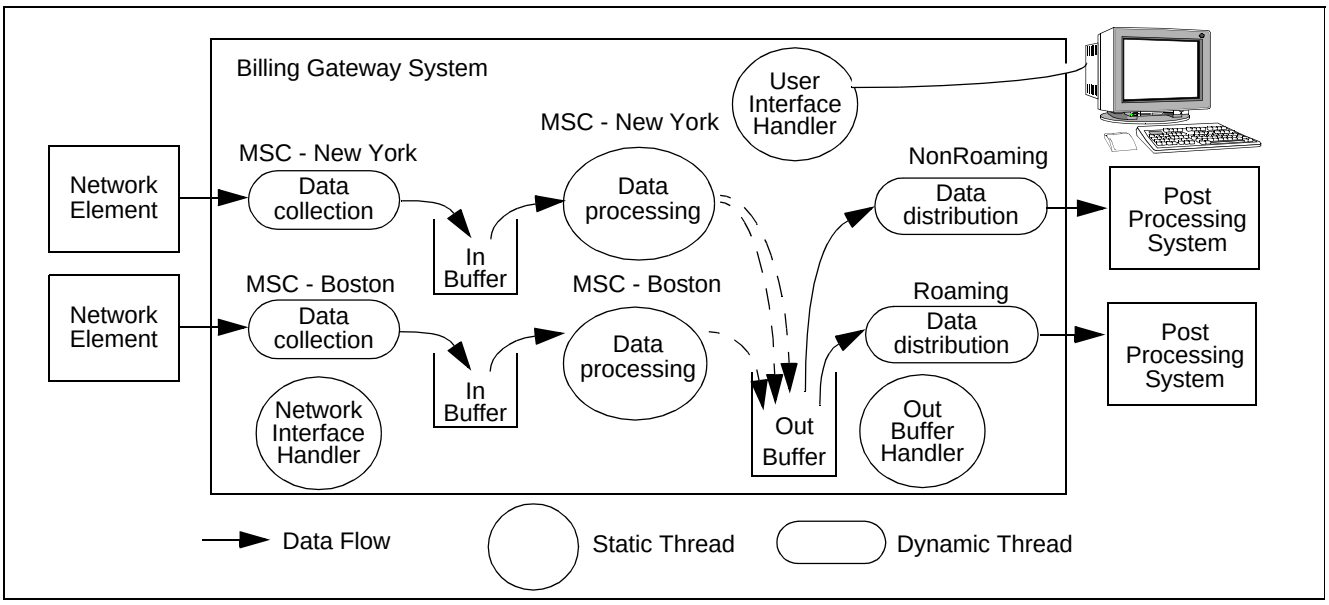

Figure 2: The thread structure of Billing Gateway.

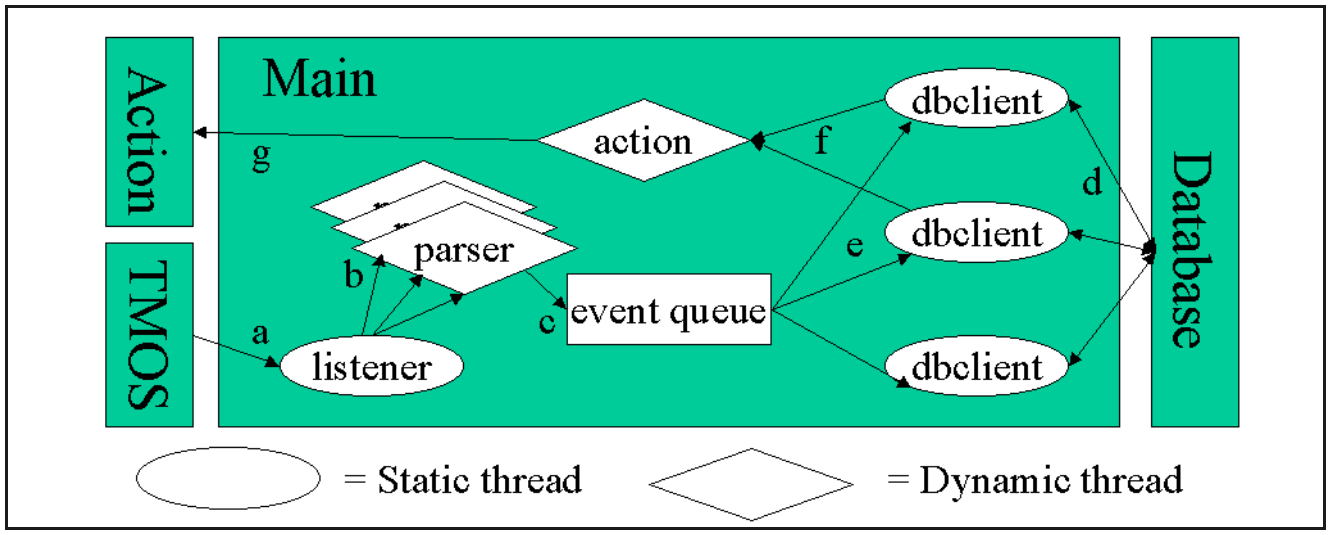

Figure 3: The architecture of FCC.

enough for call termination. FCC allows the cellular operator to decide certain criteria that have to be fulfilled before a call is terminated, e.g. the number of indications that has to be detected within a certain period of time before any action is taken. The events are continuously stored in files in the cellular network. With certain time intervals or when the files contain a certain amount of events, these files are sent to the FCC.

The FCC consists of four major software modules (see Figure 3). The TMOS module is an Ericsson propriety platform and handles the interaction with the switching network. The collected events are passed on to the main module of FCC. In the Main module the event files are parsed and divided into separate events. The events are then stored and checked against the pre-defined rules using the database. If an event triggers a rule, the action module is notified. This module is responsible for executing the action associated with a rule, e.g. to send terminating messages to the switching network.

A central part of FCC is data storage and data processing. A commercial RDBMS (Sybase) was used in the implementation. In order to improve performance, FCC has implemented parallel execution, using Solaris threads.

The processing within the Main module is based on threads. Figure 3 shows how the threads are communicating. A listener thread receives the event file (3a) and creates a parser thread (3b). After it has created the parser thread, the lister thread is ready to receive the next file. The parser threads extract the events from the file and insert the separate events into an event queue (3c), where they are waiting for further processing. When all events in a file have been extracted, the parser thread terminates. The number of simultaneous parser threads is dynamic. The parser in FCC is designed in a way which makes it flexible. It is very important for FFC to quickly support new types of events when a new network release often introduce new, or change the format of old, events. However, the flexible design results in frequent use of dynamic memory.

The Main module has a configurable number of connections toward the database server (3d). Each connection is handled by a dbclient thread. A dbclient thread handles one event at the time by popping the first event of the event queue (3e) and then processing it. The interaction with the database is made through SQL commands via a C-API provided by Sybase. Each SQL command is constructed before it is send to the database module. Since the final size of a SQL command is unknown dynamic memory has to be used for its construction. The dbclient thread is also responsible to initiate actions caused by the event (3f,3g) before it processes the next event.

2.2.2 Experiences

One important conclusion from the industrial Ericsson FCC project is that the present and future performance requirements of this kind of telecommunication systems can only be meet with multiprocessor systems.

Dynamic memory management was a performance bottleneck also for FCC. There are two reasons why FCC has an intensive use of dynamic memory: the object oriented design of the parser and the use of a string library for dynamic construction of database requests, respectively. By optimizing dynamic memory management the speedup was increased significantly.

We used two different approaches for optimizing the dynamic memory handling in FCC. One approach was to replace the standard memory handler with a parallel memory handler called ptmalloc. The other approach was to split the client into two or three Unix processes (Unix processes have different memory images). The performance characteristics of these two approaches were very similar.

2.3 Generic Database Systems

2.3.1 System description

In some database applications there is a need for very high flexibility. One may want to change the number of properties (columns) associated with an object (record) at run-time or one may want to do very flexible searches, e.g. give me all objects for which some property has the value Red. These type of functionality cannot easily be supported with a regular (standard) database design. Having a high degree of flexibility, i.e. making

the system configurable, will in many cases decrease the maintainability cost, since a lot of changes that would require redesign in a regular database system can be made by the user in a configurable system. We have investigated one such flexible application - the Promis database system [6].

Promis is based on a standard RDBMS (Oracle). The purpose of the system is to maintain information about telecommunication products and services. The development of the system was a joint effort between Swedish Telia, Dutch PTT and Swiss PTT. In order to obtain the desired flexibility a meta-data approach was used (this is very similar to the Reflection architectural design pattern <ref>).

In Promis there are four tables for meta data. The actual data is stored in two tables independent of the record format, i.e. no matter how many record types we have there are only two tables for data. The information about the record format is stored in the meta data tables.

2.3.2 Experiences

Measurements on a production database showed that the Promis approach increased the average access time with approximately a factor of 20 compared to a regular database implementation, i.e. there was a slow of approximately 20. Consequently, the price for the flexibility was dramatic (and unacceptable) reduction of performance.

However, by identifying the most common cases (e.g. reading an object was 450 times more common than creating an object), we were able to optimize the implementation in such a way that the performance degradation compared to a regular database implementation was limited to factor of two (roughly). The optimization introduced some controlled redundancy in the database. This resulted in somewhat longer times for creating an object. However, the time for reading an object was reduced significantly. Due to the optimizations the average performance could be improved from a slow down of approximately 20 to a slow down of 2-3.

2.4 Haemo dialysis Machines

2.4.1 System description

The aim of a dialysis system is to remove water and certain natural waste products from the patient's blood. Patients that have kidney problems and consequently produce little or no urine use this type of system.

The 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.

An overview of a dialysis system is presented in Figure 4. 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, 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 (rectangle with a curl).

On the right side of Figure 4, 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. The extra corporal circuit uses a number of sensors, e.g. for identifying bubbles, and actuators.

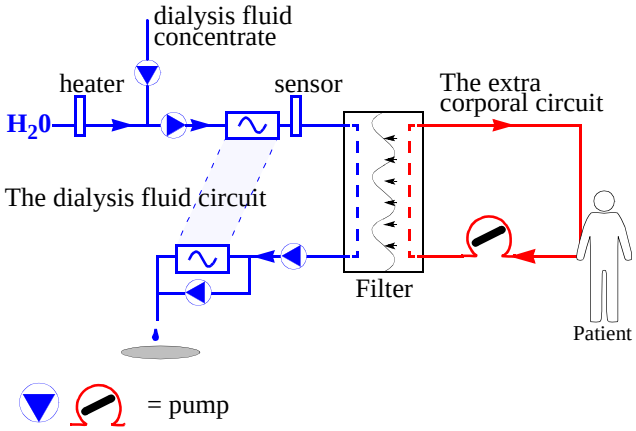

Figure 4: Schematic view of Haemo Dialysis Machine.

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 we anticipated several additional changes to the software, hardware and mechanical parts of the system that will be necessary in response to developments in medical research.

2.4.2 Experiences

We learned a number of lessons during this project:

- **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".
- **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 com-

plete analysis. And finally the techniques are generally intended for the later design phases and often require detailed information not yet available during architecture design.

- **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.
- **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. 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.

2.5 Measurement Systems

2.5.1 System description

Measurement systems [3] are a class of systems used to measure the relevant values of a process or product. A measurement system is used for quality control on produced products that then can be used to separate acceptable from unacceptable products or to categorize the products in quality categories. One example of a measurement system is a system for sorting out bad pieces of wood in a floor manufacturing process.

Although a measurement system contains considerable amounts of software, a substantial part of these systems is hardware since it is connected to the real-world through sensors and actuators. These developments in the domain of sensors and actuators changes measurement systems from small, single processor systems that are developed close to the hardware to distributed computing systems since modern sensors and actuators often contain their own processors. However, although the increased functionality of the sensors and actuators reduces the complexity of constructing measurement systems, the increased demands on these systems and the resulting increase in size make that the construction of measurement systems is a complex activity. We have been involved in the design of an object-oriented framework for measurement systems. In Figure 5, the architecture of a simple measurement system is shown. It consists of five entities that communicate with each other to achieve the required functionality.

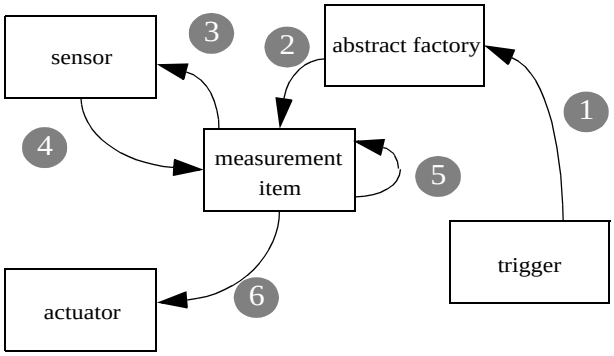

Figure 5: Architecture of a simple measurement system

1. The **trigger** triggers the **abstract factory** when a physical item enters the system.
2. The **abstract factory** creates a representation of the physical object in the software, i.e. the **measurement item**.
3. The **measurement item** requests the **sensor** to measure the physical object.
4. The **sensor** sends back the result to the **measurement item** which stores the results.
5. After collecting the required data, the **measurement item** compares the measured values with the ideal values.
6. The **measurement item** sends a message to the actuator requesting the actuation appropriate for the measured data.

Measurement systems have to fulfil a number of quality requirements:

- **Intuitive**: As any type of system the designed framework should be based on concepts that have a direct correspondence in the application domain. The way these concepts are used and combined should be logically consistent with the view of a domain expert.
- **Reusable**: The framework should provide reusable components for the construction of measurement systems. This requires a delicate balance between generality and speciality. It also means that the components and decomposition dimensions have to be chosen such that relatively general components from different dimensions can be composed to form specific components that can be used in real system with minimal extensions.
- **Flexible**: The requirements on the flexibility of measurement systems are higher than average. As described, the actual composition of the system from its components, the analysis process and the reaction by the system based on the analysis results needs to be easily adaptable both during application development as well as during system operation.
- **Real-time constraints**: Although most traditional system construction approaches deal with real-time constraints by running tests on the system and measuring the system responses, we already discussed the advantages of expressing real-time constraints directly as part of the system. The difficulty with both real-time and concurrency is the platform dependence of the implementation of these techniques.

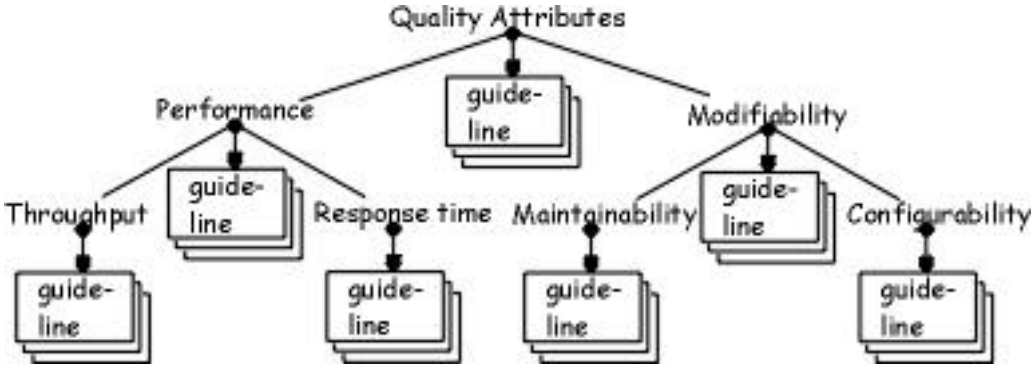

Figure 6: Taxonomy of quality requirements

2.5.2 Experiences

The following experiences were collected from the measurement systems project:

- **Quality attributes harder than functionality**: Although designing the system to include the correct functionality is not trivial, much of the architecture design effort was directed to achieving the quality attributes. In [1], we discuss the development of a measurement system into an object-oriented framework for measurement systems. Most of the transformations of the architecture were taken to improve quality attributes.
- **Boundary**: Designing a framework for a domain may seem like a well-defined task, but in practice we found that drawing the boundary for the framework is extremely difficult. In our early discussions, we constantly were extending the framework since we all agreed that the framework would be even better when it would include yet another feature. We soon realised that the size of the framework would be unmanageable, but prioritizing features is difficult.

3 Balancing Quality Attributes

The experiences from the applications in Section 2, and some other systems that we have worked with, has given us valuable insights regarding the implications certain quality attributes have on the software architecture. Some quality attributes favor the same (or at least similar) architectural solutions, whereas other quality requirements favor architectural solutions that are conflicting or at least not easy to combine.

In Section 3.1 we present a simple categorization of quality attributes. In Section 3.2 the software architecture design method that we have developed is briefly introduced. Although most of the reasoning presented in this paper is of qualitative nature, the presented design method provides, at least to some extent, means to perform quantitative assessments of the quality attributes of a software architecture. The architectural implications caused by quality requirements are discussed in Section 3.3. Section 3.4 and 3.5 discuss the relation between architecture, implementation and performance and maintainability, respectively. Based on this, Section 3.6 presents a set of design guidelines. These guidelines support the selection of appropriate architecture transformations.

3.1 Categorizing Quality Attributes

There are a number of possibilities for categorizing quality attributes [10]. In Figure 6 we suggest a categorization of a subset of quality attributes. The reason for selecting this particular categorization is that our experience enables us to define how and where the conflicts occur between these attributes.

First we divide the attributes into two major categories, viz. performance and modifiability that is concerned with the effort needed to redesign the system. Further, we separate the requirement of high throughput from the requirement of response time. Finally, modifiability is decomposed into two quality attributes, i.e. maintainability and configurability. As part of our future work, we intend to incorporate other attributes, e.g. reusability, usability and availability.

There are at least two different ways of meeting the need for future changes in the system. If it is required that an (advanced) user should be able to do a certain modification, we have a configurability requirement, i.e. although the system's behavior is modified, the executable program is not. For instance, in the BGw system the input formats and the way the input data is processed can be configured by the user, by drawing graphs like the ones in Figure 1. Another example of run time configurability is the Promis database system, which allows the user to define any record formats.

If we have a requirement that a certain modification should be inexpensive to perform for a designer, we refer to this as a maintainability requirement, i.e. in this case we will redesign the system. Consequently, configurability and maintainability both address the need for modifying the system to meet certain future changes. However, in the case of configurability the program should not be redesigned, whereas the maintainability requirement states that the redesign of the program should be cost effective.

3.2 Design Method Outline

Previously, we have developed a method for designing the architecture [1][4], presented graphically in Figure 7. The 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 [17]. 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 method uses four evaluation approaches. *Scenario-based evaluation* is techniques where the software qualities are expressed as typical or likely scenarios. Using *simulation* the architecture is modeled in a simulation environment and its behavior is used to predict the software quality attribute. *Mathematical modeling* (including metrics & statistics) is a technique where product and process data are used to make predictions about the potential qualities of a resulting product or task. *Experience-based reasoning* employs experienced designers that often intuitively identifies designs that are not addressing certain quality requirements adequately.

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. Four categories of transformations are identified. *Applying an architecture style* result in changes to the overall structure. *Applying an architecture pattern* adds certain behavioral rules to the architecture, e.g. Periodic Objects [11]. *Applying design patterns* impacts only a few elements of the architecture. *Converting quality requirement to functionality*, e.g. handling robustness by introducing exception handling.

These transformations primarily 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.

Our architecture design method has similarities to the Architecture Tradeoff Analysis Method (ATAM) proposed by Kazman et al [13]. Although several differences exist, one major difference is that we include concrete guidelines on how to transform the architecture in order to meet certain quality requirements, whereas ATAM concentrates on identifying so called tradeoff points, e.g. design decisions that will affect a number of quality attributes. However, in ATAM there are no guidelines on how to modify the software architecture.

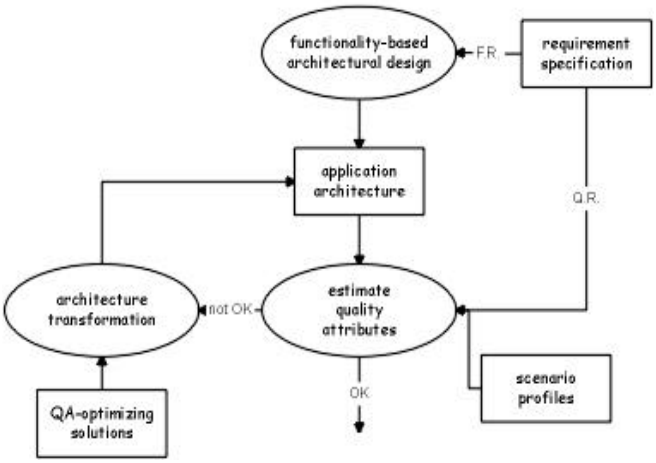

Figure 7: Outline of the architectural design method

3.3 Architectures for Meeting Quality Attributes

Our experience shows that there is generally a large conflict between configurability and performance (throughput as well as response time). However, there does not have to be any major conflicts between maintainability and performance. Due to the commonly used implementation techniques there can, however, be a conflict between maintainability and performance.

Run-time configurability is typically obtained by some interpreter like structure that operates on meta data. One typical example is the Reflection pattern [5]. There are a number of reasons why these kinds of software architectures cause performance problems. First, it is well known that interpretation is more costly than executing the same functionality directly on the target hardware. However, the interpreter structure will also generate heavy use of dynamic memory. The reason for this is that the structures on which the interpreter operates is, at least partly, unknown at compile time, i.e. in order to configure the system dynamically at run-time we need to use dynamic data structures. The object oriented design paradigm, which is prevailing today, tends to intensify the use of dynamic data structures even more [8]. Frequent use of dynamic data structures can easily generate serious performance problems when using multiprocessor platforms. The reason for this is that the dynamic memory handling often becomes a performance bottleneck [8][9].

Experience from the FCC project and some other projects that we have studied shows that the distinction between throughput and response time is important when selecting a suitable architecture. Our experience, particularly from FCC, shows that in order to obtain good throughput we need to process large chunks of data at the same time. This will, however, lead to poor response times. Based on this observation and experiences from various architectural styles, particularly the black board and the pipes and filters architecture styles, we conclude that pipes and filters tend to be more “fair” than the blackboard architecture in the sense that the variations in response times are smaller. However, the black board architecture often improves throughput since this architecture opens up the possibility to process chunks of data that do not arrive at system at (exactly) the same time, e.g. we may check some rule in FCC (and similar systems) on a group of data at the same time.

3.4 Architecture, Implementation and Performance

It is well known that the implementation techniques have a strong impact on performance, e.g. the performance difference between different sorting algorithms can be substantial [15]. The relation between performance and the software architecture is not that obvious. However, our experience shows that some architectural decisions have performance implications that cannot be fully compensated by clever implementation techniques; in particular those decisions that are caused by configurability requirements.

As discussed previously, experiences from the BGw and FCC projects show that a consequence of using object-oriented methods and a configurable design is extremely heavy use of

dynamic memory. This resulted in very poor performance; in particular when using a multiprocessor system. This performance problem can be reduced by optimizing the routines for allocating and deallocating dynamic memory, e.g. by replacing the standard memory handler with a memory handler which was optimized for shared memory multiprocessors.

In the Promis database project we were able to reduce the performance problems due to using meta data. This was done by using controlled redundancy and optimizing the common cases. However, the performance was still less than half of a standard implementation (i.e. an implementation that does not use meta data), even after these optimizations.

Consequently, some of the performance problems due to configurability requirements, can be solved at the implementation level. However, some performance problems remain even when using highly optimized implementations. This means that the performance requirements should not be completely postpone to the implementation phase, at least not if performance is a major issue. However, the implementation techniques will always have a large impact on performance.

3.5 Architecture, Implementation and Modifiability

Similar to performance, also modifiability is influenced considerably by the implementation. Nevertheless, the top-level decomposition of a system, i.e. its architecture, plays a principal role in achieving modifiable systems. The primary principle for the architecture should be that new requirements should affect as few components as possible. Thus, the more likely that a particular requirement will be added in the future, the more easy it should be to incorporate this requirement.

As discussed in Section 3.1, maintainability and configurability are both located under the modifiability heading. The advantage of exploiting configurability is that the cost of incorporating new requirements is very small, and it can be done by an (advanced) user. Two important disadvantages exist, though. First, the new requirement needs to be implemented already during the initial product development, i.e. one is incorporating potential requirements that may never be demanded. Even though this is acceptable if the cost of incorporating the requirement during the initial development is small compared to adding it during maintenance and the likelihood of the requirement is high, there is a certain risk taking involved. Second, run-time configurability often leads to decreased performance.

3.6 Design Guidelines

Our experiences this far can be formulated as a set of guidelines. These guidelines can be categorized according to the quality requirements. In Figure 6, the quality requirements studied in this paper are presented in a taxonomy. With each quality attribute in the hierarchy, one or more guidelines may be associated.

These guidelines are used during the architecture transformation phase of our architecture design method that was discussed in Section 3.2 (see Figure 7). The phase is entered after architecture assessment since the analyzed architecture did not fulfil its quality requirements. We suggest that the quality requirements are categorized and formulated according to the categories shown in Figure 6.

Architecture transformations can be performed using several different architectural styles, architectural patterns and design patterns. Based on the studied cases described in Section 2, we have formulated a number of guidelines concerned with the appropriate selection of transformations with respect to the quality requirements.

The guidelines are graphically shown in Figure 8 (N.B. Figure 8 corresponds to the architecture transformation phase in Figure 7). If the estimation phase in our method indicates that a quality requirement is unfulfilled we enter the architecture transformation phase. The guidelines are associated with quality attributes, and indicate suitable transformations that positively affect the unfulfilled quality attribute.

GL1. If incorporating new requirements in fielded systems is a competitive advantage convert modifiability and maintainability requirements to configurability requirements.

Rationale: Configurability allows for easy, e.g. run-time, incorporation of new requirements. This can be achieved by changing system settings or by distributing a binary component that, after installation, is automatically incorporated into the system. Since this can be performed at the user site and, generally, by the user against minimal effort, this provides an advantage over competing products, especially in 'feature-race' products such as mobile phones. In the dialysis system case, one would like to simplify software evolution of fielded system through remote access which is typically achieved through reconfiguring the system and adding new components. However, performance and safety were affected negatively by this and the requirement was not incorporated in the first version of the architecture.

GL2. If the performance requirements are not fulfilled, convert modifiability requirements to maintainability requirements. In addition, renegotiate configurability requirements to convert these to maintainability requirements.

Rationale: Configurability solutions, such as the interpreter architectural style, generally lead to considerable performance overhead whereas maintainability solutions do not. Implementing modifiability requirements as maintainability requirements will, consequently, lead to better performance. The dialysis system case discussed above gives one example.

GL3. If throughput is the primary performance requirement, then use the blackboard architectural style.

Rationale: Experiences from FCC indicate that throughput of the blackboard style in that, but also other systems, is better than the pipes and filters architectural style.

GL4. If response time is the primary performance requirement, then use the pipes and filters architectural style.

Rationale: Experiences from FCC indicate that the response time of individual events is higher and has less variation when using the pipes and filters style compared to the blackboard architectural style.

GL5. If the configurability requirement is not fulfilled, use the interpreter architectural style.

Rationale: Configurability is concerned with being able to adapt the system to future situations that cannot be specified at this point. A solution that deals with this is to convert the base-level functionality into a meta-model of the domain covering that and related functionality. The meta-model can be interpreted using an interpreter. Since the meta-model can be changed and extended easily, the system becomes much more configurable. However, there is generally a considerable performance penalty for using this architectural style. Typical examples of this can be found in the BGw and FCC systems where a parser was used as an interpreter to describe the system behavior.

GL6. If the maintainability requirement is not fulfilled (and cannot be converted into a configurability requirement due to performance requirements), employ abstract factory [7] and strategy design patterns to factor out behavior that is likely to be affected by future requirements.

Rationale: In the measurement and dialysis systems, maintainability was improved by separating stable functionality from functionality that was judged to be likely to be changed in the future. Factoring out code that is likely to be changed increases maintainability since rather than editing existing code, it often suffices to define new subclasses and instantiating those instead of the previous classes.

GL7. If the performance requirement is not fulfilled, convert dynamic object instantiation to static object allocation, where possible.

Rationale: As described earlier in the almost all systems described in Section 2, considerable amounts of dynamic objects are created and destroyed. Our studies, e.g. [8], show that object memory management requires substantial computational resources. Avoiding dynamic object creation and removal especially in frequently iterated execution paths will lead to substantial performance improvements.

GL8. If the performance requirement is not fulfilled, try to increase the granularity of the object definitions, e.g. by avoiding defining objects in terms of numerous sub-objects. Moreover, try to declare sub-objects by value and not by reference in composite aggregations [16].

Rationale: As described earlier in the almost all systems described in Section 2, considerable amounts of dynamic objects are created and destroyed. When objects are defined in terms of sub-objects that are declared by reference, i.e. created in the constructor of the aggregating object, this will lead to a large number of dynamic memory allocations for each creation of an aggregating object. N.B. there is a potential conflict between this guideline and GL 6, since GL 6 often results in a division of objects into a larger number of sub-objects. Consequently, the tradeoff between GL6 and GL 8 depends on the degree of unfulfillment of the maintainability and performance requirements respectively.

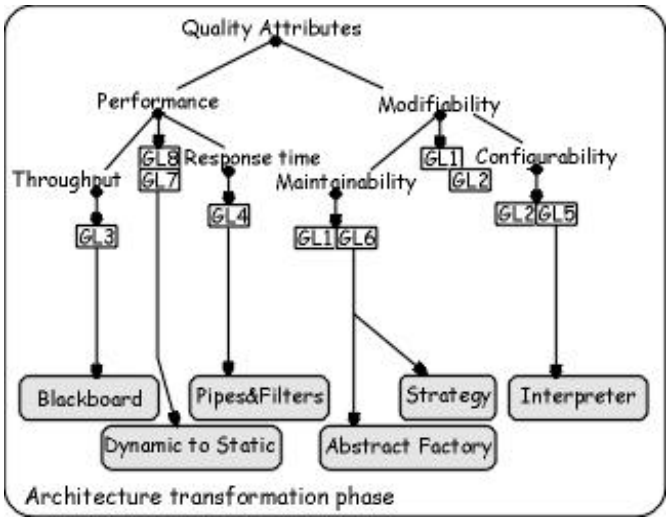

Figure 8: Guideline-based architecture transformation

4 Conclusions

We have presented experiences from five industrial applications. The experience from each individual project shows that modifiability as well as performance are important issues. Performance evaluations of the BGw, FCC and Promis projects show that there is a conflict between introducing concepts that will increase the modifiability, e.g. the new language in BGw and the meta data model in Promis, and obtaining high performance. The most serious conflicts typically occur between the configurability requirement and the performance requirement.

Based on the joint experience from all five projects, we have formulated eight design guidelines and defined a small taxonomy for some important quality attributes. Each such guideline is associated with a quality requirement in the taxonomy. We have also incorporated the guidelines in an existing design method, thus making it clear how and when the guidelines should be used.

Our experiences show that performance and to some extent modifiability are affected by the implementation techniques. Some of our performance problems, e.g. performance problems caused by excessive dynamic memory management in BGw and FCC, could, to a large extent, be compensated by appropriate implementation techniques. However, some architectural decisions have (negative) performance implications that cannot be fully compensated in the implementation phase, e.g. the meta data model in Promis. It is, therefore, important to avoid, or at least not unknowingly adopt, some architectural styles if performance is a major issue. The modifiability of a system is to an even larger extent than performance decided by the top level architectural decomposition. Consequently, it is important to obtain a reasonable balance between the different quality requirements in the top level architectural design.

Hitherto we have only considered performance and modifiability. However, in the future we plan to extent our guidelines and method to other quality requirements, e.g. reusability and availability.

References

[1] PO Bengtsson, J. Bosch, 'Scenario-based Architecture Reengineering,' Proceedings of the 5th International Conference on Software Reuse, pp. 308-317, June 1998.

[2] PO Bentsson, J. Bosch, 'Haemo Dialysis Software Architecture Design Experiences,' Proceedings of the 21st International Conference on Software Engineering, May 1999.

[3] J. Bosch, 'Design of an Object-Oriented Framework for Measurement Systems,' Object-Oriented Application Frameworks, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, 1998.

[4] J. Bosch, P. Molin, 'Software Architecture Design: Evaluation and Transformation,' in Proceedings of the 1999 IEEE Conference on Engineering of Computer Based Systems, December 1998.

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

[6] W. Diestelkamp and L. Lundberg, 'Promis, a Generic Product Inforamtion database System,' in Proceedings of the ISCA 14th International Conference on Computers and their Applications, Cancun Mexico, April 1999.

[7] E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, 'Design Patterns - Elements of Reusable Object-Oriented Software,' Addison-Wesley, 1994.

[8] D.Häggander and L. Lundberg, 'Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor,' in Proceedings of the 27th International Conference on Parallel Processing, August, Minneapolis 1998.

[9] D.Häggander and L. Lundberg, 'Multiprocessor Performance Evaluation of a telecommunication Fraud Detection Application,' Submitted for publication.

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

[11] P. Molin, L. Ohlsson, 'Points & Deviations - A pattern language for fire alarm systems,' in Pattern Languages of Program Design 3, Addison-Wesley, 1997.

[12] R. Kazman, G. Abowd, L. Bass, and P. Clements, 'Scenario-Based Analysis of Software Architecture,' IEEE Software, November 1996, pp.47-55.

[13] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, 'The Architecture Tradeoff Analysis Method,' Tech. report, CMU/SEI-98-TR-008.

[14] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzales Harbour, 'A Practitioner's Handbook for Real-Time Analysis,' Kluwer Academic, 1993.

[15] D. E. Knuth, 'The Art of Computer Programming, vol. 3: Searching and Sorting,' Addison-Wesley, 1973.

[16] J. Rumbaugh, I. Jacobson, and G. Booch, 'The Unified Modeling Language Reference Manual,' Addison-Wesley, 1999.

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

[18] C. Smith and L. Williams, 'Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives,' IEEE Transactions on Software Engineering, 19(7), pp. 720-741.