Design and Use of Industrial Software Architectures

Jan Bosch
University of Karlskrona/Ronneby
Jan.Bosch@ipd.hk-r.se
www.ipd.hk-r.se/~bosch

Excerpt from a working/draft version. Chapters 1 through 6 are included. I welcome comments and questions concerning the content of the chapters.

Design and Use of Industrial Software Architectures

CHAPTER 1

Design of Software Architectures[1]

1.<This chapter gives an introduction to the notion of architectural design and an overview of the method presented in this part of the book>

In our experience, the most complex activity during application development is the transformation of a requirement specification into an architecture for the system. The later phases also are challenging activities, but, for instance, detailed design and implementation, are better understood and more methodological and technological support is available to the software engineer. The process of architectural design is considerably less formalized and little methodological support is available. In industry, the design of a software architecture is often more like art or intuitive craftsmanship than objective engineering.

Although software systems have had architectures since the early days of computers, it has only during recent years recognized as more important to explicitly specify and analyze software architectures. One important factor is this is the fact that especially quality requirements are heavily influenced by the architecture of the system. Architectural design is a typical multiple objective design activity where the software engineer has to balance the various requirements during architectural design.

In this part of the book, we present an architecture design method that provides support for an objective, rational design process balancing and optimizing, especially, the quality requirements. The method iteratively assesses the degree up to which the architecture supports each quality requirement and improves the architecture using transformations until all quality requirements are fulfilled. The presented method complements traditional design methods in that it focuses on quality attributes whereas traditionally the functionality is prioritized.

The method consists of three phases, i.e. functionality-based architectural design, architecture evaluation and architecture transformation. In this chapter, we give a brief overview over the method, whereas chapters 3, 4 and 5 describe the method phases in detail. The remainder of this chapter is organized as follows. In the next section, we describe a common terminology for requirements; in particular quality requirements, and describe the notion of profiles and their use in the specification of quality requirements. Section 2 provides a rationale for software architecture design and handling quality requirements explicitly and presents a brief overview over the three main phases in the method. The subsequent sections describe these phases and the chapter is concluded in section 6.

1. Requirements

Requirement engineering has been studied extensively [refs] and it is not the aim of this book to address the process of identifying and specifying requirements. Instead, the requirement specification is used as an input for architectural design. There is, however, one aspect to requirement engineering that is relevant for software architecture design: the specification of quality requirements. Our experience is that most requirement specifications either do not specify the quality attributes at all, or specify them very unclear and, consequently, not measurable. In the remainder of this section, we first establish a terminology for the various requirement concepts. Subsequently, in section 1.2, we discuss the specification of quality requirements and the use of, so-called, profiles for this purpose.

1.1 Terminology

In this section, we define the terminology related to requirements that we will use throughout the remainder of this book.

System requirements are defined as the top-level requirement set consisting of software, hardware and mechanical requirements. In this book, we focus on fulfilling the software requirements and ignore other types of requirements. Software requirements can be defined as consisting of *functional requirements* and *quality requirements* (also referred to as system properties). The functional requirements are related to the domain-related functionality of the application. Typically, a functional requirement is implemented by a subsystem or a group of components, i.e.

functional requirements are traceable in the architecture. Quality requirements can be categorized in *development* and *operational* quality requirements. Development quality requirements are qualities of the system that are relevant from a software engineering perspective, e.g. maintainability, reusability, flexibility and demonstrability. Operational quality requirements are qualities of the system in operation, e.g. performance, reliability, robustness and fault-tolerance. Different from functional requirements, quality requirements can generally not be pinpointed to a particular part of the application but are a property of the application as a whole.

1.2 Quality Attributes and Profiles

As mentioned in the introduction to this section, our experience is that quality requirements such as performance and maintainability are generally specified rather weakly in industrial requirement specifications. In some of our architecture design projects with industry, the initial requirement specification contained statements such as “The maintainability of the system should be as good as possible” and “The performance should be satisfactory for an average user”. Such subjective statements, although well intended, are totally useless for the evaluation of software architectures. For example, [Gilb 88] discusses the quantitative specification of quality requirements and presents useful examples.

Some research communities, e.g. performance [Smith 90], real-time [Liu & Ha 95] and reliability [Neufelder 93], have spend considerable effort on the specification of their particular quality requirement, but the software development industry has not adopted these techniques. One of the reasons is that these techniques tend to be rather elaborate and require considerable effort to complete. Since, within an engineering discipline, each activity is a balance of investment and return, these techniques may not have provided sufficient return-on-investment, from the perspective of industrial software engineers.

However, when one intends to treat the architecture of a software system that one is working on explicitly in order be able to early predict the quality attributes of the system, it is also necessary to specify quality requirements in sufficient detail. One common characteristic for quality requirements is that stating a required level without an associated context is meaningless. For instance, the statement “Performance = 200” or “Maintainability = 0.8” is virtually meaningless.

However, one common denominator of most specification techniques is that some form of *profile* is used as part of the specification. A profile is a set of scenarios, generally with some relative importance associated with each scenario. The profile used most often in object-oriented software development is the *usage profile*, i.e. a set of usage scenarios that describe typical uses for the system. The usage profile can be used as the basis for specifying a number of, primarily operational, quality requirements, such as performance and reliability. However, for other quality attributes, other profiles are used. For example, for specifying safety we have used *hazard scenarios* and for maintainability we have used *change scenarios*.

Based on our experience, we believe that it is necessary to specify the relevant profiles for the software quality requirements that are to be considered explicitly in the architecture design process. Using the profile, one can make a more precise specification of the requirement for a quality attribute. For example, the required performance of the system can be specified using the usage profile by specifying the relative frequency and the total number of scenarios per time unit.

In chapter 4, the specification and usage of profiles is discussed in more detail and several examples are presented.

2. Architecture Design Method Overview

In software industry, our experience is that quality requirements are generally dealt with by a rather informal process during architecture design. Conventional object-oriented design methods, e.g. [Booch 94, Jacobsen et al. 92, Rumbaugh et al. 91], tend to focus on achieving the required system functionality, but do not spend much attention on quality requirements. Implicitly, these methods assume that using an object-oriented modeling approach will automatically lead to reusable and flexible systems, thus one could state that the maintainability and reusability requirements are incorporated up to some extent. However, only these quality attributes are considered and only implicitly.

Software engineers in industry, lacking support for the early evaluation of quality requirements, develop systems using the available design methods and measure the quality attributes of the system once it has been built. There are at least two problems with this approach: first, not all quality attributes can be measured before the system is put in operation. For instance, to measure maintainability of a system generally requires up to several years before one can make generalizable statements about the required effort for implementing new requirements. Second, even if one is able to measure a quality attribute once the system has been built, e.g. performance, the effort required to rework the system if it does not fulfil the requirements is generally rather costly. Often, the cost of reworking a system to incorporate the quality requirements after it has been built is one or several magnitudes of order higher than performing the evaluation and transformation of the system design early in the development. The focus on software architecture design currently experienced in the software engineering community can be explained by the above arguments, i.e. explicit evaluation of the architecture of software systems with respect to the quality requirements will minimize the risk of building a system that fails to meet its quality requirements and consequently decrease the cost of system development.

At this point it is relevant to notice that many exceptions to the observation exist. Especially companies that have been working in a particular domain for several years, e.g. embedded systems, often have remarkable success rates in achieving the quality requirements on their systems. The main explanation for this is that software architects at those companies often have a considerable understanding of the possibilities and obstacles of particular architectural designs in their domain. Since the system architects often are experienced in building systems in the domain, experience helps them to minimize system redesign. However, although this is one of the most successful means for companies to build up a competitive advantage, one can identify at least three disadvantages. First, the design expertise is generally *tacit knowledge*. Thus when the experienced software architect leaves the company, so does the design expertise. In addition, when the organization enters a new domain, the experience base has to be created again through trial and error. Second, since the expertise used as a basis for the design decisions is implicit and design is, up to some extent, performed based on ‘gut feeling’, it is virtually impossible to perform critical evaluations. This is disadvantageous for the company since it may result in design decisions based on incorrect assumptions by the software architect. In addition, the software engineering community does not benefit from the generated design expertise since it is not objectified and evaluated. Finally, it complicates the education of software architects at universities since there is no body of knowledge available that can be taught. Consequently, the students have to obtain this by hands-on design experience, hopefully under the guidance of a mentor.

The observations discussed above are by no means novel. Computer science and software engineering research have spent considerable effort on several of the quality requirements. This has lead to the formation of several ‘quality attribute-based’ research communities that focus on one quality attribute and try to maximize the systems they build with respect to their particular quality attribute. Examples of such communities are those working on real-time systems, high-performance systems, reusable systems and, more recently, maintainable systems. Several of these communities have proposed their own design methods and evaluation techniques.

For instance, in real-time systems [Liu & Ha 95], in high-performance computing [Smith 90] and in reusable systems [ref-ICSR?].

Each of the research communities address relevant and important aspects of software systems and to achieve progress in hard problems, one needs to focus and ignore issues outside the focus. However, there is a major problem in this development and that is that, since each research community has a tendency to study a single system quality requirement, it consequently does not address the composition of its solutions with the solutions proposed by research communities studying different quality requirements. Concrete industrial software systems never have only a single quality requirement to fulfil, but generally have to achieve multiple of these requirements. For instance, most real-time systems should be reusable and maintainable to achieve cost-effective operation and usage, whereas fault-tolerant systems also need to fulfil other requirements such as timeliness and maintainability. No pure real-time, fault-tolerant, high-performance or reusable computing systems exist, even though most research literature within the respective research communities tends to present systems as being such archetypical entities. All realistic, practical computing systems have to fulfil multiple quality requirements.

One may wonder why the above is a problem: if the quality-attribute oriented research communities develop their solutions and guidelines, why not just compose the solutions and guidelines in the system currently under design. The answer is that the solutions and guidelines provided for fulfilling the quality requirements tend to be conflicting, i.e. using a solution for improving one quality attribute will generally affect other quality attributes negatively. For example, reusability and performance are generally considered to be contradicting, as well as fault-tolerance and real-time computing. As a consequence, when the customer for a system has extreme performance requirements, it requires that this customer accepts that the maintainability of the system will be very low, i.e. it is very costly to incorporate new requirements. This observation has been implicitly accepted by the software engineering community, but very few examples of approaches to explicitly handling the conflicts in quality requirements exist. Consequently, lacking a supporting method, software engineers in industry design system architectures in an ad-hoc, intuitive, experience-based manner, with the consequent risk of unfulfilled system properties.

2.1 Method

As we identified in the previous section, there is a lack of software design methods that explicitly address and balance the quality attributes of a system. In this part of the book, we present our approach to addressing the identified problems, including an architecture design method that incorporates explicit evaluation of and design for quality requirements. The developed approach is part of our research efforts in the domain of software architecture. It is important to note that it is explicitly not our intention to present the final architecture design method. Instead, we report on our experiences with software architecture design, the generalizations we made based on our experiences and the validation of the generalizations that we performed. The architecture design method presented in this part of the book is a generalization of the design of three software architectures in the embedded systems domain, i.e. fire-alarm systems, measurement systems and dialysis systems. Members of our research group have been involved in the design of these systems, either while working in industry or as part of a joint research project between our research group and one or more industrial partners. Since these systems will be used extensively as examples, we present the systems and their domains in more detail in chapter 2.

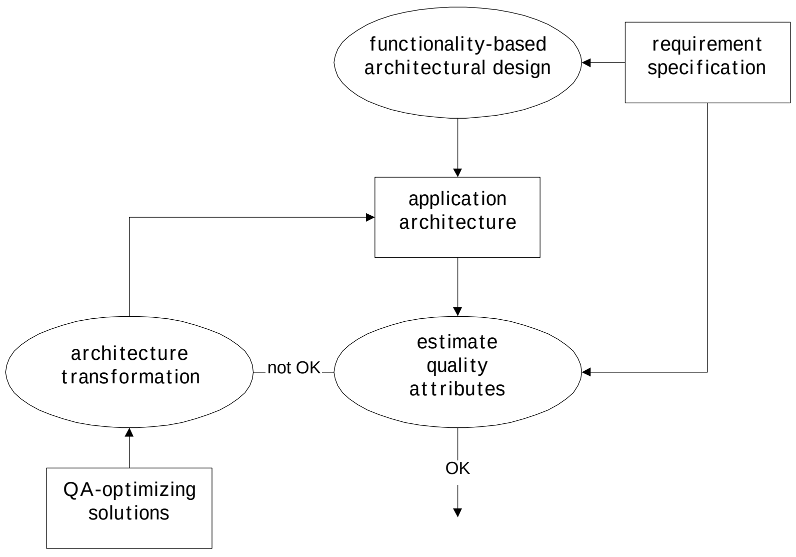

functionality-based architectural design
requirement specification
application architecture
architecture transformation
not OK
estimate quality attributes
OK
QA-optimizing solutions

FIGURE 1. ***Outline of the architectural design method***

The architecture design process can be viewed as a function taking a requirement specification that is taken as an input to the method and an architectural design that is generated as output. However, this function is not an automated process and considerably effort and creativity from the involved software architects is required. The software architecture design is used for the subsequent phases, i.e. detailed design, implementation and evolution. In figure 1, the main steps in the method are presented graphically. The design process starts with a design of the software architectural based on the functional requirements specified in the requirement specification. Although software engineers generally will not design a system less reliable or reusable, the quality requirements are not explicitly addressed at this stage. The result is a first version of the application architecture design. This design is evaluated with respect to the quality requirements. Each quality attribute is given an estimate in using a qualitative or quantitative assessment technique. The estimated quality attribute values are compared to the values in the requirements specification. If all estimations are as good or better than required, the architectural design process is finished. Otherwise, the second stage is entered: architecture transformation. During this stage, the architecture is improved by selecting appropriate quality attribute-optimizing transformations. Each set of transformations (one or more) results in a new version of the architectural design. This design is again evaluated and the same process is repeated, if necessary, until all quality requirements are fulfilled or until the software engineer decides that no feasible solution exists. The transformations, i.e. quality attribute optimizing solutions, generally improve one or some quality attributes while they affect others negatively.

The fact that the method is iterative is not novel. Some design methods for one-QR based systems, e.g. real-time or performance engineering, follow a similar iterative process. For instance, Smith [Smith 90] defines a similar method for performance engineering but she only considers performance.

In the remainder of this chapter, the three main steps in the method, i.e. functionality-based architecture design, evaluation and assessment of software architectures and the transformation of software architectures are described in more detail.

3. Functionality-based Architecture Design

The first step during software architecture design is to develop a software architecture based on the functional requirements. Based on the requirement specification, the top-level, i.e. architecture, design of the system is performed. The main issue during this phase is to identify the core abstractions, i.e., the archetypes, based on which the system is structured. Although these abstractions are modeled as objects, our experience is that these objects are not found immediately in the application domain. Instead, they are the result of a creative design process that, after analyzing the various domain entities, abstracts the most relevant properties and models them as architecture entities. Once the abstractions are identified, the interactions between them are defined in more detail.

The process of identifying the entities that make up the architecture is different from, for instance, traditional object-oriented design methods. Those methods start by modeling the entities present in the domain and organize these in inheritance hierarchies, i.e. a bottom-up approach. Our experience is that during architectural design it is not feasible to start bottom-up since that would require dealing with the details of the system. Instead one needs to work top-down.

Architecture entity identification is related to domain analysis methods [ref-domainanalysis]. However, different from these approaches, our experience is that the entities resulting from architecture design are generally not found in the domain. A second difference between architecture design and domain analysis is that the architecture of a system generally covers multiple domains.

The assumption underlying our approach is that an architectural design based on the functional requirements only does not preclude the use of transformations for optimizing quality requirements. Some researchers believe that an architectural design cannot be separated in the way proposed in this paper. We agree that no pure separation can be achieved, i.e. an architectural design based on functional requirements only will still have values for its quality attributes. However, we believe that an objective and repeatable architectural design method must be organized according to our principles since it is unlikely that an architectural design process does not require iterations to optimize the architecture. Since an architecture based on functional requirements only is more general, it can be reused as input for systems in the same domain but with different quality requirements. On the other hand, it is unlikely that a software architecture that fulfils a particular set of quality requirements will be applicable in a domain with different functional requirements.

Chapter 3 discusses the design of software architectures based on their functional requirements in more detail.

4. Assessing Quality Attributes

One of the core features of the architectural design method is that the quality attributes of a system or application architecture are explicitly evaluated during architecture design; thus without having a concrete system available. Although several notable exceptions exist, our experience is that the traditional approach in software industry is to implement the system and then measure the actual values for the quality system properties. The obvious disadvantage is that potentially large amounts of resources have been put on building a system that does not fulfil its quality requirements. In the history of software engineering, several examples of such systems can be found. Being able to estimate the quality attributes of the system already during early development stages is important to avoid such mishaps.

However, the question is how to measure system properties based on an abstract specification such as an architectural design. For obvious reasons it is not possible to measure the quality attributes of the final system based on the architecture design. Instead, the goal is to evaluate the potential of the designed architecture to reach the required levels for its quality requirements. For example, some architectural styles, e.g. layered architectures, are less suitable for systems where performance is a major issue, even though the flexibility of this style is relatively high.

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

4.1 Scenario-based evaluation

To assess a particular quality attribute, a set of scenarios is developed that concretizes the actual meaning of the requirement. For instance, the maintainability requirement may be specified by a *change profile* that captures typical changes in requirements, underlying hardware, etc. The profile can then be used to evaluate the effort required to adapt the architecture to the new situation. Another example is robustness where the architecture can be evaluated with respect to the effects of invalid input.

The effectiveness of the scenario-based approach is largely dependent on the representativeness of the scenarios. If the scenarios form accurate samples, the evaluation will also provide an accurate result. Object-oriented design methods use scenarios to specify the intended system behavior, e.g. use-cases [Jacobsen et al. 92] and scenarios [Wirfs-Brock et al. 90]. For architectural design, however, two sets of scenarios should be developed, i.e. one for design and one for evaluation purposes. Once a version of the architecture is ready for evaluation, the software engineer can ‘run’ the scenarios for the architecture and evaluate the results. For instance, if most of the change scenarios require considerable reorganizations of the architecture, one can conclude that the maintainability of the architecture is low.

In our experience, scenario-based assessment is particularly useful for development quality attributes. Quality attributes such as maintainability can be expressed very naturally through change scenarios. In [Bengtsson & Bosch 99b], we present a scenario-based technique that we developed for predicting maintainability based on the software architecture.

4.2 Simulation

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

Simulation of the architecture design is, obviously, not only useful for quality attribute assessment, but also for evaluating the functional aspects of the design. Building a simulation requires the engineer to define the behavior and interactions of the architecture entities very precise, which may uncover inconsistencies in the design earlier than traditional approaches.

Once a simulation is available, one can execute execution sequences to assess quality attributes. Robustness, for example, can be evaluated by generating or simulating faulty input to the system or by inserting faults in the connections between architecture entities.

Simulation complements the scenario-based approach in that simulation is particularly useful for evaluating operational quality attributes, such as performance of fault-tolerance by actually executing the architecture implementation, whereas scenarios are more suited for evaluating development quality attributes, such as maintainability and flexibility. Nevertheless, the implementation of the architecture in the simulation can be used to evaluate, for instance, maintainability, by changing the implementation according to change scenarios and measuring the required effort.

4.3 Mathematical modeling

Various research communities, e.g. high-performance computing [Smith 90], reliable systems, real-time systems [Liu & Ha 95], etc., have developed mathematical models that can be used to evaluate especially operational quality attributes. Different from the other approaches, the mathematical models allow for static evaluation of architectural design models. For example, performance modeling is used while engineering high-performance computing systems to evaluate different application structures in order to maximize throughput.

Mathematical modeling is an alternative to simulation since both approaches are primarily suitable for assessing operational quality attributes. However, the approaches can also be combined. For instance, performance modeling can be used to estimate the computational requirements of the individual components in the architecture. These results can then be used in the simulation to estimate the computational requirements of different execution sequences in the architecture.

4.4 Experience-based reasoning

A fourth approach to assessing quality attributes is through objective reasoning based earlier experiences and logical argumentation. Experienced software engineers often have valuable insights that may prove extremely helpful in avoiding bad design decisions. Although some of these experiences are based on anecdotal evidence, most can often be justified by a logical line of reasoning.

This approach is different from the other approaches in that the evaluation process is less explicit and more based on subjective factors such as intuition and experience. The value of this approach should, however, not be underestimated. Most software architects we have worked with had well-developed intuitions about ‘good’ and ‘bad’ designs. Their analysis of problems often started with the ‘feeling’ that something was wrong. Based on that, an objective argumentation was constructed either based on one of the aforementioned approaches or on logical reasoning. In addition, this approach may form the basis for the other evaluation approaches. For example, an experienced software engineer may identify a maintainability problem in the architecture and, to convince others, define a number of scenarios that illustrate this.

5. Architecture Transformation

Once the quality attributes of an architecture have been assessed, the estimated values are compared to the requirements specification. If one or more of the quality requirements are not satisfied, the architecture has to be changed to cover these requirements also. This requires the software engineer to analyse the architecture and to decide due to what cause the property of the architecture is inhibited. Often, the evaluation itself generates hints as to what parts or underlying principles cause low scores.

Assessment of the quality attributes is performed assuming a certain context, consisting of, certain subsystems, e.g. databases or GUI systems and one or more operating systems and hardware platforms. Whenever a quality attribute is not fulfilled, one may decide to either make changes to the presumed context of the system architecture or to make changes to the architecture itself. In the architectural design method discussed in this part of the book, changes to the architecture are performed as *architecture transformations*. Each transformation leads to a new version of the architecture that has the same functionality, but different values for its properties.

The consequence of architecture transformations is that most transformations affect more than one property of the architecture; generally some properties positively and others in a negative way. For instance, the *Strategy* design pattern [Gamma et al. 94] increases the flexibility of a class with respect to exchanging one aspect of its behavior. However, performance is often reduced since instances of the class have to invoke another object (the instance of the Strategy class) for certain parts of their behavior. However, in the general case, the positive effect of increased flexibility considerably outweighs the minor performance impact.

Four categories of architecture transformations have been identified, organized in decreasing impact on the architecture, i.e. imposing an architectural style, imposing an architectural pattern, applying a design pattern and converting quality requirements to functionality. One transformation does not necessarily address a quality requirement completely. Two or more transformations might be necessary. In the sections below, each category is discussed in more detail.

5.1 Impose architectural style

Shaw and Garlan [Shaw & Garlan 96] and Buschmann et al. [Buschmann et al. 96] present several architectural styles (or patterns) that improve the possibilities for certain quality attributes for the system the style is imposed upon and are less supportive for other quality attributes. Certain styles, e.g. the layered architectural style, increase the flexibility of the system by defining several levels of abstraction, but generally decrease the performance of the resulting system. With each architectural style, a fitness for each system property is associated. The most appropriate style for a system depends primarily on its quality requirements. Transforming an architecture by imposing an architectural style results in a complete reorganization of the architecture.

Although architectural styles can be merged up to some extent, more often a different style is used in a subsystem than at the system level, provided that the subsystem acts as a correct component at the system level. However, if, during design iteration, a second architectural style is selected for a part of the system, it is necessary to make sure that the constraints of the two styles do not conflict with each other.

In our approach, we explicitly distinguish between the components that are used to fulfil the functional requirements and the software architecture of the system that is used to fulfil the quality requirements. In practice, the distinction is generally not as explicit, i.e. also the implementation of a component influences most quality attributes, e.g. reliability, robustness and performance.

5.2 Impose architectural pattern

A second category of transformations is the use of *architectural patterns*[1]. An architectural pattern is different from an architectural style in that it is not predominant and can be merged with architectural styles without problems. It is also different from a design pattern since it affects the complete architecture, or at least the larger part of it. Architectural patterns generally impose a *rule* [Perry & Wolf 92] on the architecture that specifies how the system will deal with one aspect of its functionality.

Architectural patterns generally deal with some aspect of the system behavior that is not in the application domain, but addresses some of the supporting domains. For example, the way the system deals with concurrency, persistence, fault-tolerance or distribution. If the software architect decides to implement concurrency using an application-level scheduler that invokes the entities that need some active behavior, then this decision puts requirements on most architectural entities since each entity needs to support a particular interface and needs to limit its execution time to a small and predictable period. Architectural patterns are generally orthogonal to each other and to architectural styles, but affect most entities in the architecture.

5.3 Apply design pattern

The third class of transformations is the application of a design pattern. This is generally a less dramatic transformation than the former two categories. For instance, an *abstract factory* pattern [Gamma et al. 94] might be introduced to abstract the instantiation process for its clients. The abstract factory pattern increases maintainability, flexibility and extensibility of the system since it encapsulates the actual

1. Note that our use of the term “architectural pattern” is different from the use in [Buschmann et al. 96].

class type(s) that are instantiated, but decreases the efficiency of creating new instances due to the additional computation, thereby reducing performance and predictability. Different from imposing an architectural style, causing the complete architecture to be reorganized, the application of a design pattern generally affects only a limited number of classes in the architecture. In addition, a class can generally be involved in multiple design patterns without creating inconsistencies.

5.4 Convert quality requirements to functionality

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

This type of transformation is different from the first three in that it does not change the existing structure of the software architecture, but instead primarily adds functional entities to the structure that fulfil a particular quality requirement.

5.5 Distribute requirements

The final activity of a transformation iteration deals with quality requirements using the *divide-and-conquer* principle: a quality requirement at the system level is distributed to the subsystems or components that make up the system. Thus, a quality requirement X is distributed over the n components that make up the system by assigning a quality requirement x_i to each component c_i such that $X=x_1+ ... +x_n$.

A second approach to distribute requirements is by dividing the quality requirement into two or more functionality-related quality requirements. For example, in a distributed system, fault-tolerance can be divided into fault-tolerant computation and fault-tolerant communication.

Distributing requirements does not change the structure or functionality of the software architecture that is under design, but it facilitates the breaking down of quality requirements and their assignment to lower-level components. This process is analogous to the process of decomposition of functional requirements during conventional system design.

6. Concluding Remarks

In this chapter, we have introduced the architecture design method that is the topic of this part of the book. We started by providing a terminology for requirements in general and quality requirements in particular. Based on that discussion, we introduced the notion of *profiles* and their relevance for the definition of quality requirements. In section 2, we provided an introduction to software architecture and presented an overview over the architecture design method. In the subsequent sections, we presented an overview over the three main phases in the method, i.e. functionality-based architecture design, architecture evaluation and architecture transformation.

In the following chapters, first three systems are introduced that are used as examples in the subsequent chapters. Then, each of the main phases of the method is discussed in a separate chapter.

CHAPTER 2

Software Architecture Design Case Studies

Software architecture design is, similar to most engineering and design disciplines, very hard to discuss at an abstract level. Instead, one needs concrete examples of relevant systems to discuss alternative design solutions and ways of argumentation. In this chapter, we present three examples that will be used throughout this part of the book. All three systems are in the embedded systems domain, but still rather diverse. These systems are not just examples for illustrative purposes, but they have been the subject of architecture design projects with various industrial partners and our research group.

1. Fire-Alarm Systems

The description of fire-alarm systems presented here is based on the work that was performed by TeleLarm AB, a swedish security company. The work originally started in 1992 with the aim of taking advantage of the benefits of object-oriented technology. Concretely, the aim of the project was to develop an object-oriented framework that would be able to handle the large variety of fire-alarm products ranging from small home and office installations to large, complex systems servicing industrial multi-building sites. The first version of the framework was completed in 1994 and, on the average, 500 systems per year have been installed at client sites. The experiences from using the framework have been very positive in that it delivered the two main promises of software reuse, i.e. increased quality and decreased development effort. For example, not a single fault has been detected in the framework after beta-testing. Secondly, the software has proved to be easy to modify; a number of changes proved to be implemented with considerable less effort than expected based on earlier experience.

During 1996, a second version of the framework was developed and fielded. This version primarily improved the modularization in the framework, extended the domain covered by the framework and improved some dimensions of variability.

The notion of object-oriented frameworks was introduced in chapter 7 and has importance in the design of object-oriented software architectures as well as in the use of software architectures, but in this part of the book we are only concerned with the design aspects of the fire-alarm system framework. Finally, the description of fire-alarm system is based on [Molin 97] and [Telelarm 96].

1.1 Domain Description

The main function of a fire-alarm system is to monitor a large number of detectors and, whenever a potential fire is detected, activate a number of outputs. Several examples of outputs exist, including alarm bells, alarm texts on displays, extinguisher system and automatic alarming of the fire department. Detectors, or input devices, cover a wide variety of types, ranging from the traditional temperature and smoke sensors to ultra-sensitive laser-based some detectors. The wide variety in input and output devices presents one of the major challenges to the software architecture design.

A second dimension of variability is the range of systems that should be covered by the architecture. At the low end there are very cost sensitive systems that still should fulfil standards and regulations. High-end systems include advanced sensors, a sophisticated high-speed extinguisher control system and a large and complex graphical user interface. For instance, conventional sensors have three externally visible states, i.e. normal, alarm or fault. The advanced sensors instead transmit particle density (smoke intensity) or temperature values to a control unit that deduces, based on the input date and using various algorithms, whether there is an alarm or fault situation.

A third relevant aspect of fire-alarm systems is their highly distributed nature. Detectors and output units are distributed throughout a building and, in the case of high-end systems, over multiple buildings. The software controlling the fire alarm system has to monitor all input devices for alarms and, if alarms occur, activate the correct output devices.

Because of potential consequences in case of a failure of the fire-alarm system, continuous self monitoring is part of the system behavior. Due to this one can think of the system as consisting of two levels of functionality, i.e. domain-related functionality and system monitoring functionality.

The platforms for the fire-alarm systems range from small 8-bit micro controller systems to larger 16-bit systems that can be connected to form a distributed network with a capability of up to 10 000 addressable detectors. In addition, the installation owner can configure names and physical locations of detectors, the texts that appear on displays in case of fire and the relations between output devices and detectors, i.e. what output devices are activated when particular detectors indicate an alarm.

1.2 Quality Requirements

At the start of the design project, it was identified that the company maintained a family of fire-alarm systems that used different real-time kernels, different hardware and different programming languages. In addition, each system was available in different language versions and with functionality specific for particular countries. The goal of the project was to cover these systems and system variations using a single product-line architecture and component base, i.e. a product-line architecture.

In addition to addressing the variability described above the system had support a number of other quality requirements as well. Below, the quality requirements are described:

- **Configurability**: It should be relatively simple to instantiate specific versions of the fire-alarm system. For instance, configuration with country, language and hardware specific details should be easy.
- **Demonstrability**: Although the architecture can affect reliability of instantiated systems only up to some extent, one can require that the architecture simplifies testing and facilitates the demonstration of the reliability of resulting systems. This is particularly important since fire-alarm systems need to be certified by an external certification institute.
- **Performance**: The performance of a fire-alarm system is dependent on the size of the system, the available memory and the CPU processing capacity, making it hard to state absolute performance requirements at this level. However, the

architecture and reusable components should be efficient, i.e. not be considerably slower than a system-specific implementation.

- **Maintainability**: The system should be prepared for incorporating new requirements, e.g. by ‘factoring out’ potential variable parts and representing them as separate entities.

Since the maintainability of the architecture and provided levels of configurability are the primary requirements on a product-line architecture, we discuss these aspects in more detail. Below a list of potential new requirements or requirement categories is presented.

- **Detector technology**: New types of detectors enter the market continuously. These detectors not only vary with respect to the measured variables, but, more important, in the way they interface with the fire-alarm system.
- **Extinguishers**: Extinguisher systems are also evolving constantly, due to new technology but also due to, e.g. new environmental standards.
- **Compatibility**: Several other systems in an organization are interested in communicating with the fire-alarm system for, among others, retrieving data and setting alarm boundaries. In addition, new fire-alarm systems should be able to incorporate legacy fire-alarm systems.
- **Hardware**: Cost play a major role in the fire-alarm systems domain. Consequently, new hardware with a better price/performance ratio should be incorporated in the fire-alarm system with limited effort.
- **Man-machine interface**: There is a constant and quick development in technology used to interface with operators. Starting with LEDs and buttons, succeeded by LCD screens and numeric keypads, the current level is to have a graphical, window-based interface. However, the development is towards incorporating multiple medias in the interface with the fire-alarm system.
- **Standards**: Although standards have a tendency to lag behind the industrial practice, they do change on a regular basis and the changes need to be incorporated in the products.
- **User-adapted instantiations**: Large customers often have additional (or even conflicting) requirements on the fire-alarm system that have to be incorporated in their instantiation against limited effort. In addition, when upgrades of the product become available, these customers should have the ability to upgrade while maintaining their additions.

2. Measurement Systems

The increasing automation of the production process has begun to address processes beyond the primary production processes. During the last decade, one can recognize an increasing need for automated tools that support the quality control processes surrounding the actual production. The emergence of the ISO9000 quality standards, the quality thinking in general and the increased productivity of production technology requires the quality control systems to improve productivity as well and whereas many factories used manual quality control by personnel, nowadays the need for automated support is obvious. This development has dramatically increased the need for automated measurement systems. The advantages of measurement systems are generally improved performance/cost ratio and more consistent and accurate quality control. This development increased the needs for reusability of existing measurement system software. Although these systems, conceptually, have a rather similar structure, in practice the implementation of these systems tends to be rather diverse. This is due to the fact that real-time constraints, concurrency and requirements resulting from the underlying hardware strongly influence the actual implementation.

Despite these difficulties, we have, together with our industrial partner, EC-Gruppen, a company located in southern-swedish developing embedded systems, designed an object-oriented framework for measurement systems that would decrease their software development cost by increasing reuse of existing software and, as an important second requirement, increase the flexibility of running applications. Operators of the measurement systems often need to make some adjustments in the way the measurement system evaluates a measurement item and the system should provide this flexibility. However, traditional systems constructed in C and assembly often have difficulty to provide this functionality. Part of the results of the project are reported in [Bosch 99].

2.1 Domain Description

Measurement systems are a class of systems used to measure the relevant values of a process or product. These systems are different from the, better known, process control systems in that the measured values are not directly, i.e. as part of the same system, used to control the production process that creates the product or process that is measured. A measurement system is used for quality control on produced products that can then be used to separate acceptable from unacceptable products or to categorize the products in quality categories. In some systems, the results from the measurement are stored in case in the future the need arises to refer to this information, e.g. if customers complain about products that passed the measurement system.

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 a number of sensors and actuators. The sensors provide information about the real-world through the noticed impulses. However, whereas traditional sensors were primarily hardware and had a very low-level interface to the software system, new sensors provide increasing amounts of functionality that previously had to be implemented as part of the software. For instance, a conventional temperature sensor would only provide the A/D conversion and the software would need to convert this A/D value into the actual temperature in Celsius or Kelvin and, in addition, had to do the calibration of the sensor. Modern temperature sensors perform their own calibration and immediately provide the actual temperature in the required format. The interface between the sensor and the system is becoming more and more high-level, but also more complex since the amount of configurability of the sensors is increasing.

With respect to the actuators one can recognize a similar development. Whereas the software previously had to be concerned with the actuation through the actuators, modern actuators often only need a set value expressed in application domain concepts such as angular speed or force. For example, to control the open angle of a valve in a traditional measurement system, one would have to generate a ‘duty cycle’ in software. A duty cycle is the periodic process of sending out a ‘1’ for part of the cycle and a ‘0’ for the rest. The ratio between the time the output is ‘1’ and the time the output is ‘0’ represents the ‘force’ expressed through the actuator. When opening a valve for 70% requires that the system outputs a ‘1’ for 70% of the cyclic period and a ‘0’ for the remaining 30%. The implementation of this is often achieved through an interrupt routine that changes the output signal when required. Modern actuators contain considerably more functionality and will generate the duty cycle themselves, requiring only the set value from the software.

These developments in the domain of sensors and actuators changes measurement systems from small, single processor systems that are developed very close to the hardware to distributed computing systems since the more complex 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. The languages and tools used to construct measurement systems ought to provide powerful means to deal with this complexity.

A measurement system, however, consists of more than sensors and actuators. A typical measurement cycle starts with a trigger indicating that a product, or measurement item, is entering the system. The first step after the trigger is the data-collection phase by the sensors. The sensors measure the various relevant variables of the measurement item. The second step is the analysis phase during which the data from the sensors is collected in a central representation and transformed until it has the form in which it can be compared to the ideal values. Based on this comparison, certain discrepancies can be deduced which, in turn, lead a classification of the measurement item. In the third phase, i.e. actuation, the classification of the measurement item is used to select the appropriate actions that are associated with the classification and subsequently these actions are performed. Example actions may be to reject the item, causing the actuators to remove the item from the conveyer belt and put it in a separate store, or to print the classification on the item so that it can be automatically recognized at a later stage. One of the requirements on the analysis phase is that the way the transformation takes place, the characteristics based on which the item is classified and the actions associated with each classification should be flexible and easily adaptable, both during system construction, but also, up to some extent, during the actual system operation.

Based on the above discussion, one can wonder whether the traditional view on measurement systems as a centralized system with one main control loop is still appropriate. During the project, we became convinced that the system should to be viewed as a collection of communicating, active entities that co-operate to achieve the required system behavior. This improves decomposition of the system, decreases the dependencies between the various parts and increase system flexibility. However, decomposing the system into active entities requires processes to be available, or at least simulated, by the underlying operating system.

Another important aspect is the real-time behavior of the measurement system. Different from many real-time systems, a measurement system is not a periodic system. The real-time constraints in the system are, directly or indirectly, related to the triggering point where a product to be measured enters the system. Although, when running at maximum performance, this becomes a periodic behavior, the start is not determined by the clock, but by a physical event. In the ideal situation, the software engineer would specify the real-time constraints on the different activities in the system. Based on that specification, the system would schedule the activities such that the real-time constraints are met or, if it is not possible to schedule all activities, respond to the software engineer with a message. However, in the current situation, the software engineer implements the tasks that have to be performed and performs a test run of the system. Often, the system does not meet all deadlines at first and the software engineer has to adjust the system to fulfil the requirements by, e.g. changing the priorities of the different processes.

Finally, the requirements on modern measurement systems often result in systems that are no longer confined to a single processor. Distribution plays an increasingly important role in measurement systems. However, the presence of distribution should not require the software engineer to change the basic architecture of the system. The system should just be extended with behavior for dealing with communication over address spaces.

In figure 2, the entities that are part of a simple measurement system is shown. The system consists of five entities that communicate with each other to achieve the required functionality. Below, a sequence of events during a normal measurement cycle for an entity is shown:

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.

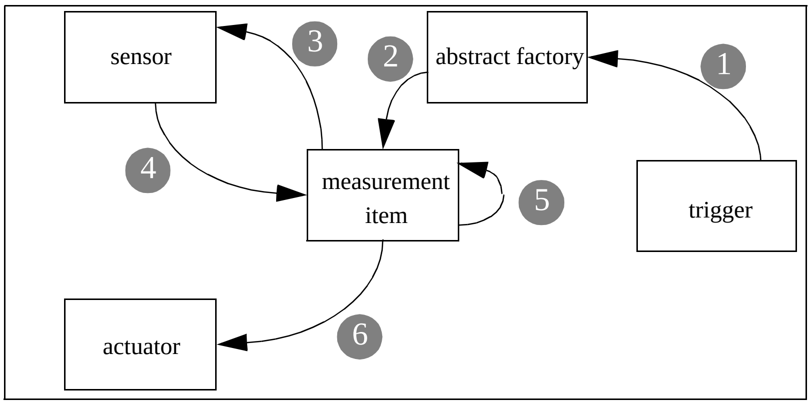


FIGURE 2. **Architecture of a simple measurement system**

2.2 Quality Requirements

Measurement systems should support the functionality described in the previous section. However, in addition to these functional requirements, one can identify a number of quality requirements as well.

- **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.
- **Configurability**: 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**: Although flexibility would be considered to be a positive aspect of any system, 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.

3. Haemo Dialysis Systems

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.

We have been involved in a research project aimed at designing a new software architecture for the dialysis machines produced by Althin Medical. The software of the existing generation of products was exceedingly hard to maintain and certify and management had become convinced that it was necessary to develop the next generation of the dialysis system software independent of the existing software. 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* [Argyris et al 85], i.e. researchers actively participated in the design process and reflected on the process and the results. The results of the research project are reported, among others, in [Bengtsson & Bosch 99a].

3.1 Domain Description

An overview of a dialysis system is presented in figure 3. 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 3, 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 in this context.

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.

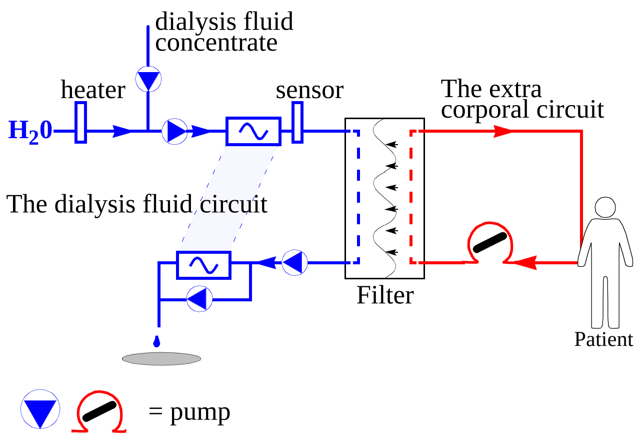

dialysis fluid concentrate
heater
sensor
The extra corporal circuit
H_20
The dialysis fluid circuit
Filter
Patient
= pump

FIGURE 3. Schematic of Haemo Dialysis Machine

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 deteriorated substantially. The three major software subsystems are the Man Machine Interface (MMI), the Control System, and the Protective system (see figure 4).

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.

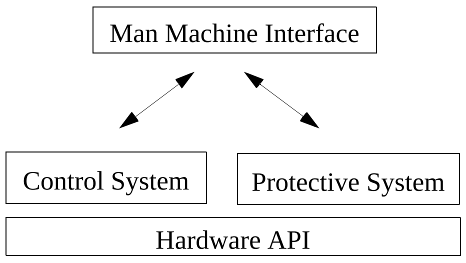


FIGURE 4. Legacy system decomposition

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.

3.2 Quality 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.

Configurability. The software developed for the dialysis machine should be easily configurable. 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. Consequently, it should be relatively simple to instantiate the product-line architecture for a particular system.

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 has to some extent already been transformed into functional requirements by the safety requirements standard for haemo dialysis machines [CEI/IEC 601-2], but only as far as to define a number of hazard situations, corresponding threshhold 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 [CEI/IEC 601-2].

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*.

4. Concluding Remarks

The design of software architectures, similar to other engineering disciplines, is hard to present and discuss without concrete examples. In this chapter, we have presented three software systems that will be used as examples in the subsequent chapters, i.e. fire-alarm systems, measurement systems and dialysis systems. All three systems are in the embedded systems domain and have to deal with rapidly developing contexts in terms of requirements, hardware and other technology and each system has to support considerable variation in terms of the instantiations that need to be supported by the architecture and its base of reusable components. For each system, we provided a domain description and discussed the primary quality requirements.

CHAPTER 3

Functionality-based Architectural Design

Architecture design is the process of converting a set of requirements into a software architecture that fulfils, or at least facilitates the fulfillment of, the requirements. The method for architecture design presented in this part of the book has a focus on the explicit evaluation of and design for quality requirements, but that does not mean that the functional requirements are irrelevant. Before one can start to optimize an architecture for quality requirements, the first step must be to design a first version of the architecture based on the functional requirements.

The first design phase in the <NAME> method consists of three main steps. The first step concerned with determining the context of the system under design, the interfaces of the system to the external entities it interacts with and the behavior the system should exhibit at the interfaces. The second step is the identification of the archetypes, i.e. the main architectural abstractions, and the relations between the archetypes. Our experience is that finding these archetypes is very important especially for the later phases. Architecture transformations tend to build additional structures around the archetypes for fulfilling quality requirements. The final step in functionality-based architectural design is the description of system instances using the archetypes and the system interfaces. Since the architecture, for instance in the case of product-line architectures, may be required to support a number of different instantiations, these have to be specified explicitly to verify that the system, in addition to the commonality also supports the required variability.

The assumption underlying our approach to architectural design is that starting from the functional requirements does not preclude the optimization of quality requirements during the later architecture design stages. There are some authors [refs] that perform an architectural design based on all requirements and believe that design cannot be separated in the way proposed in this book. We agree that no pure separation can be achieved, i.e. an architectural design based on functional requirements only will still have values for its quality attributes. However, our position is that an objective and repeatable architectural design method must be organized according to our principles since it is unlikely that an architectural design process does not require iterations to optimize the architecture. Since an architecture based primarily on functional requirements is more general and can be reused as input for systems in the same domain but with different quality requirements. On the other hand, it is unlikely that a software architecture that fulfils a particular set of quality requirements will be applicable in a domain with different functional requirements.

The remainder of this chapter is organized as follows. In the next section, we briefly introduce the notation that we will use throughout the book, i.e. UML, or at least the parts of UML that we make use of. In section 2, the definition of the system context is discussed, i.e. the first step in the method. The topic of the subsequent section is the identification of archetypes, whereas the definition of architecture instantiations is discussed in section 4. To illustrate the discussed method steps, we present a number of examples from the case studies discussed in the previous chapter. Finally, the chapter is concluded in section 6.

1. Notation

Architecture designs have to be disseminated among the architects of a system, between the architects and the software engineers and between the architects and the other stakeholders of the system. For this purpose, an architecture needs to be documented using some notation and associated descriptions. Our experience is that the documentation of a design in general, and of an architecture design in particular, requires a considerable translation from the internal representation of the software architect. Often, all kinds of relevant information seems to disappear in the process because no feasible ways of documenting these aspects are available. Undoubtedly, this is also true for the designs presented in this book, but we have no alternative ways of documenting architectures.

In our architecture design projects, we have used a subset of the Unified Modeling Language (UML) [refs]. The reasons for choosing UML are largely practical, i.e. the notation is wide-spread and enjoys a wide acceptance. In addition, most software engineers are able to interpret UML diagrams. Finally, the notion of architecture components and the concept of an object are rather similar, due to which the notation matches architecture specifications reasonably well.

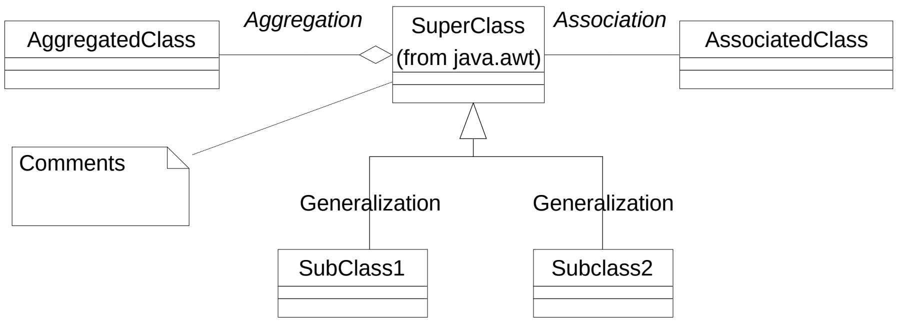


FIGURE 5. Illustrating some of the UML components

Providing a thorough introduction to UML is not within the scope of this book. However, we will briefly introduce the primary components in UML. In figure 5, a number of UML components is shown, i.e. class and three types of relations, i.e. generalization, aggregation and association. In addition, the comments component is shown. These components are used to describe structural properties of a software system. In figure 6, an example sequence diagram is shown that can be used to describe the dynamic properties of a group of classes. It shows the interaction between three objects, including a call to another object, to the object itself, a nested call and a call back to the sender.

*** This section needs to be extended once I know what notation is needed in the remainder of this part ***

2. Defining the System Context

"No object is an island", wrote XX [ref] several years ago. The same can be said to be true for software systems in general. All software has to interface with one or more external entities. Different from what one may suspect, it is the externally visible behavior of a system that is the thing that counts. All our efforts as software architects and engineers are judged from this perspective, although it is difficult to maintain this viewpoint since virtually all of our efforts are spend on the internals of software systems.

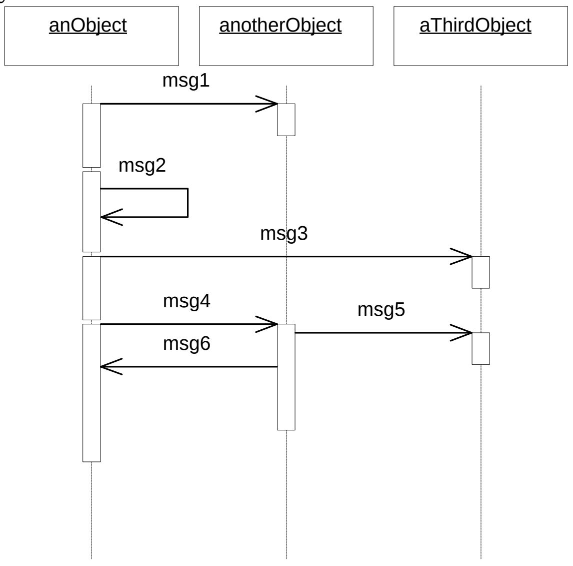


FIGURE 6. **UML Sequence diagram**

The entity at the other end of an interface may be located at a lower-level, a higher-level or at the same-level as the system that we are designing. Examples of lower-level entities include network interface and sensor or actuator interfaces, whereas same-level entities often are systems that address a different functional domain, but need to communicate because of system integration requirements. For instance, in the case of Securitas Larm, high-end fire-alarm system has to interface with other building automation systems to achieve more intelligent behavior. For instance, if no humans are supposed to be in one part of the building, the particle-density sensors should be more sensitive then when smoking persons might be walking around in that part. In the latter case, sensors should not activate the alarm when temporary peeks in particle density are detected. Higher-level entities may be system integration software, e.g. in the case of the fire-alarm system, the building automation integration system, or human beings, e.g. operators of the system or other users.

Explicitly defining the system in terms of the functionality and quality required on its interfaces is an important starting point once the requirements by the customer have been defined. It allows one to distribute requirements to the interfaces and to define the various quality requirements more precisely. Interface-specific requirements allow for the specification of both operational and development quality requirements. For example, performance, real-time and reliability requirements can be expressed on the services provided on the interface. In addition, maintainability and flexibility requirements can be expressed in terms of the likely changes at the interface.

An additional reason for explicitly defining the system context and boundaries is because our experience is that there is a natural tendency to include more and more aspects during design. In the situations that explicit effort is spent on software architecture design, there generally also is an understanding that this process should be allowed to take time. Because there is no extreme time pressure, software architects, in our experience, try to extend the domain of the design because each of these extensions will improve the applicability of the architecture and allow for likely future requirements to be integrated more easily. The problem, however, is that these extensions increase the design and development cost, resulting in the situation where the current development project budget is partially used for maintenance activities in response to likely, but not certain future requirements. Management may easily react to this development using the well-known tool, i.e. strict deadlines, not allowing for a sufficiently thought-through architecture. The more fruitful approach to this is to explicitly define the system boundaries and the functional and quality requirements, requiring software engineers to stick to their appointed domain.

In the case of a software architecture design for a product-line, the definition of the system context is somewhat more complicated since products in the product-line tend to have variations in the interfaces they provide. The product-line architecture has to support the superset over all products of the interfaces, functional requirements and quality requirements without sacrificing cost and resource efficiency for the low-end product. The latter may prove difficult to achieve in practice, but, in our experience, the reasons for these difficulties are often located in the reusable assets rather than in the architecture. Since the architecture is primarily facilitating the fulfillment of functional and quality requirements, it often quite possible to exclude or short-cut parts of the architecture for low-end products. In later chapters, we address the implementation of variability in reusable assets, including the issues related to excluding functionality for low-end instantiations of reusable components.

Concluding, the following activities are part of the system context definition step:

- Define the interfaces of the system to external entities. These entities may be located at a lower level, a higher level or at the same level.
- Associate functional and quality requirements with each interface. Both operational and development quality requirements can be associated with interfaces.
- In the case of a product-line architecture, the variability in the interfaces supported by the various products in the product-line should be explicitly identified and specified. The cost and resource efficiency of low-end products should not be sacrificed for the requirements of high-end products.

3. Identifying the Archetypes

Once the boundaries for the system has been defined, the next step is to identify and define the core abstractions based on which the system is structured. We refer to these core abstractions as *archetypes*. It is of critical importance to successful architecture design that the architect finds a small set of, often highly abstract, entities that, when combined, are able to describe the major part of the system behavior at an abstract level. These entities form the most stable part of the system and supposed not to be changed or only in very limited ways. Our experience is that even relatively large systems can be described in terms of a small number of archetypes.

It is important to note that the archetypes are radically different from subsystems. Whereas subsystems decompose system functionality into a number of big chunks, the archetypes represent stable units of abstract functionality that appear over and over again in the system. In the examples later in this chapter, the difference will be exemplified more clearly.

The process of identifying the entities that make up the architecture is different from, for instance, traditional object-oriented design methods as proposed by, among others, [Booch 94] and [Rumbaugh et al. 91]. Those methods start by modeling the entities present in the domain and organize these in inheritance hierarchies, i.e. a bottom-up approach. Our experience is that during architectural design it is not feasible to start bottom-up since that would require dealing with the details of the system. Instead one needs to work top-down.

Architecture entity identification is related to domain analysis methods (see [ref] for an excellent overview), but some relevant differences exist. First, although archetypes are modeled as domain objects, our experience is that these objects are not found immediately in the application domain. Instead, they are the result of a creative design process that, after analyzing the various domain entities, abstracts the most relevant properties and models them as architecture entities. Once the abstractions are identified, the interactions between them are defined in more detail. A second difference between architecture design and domain analysis is that the architecture of a system generally covers multiple domains.

Once one has reduced the archetype candidates to a small and manageable set that has proven some stability, the relations between the archetypes are identified and defined. The types of relations are generally domain specific and describe control and/or data flow in the system. The relations should not be generic relation types as, for instance, the generalization and aggregation relations in object-oriented modeling. The presence of these relations, especially generalization, between archetypes is suspicious and one should reconsider whether the involved archetypes should perhaps be merged.

Within the domain of architecture description languages, e.g. [Allen & Garlan 97], architectures are described in terms of components and connectors. The connectors are explicitly modeled entities representing the relations between components. When using particular architectural styles, the connectors are style specific as well. For instance, in the pipes and filters architectural style, the pipes are represented as connectors. However, connectors are not equivalent to relations between archetypes. Instead, connectors represent one way of implementing relations. When the components implementing two related archetypes provide a good match, the relation may not be represented explicitly but rather through normal message passing. However, if some mismatch between the components exists, then the necessary glue code can be implemented in a connector that then becomes a first-class entity in the implementation.

Small groups of related archetypes tend to form system-specific ‘architectural patterns’ that are applicable in many locations in the system instantiations. The ‘architectural patterns’ may prove to have a wider validity than just the system context and may actually suit more systems in an application domain.

Finally, no explicit guidelines exist that decide on when the identification and definition of archetypes is done and the team has to move on to the next phase. Our experience is that, during the process, a consensus develops within the team on when sufficient effort has been spent and the design activity can proceed to the next step. However, during later phases, it may prove necessary to return to this step and rework the archetypes. It is important to allow for this type of iteration and not to force the creative process in a waterfall model too early and easily.

Concluding, the following activities take place during archetype identification:

- The identification of archetype candidates.
- The selection of a small and stable set of archetypes from the candidates. This may require the exclusion of candidates, but especially the merging of candidates is rather common.
- The identification and selection of relations between archetypes.

4. Describing System Instantiations

The identified collection of archetypes captures the most stable and core abstractions of the system domain. However, these abstractions do not provide a system description. To describe the system under design, an instantiation of the archetypes and the relations between them is required. Since the archetypes capture core abstractions in the system domain, they are generally instantiated in many places in the system in many different concrete forms.

Since a system instantiation is concerned with the structure of concrete instances of the final product, a decomposition of the system into subsystems is generally relevant. A recursive decomposition of the system into a hierarchy of subsystems helps to deal with the complexity of software systems. The complexity of a software system does not have to be the result of sheer size, it can also result from a multitude of interfaces to the system or because of highly prioritized, but strongly conflicting quality requirements. For instance, the new generation of certain types of embedded systems, e.g. handheld devices, have extreme flexibility and performance requirements. The cost effective implementation of these conflicting quality requirements is a major challenge for the software engineers involved.

The recursive decomposition of the system into subsystems is populated with instantiated archetypes. At the leaf levels of the system, the subsystems are assigned individual instances of archetypes. The generic relations between the archetypes identified during the previous step are instantiated together with the instantiation of the involved archetypes. The assignment of archetypes to subsystems and specification of relations between subsystems allows for a verification of the match between the domain abstractions represented by the archetypes and the concrete system instantiation. If mismatches are identified, this is generally the indication of a problem that needs to be investigated to make sure that no fundamental mistakes have been made that often prove extremely costly to repair at a later stage.

It is the description of the system instantiation that we consider to be the architecture of a software system, i.e. its decomposition into its main components. However, this decomposition does not need to be single level, but may incorporate a two or more levels of decomposition for critical parts of the system. As we will discus in later chapters, the goal of software architecture design is to specify, early in the development process, a system structure that allows for the fulfillment of the system requirements. In certain cases, it is required to perform more detailed analysis of critical parts of the system in order to be able to state with sufficient certainty that the system will fulfil its quality requirements.

Product-line architectures need to support multiple system instantiations, since the individual products in the family have unique requirements. During the definition system instantiations, explicit attention has to be directed to the variation in system instantiation for each product. Although the difficulties of providing the required variability are primarily found in the implementation of the reusable assets, uncareful design of the software architecture can lead to unwanted rigidness in dimensions where flexibility is required.

Summarizing, the following activities take place during the definition of system instantiations:

- The system is recursively decomposed into subsystems.
- Each subsystem is either populated with instantiated archetypes that fulfil the functionality required from the system or is represented by an individual instantiated archetype.
- The generic relations between the archetypes are instantiated for the instantiated archetypes and a verification of the match between abstractions and the concrete system decomposition is performed.
- Sufficient variability of product-line architectures is verified by the definition of multiple system instantiations, representing different products.

5. Illustrating Functionality-Based Design

In the following sections, we illustrate the functionality-based design for the example systems discussed in the previous chapter. Although based on real-world systems, the designs presented here have been modified and simplified for illustrative purposes. Because of this, the designs presented here may seem somewhat small and naive, but since these designs will be transformed in chapter 5 to incorporate the quality requirements we are forced to keep the size of the initial designs small.

5.1 Fire-alarm systems

Defining the system context. A fire-alarm system is a relatively autonomous system, but it does provide a number of interfaces to its context. The first issue to decide whether the mechanical and hardware parts of the detectors and alarm devices are part of the system or not. Since we are concerned with the architecture of the *software* system, we consider those parts to be external and consequently interfaces exist between the software system and the physical detectors and alarm devices.

As second issue that we need to decide upon is whether the communication system is part of the system at hand, because the fire-alarm system is highly distributed in nature. In this case, we decide that communication is included as a part since it forms an integral part of the fire-alarm system functionality.

A second interface of the system is towards the operator of the system. In the case of an alarm, but also for activating and deactivating parts of the system and monitoring its behavior. This amount of variability of the functionality of the interface is very large, but one can identify a number of core issues that need to be retrievable via the operator interface, such as the location of an alarm or fault warning in the building. Part of this interface is the interaction with external contacts that need to be notified when the system enters certain states, e.g. alarm, such as the local fire station.

A third interface, although related to the previous, is concerned with the interaction to other building automation systems. Other systems may be interested in certain events that take place in the fire-alarm system and may request to be notified. Similarly, the fire-alarm system may want to affect the state and behavior of other systems, e.g. in case of a fire in a part of the building, the passage-control system may be ordered to unlock all doors in that part allowing people to leave the building without having to use their cards and codes at every door.

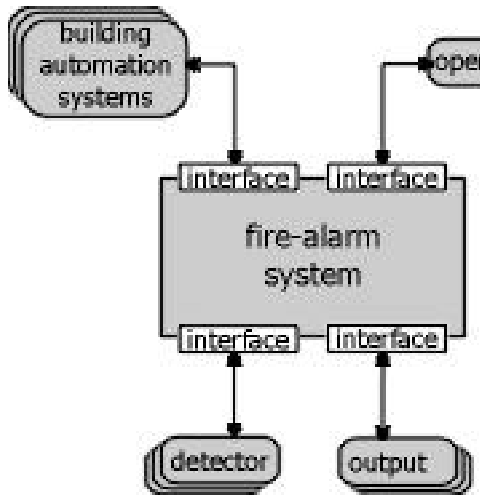


FIGURE 7. **Interfaces of the fire-alarm system**

In figure 7, the interfaces provided by the fire-alarm system are presented graphically. As discussed earlier, in a real design, one would assign functional and quality requirements to the identified interfaces and define the interaction at these interfaces in more detail. However, we leave this step here to avoid exposing unnecessary details of the system.

Identifying the archetypes. When searching for entities that grasp the behavior of several entities and are still abstract, one can detect a number of candidates. Among these, we will use the following as archetypes:

- **Point**: The notion of a point represents highest-level abstraction concerning fire-alarm domain functionality. It is the abstraction of the two subsequent archetypes.
- **Detector**: This archetype captures the core functionality of the fire-detection equipment, including smoke and temperature sensors.
- **Output**: The output archetype contains generic output functionality, including traditional alarms, such as bells, extinguishing systems, operator interfaces and alarm notification to, e.g., fire stations.

- **Control Unit**: Since a fire-alarm system is a distributed system by nature, small groups of points are located at control units that interact with the detectors and outputs in the group. Control units are connected to a network and can communicate. The latter is of crucial importance since the detector alarms in one control unit should often lead to the activation of outputs in other control units.

In figure 8, the relations between the archetypes are shown. As discussed earlier, detector and output are specializations of point and points are contained in control units. Control units communicate with other control units to exchange data about detectors and to activate outputs.

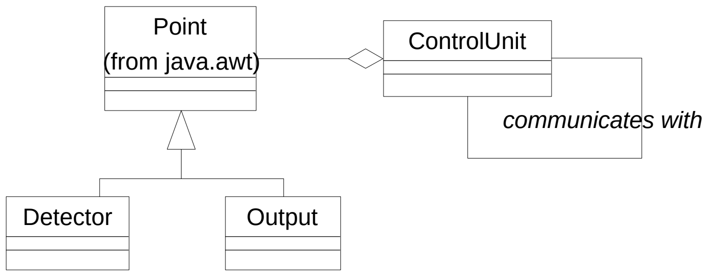


FIGURE 8. **Relations between the fire alarm system archetypes**

Describing system instantiations. The first activity in this step is to identify subsystems. The actual system design is decomposed in six main subsystems. However, since we scaled down the actual system for illustrative purposes, we identify only those subsystems directly related to the identified archetypes for the example fire-alarm system.

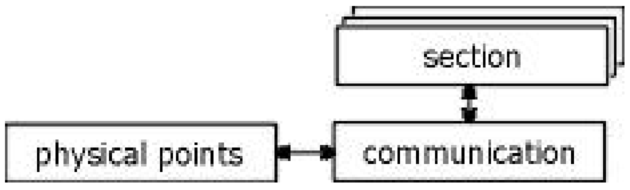


FIGURE 9. **Subsystems of the fire-alarm system**

To understand the instantiation of the fire-alarm system, we present two system instantiation that are at the two extreme ends of the complexity scale. The first system, shown in figure 10, represents a small system that might be found, for example, in a single family house. It consists of a small set of detectors, five smoke detectors in the example, one control unit and two outputs, i.e. a sound alarm and a simple LED-based user interface. The functionality available to the user is to activate or deactivate the system and the feedback from the system is an indication for alarm and one for faults, i.e. internal system errors.

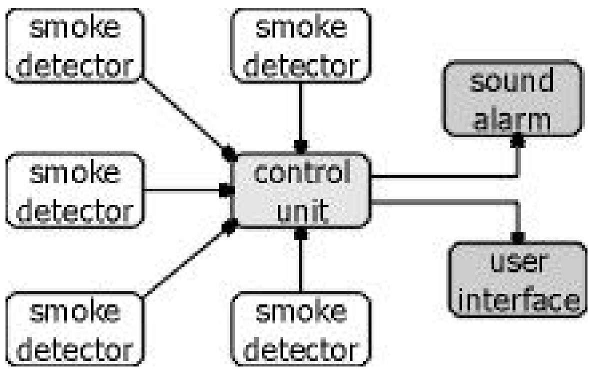


FIGURE 10. Example small fire-alarm system instantiation

A considerably larger example of an instantiated fire-alarm system is shown in figure 11. The fire-alarm system covers a site consisting of two buildings and each building is divided into 4 sections. Each section is supervised by a control unit. One of the control units has an operator interface as a point connected to it. Since the control units are able to communicate with each other, the operator can monitor the complete system from the control unit that the interface is connected to.

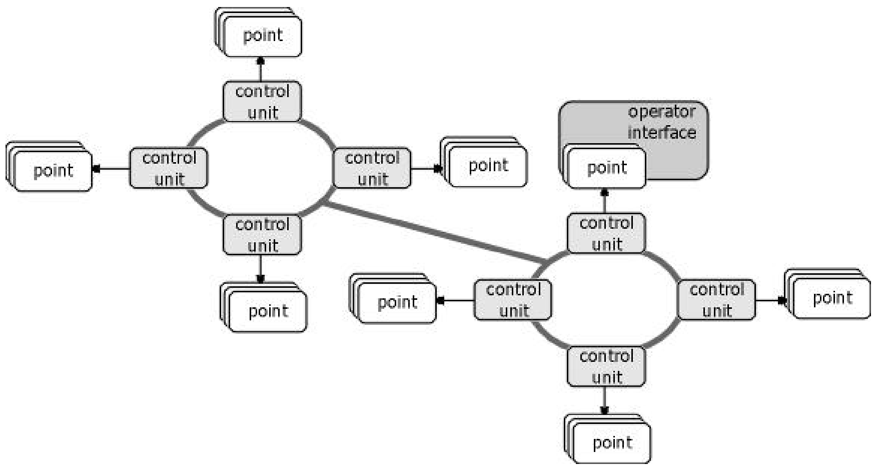


FIGURE 11. **Example two-building fire-alarm system**

5.2 Measurement systems

Defining the system context.

Identifying the archetypes.

Describing system instantiations.

5.3 Dialysis systems

Defining the system context.

Identifying the archetypes.

Describing system instantiations.

6. Summary

Architecture design connects the activity of requirements engineering to conventional detailed design by providing a top-level design incorporating the main design decisions. In this chapter, we discussed the first step in the architecture design process, i.e. designing the first version of the architecture based on the functional requirements. The method for architecture design presented in this part of the book has a focus on the explicit evaluation of and design for quality requirements, but that does not mean that the functional requirements are irrelevant. Before one can start to optimize an architecture for quality requirements, the first step must be to design a first version of the architecture based on the functional requirements.

The first design phase discussed in this chapter consists of three main steps. The first step concerned with determining the context of the system under design, the interfaces of the system to the external entities it interacts with and the behavior the system should exhibit at the interfaces. In addition, the variability required from the components is specified. The second step is the identification of the archetypes, i.e. the main architectural abstractions, and the relations between the archetypes. Our experience is that finding these archetypes is very important especially for the later phases. Architecture transformations tend to build additional structures around the archetypes for fulfilling quality requirements. The final step in functionality-based architectural design is the recursive decomposition of the system into subsystems and the description of system instances using the archetypes and the system interfaces. Since the architecture, for instance in the case of product-line architectures, may be required to support a number of different instantiations, these have to be specified explicitly to verify that the system, in addition to the commonality also supports the required variability.

CHAPTER 4

Assessing Software Architectures

In the previous chapter, a first version of some software architectures, based on the functional requirements, was designed. In order to decide whether these architecture fulfil their quality requirements as well, the architectures needs to be assessed. In this chapter we discuss a number of different approaches to architecture evaluation that we have experienced as being useful. These techniques are illustrated using the example architectures presented in the previous chapter.

1. Introduction

One of the core features of the architectural design method is that the quality attributes of a system or application architecture are explicitly evaluated during architecture design; thus without having a concrete system available. Although several notable exceptions exist, our experience is that the traditional approach in software industry is to implement the system and then measure the actual values for the quality system properties. The obvious disadvantage is that potentially large amounts of resources have been put on building a system that does not fulfil its quality requirements. In the history of software engineering, several examples of such systems can be found. Being able to estimate the quality attributes of the system already during early development stages is important to avoid such mishaps.

However, the question is how to measure system properties based on an abstract specification such as an architectural design. For obvious reasons, it is not possible to measure the quality attributes of the final system based on the architecture design. Instead, the goal is to evaluate the potential of the designed architecture to reach the required levels for its quality requirements. For example, some architectural styles, e.g. layered architectures, are less suitable for systems where performance is a major issue, even though the flexibility of this style is relatively high.

The assessment of an architecture can have different goals, depending on the ambition level of the software architect and the applicability of the assessment techniques used:

- **Relative assessment**: At the lowest ambition level, the software architect is interested in the comparison of two candidate architectures and is concerned with what architecture is more suited for a particular quality attribute. These two architectures may either be two completely different alternatives, but it may also be two subsequent versions of an architecture, where the latter has been transformed to improve the assessed quality attribute or another. For instance, one may assess two architectures for maintainability to decide which of the two is easiest to maintain.

  The main disadvantage is that relative assessment only gives a 'boolean' answer, e.g. architecture A is more suited than architecture B for a given quality attribute. In the situation where the software architect has two alternative architectures, A and B, and two quality attributes, e.g. performance and maintainability, and one of the architectures is more suited for performance and the other more suited for maintainability, the architect has too little information to make a decision concerning which alternative is more viable. The one architecture may be only slightly worse for performance, but considerably better for maintainability, but relative assessment gives no information concerning this.
- **Absolute assessment**: At a higher ambition level, the software architect is interested in making absolute statements about the quality attributes of the software architecture. Examples of these are statements about the throughput of the system, average response times of individual actions and the maintenance cost of the system. If the architect is able to perform assessments at the absolute level with an acceptable accuracy, then it is possible to compare the assessment results to the requirement specification and decide on whether the system will fulfil all its requirements, including the quality requirements before the system is actually build. In addition, the comparison of alternative architectures or subsequent versions of an architecture become much more informed and the architect has quantifiable, objective means to select alternatives.

A disadvantage of this approach is that although we know what level the architecture provides for the assessed quality attributes, we have no information about the theoretically maximum (or minimum) values for the quality attributes. Thus, we have assessment results that, for instance, predict performance and maintainability levels, but we have no clue whether evolving this architecture or a fundamentally different architecture will provide, potentially, much higher performance and much lower maintenance cost.

- **Assessment of theoretical maximum**[1]: At the highest level of assessment we assess an architecture both for its current level and for its theoretical maximum (or minimum) for the relevant quality attributes. The gap between current and maximum levels allows us to determine whether evolving the architecture is still useful and whether we need to start making trade-offs between quality attributes, and potentially renegotiate with the stake-holders to change the requirement specification, or that either evolving the current architecture or a fundamentally different architecture would be able to incorporate both quality requirements without conflicts.

In our experience, the currently available techniques for architecture assessment allows us to make absolute statements about quality attributes at the software architecture level (although perhaps not at the level of accuracy that we would like), but we have no means to predict theoretical maximum (or minimum) values for software architectures.

This chapter is organized as follows. In the next section, the notion of scenario profiles is introduced and explained in detail. These profiles are used for the assessment of quality attributes. Three approaches for assessing quality requirements have been identified, i.e. scenarios, simulation and mathematical modeling. For each quality requirement, the engineer can select the most suitable approach for evaluation. Each assessment approach is discussed in a separate section. Section 6 discusses the importance of experience and creative insight of software architects and engineers. This section is followed by a discussion of architecture assessment in the broad sense and the paper is concluded in section 8. Some relevant further reading is discussed in section 9.

1. Thanks to PO Bengtsson for pointing this out to me.

2. Profiles

Quality requirements such as performance and maintainability are, in our experience, generally specified rather weakly in industrial requirement specifications. In some of our cooperation projects with industry, the initial requirement specification contained statements such as “The maintainability of the system should be as good as possible” and “The performance should be satisfactory for an average user”. Such subjective statements, although well intended, are useless for the evaluation of software architectures. For example, [Gilb 88] discusses the quantitative specification of quality requirements and presents useful examples.

Several research communities, e.g. performance [Smith 90], real-time [Liu & Ha 95] and reliability [Neufelder 93], have developed techniques used for the specification and assessment of their particular quality requirement. Typical for these techniques is that they tend to require considerable effort from the software engineer for creating specifications and predictions. Secondly, since the ambition is to produce detailed and accurate results, these techniques generally require information about the system under development that is not available during architectural design, but earliest during detailed design. Since we are interested in making predictions early in the design process, before the important, architectural design decisions cannot be revoked, other techniques are required that do not require as much detailed information and, consequently, may lead to less precise results, but give at least indications of the quality of the prediction. Finally, it is important to note that the software engineering industry at large has not adopted the techniques developed by the quality-attribute research communities. One explanation might be that within an engineering discipline, each activity is a balance of investment and return. These techniques may not have provided sufficient return-on-investment, from the perspective of industrial software engineers, to be economically viable.

However, when one intends to treat the architecture of a software system that one is working on explicitly in order be able to early predict the quality attributes of the system, one is required to specify quality requirements in sufficient detail. One common characteristic for quality requirements is that stating a required level without an associated context is meaningless. For instance, the statement “Performance = 200” or “Maintainability = 0.8” is meaningless for architecture evaluation.

However, one common denominator of most quality attribute specification techniques is that some form of *profile* is used as part of the specification. A profile is a set of scenarios, generally with some relative importance associated with each scenario. The profile used most often in object-oriented software development is the *usage profile*, i.e. a set of usage scenarios that describe typical uses for the system. The usage profile can be used as the basis for specifying a number of, primarily operational, quality requirements, such as performance and reliability. However, for other quality attributes, other profiles are used. For example, for specifying safety we have used *hazard scenarios* and for maintainability we have used *change scenarios*.

Based on our experience, we believe that it is necessary to specify the relevant profiles for the software quality requirements that are to be considered explicitly in the architecture design process. Using the profile, one can make a more precise specification of the requirement for a quality attribute. For example, the required performance of the system can be specified using the usage profile by specifying the relative frequency and the total number of scenarios per time unit.

2.1 Complete and Selected Profiles

There are two ways of specifying profiles for quality attributes, i.e. the complete and the selected profiles.When defining a complete profile for a quality attribute, the software engineer defines all relevant scenarios as part of the profile. For example, a usage profile for a relatively small system may include all possible scenarios for using the system (perhaps excluding exceptional situations). Based on this complete scenario set, the software engineer is able to perform an analysis of the architecture for the studied quality attribute that, in a way, is complete since all possible cases are included.

It should, at this point, be clear to the reader that complete profiles only work in a limited number of cases. It requires systems to be relatively small and one is only able to predict complete scenario sets for some quality attributes. For instance, in order to predict the maintainability of the system, the definition of a complete profile assumes that one is able to define all changes that will be required from the system during its operation, or during a predefined period of time. It is safe to assume that this is impossible in all but highly artificial situations.

The alternative to complete profiles are selected profiles. Selected profiles are analogous to the random selection of sample sets from populations in statistical experiments. Assuming the sample set is selected according to some requirements, among others randomness, the results from the sample set can be generalized to the population as a whole. One of the major problems in experiment-based research is the truly random selection of elements from the population, because of practical or ethical limitations. However, even if one does not succeed to achieve random selection, but is forced to make a structured selection of elements for the sample set, the research methodology developed for a weaker form of experimentation, known as quasi-experimentation, allows one to still make scientifically validated statements.

The notion of selected profiles makes use of the above principles. Assuming a large population of possible scenarios, e.g. change scenarios, one selects individual scenarios that are made part of the sample set. A complete sample set is what we, so far, have referred to as a profile. Assuming the selection of scenarios has been done careful, one can assume that the profile represents an accurate image of the scenario population. Consequently, results from architecture analysis based on the profile, will provide accurate results for the system and not just for the profile.

2.2 Defining Profiles

The most difficult issue in defining a profile is, obviously, the selection of scenarios that become part of the profile. Since these scenarios are selected and defined by software engineers and other stakeholders, it is hard to claim that this process is random. In our experience, totally unsupported selection and definition of scenarios leads, in some cases, to situations where particular types of scenarios, e.g. changes to the user interface, become overrepresented. To address this, we divide the process of profile specification into two main steps:

- **Definition of scenario categories**: The first step in profile specification is the decomposition of the scenario ‘population’ into a number of smaller populations that cover particular aspects of the system. For instance, in the case of usage profiles, one may identify different users of the system, e.g. local user, remote user, operator, etc. To give a second example, in the case of maintenance profiles, one may identify changes to the different interfaces to the context of the system, e.g. the hardware, the communication protocols, the user-interface, etc. In our experience, we normally define around 6 categories, but this is heavily dependent on the type of system and the intentions of the software architects for defining the profiles.
- **Selection and definition of scenarios for each category**: In the second step, the software architects select, for each category, a set of scenarios that is representative for the sub-population. Of course, we moved the problem of representativeness one level down from the profile as a whole to the category. However, in our experience, when dealing with a particular category, e.g. hardware changes in a maintenance profile, it is considerably easier to cover all relevant aspects in that category. In addition, even if the scenarios within a category are not representative, the resulting profile will still be closer to the ideal compared to not using categories. Finally, in our experience, we select and define up to 10

scenarios per category for quality attributes that are crucial for the system and 3 to 5 scenarios for important quality attributes.

The fact that humans are part of the process of selecting scenarios and categories can be considered a weakness, especially compared to automated random selection. However, it is not possible to perform random selection for the definition and the alternative is to not use scenarios and architecture analysis at all. And, as discussed in the introductory chapter, we know what the lack of assessment early in the design process leads to.

Once the categories and scenarios that are part of the profile have been selected and defined, the next step is to assign weights to the scenarios. The weights indicate slightly different things for different profiles, e.g. in the case of a usage profile, the weight indicates the relative frequency of executing a particular scenario whereas, in the case of a maintenance profile, the weight indicates the predicted relative likelihood of a particular change scenario.

There exist many different approaches to assigning weights to scenarios. For instance, scenarios can be rated on a scale from 1 to 10 or 1 to 100 or an approach using (--, -, o, +, ++) can be used. We have no preferences on the particular approach to use, but we do require that the approach used is quantifiable and that the weights can be converted to relative weights.

For example, assume that the scenarios in a profile have been rated on a scale 1 to 10. Once all weights have been assigned, we calculate the relative weight by adding the scores of all scenarios to achieve a total X. The relative weight of each scenario is than calculated as assigned weight divided by X. This results in a number between 0 and 1 and indicates the relative importance of the scenario in the profile. The sum of relative weights of all scenarios in the profile is, obviously, 1. Later in this chapter, we will describe how the weights are used in architecture assessment.

2.3 Quality Attribute Profiles

In the disposition, we have, up to this point, implicitly indicated that each quality attribute has an associated profile. Although this is true for several system attributes, some profiles can actually be used for assessing more than one quality attribute. In this section, we briefly describe the most important quality attributes and their associated profiles.

The following quality attributes can be considered as the most relevant from a general software system perspective.

- **Performance**: The general efficiency with which the system performs its functionality, measured in throughput, i.e. number of use scenarios per time unit, or the response time of use scenarios, is generally considered to be a very important property of any software system. Especially in large systems, the architecture plays a central role in achieving high performance since performance bottlenecks are not caused by the actual computation related to the domain functionality, but are due to context switches, synchronization points, dynamic memory management, etc.

  Profile: Usage profile describing either a complete or selected set of functional scenarios, describing a particular instance of system usage by one of the users. Scenario categories generally decompose the use scenario space based on user types and/or system interfaces. Scenario weights represent the relative frequency of the scenario.

  Architecture description data: The architecture should contain, in addition to the base information concerning component and component functionality, the system behavior in response to the use scenarios in the profile, the required computation at each component, the average (and worst-case) delays due to e.g. synchronization and the general overhead in the system. This information can either be generated by the software architects based on intelligent guessing or based on historical data from existing systems.

- **Maintainability**: Similar to performance, also maintainability is a quality attribute fundamentally affected by the architecture of a software system. The way the system functionality is decomposed into components leads to highly different efforts in response to requirements changes. This because a requirement change may lead to a local change in one architecture and may require changes in several components in another architecture. There are considerable cost differences between the two examples, due to the additional effort of changing more than one component, but also due to the increased architecture erosion and other effects. Finally, there are requirement changes that have so-called architectural impact, i.e. they require the architecture to be changed. It should be possible to incorporate all likely requirement changes without changes to the software architecture since changes with architectural impact tend to be prohibitively expensive.

  Profile: Maintenance profile, consisting of change categories and change scenarios. Change categories tend to be organized around the interfaces the system has to its surroundings. Generally, the hardware and operating system, the interface to other systems the system under design is intended to communicate with and the user interface to each of the user types tend to become change categories. The change scenarios describe concrete requirement changes that lead to changes to the software. The relative weight of a scenario indicates its relative likelihood, i.e. the chance that the scenario (and the scenarios it represent) occur during a time period. Note that since we assume that multiple requirement changes may take place during this time period, it is more than possible that, especially likely, change scenarios happen multiple times. This does not mean that the concrete change scenario happens more than once (the requirement change can only be incorporated once), but that multiple concrete scenarios may occur from the set represented by the change scenario. Since these scenarios are assumed to have a similar impact, it does not matter for the assessment which concrete scenario is counted.

Architecture description data: As we will see in the examples later on in the chapter, change scenarios are evaluated with respect to their impact on the architecture. Impact is calculated in terms of number of lines of code that have to be changed. In order to determine this, the estimated size of the architectural components in terms of lines of code needs to be available for maintainability assessment.

- **Reliability**: More than the quality attributes described earlier is reliability a function of many factors, including the architecture, the detailed design, the implementation, the education and experience levels of the people involved, etc. During architecture assessment, we are, naturally, primarily concerned with the architectural dimension of reliability. The architectural aspect of reliability is concerned with the component interaction during operation and the effects of component errors on the system reliability as a whole. Use scenarios that use multiple components will have a lower reliability than use scenarios using fewer components since the combined component reliabilities lead to lower use scenario reliabilities. (*** Need to give Claes a call/email to figure out how this works ***)

  Profile: Usage profile, where the use scenarios are evaluated with respect to the component reliabilities, resulting in use scenario reliabilities. Based on these reliabilities and the relative weights, a system reliability can be calculated.

  Architecture description data: In addition to the knowledge about the architectural components and information about the component interaction in response to each use scenario, the software architect needs component reliability data. This data can either be based on historical data, on intelligent guessing or the reliability data can be converted into component requirements. The component designers and implementers then need to assess component reliability and make sure that the required levels are achieved.
- **Safety**: Safety is concerned with the negative or even destructive effects the system under design may have on the real-world and entities in the real-world. It is, consequently, not concerned with the system functionality, but with the effects the system may have on the real-word. The effects may be physical, e.g.

a dialysis machine not detecting air bubbles in the extra-corporeal blood flow, but need not be, e.g. a banking system performing incorrect money transfers, thereby damaging customers and/or the banking corporation itself. In addition, the safety requirement is supposed to both detect internal system errors and incorrect system input.

Profile: Hazard profile, containing hazard scenarios, i.e. situations where not detecting a fault may lead to negative or disastrous consequences. The hazard categories can be organized according to certification documents, as is the case for medical systems, the interaction points of the system with the real-world or the critical system components, whose failure may lead to hazard situations.

Architecture description data: Safety is often an issue in embedded systems, i.e. systems including mechanical, hardware and software parts. Since safety is primarily a system attribute and the software safety is derived from the system safety, the relation between system architecture and the software architecture should be explicit and clear.

- **Security**: Software systems may become subject to unintentional or intentional attempts to access the system or parts of the system by unauthorized entities, be it other systems or persons. Security is not just an issue for military (intelligence) systems, but is relevant for most business and governmental systems since most organizations both have the right and the obligation to keep data inaccessible for all but authorized users. This includes the situation where users of the system are only authorized to perform those tasks that are part of their role within the organization. Especially intentional access attempts by unauthorized entities can take many forms, including attacks at the level of system hardware, but there is definitely an architectural component in security. One issue is that security should preferably be handled consistent throughout the system; another being that ‘compartmentalization’ needs to be handled at the system level. Finally, software security is part of system security, which in turn may be part of other security schemes.

  Profile: Authorization profile, at least divided in a matrix of two by two main categories, i.e. unintentional versus intentional unauthorized access attempts and internal versus external unauthorized access attempts. In addition, all system interfaces can be used for categorization. The usage profile may be used as a secondary profile to provide information on the actual usage of the system.

  Architecture description data: Except for the base architecture description data, no additional information is required. (**check **)

2.4 Example

To illustrate the selection and definition of a profile, we use the dialysis system as an example. As described in chapter 2, one of the driving quality attributes for the dialysis system is *maintainability*. The company had learned the hard way the importance of developing maintainable software. In table 1, a summary of the maintenance profile for the dialysis system is shown. Six change categories are presented, i.e. changes driven by the market, the hardware, safety regulations, medical advances, communication and I/O and, finally, algorithm implementation. For each of the scenario categories, one or a few scenarios are presented. The example is discussed in more detail in [Bengtsson & Bosch 99].

Table 1: Dialysis System Maintenance Profile

Category	Scenario Description	Weight
Market Driven	C1 Change measurement units from Celsius to Fahrenheit for temperature in a treatment.	0.043
Hardware	C2 Add second concentrate pump and conductivity sensor.	0.043
Hardware	C3 Replace blood pumps using revolutions per minute with pumps using actual flow rate (ml/s).	0.087
Hardware	C4 Replace duty-cycle controlled heater with digitally interfaced heater using percent of full effect.	0.174
Safety	C5 Add alarm for reversed flow through membrane.	0.087
Medical Advances	C6 Modify treatment from linear weight loss curve over time to inverse logarithmic.	0.217
Medical Advances	C7 Change alarm from fixed flow limits to follow treatment.	0.087
Sum		1.0

Table 1: Dialysis System Maintenance Profile

Category	Scenario Description	Weight
Medical Advances	C8 Add sensor and alarm for patient blood pressure	0.087
Com. and I/O	C9 Add function for uploading treatment data to patient's digital journal.	0.043
Algorithm Change	C10 Change controlling algorithm for concentration of dialysis fluid from PI to PID.	0.132
Sum		1.0

2.5 Summary

The definition of profiles for the quality attributes considered most relevant for the software architecture design allows for concrete and precise description of the meaning of statements about quality attributes. For instance, a maintenance profile describes what changes are most likely to occur and should be easy to incorporate. For instance, using the profile as input, the software architect is able to optimize the architecture for the most likely changes, thereby improving the maintainability of the system.

In order to achieve the above situation, however, the software architecture needs to have two important tools available, i.e. techniques to assess the quality attributes and techniques to transform the architecture to improve its quality attributes. These two issues are the subject of the current and next chapter, respectively.

The software architect can decide to define a complete scenario for a particular quality attribute, but often this is not feasible and one is required to define a selected profile. The definition of a selected profile for a quality attribute consists of the following steps:

- **Define scenario categories**: As a first step, the scenario population for the quality attribute is divided into categories. Generally speaking, five to six categories can be used, but this depends on the type of system.
- **Define scenarios**: For each category, a set of scenarios is selected by the software architect that cover the category as well as possible. In our experience, up

to ten scenarios can be used for a detailed assessment, but three to five suffice for obtaining a good indication.

- **Assign weights**: Each scenario is assigned a weight indicating its ‘likelihood’. For instance, in the case of performance, the weight denotes the relative frequency of the scenario, whereas in the case of maintainability, the weight expresses the chance that this change scenario occurs. No approach to assigning weights is enforced, but, as a minimum, the weights should quantifiable.
- **Normalize the weights**: To simplify the use of the profile in the assessment techniques described in the subsequent sections, the weights of the scenarios are normalized, such that the sum of all scenario weights is one.

3. Scenario-based Assessment

In the remainder of this chapter, a number of approaches to architecture assessment are discussed. The approaches have different advantages and disadvantages, but tend to complement each other. In this section, we discuss scenario-based assessment of software architectures.

Scenario-based assessment is directly depending on the profile defined for the quality attribute that is to be assessed. The effectiveness of the technique is largely dependent on the representativeness of the scenarios. If the scenarios form accurate samples, the evaluation will also provide an accurate result. Scenario-based assessment of functionality is not new. Several object-oriented design methods use scenarios to specify the intended system behavior, e.g. use-cases [Jacobsen et al. 92] and scenarios [Wirfs-Brock et al. 90]. The main difference to the object-oriented design methods is twofold. First, we use scenarios for the assessment of quality attributes, rather than for describing and verifying functionality. Second, in addition to use scenarios, we also use other scenarios that define other quality attributes, e.g. change and hazard scenarios.

If, however, the software architect decides to use traditional use-cases during architectural design and the use profile is a selected profile, it might be important to define the use-cases independent of the use profile. The reason is that the architecture design will, most likely, be optimized for the set of use-cases. If the set of use-cases and the use profile are the same, then one can no longer assume that the assessment of the architecture based on the profile is representative for the scenario population as a whole. However, while developing the scenarios, it is not necessary to develop two sets. The two sets could be generated later by randomly dividing the initially specified set of scenarios. In addition, depending on the system, it might be necessary to develop new scenarios for evaluation purposes if the design is iterated a number of times.

Scenario-based assessment can be used for comparing two architectures and for an absolute assessment of a single architecture. The main difference is the amount of quantitative data or estimates necessary to perform the assessment, which is considerably larger in the latter form. Below, we will describe first describe the absolute assessment and then discuss how one can scale down the approach for comparative assessment.

(** figure presenting scen. based assessment process **)

Scenario-based assessment consists of two main steps:

- **Impact analysis**: As an input to this step, the profile and the software architecture are taken. For each scenario in the profile, the impact on the architecture is assessed. For a change scenario, the number of changed and new components and the number of changed and new lines of code could be estimated. For performance, the execution time of the scenario could be estimated based on the path of execution, the predicted component execution times and the delays at synchronization points. The results for each scenario are collected and summarized.
- **Quality attribute prediction**: Using the results of the impact analysis, the next step is to predict the value of the studied quality attribute. For performance, the scenario impact data can be used to calculate throughput by combining the scenario data and the relative frequency of scenarios. For maintainability, the impact data of the change scenarios allows one to calculate the size in changed and new lines of code for an average change scenario. Using a change request frequency figure that is either estimated or based on historical data, one can calculate a total number of changed and new lines of code. Using historical data within the company or figures from the research literature, the software architect can calculate the maintenance cost by, for instance, multiplying the number of work hours per maintained line of code with the total number of maintained lines of code.

To illustrate scenario-based assessment, we present maintainability assessment and an associated maintenance profile as an example. We use dialysis system example discussed in chapter 2 and the maintenance profile in table 1. For each of the change scenarios in the maintenance profile, an impact analysis is performed. In this case, the changed code is expressed in the percentage of lines of code affected in an existing component. The component sizes are taken from a small prototype that was constructed as part of the joint research project with the involved companies, i.e. EC-Gruppen and Althin Medical. The result of the impact analysis is shown in table 2.

Table 2: Impact Analysis per Scenario

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

Using the impact analysis data, we can calculate the average number of lines of code (LOC) for a change scenario:

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

Assuming a number of change requests per year of 20 and an average productivity of 1 LOC/hour (which is high for medium-sized and large systems, where studies have shown a productivity of 0.2 LOC/hour), one can calculate a maintenance effort for the architecture of:

20 change requests * 145 LOC/change request * 1 hour/LOC = 2800 hours

This equals about 1.5 person working full-time on the maintenance of the software system. Note that these figures incorporate all maintenance related activities, including updating the requirement specification, design documents and user documentation, the actual design and implementation, regression testing and testing of the new functionality and the deployment of the software.

As mentioned earlier, scenario-based assessment can also be used for the comparison of two or more alternative architectures. In that case, the quantification of impact data is less important and no prediction of quality attribute is done. Instead, for each architecture, impact data is collected, which can be on a (--, ..., ++) scale and summarized. Using this model, each architecture is assigned a score and the software architect can select the most appropriate architecture by comparing the scores. However, the disadvantages of relative assessment discussed in the introduction remain.

Finally, in our experience, scenario-based assessment is particularly useful for development quality attributes. Quality attributes such as maintainability can be expressed very naturally through change scenarios. Although operational quality attributes, such as performance, can be assessed using this technique as well, we have experienced that other assessment techniques are sometimes preferable.

Summarizing, scenario-based assessment can be used both for relative (or comparative) and absolute assessment. It consists of two major steps, i.e. impact analysis, taking the software architecture and the profile as input and generating impact data, and quality attribute prediction, using the impact data to make a statement about the level of the quality attribute. In the case of comparative assessment, the quality attribute prediction phase does not lead to a figure, but to a ranking of alternative architectures.

4. Simulation-based Assessment

The assessment technique discussed in the previous section is rather static in that no executing dynamic model is used. An alternative is *simulation-based assessment* in which a high-level implementation of the software architecture is used. The basic approach consists of an implementation of the components of the architecture and an implementation of the context of the system. The context, in which the software system is supposed to execute, should be simulated at a suitable abstraction level. This implementation can then be used for assessing the behavior of the architecture under various circumstances.

Once a simulated context and high-level implementation of the software architecture are available, one can use scenarios from the relevant profiles to assess the relevant quality attributes. Robustness, for example, can be evaluated by generating or simulating faulty input to the system or by inserting faults in the connections between architecture entities.

The process of simulation-based assessment consists of a number of steps:

- **Define and implement the context**: The first step is to identify the interfaces of the software architecture to its context and to decide how the behavior of the context at the interfaces should be simulated. It is very important to choose the right level of abstraction, which generally means to remove most of the details normally present at system interfaces. For instance, for an actuator in the context of the system requiring a duty cycle to be generated, the simulated actuator would accept, for instance, a flow or temperature setting.

  Especially for embedded systems where time often plays a role, one has to decide whether time-related behavior should be implemented in the system. For instance, when increasing the effect of the heater in the dialysis system, the actual water temperature will increase slowly until a new equilibrium is met. The software architect has to decide whether such delays in effects between actuators and sensors should be simulated or not. This decision is depending on the quality attributes that the software architect intends to assess and the required accuracy of the assessment.

  Finally, for architecture simulation as a whole, but especially for the simulation of the system context, one has to make an explicit balance between cost and benefit. One should only implement at the level of realism required to perform the assessments one is interested in.
- **Implement the architectural components**: Once the system context has been defined and implemented, the components in the software architecture are con-

structed. The description of the architecture design should at least define the interfaces and the connections of the components, so those parts can be taken directly from the design description. The behavior of the components in response to events or messages on their interface may not be specified as clearly, although there generally is a common understanding, and the software architect needs to interpret the common understanding and decide upon the level of detail associated with the implementation.

Again, the domain behavior that is implemented for each component as well as the additional functionality for collecting data is again dependent on the quality attributes that the software architect intends to assess. For some quality attributes, additional architecture description data needs to be associated with the components. For instance, in the case of performance assessment, estimated execution times may be associated with operations on the component interfaces. In that case, generally a simulated system clock is required in order to be able to calculate throughput and average response time figures.

- **Implement profile**: Depending on the quality attribute(s) that the software architect intends to assess using simulation, the associated profile will need to be implemented in the system. This generally does not require as much implementation effort as the context and the architecture, but the software architect should be able to activate individual scenarios as well as run a complete profile using random selection based on the normalized weights of the scenarios.

  For example, in virtually all cases, the use profile needs to be implemented. The software architect generally is interested in performing individual use scenarios to observe system behavior, but also to simulate the system for an indefinite amount of time using a scenario activator randomly selecting scenarios from the profile.

- **Simulate system and initiate profile**: At this point, the complete simulation, including context, architecture and profile(s) is ready for use. Since the goal of the simulation is to assess the software architecture, the software architect will run the simulation and activate scenarios in a manual or automatic fashion and collect results. The type of result depends on the quality attribute being assessed.

  It is important to note that for several quality attributes, the simulation will actually run two profiles. For instance, for assessing safety, the system will run its use profile in an automated manner, and the hazard scenarios will, either manually or automatically, be activated. Whenever a hazard scenario occurs, data concerning the system context is collected. For example, in the dialysis system example, all values relevant to the safety of the patient, e.g. blood temperature, concentrate density, air bubbles, heparin density, etc. are collected from the simulation to see if any of these values, perhaps temporarily, exceed safety boundaries.

- **Predict quality attribute**: The final step is to analyze the collected data and to predict the assessed quality attribute based on the data. Depending on the type of simulation and the assessed quality attribute, excessive amounts of data may be available that need to condensed. Generally, one prefers to automate this task by extending the simulation with functionality for generating condensed output or using other tools.

  To give an example, for performance assessment, the system may have run tens (or hundreds) of thousands of use scenario instances and collected the times required to execute the scenario instances. All this data needs to be condensed to average execution times per scenario, perhaps including a standard deviation. This allows one to make statements about system throughput, scenario-based throughput and average response times of individual scenarios.

Simulation of the architecture design is not only useful for quality attribute assessment, but also for evaluating the functional aspects of the design. Building a simulation requires the engineer to define the behavior and interactions of the architecture entities very precise, which may uncover inconsistencies in the design earlier than when using traditional approaches. In our experience, it is extremely useful to be forced to express exactly what the functionality of an architecture component should be. Simulation has been used in all three examples systems described in chapter 2 and the experiences have been very positive. Although there is an overhead involved in the implementation of, especially, the system context, our experience is that the advantages easily outweigh these.

The fact that an executable specification of the system is available early in the design process often proves to be highly relevant. It is important to note that it is possible to have a simulation of the system available during the whole design process. This is achieved through iterative refinement of the simulated context and the system implementation. Using this approach, the software architect iteratively details the design and the context in which the design operates until the context is no longer simulated, but the actual context. If this level is achieved, the system implementation is also complete. It is possible to use parts of the actual system context early in the design and use simulated parts where the system is not sufficiently detailed yet. Although our experience is that this approach is very interesting for, especially, the implementation of embedded systems with high availability and strict safety requirements, it is outside the scope of this book. We refer to, among others, [ref] for more elaborate discussions.

Simulation complements the scenario-based approach in that simulation is particularly useful for evaluating operational quality attributes, such as performance of fault-tolerance by actually executing the architecture implementation, whereas scenarios are more suited for evaluating development quality attributes, such as maintainability and flexibility.

Simulation can be used for the assessment of reliability in two ways. First, using component reliability figures, one can simulate component failures during automated execution of the usage profile and collect reliability figures for the architecture based on this data. Secondly, some research results, e.g. [ref-claes-SDL] has shown that correlations exist between the reliability of specifications and the systems implemented based on those specifications. Thus, the reliability of the architecture implementation in the simulation gives an indication of the reliability of the final system.

Although simulation is primarily suited for the assessment of operational quality attributes, the implementation of the architecture in the simulation can be used to evaluate development quality attributes, such as maintainability. Once the simulattion is available, the software architect can actually implement scenarios from the maintenance profile and measure the required effort and identify the affected components and the extent of change. Using this data, one can extrapolate the required effort for the complete system.

Finally, the accuracy of simulation-based assessment depends on a number of factors. First, the accuracy of the profile used to assess the quality attribute. Second, it is dependent on how well the simulated system context reflects real world conditions. Finally, the relation between the architecture implementation and the implementation of the final system. Consequently, the factors influencing accuracy are considerably more than for scenario-based assessment, but the early availability of an executable implementation of the system has, as discussed, several advantages as well.

*** example - measurement system paper? Main problem, nice functional model, but what quality attribute? ***

Summarizing, simulation-based assessment makes use of an executable model of the software architecture and a simulation of the context of the system. Using the model and its context, one can execute the profile for the quality attribute that is to be assessed. Using the data collected during profile execution, one can predict the level of quality attribute for the software architecture. The simulation-based assessment process consists of the following steps:

- Define and implement the context
- Implement the architectural components
- Implement profile
- Simulate system and initiate profile
- Predict quality attribute

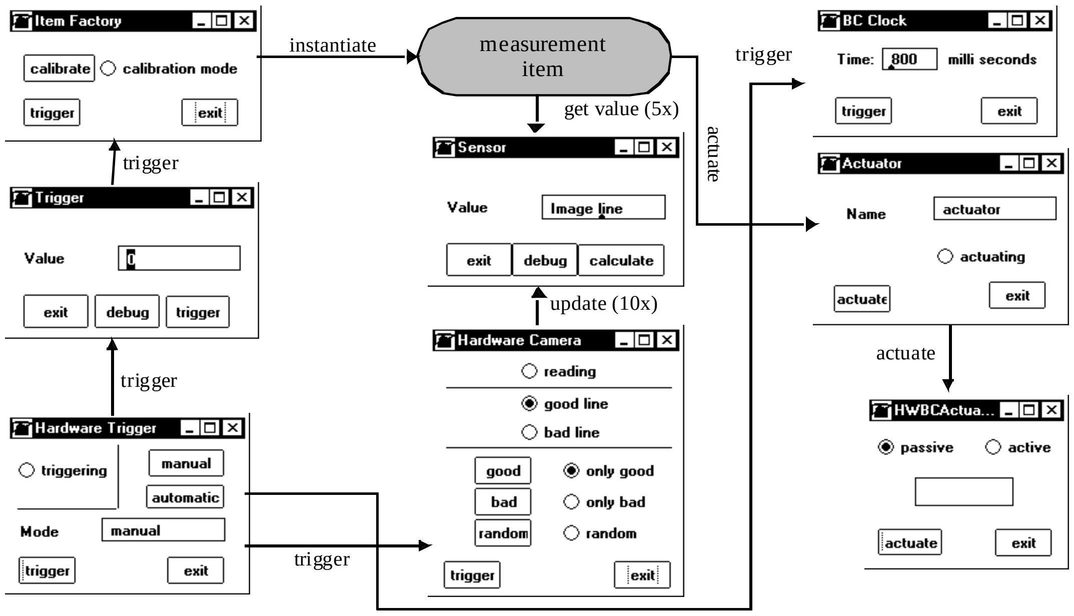


FIGURE 12. Simulation of a beer can inspection system

5. Mathematical Model-based Assessment

Various research communities, e.g. high-performance computing [Smith 90], reliable systems [Neufelder 93], real-time systems [Liu & Ha 95], etc., have developed mathematical models that can be used to evaluate especially operational quality attributes. Different from the other approaches, the mathematical models allow for static evaluation of architectural design models. For example, performance modeling is used while engineering high-performance computing systems to evaluate different application structures in order to maximize throughput. Using, for instance, queuing network theory [Smith 90], the software engineer can develop a mathematical representation that can be analyzed.

Mathematical modeling is an alternative to simulation since both approaches are primarily suitable for assessing operational quality attributes. However, the approaches can also be combined. For instance, performance modeling can be used to estimate the computational requirements of the individual components in the architecture. These results can then be used in the simulation to estimate the computational requirements of different use scenarios in the architecture.

The process of model-based assessment consists of the following steps:

- **Select and abstract a mathematical model**: As mentioned in the introduction, most quality attribute-oriented research communities have developed mathematical models for assessing ‘their’ quality attribute. The models are generally well-established, at least within the community, but tend to be rather elaborate in that much, rather detailed, data and analysis is required. Consequently, part of the required data is not available at the architectural level and the technique requires too much effort for architecture assessment as part of an iterative design process. Thus, the software architect is required to abstract the model. This may result in less precise prediction, but that is, within limits, acceptable at the software architecture design level.
- **Represent the architecture in terms of the model**: The mathematical model that has been selected and abstracted does not necessarily assume that the system it models consists of components and connections. For instance, real-time task models assume the system to be represented in terms of tasks. Consequently, the architecture needs to be represented in terms of the model.
- **Estimate the required input data**: The model, even when abstracted, often requires input data that is not included in the basic architecture definition. This data, then, has to be estimated and deduced from the requirement specification and the designed software architecture. For instance, real-time task models require data about, among others, priority, frequency, deadline and computational requirements of tasks and information about the synchronization points in the system.
- **Predict the quality attribute**: Once the model is defined, the architecture expressed in terms of the model and all required input data available, the software architect is able to calculate the resulting prediction for the assessed quality attribute. In some cases, for instance non-trivial performance assessments based on the performance engineering method developed by [Smith 90] may require more advanced approaches.

The software metrics research community has developed a variety of product metrics of which at least part can be used for the assessment of software architectures. Validation of several of these metrics has shown correlation to quality attributes. For instance, McCabe's cyclomatic complexity metric has shown correlation to the maintenance cost of software systems. However, it is important to note that correlations are statistical relations, meaning that although true on the average, a metric does not have to predict accurately for the software architecture at hand.

Different from the two aforementioned assessment techniques, mathematical model-based assessment does not make use of profiles, or at least not of selected profiles. This means that the assessment technique provides an immediate translation from the software architecture to the quality attribute, without incorporating the actual meaning of the quality requirement as specified by the customer. (** extend argument **)

*** example: real-time task model a la Liu for, e.g. fire alarm system? ***

Concluding, mathematical model-based assessment provides an alternative primarily to simulation-based assessment in that these models are primarily available for operational quality attributes. Which assessment technique to choose depends upon, at least, two factors. First, no appropriate mathematical models may be available for the relevant quality attributes. Second, although the development of a complete simulation model may require substantial effort, but if the model is used to assess more than one quality attribute, the effort can be divided. Mathematical models are unique for each quality attribute.

6. The Role of Experience

In the previous sections, we have discussed three approaches to *quantitative* architecture assessment. The reason we stress these approaches is because we hope to progress the state of the art towards quantitative, objective assessment rather than the current state-of-practice, that often is subjective and qualitative. However, it is by no means our intention to diminish the value of architecture assessment through objective reasoning based on earlier experiences and logical argumentation. At numerous occasions, we have encountered experienced software architects and engineers who provided valuable insights that proved extremely helpful in avoiding bad design decisions. Although some of these experiences are based on anecdotal evidence, most can often be justified by a logical line of reasoning.

This approach is different from the other approaches in that the evaluation process is less explicit and more based on subjective factors such as intuition and experience. The value of this approach should, nevertheless, not be underestimated. Most software architects we have worked with had well-developed intuitions about 'good' and 'bad' designs. Their analysis of problems often started with the 'feeling' that something was wrong. Based on that, an objective argumentation was constructed either based on one of the aforementioned approaches or on logical reasoning. In addition, this approach may form the basis for the other evaluation approaches. For example, an experienced software engineer may identify a maintainability problem in the architecture and, to convince others, define a number of scenarios that illustrate this.

To give an example, during the design of the fire alarm system architecture, it was identified that the system is inherently concurrent. Consequently, it was necessary to choose a concurrency model. Earlier experience by some team members in earlier small embedded systems had shown that fine-grain concurrency with a preemptive scheduler could be error-prone considering the possibility of race conditions. The argumentation by those team members convinced the team as a whole and lead to a transformation of the architecture that was later named the Point pattern [Molin and Ohlsson 97?]. In the next chapter, the Point pattern and the associated architecture transformation are discussed in more detail.

7. Performing Architecture Assessment

So far, in this chapter, we have primarily discussed techniques and approaches that can be used as part of architecture assessment, but we have not shown how these parts are integrated in a full-scale architecture assessment process. Although the reader may have deduced this from the discussion in this and previous chapters, we will here explicitly define the main steps in architecture assessment.

First, one should observe that architecture assessment is an iterative activity that is part of an iterative design process. Once the architecture is assessed for the first time, it will enter the transformation phase, assuming it does not already fulfil all its requirements. After transformation, the architecture will, again, be assessed for its quality attributes.

The first time the architecture is assessed, or possibly even before functionality-based design is performed, the profiles for the relevant quality attributes should be defined. It is important no notice that it is generally not feasible nor useful to assess all or many quality attributes. As in any engineering discipline, the benefit should outweigh the cost for each activity. Since both the definition of the profile and the, repetitive, assessment process are time-consuming activities, only those quality attributes should be selected for explicit assessment that are crucial for system success and for which it is unclear whether they will be fulfilled.

Once the relevant quality attributes have been selected and the profiles for these quality attributes have been defined, the next step is to select an assessment technique. As a very general rule, our experience is that development quality attributes are generally most easily assessed using a scenario-based approach whereas operational quality attributes can be assessed using either simulation-based assessment or a mathematical or metrics-based model. However, each system is unique and may require deviation from this general rule. In certain cases, one may decide to use two techniques to assess the same quality attribute. This allows the software architect to cross-reference results and to increase confidence in the assessment or, alternatively, investigate inconsistencies.

The above steps are generally performed once during architecture design, for instance the first time assessment of the architecture is performed. During the design iterations, the actual architecture assessment is performed during every iteration and for each quality attribute. Assuming one is able to achieve quantitative predictions for each quality attribute, the result is a table containing, for each version of the architecture, the required level, the predicted level and an indication for each quality attribute. The indication may simply show that the attribute is or is not fulfilled, but also that the attribute needs to be renegotiated with the customer or that a, generally negative, relation exists to another quality attribute.

Concluding, the process of architecture assessment can be divided into two components, i.e. a part that is performed once and a part that is executed for every design iteration:

- Select the relevant quality attributes and define the required levels
- Define a profile for each quality attribute
- Select an assessment technique for each quality attribute

For each design iteration:

- Perform the quality attribute assessment for the current version of the architecture
- Assemble the results and decide upon continuation, renegotiation or termination of the design project

8. Concluding Remarks

Assessment of software architectures is the process of predicting quality attributes of the system developed based on a software architecture. We have identified three different goals with architecture assessment. First, *relative assessment* is used to compare two alternative architectures. Although useful, the disadvantage of relative assessment is that when comparing alternative architectures for more than one quality attribute, one has only 'boolean' data to base the selection on. Second, the software architect can perform *absolute assessment* resulting in quantitative statements about the quality attributes for the assessed architecture. This allows the software architect to decide whether the requirements are met by the assessed architecture. However, absolute assessment provides no means to determine upon the theoretical limits for the architecture and the distance between the current level and the theoretical maximum or minimum. The third goal of architecture assessment is to determine the theoretical maximum of a software architecture for a particular quality attribute. In our experience, techniques are available for the first two assessment goals, but no work, to the best of our knowledge, is currently available with respect to the third goal.

The meaning of quality requirements in the requirement specification is often rather vague and imprecise. In this chapter, we propose to define scenario profiles that define the meaning of quality requirements more precisely. Two approaches to defining profiles exist, i.e. complete profiles and selected profiles. The first defines all relevant scenarios for a particular quality attribute, whereas the second selects a limited number of scenarios from a large population of possible scenarios. To structure the selection process, scenario categories are defined to divide the population into parts.

We have presented three architecture assessment techniques, i.e. scenario-based, simulation-based and mathematical model-based architecture assessment. The scenario-based approach assesses the impact of the scenarios in the profile and predicts the quality attribute based on the impact data. Simulation-based assessment develops an abstract system context that is simulated and a high-level implementation of the architecture. Generally, for practical reasons, also the profile that is used for the assessment is implemented. During the simulation, relevant data is collected and the quality attribute can be predicted using the collected data. Finally, the software architect can use a, often adapted, mathematical model developed by one of the quality attribute research communities. The adapted model can be used to predict the quality attribute.

We have discussed the importance of experience in software architecture assessment and design. Although our goal is to improve the state of practice by providing objective and quantitative means to reason about architectures, it is explicitly not our intention to diminish the value of experience and creative insight in the architecture design process. Experienced and creative software architects and engineers are a necessary ingredient in any successful software development project.

Finally, we have briefly mentioned the overall software architecture assessment process. This process can be divided in two parts. The first part is performed once during the design of a software architecture and includes activities such as selecting and defining the relevant quality attributes, developing the associated profiles and selecting an assessment technique for each quality attribute. The second part of the process is performed for each iteration of the architecture and consists of performing the assessment for the relevant quality attributes, collecting the results and to decide upon continuation, renegotiation or termination of the design project.

9. Further Reading

< to be written >

CHAPTER 5

Transformation of Software Architectures

The approach to architectural design presented in this book consists of three major phases, i.e. functionality-based architectural design, architecture assessment and architecture transformation. In the previous chapters, the first two phases have been presented. In this chapter, the notion of transforming an architecture to improve one or more of its quality attributes is discussed. We identify four types of architectural transformations that can be used change the properties of the system. The transformations are illustrated using the example software architectures discussed in earlier chapters.

1. Introduction

During functionality-based architecture design, the structure of the system has been determined by the application domain and the functional requirements. The identified archetypes and the system instances described using the archetypes are based on the software architect's perception of the domain. Since the perception of the software architect is largely formed based on the culture in which the architect lives and the education that he or she has received. Consequently, it is likely that other software architects and engineers will share the perception of the software architect that designed the initial version of the architecture. The fact that domain understanding is shared among, at least, the software engineering community is an important property of the functionality-based design, since it allows for easy communication between members of the community. Thus, when software development based on the software architect design is initiated, it is relatively easy for the software architect to explain the important concepts underlying the design. In addition, if the software architect disappears during the development, the persons taking over will have an easier task to understand the architecture design and maintain the conceptual integrity [Brooks 95]. Finally, during maintenance, software maintainers will have easier understanding of the constraints, rules and rationale underlying the architecture, thereby maintaining conceptual integrity and, consequently, slow the software aging process.

Thus, the functionality-based design of the architecture is based on domain analysis that is formed by the culture and education and thus, up to some extent, shared by, at least, the software engineering community. As mentioned, this has important advantages. However, there is an important issue that has remained implicit: the functionality-based architecture design may not fulfil the quality requirements put on the system! Performance, maintainability and other quality attributes of the architecture may not be satisfactory.

Assessment of the software architecture, as discussed in the previous chapter, is performed to collect information on the quality attributes of the architecture so that these can be compared to the requirements. If one or more of the quality requirements are not satisfied, the architecture has to be changed to improve the quality attributes. This is the process of architecture transformation, which is the topic of this chapter.

Architecture transformation requires the software engineer to analyze the architecture and to decide due to what cause the property of the architecture is inhibited. Often, the assessment generates hints as to what parts or underlying principles cause low scores. The assessment of the quality attributes is performed assuming a certain context, consisting of certain subsystems, e.g. databases or GUI systems and one or more operating systems and hardware platforms. Consequently, whenever a quality attribute is not fulfilled, one may decide to either make changes to the presumed context of the system architecture or to make changes to the architecture itself.

If it is decided that the software architecture, rather than the context or the requirement specification, should be changed the architecture is subjected to a series of one or more architecture transformations. Each transformation leads to a new version of the architecture that has the same domain functionality, but different values for its properties.

The consequence of architecture transformations is that most transformations affect more than one property of the architecture; generally some properties positively and others in a negative way. For instance, the *Strategy* design pattern [ref-Gamma] increases the flexibility of a class with respect to exchanging one aspect of its behavior. On the down-side, performance is often reduced since instances of the class have to invoke another object (the instance of the Strategy class) for certain parts of their behavior. In the general case, however, the positive effect of increased flexibility considerably outweighs the minor performance impact.

We have identified four categories of architecture transformations, organized in decreasing impact on the architecture, i.e. imposing an architectural style, imposing an architectural pattern, applying a design pattern and converting quality requirements to functionality. One transformation does not necessarily address a quality requirement completely. Two or more transformations might be necessary. In the sections below, each category is discussed in more detail.

Although the transformation of a software architecture is necessary to fulfil its quality requirements, there are two important disadvantages of changing the functionality-based architectural design. First, the transformed architecture will not be as close to the shared understanding of the domain, requiring software architects and engineers to spend more time to understand the ‘philosophy’ underlying the architecture. For instance, the functionality-based architectural design of the measurement system consists of four components. The object-oriented framework that we designed for the measurement system domain consists of more than 30 components. These components have been added through architecture transformations improving the quality attributes, but not changing the domain functionality represented by the architecture. Second, the design tends to blow up in the number of components. Most transformations will take one or a few components and reorganize the functionality by dividing it over more components. To use the aforementioned Strategy design pattern as an example, the pattern transforms one (class) into at least three classes, i.e. the original class without the factored out behavior, the abstract strategy class defining the interface and, at least, one concrete strategy class providing one variety of the factored out behavior. Since most transformations increase the number of components in the architecture, it easily becomes the case that an elegant and simple functionality-based architectural design is transformed into a large and complex set of components that bears no visible relation to the initial architecture. During the complete architecture design process, it is of crucial importance to keep things as simple as possible and to search for conceptual integrity, a notion hard to quantify but understood by each software engineer.

One may wonder whether the architecture design method presented in this book is not making design overly complicated. It is important to observe that we defined this method based on a number of architecture design projects that we have been involved in. Generally, researchers from our research group formed an architecture design team with software architects from the companies we cooperated with. We tried to reflect about the way the team and its members performed architectural design. Based on the experiences from the projects, we have made the implicit architecture design process as we experienced it explicit.

A second disadvantage that we try to attack in our explicit approach to architecture design is that ad-hoc architecture design approaches tend to lead to mismatches between the perceived problems and the actual problems and between problems and the solutions selected to address those problems. In several industrial architecture designs we have seen the use of styles, patterns or other solution technology that did not match with the problems, i.e. unfulfilled quality requirements, the project members were trying to solve. Subsequently, further analysis showed that the quality requirements the project members tried to solve were not the problematic quality attributes of the architecture. Converting the ad-hoc, implicit architecture design approach into an architecture design method explicitly organizing the process into a number of well-defined steps will help avoid problems as described above.

Finally, a note on the level of detail that should be achieved during architectural design. This is a function of the size of the system, the quality requirements and the required level of certainty. If the system is very large, architectural design is unable to penetrate the challenges of the individual parts of the system. The quality requirements, however, are the most important factor deciding on the level of detail. The *goal of architectural design* is to develop a software architecture that, with a sufficient level of certainty, will fulfil all its requirements, including the quality requirements. Thus, if the quality requirements are very challenging and close to the boundary of the technical capabilities, architectural design needs to go into considerable detail for the critical parts of the system, e.g. down to the behavior of individual classes. The important issue is not to avoid performing design tasks normally considered to be part of detailed design, but to make sure that the final system will fulfils its quality requirements in addition to its functional requirements.

The remainder of this chapter is organized as follows. The next section discusses the process of transforming a software architecture. Section 3 and onward discuss the categories of architecture transformations that we have identified. Section 7 discusses the balancing of requirements for the software architecture and the requirements for the components that are part of the architecture. The chapter is concluded in section 8 followed by a section discussing interesting material on the topic.

2. The Architecture Transformation Process

Before presenting the categories of architecture transformation techniques that we have identified, we present a process of architecture transformation that intends to put these categories into a context. Transforming the architecture according to an architectural style or a design pattern is not an independent event, but occurs as part of a larger process of problem identification, solution selection and solution application.

The first step of the process is to identify what quality requirements are not fulfilled by the current version of the software architecture. In addition, information about the discrepancy between the assessed level for the quality attribute and the requirement is collected. Based on difference between assessed level and the requirement for each quality attribute and their relative importance, we define a ranking of the quality attributes. The ranking indicates what quality attributes should be addressed first. Note that, as mentioned earlier, only those quality attributes are part of the assessment and transformation process that have explicitly been selected by the software architect as having crucial importance for the success of the system, normally up to about five attributes. Finally, the assessed and required levels for the quality attributes that are fulfilled are also noted. This information is used later on in the selection of transformations.

The following steps of the process are, in principle, repeated for each quality attribute. However, since transformations affect more than one quality attribute, decisions concerning the selection of transformation will be based on all the relevant quality attributes.

The second step is to identify, for the quality attribute currently addressed, at what components or locations in the architecture the quality attribute is inhibited. The assessment performed in the previous phase has lead to a quantitative prediction, but while assessing the architecture the software architect normally get several hints on what components represent bottlenecks for the quality attribute. For instance, when performing impact analysis during scenario-based maintainability assessment, there often is one (or a few components) that play a role in multiple scenarios. For some reason, the functionality captured by that component is sensitive to requirement changes. Such kinds of indications often give valuable input on what aspects of the architecture need to be changed.

The third step is the selection of a transformation that will solve the identified problem spots in the architecture. Generally, several different transformations can be used, differing in scope and impact, but also on their effects on the other quality attributes. For the selection of the most appropriate transformation, it is important to explicitly analyze the effects of the transformation on the other quality attributes. Based on this analysis, we select the transformation that does not affect any unfulfilled quality attributes in a negative manner, but only quality attributes for which there is a satisfying (positive) difference between the assessed and required level. For instance, if the system performance is well satisfied by the current architecture, it is acceptable to select transformations that improve maintainability at the expense of performance. However, if both are currently not fulfilled or the assessed level very close to the required level, one should search for other alternatives.

The fourth and final step in the process is to perform the transformation, meaning that the functionality is reorganized according to, e.g. the selected style or pattern, and that the description of the architecture is updated to incorporate the changes. It is important to keep a record of the versions of the software architecture, the assessed levels for each of the relevant quality attributes and the rationale for each transformation. This is both useful when it proves necessary to backtrack to an earlier version since the team reached a dead-end in the design and for future reference by software engineers doing detailed design, implementation or software maintenance.

The architecture design method presented in this part of the book assumes a fully objective and quantitative approach. It presents a picture of an idealized software architecture design process because the technology for several aspects of the method are currently not available or have not been disseminated to software industry. For instance, for some quality attributes no validated assessment techniques are available. In addition, for several of the transformations discussed in the remainder of this chapter, the exact effect on the quality attributes of a software architecture is not obvious. Part of this is due to the fact that performing a design, basically any design, is fundamentally a creative process that cannot be formalized and automated. However, many parts surrounding the creative process can and should be formalized in order to become objective and, potentially, automated. Thus, the method presented here is as much a vision on how we would want to perform software architecture design as it is a viable way of working today. However, it requires at times that one resorts to, e.g. qualitative reasoning or experience-based decisions.

Concluding, the software architecture transformation process consists of four major steps:

- Identify the QAs that are not fulfilled
- For each QA, identify the locations where the QA is inhibited
- Select the most appropriate transformation
- Perform the transformation

3. Impose Architectural Style

The first category of architecture transformation is concerned with imposing an architectural style on the software architecture. Shaw and Garlan [Shaw & Garlan 96] and Buschmann et al. [Buschmann et al. 96] present several architectural styles[1] that improve the possibilities for certain quality attributes for the system the style is imposed upon and are less supportive for other quality attributes. Certain styles, e.g. the layered architectural style, increase the flexibility of the system by defining several levels of abstraction, but generally decrease the performance of the resulting system. With each architectural style, there is an associated fitness for the quality attributes. The most appropriate style for a system depends primarily on its quality requirements.

Transforming an architecture by imposing an architectural style results in a complete reorganization of the architecture. Often virtually all architectural components are affected and the assignment of functionality to the components is reorganized. In addition, the original connections between the components are often affected. Consequently, imposing an architectural style is a transformation with major, architecture-wide impact.

Although architectural styles can be merged up to some extent, styles are not orthogonal in the sense that they can be merged arbitrarily. If a second architectural style is selected for a part of the architecture, it is necessary to make sure that the constraints of the two styles do not conflict with each other. A typical example of a software architecture using two styles is the compiler example in [Perry & Wolf 92] where the standard pipes-and-filters compiler architectural style is complemented with a black-board style. The blackboard contains data that needs to be accessible

1. Buschmann et al. [Buschmann et al. 96] uses the term architectural patterns, but we use that term for another category of transformations.

by multiple filters. Although the resulting software architecture contains both styles, constraints of both styles are violated.

The more common case is where a subsystem uses an architectural style that is different than the style used at the system level. This use of different architectural styles leads to less conflicts between styles, provided that the subsystem acts as a correct component at the system level. However, when considering conceptual integrity, our experience is that one is able to use the same archetypes and organizing principles at all levels of the system. Thus, if it is possible to use the same style throughout the system, this is preferable.

In our approach, we explicitly distinguish between the components that are used to fulfil the functional requirements and the software architecture of the system that is used to fulfil the quality requirements. In practice, the distinction is generally not as explicit, i.e. also the implementation of a component influences most quality attributes, e.g. reliability, robustness and performance.

3.1 Styles and Quality Attributes

Architectural styles have been discussed at length in other publications, see e.g. [Buschmann et al. 96] and [Bass et al. 98], although the suitability of styles for the various quality attributes is not always discussed to the same extent. In this section, we briefly discuss the most fundamental styles that are generally recognized:

Pipes-and-filters. The pipes and filters of this style can be viewed as analogous to a chemical plant, in which the filters initiate chemical processes on the material transported through the pipes. The pipes-and-filters style assumes a data-flow network where data flows through the pipes and is processed by the filters. The most well-known instance of this style is implemented in the Unix operating system, in particular the associated command shells. A second example is the standard compiler architecture taught in virtually each computer science study program. In a standard compiler, the scanner, parser, optimizer and code generator form the filters whereas pipes transport, for instance, character and token data.

There exist many varieties of pipes-and-filter implementations and definitions. Variations include pipeline (linear), systems without feedback loops (a-cyclic) and arbitrary (cyclic graph) [ref-shaw-clements]. The common denominator is that the filters operate asynchronously and have little or no state. However, filters do exchange data through pipes and, consequently, some synchronization occurs in that manner. The way data is transported through pipes can be pushing, pulling or asynchronous. In the first approach, entry of data by the source filter will activate the sink filter. This is typically useful when the system is processing data from an external source. The opposite occurs for the pulling approach, where the sink filter activates the source filter. This typically occurs in a compiler where the parser will ask for tokens from the lexer. Finally, when using the asynchronous approach, the pipe will store data entered by the source filter until the sink filter requests this, thereby decreasing the synchronicity in the system.

The application of each style to transform a software architecture has an associated effect on the quality attributes of the architecture. However, the actual effect is as much depending on the type of system modeled by the software architecture as the selected style. Nevertheless, below, we discuss the general effects of the pipes-and-filters style on the quality attributes:

- **Performance**: The advantage of the pipes-and-filters style from a performance perspective is that the filters form excellent units of concurrency, allowing for parallel processes which generally improves performance, assuming it is used with care. In addition, since the pipes connect the components, the interface for each component is very narrow, reducing the number of synchronization points.

  The advantage of the pipes-and-filters style discussed above may, however, turn into a disadvantage for performance as well. If each filter only performs a very small unit of computation for each unit of data, the style will lead to many context switches and copying of data, which affects performance negatively.
- **Maintainability**: The maintainability of a pipes and filters system also has two sides. On the positive side, the configuration of filters is generally very flexible, allowing for even run-time reorganization of pipes and filters. Thus, as long as new requirements can be implemented by new filters and reorganization of the network, maintainability of this style is very good.

  The disadvantage is that requirement changes often affect multiple filters. A typical example are syntax changes or extensions in a compiler. These are generally orthogonal to the compiler components and, consequently, require changes to the lexer, parser, optimizer and code generation component. Especially in larger systems, the disadvantage of this style is that real-world entities represented by the system are decomposed and part of the functionality of multiple filters. Our experience is that, for the systems that we have worked with, the majority of requirement changes affect more than one filter and that, consequently, the pipes-and-filters style is not particularly suitable for maintainability.
- **Reliability**: The reliability of a pipes-and-filters system is dependent on its topology and, as a result, it is hard to generalize over it. However, since the pipes-and-filters style assumes that each external event causes computation in a

series of filters, one may deduce that the reliability may be less than in styles where most events lead to computation in only one, or perhaps a few components. The series of filters requires each filter to deliver the specified result in order for the system to be successful, i.e. analogous to an ‘and’-function in boolean logic. In other styles, the primary component handling the event may still be able to deliver a result even if some of the secondary components used by the primary component fail.

- **Safety**: The line of reasoning used for reliability also holds for safety. The fact that correct computation is dependent on several components, increases the chance that some failure will occur. Thus, in cases where passiveness of the system may cause hazardous situations, this causes decreased safety. On the other hand, the fact that the output of the system generally will occur through one or a few filters allows for local verification of reasonable output values.
- **Security**: Pipes-and-filter system generally have small and explicitly defined input and output interfaces and a well-defined component topology. This means that access to the system is only available through the defined interfaces, where identity verification, authorization and encryption/decryption can be performed. The same technique can be used at the component level in systems where information of different security levels is present.

Layers. The layered architectural style suggests to decompose a system into a set of horizontal layers where each layer provides an additional level of abstraction over its lower layer and provides an interface for using the abstraction it represents to a higher level layer. As a consequence, an atomic task at the highest level of abstraction is decomposed into a number of tasks at the lower level layer, which in turn is decomposed into yet lower-level tasks, thus forming a hierarchy of tasks becoming smaller and simpler lower in the hierarchy.

The probably best known example of a layered architecture is OSI 7-layer model for communication protocols [ref-tanenbaum?]. In figure 13, the layers of the OSI standard are presented. Each layer deals with one particular aspect of communication and builds upon the lower-level aspects to be available.

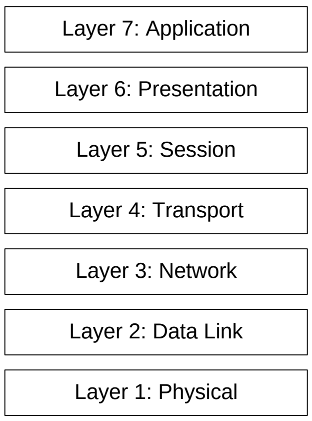


FIGURE 13. **The OSI 7-layer model for communication protocols**

Several variants of the layered style exist. The pure style allows layers to call only their immediate subordinate layer. Assuming this fits the application domain, this leads to the lowest level of dependency between layers. The relaxed style exists in two forms. In the first, each layer can invoke all lower level layers, rather than just the layer immediate below it. In the second form, the layer can invoke higher level layers.

Imposing a layered style involves at least the following steps. First, the identification of a number of abstraction levels and representing them as layers. Second, the assignment of components to the layers and, finally, the remodularization of components that contain functionality belonging on different levels. The latter may result in a real-world entity being represented in multiple layers, which has a negative effect on maintainability, but this may be outweighed by the advantages. A typical example is the three-tier architecture in business applications, where the lowest level provides persistence and atomic transactions, the middle layer provides the business logic and the top layer provides presentation and user interaction. Despite its advantages, especially in client-server type applications, it does cause a real-world entity to be represented in all layers of the architecture.

Below, we discuss the relation between the layered style and the selected quality attributes:

- **Performance**: The layered style organizes computational tasks based on level of abstraction, rather than their computational relation. This generally causes functionality related to an external event or request to be divided over multiple layers. For instance, in the case of communication protocols, over all layers. As a consequence, the computation in response to the external event covers multiple layers as well, requiring several method context switches. This leads to decreased performance and experience has shown that the layered architecture does cause a performance penalty when used. Consequently, communication protocols are generally not implemented according to the OSI 7-layer model, but rather in a reorganized fashion avoiding the disadvantages associated with passing several layer boundaries.

  With respect to concurrency, there is an important note to make. The naive approach to adding concurrency to a system built using the layered style is to assign each layer its own thread of control. In general, this does not lead to an increase in performance, but may even lead to decreased performance, due to the number of task context switches required to react to a single event. The alternative approach is to assign the events processed by the layer stack their own thread of control and to implement the layers in a re-entrant fashion. This generally leads to an increase in performance.
- **Maintainability**: Maintainability of a system is influenced by the way requirement changes affect the system. If most changes can be implemented by changing one or a small number of components or by adding a new component, then the maintainability of the system will be high. Assuming the way the functionality of a layered system is assigned to its layers in an appropriate way, the maintainability of a layered system is generally relatively high. Layers have few dependencies on other layers, allowing for replacement of a layer without changing its superior and subordinate layer. If, however, the functionality of the system is not organized according to the expected requirement changes, maintainability may be compromised since several layers will need to be changed for requirement changes.
- **Reliability**: Similar to the pipes-and-filters style, the layered style requires computation in all or most layers for each external event or request. Due to this, failure of one layer may cause the system as a whole to fail. Consequently, the reliability may be lower than when using some of the subsequent styles. On the positive side, a higher-level layer may contain functionality for handling faults occurring at its lower-level layer. Because the higher-level layer has a better overview of the ongoing computation, it may be able to deal with failures the lower-level layer would not have been able to manage on its own.
- **Safety**: See pipes-and-filters.

- **Security**: The layered style supports security rather well due to the fact that all computation starts at the top-level or at the bottom-level, as in the case of communication protocols. This allows for the insertion of security layers performing authorization and, possibly, encryption and decryption of data.

Blackboard. The blackboard style originates in the artificial intelligence domain, where it was used as a data or knowledge sharing mechanism between a number of intelligent entities. The computational model used by the blackboard style assumes a central data repository and a set of active components surrounding the data repository. The components scan the blackboard for data items that they are able to take as input, take these items from the blackboard, process them and places the results on the blackboard. Different from the styles discussed earlier, the control flow of the system is not explicitly designed, but evolves as the components are able to execute. Consequently, giving different priorities to the components generally affects the control flow.

The blackboard style was initially typically used in cases where no overall solutions to a problem was available, but only elements that addressed on aspect of the problem. The solution elements are implemented as components that roam the blackboard for problem pieces that they know how to handle. This may result in multiple components selecting a data element and trying to process it. Often, however, this results in all but one component failing to process the data. More recently, the blackboard style is also used in cases where there exist overall solutions, but where the solution may change so often due to changing requirements that it is more feasible to use a blackboard and let the components dynamically arrange the composition of solution elements into an overall solution. For instance, the fire-alarm system uses a blackboard style to achieve real-time and performance quality attributes. Finally, even more close to home, systems that employ a database subsystem make use of the blackboard style at least up to some extent. The database removes the connections between the components in the system, in that most components can work independently towards the database and communicate through the additions and changes made to the data in the database.

The simplest form of blackboard system employs a single central data repository, but more complex forms in which multiple repositories exist organized according to type of data or location in a distributed system. In the more complex cases, some of the components move data between repositories, possibly after having processed it.

Finally, in addition to the blackboard and the components processing data elements on the blackboard, a control component may be present that determines in what order the processing components can access the blackboard and compute. The control component may activate processing components according to a predefined schedule or inspect the blackboard to determine what processing component would be most suitable to activate next.

The effect of using the blackboard style on the quality attributes difficult to describe due to the wide variety of ways this style can be used. However, below we present some general guidelines:

- **Performance**: Although one can build blackboard-based systems with high performance, generally performance is not one of the strong points of this style. One can identify two main reasons for this. First, considerable amounts of computation may be spent on behavior that is not related to the application domain, such as roaming the blackboard, or behavior that is redundant, e.g. multiple components trying to process a data element. Second, because there is no explicitly defined control flow, but, in the best case a control component that attempts to optimize the flow, computation is generally not performed in an optimal order, leading to decreased performance for the critical paths in the system.

  Nevertheless, examples of blackboard-based systems with high performance do exist. For instance, in [ref-boasson/cherki] an architecture is presented that consists of a highly structured blackboard with a fixed and identifiable set of data fields, a set of tasks with specified interfaces, in terms of what data fields are read and what fields are written, and, finally, a control component that contains an optimized and hard-coded control flow. This architecture is, among others, used in simulation systems where timing and performance are very important. The ‘factored-out’ control can be optimized late in the development process to obtain the highest refresh rates while keeping all simulated entities synchronized. *** search Boasson’s paper on military system/Cherki’s paper on simulators ***
- **Maintainability**: A considerably stronger aspect of the blackboard style is maintainability. The blackboard allows for easy, even dynamic, addition and removal of types of data as well as the number of instances of the type. Since processing components are independent of each other, components can be added and removed without having to change other processing components. Depending on how it is constructed, the control component is the only element in the system that would need to be changed in order to incorporate changes in the processing component set. However, one can define the control components that use meta-models to capture what types of data are present on the blackboard and what data each processing component needs in order to process. In

that case, the control component is able to some types of new processing component without having to be changed.

However, also for blackboard-based systems it is important to identify that naive design may lead to systems that are hard to maintain. Likely requirement changes should be incorporated by changing or adding, preferably, a single components. In addition, blackboard-based systems that hard-wire many of the aspects of the system, such as the example simulation system discussed earlier, will be harder to maintain than very flexible systems.

- **Reliability**: The reliability of blackboard style systems has two sides to it. On the one side, the independence of the processing components and the fact that the control component iteratively activates the various components increases reliability since it makes the system more tolerant to faults and robust with respect to invalid data on the blackboard. However, on the other side, there is no central or explicit specification of the system behavior, which may make it hard for the system to identify that certain responsibilities are not fulfilled. The control component plays an important role in addressing this problem.
- **Safety**: The fact that the system has no central or explicit specification of overall system behavior may also compromise safety in safety-critical systems. Processing components may write incorrect data on the blackboard that can lead to potentially dangerous external actions. Since all components, at least in the pure model, may read and write all data, one component may compromise both safety and reliability.
- **Security**: The fact that the blackboard style employs a central data storage that can be accessed by all components in the system and the ability of the system to dynamically incorporate new components may lead to security problems if no precautions are taken. On the other hand, having all classified data in a single location simplifies the control of access.

Object-Orientation. The object-oriented style organizes the system in terms of communication objects. Objects are entities that contain some state and operations to access and change this state. Whereas the state and operations are encapsulated by the object, the signatures of the operations are accessible on the interface of the object. Operations can be accessed by sending messages to an object. A message causes the activation of an operation, which may lead to changes to the internal object state and messages to other objects. Messages are synchronous in that the object sending a message waits until it receives a reply and only then continues with its own computation.

Several models extend the basic object-oriented style with various aspects. The concurrent object-oriented style, for example, assumes all objects to be potentially active and an object sending a message to another object will delay the thread sending the message, but other threads may be active within the object.

Although objects do not need to be aware of the sender of a message, an object is required to have the identity of an object it intends to send a message to. Since objects, in the course of their computation, generally need to send messages to several other objects, each object is required to maintain references to its acquaintances. Consequently, an object-oriented system can be viewed as a network of connected objects.

The object-oriented style, similar to the other styles, affects the quality attributes of the system. Below, these effects are discussed:

- **Performance**: The performance of object-oriented programming has received considerable attention in the literature, especially in comparison to conventional structured programming. Although new object-oriented languages, at their introduction, generally are less efficient than traditional languages, this disadvantage is generally largely removed at subsequent versions. Typical examples are C++ and Java. However, using the object-oriented style as an organizing principle at the architectural level is even less controversial since it is generally accepted that a system needs to be broken down into components. The question is just whether these components should be filters, layers, objects or of yet another type.

  The performance of systems based on the object-oriented style is very much dependent on the principles the designer used to define the objects, but there is not necessarily a fundamental conflict. For optimal maintainability, objects should be selected so that the most likely requirement changes affect as few objects as possible. For optimal performance, objects should be selected so that the most frequent use scenarios cause computation at as few objects as possible, since performing context switches between objects is expensive. Since requirement changes often affect the use scenarios, the optimal system organization for maintainability and for performance may actually be very close to each other. In our experience, we have seen several examples of this.

  One important note to make is that, similar to the layered style, the naive way of adding concurrency to an object-oriented system is to use the homogenous approach, i.e. each object has its own thread of control. This is generally not optimal if there are many objects in the system due to the large number of context switches and synchronization points for each use scenario. Instead, threads should be attached to external events that cause considerable amounts computation in the system.

- **Maintainability**: The object-oriented programming paradigm became popular due to claims of increased reusability and maintainability. In our cooperation with industry, we have seen many examples of cases where these claims were actually fulfilled. However, the main issue in achieving maintainability in object-oriented systems is modeling the right objects. As discussed in the previous section, the likely change scenarios should affect the system as little as possible, and preferably lead to the definition of a new subclass or a new type of object aggregation.

  One reuse inhibitor is the fact that an object require references to the objects it sends messages to, i.e. its acquaintances. Since the types and number of acquaintances is hard-coded in the class specification, changes will always require class specification to be changed. The implicit invocation style discussed below addresses this problem.
- **Reliability**: The object-oriented style is not particularly positive or negative with respect to reliability. One disadvantage that could be mentioned is that fault handling generally has to be managed inside the object, due to the encapsulation. This may make it harder to have fault handling at higher levels in the system, where more information is available. However, the fact that the system is modeled in terms of relatively independent entities is positive for reliability, since no central entity can cause the system to fail.
- **Safety**: One of the basic organizing principles of the object-oriented style is that real-world entities should, as much as possible, be represented as objects. As a consequence, the real-world entities that may compromise or assure safety are also modeled as objects. The fact that each real-world entity has a one-to-one correspondence to a system entity, is positive for safety since the system entity is better suited to identify hazardous situations and react to them than an organization where the behavior of the system entity is divided over multiple entities.
- **Security**: The object-oriented style both encapsulates and fragments the data contained in the system, being positive and negative aspects, respectively. Authorization of access to the system may be simplified by the fact that system interfaces generally will be represented by objects.

Implicit Invocation. The fact that an object in the object-oriented style needs to know the identity of an object it sends a message to increases the dependencies between objects which leads to a more rigid organization, with consequent negative effects on maintainability. To address this, a style developed based on the object-oriented style has recently become more popular: *implicit invocation* [ref-notkin, shaw]. The implicit invocation style organizes the system in terms of components that generate events, possibly containing data, and that consume events. Components register their interest in receiving events of certain types and, depending on the type of system, publish their ability to generate certain types of events. An event handling mechanism implicitly present within the system handles all generated events and delivers the events to interested components. The JavaBeans standard [ref] is typical example of the implicit invocation style.

Events received by components are bound to operations much in the same way as messages are bound to operations in the object-oriented style and lead to computation within the component and, possibly, the generation of new events. The main difference between events and messages is that events are asynchronous, i.e. the component generating an event continues its computation immediately after sending the event, and that events are undirected, i.e. the identity of the receiving component or components is unknown to the sender of the event.

System build on the implicit invocation style can vary in a number of aspects. The first is whether an event, in the case of multiple consumers, should be sent to one or to all components. In the case of observing components, all components should receive the event, but in other cases only one component should consume the event, that is receive it and remove it from the system. A second aspect is whether events of certain types generated by different components should be treated equal. For example, assume two button components in a graphical user interface environment. Both buttons can be clicked, leading to the generation of a 'clicked' event. However, the system behavior in the case of a clicked event from button 1 should be different from that in response to a clicked event from button 2. Three alternative solutions can be chosen. First, rather than using the clicked event, one defines two events, i.e. 'button1_clicked' and 'button2_clicked'. This solution allows one to treat events similar independent of generator of the event, but forces components to incorporate system specifics, e.g. the name of system specific events. The second solution is to generate a 'clicked' event, but to attach the identity of the generating component to the event. This solution also allows the system to treat all events of a certain type as equal, but the components need to contain system specifics, e.g. component identities to be able to exhibit different behavior, dependent on the component generating the event. Finally, one may choose to not broadcast events in a system-wide manner, but to explicitly configure the system in terms of what components will receive events generated by what other components. This solution allows components to be generic since they need not incorporate system specifics, but an explicit system configuration is required to connect components.

One can identify a relation between the implicit invocation style and the blackboard style in that both avoid explicit specification of the control and data flow. The control component may exhibit some influence on the control flow, which is similar to the implicit event handler that may prioritize certain event types over others. Consequently, the quality attributes are, up to some extent, affected similar by both styles.

- **Performance**: The event handling mechanism requires a certain amount of computation that is unrelated to the actual domain functionality, thus negatively affecting performance. In addition, component communication where an answer is required from another component requires two events to be sent and processed and may lead to fragmentation of logical operations into multiple implemented operations, which is negative both for performance and for maintainability. These negative effects can be addressed up to some extent through explicit system configuration, since it removes the, implicit, central event handler required otherwise.
- **Maintainability**: The implicit invocation style allows for run-time addition, removal and replacement of components, in addition to easy compile time flexibility. Whether this is leads to high maintainability is dependent on the modeling of components, similar to the object-oriented style. Likely change scenarios should lead to changes in one or only a few components.
- **Reliability**: The reasoning with respect to reliability is similar to the object-oriented and blackboard styles. One advantage with respect to the object-oriented style, however, is that the implicit invocation mechanism can also be used for broadcasting events indicating faults, which may be used for system-wide fault handling.
- **Safety**: See the object-oriented style.
- **Security**: See the object-oriented style.

Concluding remarks. In this section, we have presented five architectural styles that can be imposed on a software architecture. We have discussed the effects of using these styles on the quality attributes of the software architecture. It is important to observe that the actual effect is dependent not only on the style, but also on the type of system and on the way the style is imposed on the architecture. Our intention in the discussion was to avoid making absolute statements, but instead to clarify how different uses of the style may affect the quality attributes. Finally, even if an architectural style generally affects a particular quality attribute in a positive or negative manner, it is very well possible to address this using different transformations or by careful detailed design. For instance, some studies performed by members of our research group [refs] have shown that performance problems in object-oriented systems are largely due to the excessive use of dynamic memory for object creation and deletion (in C++). These problems, however, can be easily handled by either selecting a smarter memory handling library or by adding memory pools to the system and requesting objects from the memory pool rather than creating them. This can lead to up to an order of magnitude difference in performance when compared to a naive implementation. Thus, software architects should not avoid selecting styles that affect some of the driving quality attributes negatively, but select the style based on its positive effect on quality attributes that are hardest to achieve and its negative effect on quality attributes that are easiest to affect later in the design. Finally, many systems uses more than one style. Although one can argue that this reduces intuitiveness, using more than one style may be the best way to achieving the quality requirements.

3.2 Example

To illustrate the imposition of an architectural style on the system, we use the fire-alarm system presented in chapter 2. The functional architecture of the fire alarm system can be represented as shown in figure 14. During the architecture assessment phase, the performance and real-time characteristics of the system were evaluated. Since each output is depending on a potentially large collection of inputs, the output will request the status at each sensor it is depending on. This involves sending a request to the sensor via the communication loop, waiting for the answer and processing the answer to determine whether the output should activate. The result of the evaluation was that the response time of the system in case of a fire would be far above the maximum required in international standards. Obviously, this assumed the use of the intended hardware, i.e. a small 8-bit processor and a very low bandwidth network.

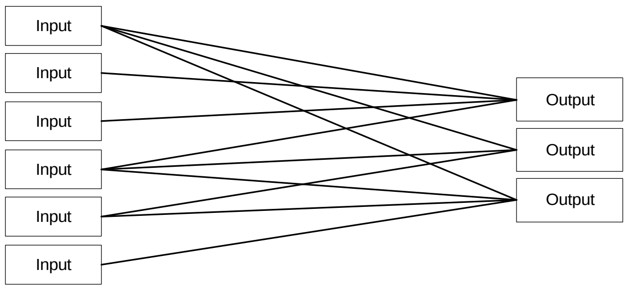


FIGURE 14. **Functional view of the fire-alarm system**

To address the evaluation results and to improve the performance and real-time attributes of the architecture, it was decided to impose a blackboard style on the software architecture. The blackboard would not contain sensor values, but rather deviations. *Deviations* are put on the blackboard only by those Inputs that are in a state different from normal. An output only needs to investigate the blackboard in order to establish its behavior. The resulting software architecture is shown in figure 15. The consequence of this transformation was that the response time of the system was well below the required levels, as well as the performance, in terms of refresh rates for the inputs.

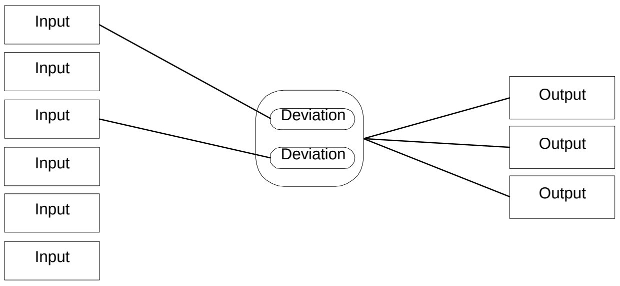


FIGURE 15. Fire alarm system architecture based on the blackboard style

4. Impose Architectural Pattern

In the previous section, imposing an architectural style was presented as a way to transform a software architecture and improve its quality attributes. A second category of transformations is the imposition of an *architectural pattern*[1]. An architectural pattern is different from an architectural style in that it is not pre-dominant and can be merged with architectural styles without problems. It is also different from a design pattern, discussed in the next section, since it affects the complete architecture, or at least the larger part of it. Architectural patterns generally impose a *rule* [Richardson & Wolf 96] on the architecture that specifies how the system will deal with one aspect of its functionality, e.g. concurrency or persistence.

An architectural pattern does, generally, not reorganize the fundamental components of the architecture, but rather extends and changes their behavior, as well as adds one or a few components that contain functionality needed by the pattern. The architectural pattern, however, constrains the behavior of the components in the software architecture, or at least those components that are affected by it, and does,

1. Note that our use of the term “architectural pattern” is different from the use in [Buschmann et al. 96].

generally, not allow the use of another architectural pattern that addresses the same aspect.

As mentioned, architectural patterns deal with one aspect of the behavior of the system. Generally this aspect is not in the application domain, but rather in the ‘computer science’ domain. Examples of aspects the way the system deals with concurrency, persistence, synchronization, transactions, distribution, run-time binding, real-time behavior and graphical user interfaces. However, these are just examples of aspects and other aspects may be relevant for particular systems.

For each aspect, several architectural patterns exist that can be used to deal with the aspect. For instance, in the case of concurrency, one can decide to use a concurrent operating system with preemptive tasks, a real-time kernel with non-preemptive tasks or one can implement an application-level scheduler as part of the system and integrate handling of concurrency in the software architecture. Each solution has its advantages and disadvantages and the optimal choice depends on the system requirements.

In the next section, we discuss architectural patterns for the aspects that we mentioned earlier in this section. In section 4.2, we present an example of the imposition of an architecture pattern.

4.1 Architectural patterns and quality attributes

Architectural patterns impose a rule on the system that requires one aspect to be handled as specified in the architectural pattern. In this section, we describe some examples of architectural patterns for the primary aspects that most systems have to deal with. The effect of each architectural pattern on the quality attributes is also briefly described.

Concurrency. Many applications have a need for simulating parallelism even on a single processor system. But even in distributed systems, generally more threads are required than available processors. This has to be incorporated into the architecture by adding an architectural pattern for handling concurrency. Several solutions are available for achieving this, and the most suitable one depends on the requirements and type of system. The use of a particular architectural pattern imposes a rule on the system, since each pattern requires the components to behave in a particular way in order to create, synchronize and remove units of concurrency. Below, we briefly describe the most common approaches to incorporating concurrency into a software architecture

- **Operating system processes**: Assuming the system will execute on top of a general-purpose operating system, processes will be available that allow components to run in separate address spaces and concurrently. The use of processes to contain components requires additional functionality, or even components, to handle the communication between components. Several models for communication between processes are available, including streams, shared memory areas and message passing.
    - Although **concurrency** will generally increase performance for systems that are I/O-bound, the use of processes has disadvantages for components that exchange much data as a result of the necessary copying of data between processes. Even if shared memory areas are used, this often requires additional specific code to handle the shared memory which is different from the implicit use of local memory.
    - **Maintainability** is influenced positively by the increased separation of components. Modeling components as processes requires the components to be more autonomous. On the other hand, additional functionality and, possibly, additional components are required for the communication and synchronization between components. This increases both the distance to the conceptual, domain-based model and the size of the system. Since maintenance cost are directly related to system size, this leads to decreased maintainability. Finally, if groups of objects are modeled as processes, the difference in intra-process communication and inter-process communication will make the organization of components more rigid since moving a component from one process to another will require changes to the way the component communicates with its related components.
    - The fact that the system is organized as independent processes is positive for **reliability** since the failure of one component and its associated process, leaves the other components in operation. A disadvantage is that communication between components is more complicated, increasing the likelihood of communication failures.
    - The fact that the failure of one component generally leaves other processes unaffected is positive for **safety** as well. In addition, the ability to restart processes that have failed allows for relatively quick recovery of the system in response to failures.
    - **Security** is influenced positively since achieving access to one process will not automatically lead to access to the other parts of the system. On the other side is the fact that classified data may be fragmented throughout the processes.

- **Operating system threads**: In addition to processes that are relatively heavy-weight and create separate address spaces, most general-purpose operating systems support threads. A thread, or light-weight process, does not create a separate address space, but executes in the same address space as the other threads. If the handling of external events should be the source of concurrency, rather than the components, threads are a more suitable model than processes. Also, if the level of concurrency should vary dynamically, threads provide a more flexible solution than processes. However, threads can access components simultaneously, leading to racing conditions, if not controlled by appropriate synchronization mechanisms.
  - Compared to a single threaded solution, the use of threads generally increases **performance**. Also, when compared to processes, threads require no additional overhead for communication between components, since the components remain in the same address space. However, in the case of multi-processor machines, studies have shown, e.g. [ref-Lars], that the performance of thread-based systems is less than process-based systems, due to the fact that the threads operate in the same address space and processors are delayed by conflicts due to simultaneous access of memory locations in the same proximity.
  - Threads may access components simultaneously and to avoid race conditions, synchronization mechanisms have to added to the components. This is negative for **maintainability** not only because of the additional code that is required, but also due to the additional complexity of multiple threads in a single address space. To optimally employ concurrency, synchronization is only added where it is absolutely necessary, which may lead to complex schemes that are defined based on knowledge of the control flow. Incorporating new requirements in the system may invalidate these schemes, but since the assumptions are very much implicit and often not well documented, this easily leads to incorrect synchronization. This is further complicated by the fact that concurrent applications are notoriously difficult to debug, leading to increased maintenance cost, since testing and corrective maintenance are part of maintenance.
  - Threads that fail will generally not affect other threads directly, which is positive for **reliability**, but failed threads may leave parts of the system in an inconsistent state. In addition, the aforementioned increased complexity of the system due to synchronization will affect reliability negatively.
  - The use of threads allows for concurrent monitoring threads that may identify, e.g. the failure of important threads controlling safety-critical parts. Since the monitoring thread, upon identification of a failure, can control the system in order to reach a safe state, **safety** is improved, compared to the

single-threaded solution, where the complete system would become passive. However, the increased complexity of using threads affects reliability, and consequently, safety negatively.
  - The use of threads has no major effects on **security**.

- **Non-preemptive threads**: Processes and threads in general-purpose operating systems are often preemptive, i.e. the system may pre-empt tasks at any point, based on clock interrupts. The disadvantage, especially in embedded and time-critical systems, is that synchronization mechanisms are required and that it is hard to make any statements about the timing of system behavior. An alternative approach is the use of non-preemptive threads, i.e. threads that give up the processor through an explicit statement. This gives more control to the software engineer since it is known what behavior is executed as an atomic unit. However, it also requires more responsibility since not giving up the processor due to, e.g. an infinite loop, will stall the system.
  - One single processor systems, non-preemptive threads may provide the highest **performance** in terms of, especially, throughput, since the delays at synchronization points are not present. The response time to high-priority events is determined by the size of the atomic units of computation defined by the software engineers. A high-priority event will be scheduled immediately after the current task gives up the processor, whereas the preemptive thread model would allow for immediate execution.
  - Although no explicit synchronization mechanisms are put into the code that affect **maintainability** negatively, the design of the code, and sometimes even the architecture, is influenced by the fact that tasks have to release the processor at frequent, but safe locations in the code. This may lead to less intuitive code, affecting maintainability negatively since it becomes harder to understand.
  - The main negative influence on **reliability** is the fact that one task failing to release the processor may stall the complete system. Embedded systems often employ watchdog mechanisms in hardware, but these will just reboot the system, not handle the failure.
  - Also **safety** is affected negatively by the ability of one task to stall the system. In cases where passiveness from the system may lead to hazardous situations, non-preemptive threads are generally less suitable.
  - **Security** is not affected substantially.
- **Application-level scheduler**: The last architectural pattern that we will discuss with respect to concurrency is the use of an application-level scheduler. This approach is typically used in embedded systems with very tight resources in terms of memory and CPU cycles. Rather than incorporating a real-time kernel

or other operating system, the management and scheduling of tasks is performed at the application level, as part of the system. Whereas non-preemptive threads give the software engineer control on when to reschedule, but not on what task to run next, this pattern allows the software engineer to control the scheduling of tasks as well. This, obviously, also leads to increased responsibility for the software architects and engineers since both aspects are now handled by the software architecture.

A typical instance of this pattern is described in [Molin and Ohlsson 97?]. In their model, the system contains a scheduler and a set of active objects that have a tick() method on their interface. The tick method of each active object is invoked periodically by the scheduler. The method performs the necessary tasks it requires and subsequently returns to the caller, i.e. the scheduler. The cycle time for the system is the sum of the execution time of all tick methods plus some minor overhead by the scheduler. If the cycle time is too long for some active objects, these objects can be put twice into the list, so that they are invoked twice during each cycle, although the total cycle length, obviously, increases. An application-level scheduler allows for the highest level of control by the software engineer, but also imposes the responsibility for the correct operation of the system with respect to concurrency fully on the shoulders of the engineer.

- Since there is no overhead from the operating system or kernel, except for some minor overhead for the application-level scheduler, **performance** is generally very high using this model. Resource efficiency is maximized at the expense of additional development time, since the software architect and engineers are responsible for task management and scheduling. A disadvantage may be, similar to non-preemptive threads, that the response time of the system to high-priority events is not optimal, although this can partially be handled by using interrupt routines.
- As mentioned above, this pattern maximizes resource efficiency at the expense of development time, system size (in terms of system-specific code) and system complexity. This has a negative impact on maintainability. Otherwise, the disadvantages are the same as for non-preemptive threads.
- See the above section concerning non-preemptive threads for the effects on **reliability**, **safety** and **security**.

Concluding, although not more than one of the architectural patterns discussed above can be applied to a component, it is possible use different patterns for different levels of the system. For instance, the system can be decomposed at the top-level into a small number of processes, but, within the processes, concurrency is achieved by using, e.g. non-preemptive threads. In addition, one can identify a direct relation between control over the behavior of the system and the responsibility that is put on the software architect.

Persistence. The second aspect for which we will discuss architectural patterns is persistence. In addition, we will also discuss a topic of mentioned in direct connection to persistence, i.e. transactions. Persistence is the ability of data to survive the process in which it was created. This allows data to be stored on permanent storage and read from this storage by other processes, at later points in time. With the mergence of object-oriented systems, the ambition expanded from persistent data to include persistent objects as well. This creates a number of challenging problems, especially with respect to object references. The first problem is that it is not always clear whether an object reference points to a part object, that should be saved together with the object containing the reference, or that the reference points to an acquaintance and thus should be rebound when the object is recreated from storage. The second issue is how to bind object references when recreating objects from permanent storage. This requires objects to have permanent identities, if the reference should be bound to the same object. However, in several cases, it is not the identity, but rather the capabilities of an object that determine whether it can be bound as an acquaintance of a recreated object. In that case, other rules for rebinding object references apply. Two architectural patterns for achieving persistence will be discussed, i.e. the use of a database management system and application-level persistence handling.

Transactions describe operations covering multiple data elements or objects that should be handled atomically. Generally, transactions should fulfil the ACID properties, i.e. atomicity, consistency, isolation and durability. These properties are primarily useful for the traditional database applications, such as banking and accounting systems. However, for several of the new types of systems, such as systems for computer-aided design or internet-based systems, not all properties are relevant or even useful. For instance, in computer-aided design systems often object-oriented databases are used that allow for versioning of objects. In such systems, even objects that are part of ongoing ‘transactions’, i.e. that are under design, can be accessed for reading, new versions of the object can be created and multiple versions of an object can be merged at later occasions. Some computer-supported software engineering environments support such approaches as well. In the aforementioned systems, the isolation property is not just useless, but even counter-productive. As an alternative, in real-time systems, rather than the isolation property, the durability property may be irrelevant. Transactions may be used to coordinate actions on two or more devices, e.g. two valves that need to close or open synchronously for correct system behavior. Concluding, although transactions provide important functionality for achieving correct system behavior, it is important to understand what properties one aims to achieve using the transaction mechanism. We describe two architectural patterns for achieving transaction-like functionality, i.e. a database management system application-level transaction management.

Before we discuss the architectural patterns and their relation to quality attributes, it is important to note that persistence and transactions are really two independent aspects of a system and can be used independent of each other.

- **Database management system**: A database management system (DBMS) extends the system with an additional component, but also imposes rules on the original architecture components. Both for persistence and for transactions, the entities that should be persistent and/or part of transactions need to be extended with additional functionality to support these aspect. In addition, components need handle requests from other components in accordance to the rules imposed by the DBMS. We leave this section relatively short since most readers will be rather familiar with DBMSs.
    - Despite the fact that a DMBS requires considerable resources in terms on primary and secondary memory is **performance** of these systems often very high. For instance, ObjectStore [ObjectStore 93] claims to be able to provide access rates to persistent data that are as high as data access to transient data, while providing the advantages of databases, i.e. persistence and transaction semantics. Obviously, when conflicts between, e.g. concurrent threads, arise, the response time for individual threads will be affected. However, in general, the many years of effort spent in optimizing the internals of DBMSs has removed much of the overhead that was present early systems, although this is primarily the case for larger systems in which much data is moved around. The primary disadvantage of using a DBMS, especially in embedded and real-time systems, is the amount of resources required by the subsystem. In addition, since DBMSs generally are optimized for large amounts of data, considerable overhead may be incorporated in handling small amounts of data under real-time constraints.

      Transaction management is, obviously, incorporated in database management systems. However, transaction semantics have to be specified explicitly in the system and the quality of the implementation influences performance considerably. Transaction conflicts cause restarts of transactions, which invalidate, possible considerable amounts of computation.
    - The effect of using a DBMS on **maintainability** depends up to a large extent on the types of changes that have to be incorporated into the system and on the type of database, i.e. relational or object-oriented, that is used in the system. Assuming that most applications are at least object-based, the use of a

relational database requires a transformation process to take place each time data is stored and retrieved. The manual transformation is sensitive to changes in the structure of the objects, thus requiring considerable effort. Changes to transaction semantics are also costly, especially since the transaction code is embedded in the code of the various system entities.

- Writing database interaction code is not trivial and often rather complex, which is negative for **reliability**. On the other hand, DBMS functionality has generally rather high reliability since it is extensively tested and used by many users.
- **Safety** is not affected significantly by the use of a database management system.
- Most database management systems have means for authorization as part of their functionality, which is a positive aspect with respect to security.

- **Application level persistence and transaction handling**: The use of a database management system requires, as mentioned earlier, considerable amounts of resources in terms of primarily memory management, but also other resources, such as performance, for certain systems. For embedded and other systems with small resources, this overhead may be unacceptable. Secondly, a system may only need some of the functionality provided by a database management system, e.g. persistence, but no transactions, or coordinated action through the use of transactions, but no roll-back. Using only a part of a full-fledged DBMS with associated resource requirements may not be a feasible solution in such situations.

  The alternative approach is to incorporate the required persistence and transaction functionality as part of the system, rather than by using a third party DBMS product. To achieve this, the software architect and engineer may make use of language and operating system features, such as serialization of objects in the Java language and semaphores or some other synchronization mechanism that is part of most operating systems. The advantage of this approach is that only the functionality required for the system is implemented.

  - It is difficult to make statements about the **performance** of systems using application level persistence and/or transaction implementation, since it heavily depends on the amount of functionality required by the system, the quality of the implementation and the characteristics of the system usage. However, assuming a reasonable implementation, performance should not be worse than when using a DBMS, while avoiding the resource requirements.
  - Similar to performance, the effect of using application-level persistence and transaction handling on **maintainability** depends on the implementation.

However, assuming that the amount of code required to implement the functionality is larger than the DBMS interaction code and the fact that this code is distributed over all components requiring persistence or transaction semantics, it is reasonable to assume that this has a negative effect on maintainability.

- For **reliability**, a similar line of reasoning holds: a larger amount of system-specific code distributed over the system results in lower reliability, due to the increased complexity of the software.
- The negative effect of this architectural pattern on reliability, has negative effects on **safety** as well. However, the fact that the persistence and transaction functionality is present at the system level, allows for application-specific failure handling, which may well improve safety.
- Since persistence and transaction semantics have no explicit relation to authorization and other security issues, no major effects of this architectural pattern on **security** are expected.

Distribution. One of the observations with respect to the current state of practice in systems development is that distribution is becoming ubiquitous. Most systems consist of parts distributed over multiple nodes or need to communicate with other systems via networks. Consequently, distribution is an integrative part of most systems.

The problem of distribution consists of two major aspects. The first is the way in which entities connect to each other. This can be achieved through predefined addresses and connections or, more flexible, through a central broker. The second aspect is the actual communication between remote entities. Again, several solutions exist including remote procedure calls and remote method invocation, distributed streams, a web interface, etc. Finally, one way to deal with distribution is by making it transparent, that is the system entities are unaware of their acquaintances being remote or local. Although much of the functionality related to distribution is transparent in today’s approaches to distribution, components are often aware of acquaintances being distributed or not, both when binding and when communicating.

The solutions used to achieve distribution in a system are typically architectural patterns since they require all entities in the system that are concerned with communication over address spaces to follow the same set of rules and constraints. Below, we discuss some architectural patterns that are typically used in the context of distributed systems, i.e. brokers, remote method invocation and HTTP.

- **Broker**: Brokers provide functionality for distributed components to find each other. A client component sends a message to the broker requesting a reference to a server component that fulfils certain requirements. These requirements include the component name and its interface, but possibly also other aspects, such as the state or the location of the component. CORBA [ref] is one of the best known broker architectures, but other alternatives exist, such as DCOM/AxtiveX [ref]. However, as presented in [Buschmann et al. 96], one can even implement the broker as an architectural pattern at the system level, meaning that the broker is part of the system rather than of the layer supporting the system.

  It is important to note is that not using a broker requires the system to hard-code the remote references and port addresses to at least each distributed part of the system in order to be able to communicate between the distributed parts.

  - The broker is used to connect distributed entities at run-time. Typically, a broker provides a reference which is then used for during the life of the component requesting the reference. Exceptional situations, i.e. the reference losing validity due to, e.g. a system crash, may require the component to invoke the broker again for a reference, but generally this happens only occasionally. Consequently, **performance** is not affected in any major way by using a broker architecture, assuming that references are requested relatively infrequent.
  - As mentioned above, the alternative to using a broker in a distributed system is to hard-code references between machines in the code or configuration files. This is negative for **maintainability** since relocation of services in the system requires an explicit effort for each client depending on that service, whereas in a broker architecture no effort would be required.
  - The broker architecture affects **reliability** both positively and negatively. On the positive side, the broker disconnects logical services from physical locations. This allows for clients to connect to a service has failed and restarts on a different node in the network. On the negative side, the broker itself is a central entity in the network and when it fails, the complete system may seize to function. However, generally brokers have backups that take over when the primary broker fails, thus eliminating this weakness.
  - A general property necessary for achieving **safety** is that it is easy to determine the behavior of the system in critical situations. The broker architectural pattern breaks a large system down into a set of smaller, but communicating systems, thus complicating the prediction of system behavior. However, the broker does allow the system to dynamically reconfigure itself, which may improve safety since clients of a failing service may dynamically connect to a new service, thus maintaining system behavior.

  - The broker is a central point for handing out references to system services. This allows for authorization at that point which is positive for **security**. Secondly, services need to register at the broker before they can be found by the clients. The broker can perform security checks for registering services so that trusted clients do not use untrusted services. However, once references have been exchanged, the broker is not involved in the continued communication between system entities, thus security is not influenced by the broker architectural pattern.

- **Remote method invocation**: Distribution has two primary aspects, i.e. finding remote entities and communicating with these entities. The broker architectural pattern is concerned with the first aspect, but for the latter aspect a remote communication mechanism is required. Traditionally, remote procedure calls were a typical communication mechanism between address spaces, but with the emergence of object-oriented programming languages, remote method invocation as in Java [ref] become more relevant alternatives. However, it is possible to make use of, e.g. sockets, for communication. The disadvantage is that this requires application-level 'interpreters' of the data that perform the work typically done in, for instance, the Java RMI layer. The latter depends, however, on the type of architectural style used for the system. For instance, a stream-based connection mechanism as sockets fits the pipes&filters style very well, making it the preferred approach.
  - Naturally, the **performance** of a remote method invocation compared to a local invocation is much lower. However, the use of distribution may be necessary for the system at hand or has advantages that outweigh the performance loss on a (small) subset of the method invocations. Even compared to other distributed communication mechanisms, remote method invocations may slow, due to the overhead associated with, among others, marshalling and demarshalling of the arguments. Therefore, it is important to explicitly investigate the type of distribution mechanism needed for the system. For instance, if a natural flow of data exists in the system, a stream-based solution, with less overhead for 'flattening' data, may be preferable from a performance perspective.
  - Assuming the acquaintance handling, i.e. selection and binding, is separated from the actual functionality, the **maintainability** of a system using remote method invocation is likely to be good. This is because no distinction has to be made between communicating with remote and local entities, allowing for flexible reallocation of objects. Generally, it has to be noted, the use of distribution is negative for maintainability since it increases the system size and because the code associated with distribution tends to be mixed with

    other code which complicates changing either the distribution code or the other code related to, e.g. domain functionality.
    - The use of remote method invocation has, compared to other distribution mechanisms, no major effects on **reliability**, **safety** or **security**.

- **HTTP and HTML**: The third, and last, architectural pattern related to distribution that we briefly intend to discuss is the use of the HyperText Transfer Protocol (HTTP) as a means of both finding and communicating with remote entities. Lately, we have seen an increasing use of the HTTP protocol as a means for networked embedded systems to communicate with the outside world and, up to some extent, communicate among system components. Especially for systems with very loosely coupled components, the use of the HTTP protocol minimizes dependencies and allows for easy observation from the outside using normal web browsers. However, selecting the HTTP protocol as a means for distributed communication imposes, as any architectural pattern, considerable constraints and design rules on the system. Generally, a considerable amount of translation between the internal representation of the domain model and the HTML format is required.
    - Due to the amount of translation required for using HTTP and the HTML format, **performance** is generally affected negatively by using this approach.
    - On the server side, the translation of the internal representation to HTTP and HTML formats causes lower **maintainability** due to increased code size and to the complexity of the translation. Although the domain functionality and HTTP translation usually are modeled as separate parts, considerable dependencies exist since extensions to the domain functionality often need to be accessed by distributed entities. On the client side, however, a standardized and relatively simple interface allows to communicate with all service-providing components in the same way, which greatly reduces interaction complexity.
    - **Reliability** and **safety** are not affected by using this architectural pattern.
    - The HTTP protocol and the HTML format have no or little means for handling **security**. Thus in systems where high security is required, this architectural pattern should not be used unless it is extended with functionality that guarantees sufficient security levels.

Graphical user interface. It may be surprising to the reader to find the topic of graphical user interfaces (GUIs) on the list of architectural patterns. The interface of the system to its users may as well be considered a functional requirement and, up to a considerable extent, it is. However, the GUI is concerned with presenting and controlling the domain functionality which requires the entities representing the domain functionality to provide interfaces that support this. Thus, deciding upon a particular approach to incorporating means to obtain domain data and controlling behavior imposes constraints.

One can identify two main approaches to incorporating interactivity in systems, i.e. model-view-controller (MVC) and the presentation-abstraction-control (PAC) [Buschmann et al. 96]. Both consist of a model (abstraction), a view (presentation) and a controller (control), but the way these components are organized is different. The MVC architectural pattern adds a view and a controller component to the current architecture, which is considered to be the model since it contains the domain functionality. Both the view and the controller interact with the model and the controller governs with the view as well. The PAC pattern organizes the architecture into a hierarchy of cooperating agents that internally consist of a presentation, an abstraction and a control component. The control component is the primary external contact for the agent as well as the internal coordinator that interacts with both the abstraction and the presentation component. However, these components do not directly interact with each other. Below, we briefly describe the effect of using these patterns on the quality attributes.

- Although it depends on the type of system and the implementation, **performance** is affected negatively by using either of these patterns. The MVC pattern tends to result in large numbers of update messages between the model and the view components. In addition, the access of data in a view may require several messages to different parts of the model component. The control component in the PAC pattern tends to be the bottleneck for communication since all messages need to pass this component. Especially requests that travel up and down the hierarchy often experience considerable overhead.
- The primary reason for using these patterns is to improve **maintainability**. The intention is that by separating the domain model from the presentation of the domain model and from the control of the system, each component can evolve independently, thus simplifying maintenance. Although this is the case, there are a few negative aspects to consider as well. Both patterns increase the complexity of the system since functionality related to domain concepts is divided over different components. In addition, the view and the controller in the MVC pattern often are connected rather intimately, which complicates changing one component without affecting the other. Finally, the PAC pattern tends to result in a complex control component that is hard to change.

- Compared to a traditional approach in which GUI functionality is mixed with domain functionality, the advantage of the MVC and PAC architectural patterns is that computation related to the application domain, to a large extent, takes place independent of other types of computation. This is positive for **reliability** since failures in one part of the system do not automatically affect other parts. A negative aspect is that the increased complexity of the system tends to decrease reliability.
- The relative independence of the system components is also positive for **safety**. The central role of the control component in the PAC pattern, however, partially neutralizes this.
- The controller in the MVC pattern can, relatively easy, be extended with authorization functionality, which is positive for **security**. Due to its organization, this is slightly less easy in the PAC pattern, but if all access to the system starts at the top agent, the control component of that agent can be extended with authorization functionality.

4.2 Example

An example from the fire alarm system domain is related to concurrency. In the functional architecture in figure 15, it is assumed that reading of inputs and potentially generating corresponding outputs take place concurrently. Assuming that light-weight pre-emptive threads are used, this solution can be evaluated with respect to efficiency and reliability. The cost of threads and the fact that pre-emptive threads are error-prone since they may cause racing conditions when accessing shared data, necessitates investigation of other solutions.

To address this, we decide to make use of an application-level scheduler and the notion of a periodic object. A *periodic object* is an interface containing a *Tick* method and that is regularly activated by the scheduler. Concrete subclasses implement their own *Tick* method that defines one slice of the periodic execution of an active object. The degree of concurrency achieved by this solution depends on the “thinness” of the largest slice. This design rule is an example of an architectural pattern that influences the complete architecture since all inputs and outputs are affected. In figure 16, the result of the transformation is presented.

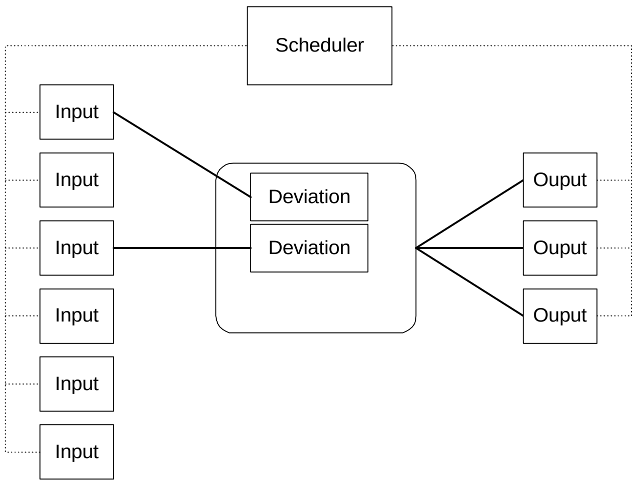


FIGURE 16. **Application-level scheduler for the fire-alarm system**

5. Apply design pattern

In the previous sections, we have discussed architecture transformation techniques that have an architecture-wide impact, i.e. the imposition of a style or architectural pattern.

A less dramatic transformation is the application of a design pattern on a part of the architecture. For instance, an *abstract factory* pattern [Gamma et al. 94] might be introduced to abstract the instantiation process for its clients. The abstract factory pattern increases maintainability, flexibility and extensibility of the system since it encapsulates the actual class type(s) that are instantiated, but decreases the efficiency of creating new instances due to the additional computation, thereby reducing performance and predictability. Different from imposing an architectural style, causing the complete architecture to be reorganized, the application of a design pattern generally affects only a limited number of components in the architecture. In addition, a component can generally be involved in multiple design patterns without creating inconsistencies.

Design patterns have received considerable amounts of attention in the literature both in books, e.g. [Gamma et al. 94] and [Buschmann et al. 96], and conference proceedings, e.g. the PLOP conferences [Coplien & Schmidt 95], [Vlissides et al. 96] and [Martin et al. 98]. Patterns have been described for a variety of problems, including persistence, distribution, user-interfaces, reactive systems and processes, but also for specific domains, such as business objects, hypermedia and transport systems. These patterns, generally, describe a solution that improves reuse and maintainability, while, up to some extent, sacrificing operational quality attributes, such as performance and real-time behavior, and understandability, i.e. patterns often make the design of a system more complex.

Design patterns can be categorized in many different ways. [Gamma et al. 94] use a two dimensional classification, where one dimension is class and object patterns and the second dimension addresses creational, structural and behavioral patterns. [Buschmann et al. 96] uses a different classification and defines the following categories: structural decomposition, organization of work, access control, management and communication.

In the next section, we present a few general-purpose design patterns and discuss the effect of these patterns on the quality attributes. Since design patterns make local rather than architecture-wide transformations, the quality attributes are not affected as much by a design pattern either. An example of applying a design pattern is presented in section 5.2.

5.1 Design Patterns and Quality Attributes

Design patterns are used to improve quality attributes. Patterns do not change the functionality of the system; only the organization or structure of that functionality. Consequently, when applying a design pattern, it is important to consider the effects on the quality attributes. An impressive number of design patterns has been proposed during recent years, published in, among others, the proceedings of the aforementioned PLOP conferences. In this section, we briefly discuss three of the classical design patterns presented in the Gang of Four (GoF) book [Gamma et al. 94], i.e. Facade, Observer and Abstract Factory. The primary difference between the presentation of design patterns in the GoF book and other publications and in this book is the fact that we explicitly treat the use of design patterns as transformations taking an architectural design from one version to the next, rather than presenting patterns as static structures.

Facade. he Facade design pattern is used to provide a single, integrated interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that simplifies the use of the subsystem. The structure of a subsystem incorporating the Facade design pattern often looks as in figure 17. The subsystem is defined as a component containing the entities that are part of the subsystem. The function of the subsystem component is basically twofold. The first is the coordination between the entities in the subsystem, whereas the second function is to provide an integrated interface to clients of the subsystem. Below, the effects of the Facade pattern on the quality attributes are discussed.

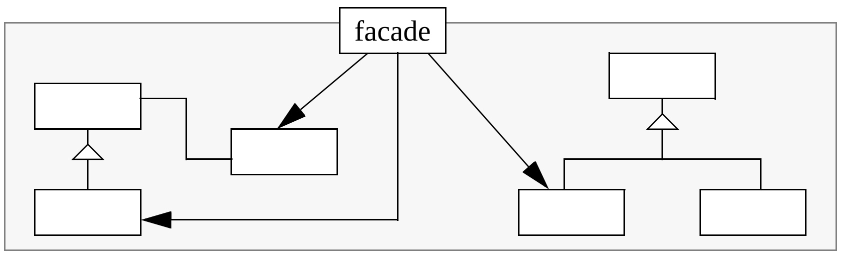


FIGURE 17. Structure of the Facade **design pattern**

- The **performance** of the architecture employing the facade pattern is reduced due to the indirection in the communication between external and internal components, but also due to the coordinator role the facade plays within the subsystem.
- The facade decreases the coupling between external components and components inside the facade. The decreased dependency is positive for **maintainability**. However, the evolution of the internal components often leads to many changes at the facade interface.
- The reduced complexity of interaction between subsystems may be positive for **reliability**, i.e. rather than n-to-n, the connections are reduced to n-to-1. However, the facade interface may easily grow complex, which affect reliability negatively.
- The facade pattern has no major effects on **safety**.
- The fact that the facade pattern provides a single point of access to the subsystem functionality is positive for **security** since, among others, authorization can be performed more easily.

Observer. The Observer design pattern deals with the situation where several components are depending on state changes in another component; when the component changes state, all its dependants are notified. The Observer pattern is a widely used in object-oriented systems since is significantly decreases the dependency between an object and its dependent objects. The structure of the Observer pattern is shown in figure 18.The quality attributes are affected as discussed below by the pattern.

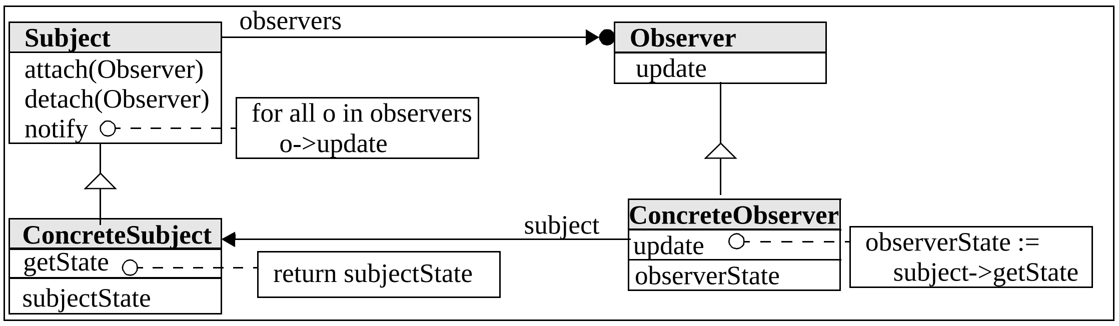
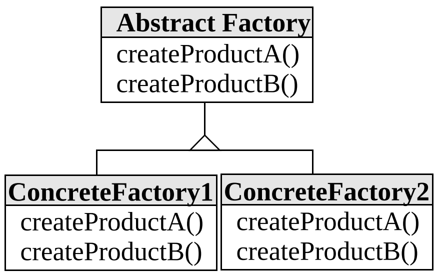


FIGURE 18. **Structure of the** Observer **pattern**

- The observer pattern avoids the situation where dependent components poll the component for state changes, which is positive for **performance**. However, the pattern does by default updates all dependent components, possibly leading to unnecessary computation. The situation where the subject would send the data that the dependent component requires whenever a relevant state change took place and the dependent component currently was interested in state changes would be more efficient, but also increase the dependencies between the subject and the observers.
- Compared to alternative approaches, such as discussed above, the **maintainability** of the observer pattern is positively influenced. Observing components can be dynamically added and received without changes to the system.
- The decreased dependencies between the subject and the observers is generally positive to **reliability** since a failure at one of the observing components does not affect the other components, assuming the failure is handled appropriately.
- **Safety** and **security** are not affected noteworthy by the pattern.

Abstract Factory. The abstract factory pattern provides an interface to creating a family of related objects without specifying their concrete classes. Often when using reusable software, such as object-oriented frameworks, selecting one type of component in part of the framework restricts the selection of component types in other parts of the framework. The typical example are user-interface frameworks. When one selects an X-windows component such as a window, one is required to select all component types from the X-windows sub-hierarchies. If one would explicitly use the names of the component types, this would complicate the use of the software for different platforms or the evolution of the software since changing replacing a component type with an updated one will require the software engineer to replace the name of the component type at all points of usage. When the abstract factory pattern is used, the name of the component type is only used in the factory. In figure 19, the structure of the pattern is shown. Since the abstract factory pattern is only used when instantiating objects, it affects the quality attributes even less than the design patterns discussed earlier. Below, this is discussed in more detail.



<To be finished>

FIGURE 19. **Abstract Factory design pattern**

- The abstract factory pattern requires more computation for the creation of new components, which affects **performance** negatively. However, once the component is created, the pattern does not play a role, so the impact on performance is minimal in most systems.
- Since explicit references to component types are avoided in the majority of the code and concentrated in the factory components, **maintainability** is improved. Evolving the system by adding new component types will result in a single point of change, rather than many changed distributed throughout the code.
- **Reliability**, **safety** and **security** are not affected by the pattern.

5.2 Example

To illustrate the use of design patterns, an example from the measurement system architecture is used. Sensor components represent physical entities that measure some aspect of the real-world. The relation between the physical entity and the software representation in the form of the sensor component needs to be maintained. In general, one can identify three alternative approaches to achieving this. The first is to wait for a client to ask for the value of the sensor. At that point the sensor communicates with its real-world counterpart, calculates the value and returns it to the caller. The second approach is for the sensor to periodically request the current value from the real-world sensor and store the calculated value. Clients requesting the value will then receive the stored value, that may be slightly outdated. Finally, the real-world entity may notify its software representation whenever it changes state, e.g. through an interrupt. When notified, the sensor component retrieves the new data and calculates the new value, which is stored and send to a client whenever it requests the value.

The problem is that the approach to use depends on the situation and the approach may vary independently of the domain functionality of the sensor. To avoid maintainability problems, we made use of the Strategy design pattern that factors out the different update approaches as separate update strategies. Upon instantiation, but even at run-time, the sensor is configured with a particular update strategy that is used by the sensor. The resulting structure is shown in figure 20.

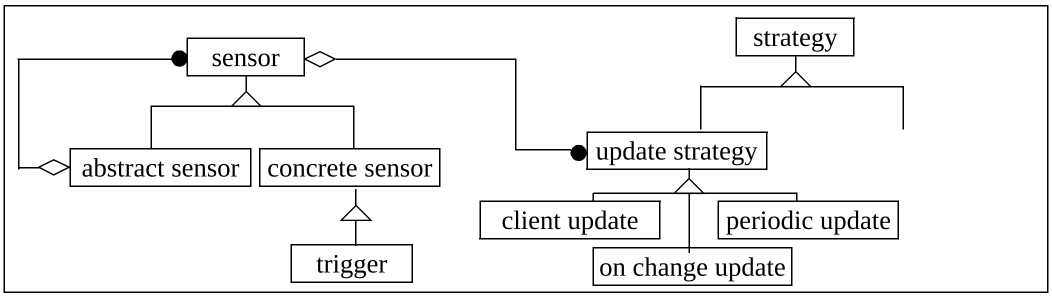


FIGURE 20. **The Strategy pattern applied to the Sensor component**

6. Convert quality requirements to functionality

The three types of architecture transformation techniques discussed in the sections above are primarily concerned with reorganizing the functionality already present in the architecture. By reorganizing the functionality in new structures, the quality attributes of the architecture are improved. However, this is not always sufficient. Often, it is necessary to add additional functionality to the system not concerned with the application domain but primarily with improving the quality attributes.

The fourth and final type of transformation is the conversion of a quality requirement into a functional solution that consequently extends the architecture with functionality not related to the problem domain but used to fulfil the requirement. Although this type may require minor reorganizations of the existing architecture, the primary effect is the addition of new components and functionality. Exception handling is a well-known example that adds functionality to a component to increase the fault-tolerance of the component.

Many examples of this type of transformation exist, although we have traditionally not necessarily considered these examples as quality attribute-improving transformations. In the next section, two examples are discussed, i.e. self-monitoring and redundancy. Section 6.2 presents an example of this type of transformation.

6.1 Converted quality requirements and quality attributes

The type of transformation discussed in this section is different from the types discussed earlier since it adds functionality to the architecture, rather than reorganizes the already present functionality. In this section, we discuss two examples of this type of transformation.

Self-monitoring. Although an architecture may provide the required functionality under ideal circumstances, it often is unclear how well it handles unexpected situations, such as failing components, hardware that breaks and external systems that go down. To address this, one solution is to add self-monitoring to the system. Analogous to the architectural patterns for GUI, i.e. MVC and PAC, two alternatives exist. One can add a layer on top of the system that monitors the behavior of the system, similar to the base-layer of the system that monitors relevant behavior in the real-world. Alternatively, the self-monitoring behavior is modeled as a hierarchy that mirrors the existing component hierarchy. Independent of the alternative chosen, the behavior of the monitoring includes not just identifying and reporting problems, but generally also actions to solve the problems. Below, the effect of adding self-monitoring functionality to the architecture on the quality attributes is discussed.

- Self-monitoring adds an additional subsystem (either logical of virtually) to the architecture that requires computational resources, but is unrelated to the application domain. Consequently, there is a, sometimes considerable, **performance** impact.
- The subsystem for self monitoring increases the system size which has an immediate negative effect on **maintainability**. However, there is an additional effect as well: the code related to the application domain has to be

mixed with the code for monitoring in order to detect erroneous situations. This increases the complexity of existing architecture, with corresponding effects on maintainability.

- Although this transformation has a negative impact on both performance and maintainability, **reliability** is definitely improved. The ability of the system to detect problems and take countermeasures is increased.
- The same line of reasoning holds for **safety**. The ability to detect potentially hazardous situations, increases the likelihood that the system, or part of it, can be brought to a fail-safe state.
- Self monitoring can be used to identify errors, but also to detect suspicious patterns of behavior, which could be used for improving **security**. However, this type of application is less frequent than the earlier two.

Redundancy. To increase the fault-tolerance of software, the notion of redundancy, originating from hardware, has also been applied to software. Since it serves little use to use multiple copies of the same software module, different implementations of the same requirement specification are required, i.e. N-version programming [Storey 96]. N-version programming is often complemented with recovery blocks, allowing the system or a module, upon the detection of an error, to abort and return to a system or module state saved earlier at a recovery point. Typical computation consists of creating a recovery point, for each version of a module, compute the results and test the values for acceptance, and finally accept the values as correct data, allowing the module to create the next recovery point. An alternative approach is to order the implementations as primary, secondary, etc. and to only execute the, e.g. secondary, version if the acceptance test for the, e.g. primary, version failed.

- The **performance** of the system is seriously decreased using N-version programming, especially when using the first approach. However, this can, at least partially, be addressed by using a multi-processor system and executing the versions in parallel on different processors.
- Obviously, N versions of the same module will also need to be changed when the requirements change, thus having a considerable impact on **maintainability**. In addition, the use of recovery points complicates the code considerably.
- Obviously, the primary reason for using redundancy is to increase the **reliability** and the **safety** of the system. The risk for the system failing due to software errors is decreased considerably.
- Redundancy has no major effects on **security**.

6.2 Example

In the example fire alarm system, there are quality requirements related to self-monitoring and availability. In certain cases, detected faults should be handled using hardware redundancy, whereas in other cases problems should be indicated to the fire brigade or the persons responsible for system maintenance. These requirements are, at least, partially fulfilled by transforming them to functional requirements similar to the basic alarm requirements. The corresponding architecture extended with entities dealing with self-monitoring is presented in figure 21.

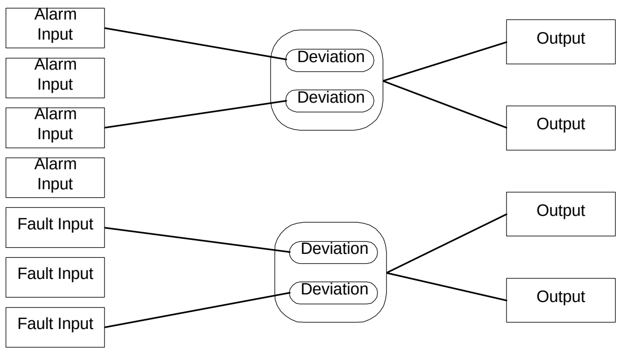


FIGURE 21. *Self-monitoring in the fire-alarm system architecture*

The above solution is a typical case where a logical additional layer is added to the system for self-monitoring. In the dialysis system, we used an alternative approach. As discussed in chapter 3, the primary archetype in the dialysis system architecture is the Device. As shown in figure 22, each domain device may have one or more alarm detector devices associated with it that monitor for potentially hazardous situations. Once an alarm has been detected, the alarm detector device activates an appropriate alarm handler by sending an alarm event.

7. Distribute requirements

In the previous sections, we have discussed four types of architecture transformation that improve, or at least affect, the quality attributes of the architecture, while leaving the domain functionality in tact. However, the architecture can only facilitate the fulfillment of quality requirements. The design and implementation of the components are of crucial importance for the quality attributes of the final system.

The increased awareness of the importance of an explicit design of the architecture of a software system has not decreased the importance of the components that make up the architecture.

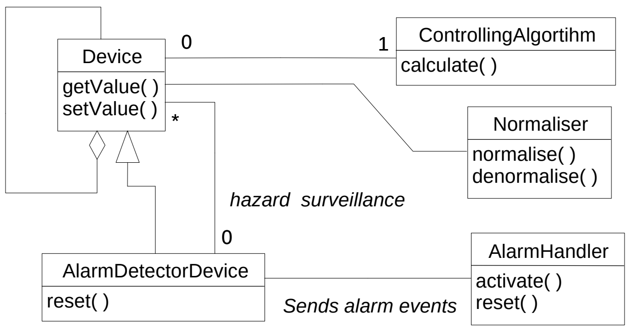


FIGURE 22. Self-monitoring in the dialysis system architecture

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

Fire alarm systems are often implemented as a distributed system where one CPU-based system controls one building. Several such systems communicate with each other and the basic requirement is that an alarm detected on one system should be indicated on all other systems. This requirement can be achieved by enforcing a copy of the ”blackboard” to be available on all systems. This distribution can be effectuated by means of communication software operating at a lower layer and assuring that consistent copies of the blackboard are distributed throughout the system. The resulting architecture is shown in figure 23.

The quality requirements stating how well the fire alarm system should cope with communication problems is assigned to the communication software, effectively distributing a system requirement to a system component.

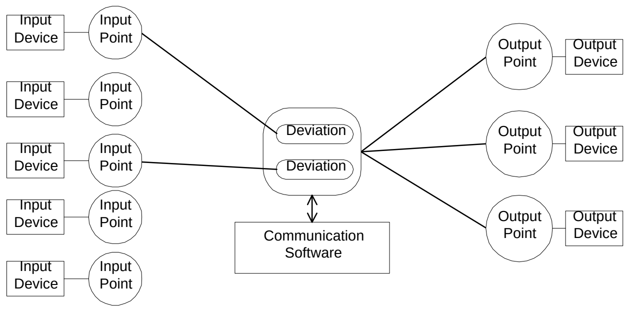


FIGURE 23. ***Dividing fault-tolerance over computation and communication***

8. Conclusion

The design of software architectures as proposed in this book consists of three main phases, i.e. functionality-based architectural design, assessment of the quality attributes of the software architecture and transformation of the architecture to improve the quality attributes that do not fulfil the requirement specification. In this chapter, we have discussed the final phase, i.e. architecture transformation.

Architecture transformation is concerned rearranging the software architecture designed based on the domain model and the functional requirements and to extend this architecture with additional components that address quality attributes rather than functional requirements. In this chapter, we have discussed four categories of architecture transformation. The first category is the imposition of an architectural style, such as a pipes&filters or layered architectural structure. Although architectural styles can be merged, styles tend to be predominant in the architecture and merging should be performed with care. The second category is the imposition of an architectural pattern. An architectural pattern is different from a style in that it is not predominant, but can be merged with most styles and other architectural patterns. An architectural pattern declares design rules on how an aspect of the software architecture is solved, such as concurrency or persistence. The third category of architecture transformation is the use of a design pattern. Design patterns do not have an architecture-wide impact, but tend to have more local effects. Although many design patterns have been proposed, most patterns improve reusability and maintainability while sacrificing the performance and real-time attributes. The final category of architecture transformation is the conversion of a quality requirement into functionality. Different from earlier categories that primarily focus on rearranging the existing functionality, this category mainly extends the software architecture with new functionality. Examples of the category include self-monitoring and redundancy.

Finally, it is important to understand that the discussed architecture transformations are part of a larger process that starts with identifying what quality requirements are not fulfilled. Subsequently, for each quality attribute, the inhibiting factors and locations in the architecture are identified. Thirdly, the most appropriate transformation is selected and, finally, the transformation is performed. Generally, the software architecture is subject to multiple transformations that need to be combined to lead to an integrated result. As a last step, the quality requirements for the software architecture need to be distributed to the architecture components. The system as a whole will only function as predicted if the components provide their required quality levels as well as the architecture.

9. Further Reading

<to be written>

CHAPTER 6

References

[Allen & Garlan 97]. R. Allen, D. Garlan, 'The Wright Architectural Specification Language,' *Draft paper*, CMU, 1997.

[Argyris et al 85]. C. Argyris, R. Putnam, D. Smith, *Action Science: Concepts, methods, and skills for research and intervention*, Jossey-Bass, San Fransisco, 1985.

[Bass et al. 98]. L. Bass, P. Clements, R. Kazman, *'Software Architecture In Practise'*, Addison Wesley, 1998.

[Bengtsson & Bosch 99a]. PO Bentsson, J. Bosch, 'Haemo Dialysis Software Architecture Design Experiences', *Proceedings of the 21st International Conference on Software Engineering* (ICSE'99), 1999.

[Bengtsson & Bosch 99b]. PO Bentsson, J. Bosch, 'Architecture Level Prediction of Software Maintenance', *The 3rd European Conference on Software Maintenance and Reengineering* (CSMR'99), 1999.

[Binns *et al.* 94]. P. Binns, Matt Englehart, M. Jackson, S. Vestal, 'Domain-Specific Software Architectures for Guidance, Navigation and Control,' *Honeywell Technical Report*, 1994.

[Bengtsson & Bosch 99]. PO Bengtsson, J. Bosch, 'Architecture Level Prediction of Software Maintenance,' Proceedings of the Third EuroMicro Conference on Software Maintenance and Reengineering, pp. xx-xx, 1999.

[Boehm 96]. B. Boehm, ‘Aids for Identifying Conflicts Among Quality Requirements,’ *International Conference on Requirements Engineering (ICRE’96),* Colorado, April 1996, and IEEE Software, March 1996.

[Booch 94]. G. Booch, *Object-Oriented Analysis and Design with Applications* (2nd edition), Benjamin/Cummings Publishing Company, 1994.

[Bosch & Molin 99]. J. Bosch, P. Molin, ‘Software Architecture Design: Evaluation and Transformation,’ *Proceedings of the Engineering of Computer-Based Systems Conference,* August 1999.

[Bosch 98a]. J. Bosch, ‘Design Patterns as Language Constructs,’ *Journal of Object-Oriented Programming,* Vol. 11, No. 2, pp. 18-32, May 1998.

[Bosch 98b]. J. Bosch**, ‘**Object Acquaintance Selection and Binding,’ accepted for publication in *Theory and Practice of Object Systems,* February 1998.

[Bosch 98c]. J. Bosch, ‘Product-Line Architectures in Industry: A Case Study,’ submitted, June 1998.

[Bosch 99]. J. Bosch, ‘Design of an Object-Oriented Framework for Measurement Systems,’accepted for publication in Object-Oriented Application Frameworks, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, (forthcoming) 1999.

[Brooks 95]. F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering,* Addison Wesley Longman, 1995.

[Buschmann *et al.* 96]. F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns, John Wiley* & Sons, 1996.

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

[Coplien & Schmidt 95]. J.O. Coplien, D.C. Schmidt, Pattern Languages of Program Design, Addison-Wesley, 1995.

[Dijkstra 76]. E.W. Dijkstra, *A Discipline of Programming,* Prentice-Hall International, 1976.

[Dikel et al. 97]. D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, ‘Applying Software Product-Line Architecture,’ *IEEE Computer*, pp. 49-55, August 1997.

[Fenton 96]. N.E. Fenton, S.L. Pfleeger, *Software Metrics - A Rigorous & Practical Approach* (2nd edition), International Thomson Computer Press, 1996

[Gamma *et al.* 94]. E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Garlan et al. 94]. D. Garlan, R. Allen, J. Ockerbloom, ‘Exploiting Style in Architectural Design Environments,*’ Proceedings of SIGSOFT ‘94 Symposium on the Foundations of Software Engineering,* December 1994.

[Gilb 88]. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.

[Jacobsen et al. 92]. I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-oriented software engineering. A use case approach*, Addison-Wesley, 1992.

[Jacobsen et al. 97]. I. Jacobsen, M. Griss, P. Jönsson, *Software Reuse - Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.

[Johnson & Foote 88]. R. Johnson, B. Foote, ‘Designing Reusable Classes,’ Journal of Object-Oriented Programming, Vol. 1 (2), pp. 22-25, 1988.

[Jones 86]. C.B. Jones, *Systematic Software Development using VDM*, Prentice-Hall Series in Computer Science. Prentice-Hall International, 1986.

[Kazman et al. 94]. 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.

[Kazman et al. 98]. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, ‘The Architecture Tradeoff Analysis Method,’ *Proceedings of ICECCS’98,* (Monterey, CA), August 1998.

[Kiczales et al. 97]. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, ‘Aspect-Oriented Programming,’ *Proceedings of ECOOP’97* (invited paper), pp. 220-242, LNCS 1241, 1997.

[Kruchten 95]. P.B. Kruchten, 'The 4+1 View Model of Architecture,' IEEE Software, pp. 42-50, November 1995.

[Liu & Ha 95]. J.W.S. Liu, R. Ha, ‘Efficient Methods of Validating Timing Constraints,’ in *Advanced in Real-Time Systems*, S.H. Son (ed.), Prentice Hall, pp. 199-223, 1995.

[Luckham et al. 95]. D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann, ‘Specification and Analysis of System Architecture Using Rapide,’ *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, 21(4):336-355, April 1995.

[Macala et al. 96]. R.R. Macala, L.D. Stuckey, D.C. Gross, ‘Managing Domain-Specific Product-Line Development,’ *IEEE Software*, pp. 57-67, 1996.

[Martin et al. 98]. R.C. Martin, D. Riehle, F. Buschmann, *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

[Molin and Ohlsson 97?]. P. Molin, L. Ohlsson, ‘Points & Deviations - A pattern language for fire alarm systems,’ in *Pattern Languages of Program Design 3*, Addison-Wesley, 1997?.

[Molin 97]. P. Molin, ‘Towards Local Certifiability in Software Design,’ Licentiate Thesis, Lund University, ISSN 1101-3931, 1997.

[Neufelder 93]. Ann Marie Neufelder, *Ensuring Software reliability*, Marcel Dekker, inc., 1993.

[ObjectStore 93]. ObjectStore, Documentation ObjectStore Release 3.0 for Unix Systems, December 1993.

[Ogden *et al.* 94]. W.F. Ogden, M. Sitaraman, B.W. Weide, S.H. Zweben, ‘Part I: The RESOLVE Framework and Discipline - A Research Synopsis,’ *Software Engineering Notes* 19, 4, pp. 23-28, October 1994.

[Perry & Wolf 92]. D.E. Perry, A.L.Wolf, ‘Foundations for the Study of Software Architecture,’ *Software Engineering Notes*, Vol. 17, No. 4, pp. 40-52, October 1992.

[Raise 95]. The RAISE Method Group, *The RAISE Development Method*, Prentice Hall, 1995.

[Richardson & Wolf 96]. D.J. Richardson, A.L. Wolf, ‘Software Testing at the Architectural Level,’ *Proceedings of the Second International Software Architecture Workshop*, pp. 68-71, San Francisco, USA, October 1996.

[Rumbaugh et al. 91]. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.

[SEI 97]. L. Bass, P. Clements, S. Cohen, L. Northrop, J. Withey, ‘Product Line Practice Workshop Report, *Technical Report CMU/SEI-97-TR-003*, Software Engineering Institute, June 1997.

[Shaw & Garlan 94]. M. Shaw, D. Garlan, ‘Characteristics of Higher-level Languages for Software Architecture,’ *CMU-CS-94-210*, December 1994.

[Shaw et al. 95]. M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik, ‘Abstractions for software architecture and tools to support them,’ *IEEE Transactions on Software Engineering*, April 1995.

[Shaw & Garlan 96]. M Shaw, D. Garlan, *Software Architecture - Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[Shlaer & Mellor 97]. S. Shlaer, S.J. Mellor, 'Recursive Design of an Application-Independent Architecture,' IEEE Software, pp. 61-72, January/February 1997.

[Simos 97]. M.A. Simos, ‘Lateral Domains: Beyond Product-Line Thinking,’ Proceedings Workshop on Institutionalizing Software Reuse (WISR-8), 1997.

[Smith 90]. C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[Storey 96]. N. Storey, Safety-Critical Computer Systems, Addison-Wesley, 1996.

[Szyperski 97]. C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

[Taligent 95]. Taligent, *The Power of Frameworks*, Addison-Wesley, 1995.

[Telelarm 96]. TeleLarm, Framework Design Document U00269, 1996.

[Terry *et al.* 94]. A. Terry, F. Hayes-Roth, Erman, Coleman, Devito, ‘Overview of Teknowledge’s DSSA Program,’ *ACM SIGSOFT Software Engineering Notes,* October 1994.

[Vlissides et al. 96]. J.M. Vlissides, J.O. Coplien, N.L. Kerth, Pattern Languages of Program Design 2, Addison-Wesley, 1996.

[Wirfs-Brock et al. 90]. R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[Wirth 71]. N. Wirth, ‘Program Development by Stepwise Refinement,’ *Communications of the ACM*, (14):221 - 227, 1971.