Chapter 6
From Problem to Solution using Quality Attributes and Design Aspects

Abstract

It is commonly accepted that quality attributes shape the architecture of a system. There are several means via which the architecture can support certain quality attributes. For example, to deal with reliability the system can be decomposed into a number of fault containment units, thus avoiding fault propagation. In addition to structural issues of architecture, qualities also influence architectural rules and guidelines, such as coding standards. In this paper we will focus on design aspects as a means of supporting quality attributes. An example of a design aspect is error handling functionality, which supports reliability. Quality attributes play a role in the problem domain; design aspects are elements in the solution domain.

We will use an industrial case to illustrate our ideas. The discussion ranges from how design aspects are defined in the architecture based on quality attributes, to how design aspects can be used to verify the realized system against the prescribed architecture. The industrial case is a product family of medical imaging systems. For this product family, the family members are constructed from a component-based platform. Here, it is especially useful to achieve aspect-completeness of components, allowing system composition without worrying about individual design aspects.

6.1 Introduction

Over the years, the complexity of software in embedded systems has increased dramatically; more and more functionality has to be realized in software. A consequence of the growing complexity is that it is becoming increasingly difficult to understand how the system as a whole behaves, how the various parts interact, what the impact of a change is, etc. This is mainly due to the fact that a system is usually considered from only one perspective, i.e. there is usually only one main architectural view. This view may, for example, focus on the functional parts into which the system is decomposed. It is relatively easy to relate some of the requirements of the system to such a view, for example, what it means if I add an additional functional block. Other requirements do not directly map onto such a view, for instance whether the system meets the reliability requirements. As a result, it is becoming more difficult to see whether the defined system architecture can meet the imposed requirements.

In addition to the difficulties in defining the right system architecture, the system realization also becomes harder to understand. We have to deal with systems that consist of several million lines of code. This code is usually structured in a one-dimensional way, that is, the system consists of components, which in turn consist of classes. This structuring is usually based on the main functionality of the system. Other concerns, such as the handling of errors, cannot easily be related to this structure, which adds to the complexity. This may lead to overly complex code.

So, improvements are needed both in the area of architecture and its requirements, and in the area of system realization. In this paper we want to show how quality attributes [4] and design aspects can contribute to this improvement, and how these two concepts are related. These concepts introduce additional views, making it easier to reason about specific concerns. Our experience is related to a number of industrial product families. In this paper we will use a medical imaging product family as an example. Especially in the context of a large product family, we propose to manage the complexity via multiple views. In this paper we want to explain:

- that multiple views are needed to master the complexity, both in requirements and design;
- that the concept of design aspects is very useful in helping to realize the quality attributes;
- that using design aspects adds structure to the architecture and the design process;
- how to use design aspects in system development;

- that aspect-completeness is important for a product family approach with a component-based platform.

This paper is structured as follows. In section 6.2 we discuss the basic ideas of quality attributes and design aspects and their relationship. After this more theoretical part, section 6.3 illustrates how quality attributes and design aspects can be applied in different stages during development, based on the industrial product family case study. In section 6.4, we describe related work. Conclusions can be found in section 6.5.

6.2 Quality Attributes and Design Aspects

In this section, we will start with an example of a so-called 'thread of reasoning' for the design of the medical imaging system product family (section 6.2.1). We will then briefly deal with the IEEE standard for architectural description, which deals with views and concerns (section 6.2.2), and finally define what we mean by quality attributes and design aspects and how they are related (sections 6.2.3 and 6.2.4).

6.2.1 Example of a Thread of Reasoning

The example in this section shows one thread of reasoning for the design of the medical imaging product family. This thread of reasoning starts with a stakeholder and ends with technical structures and mechanisms in the system realization, as depicted in Figure 6-1.

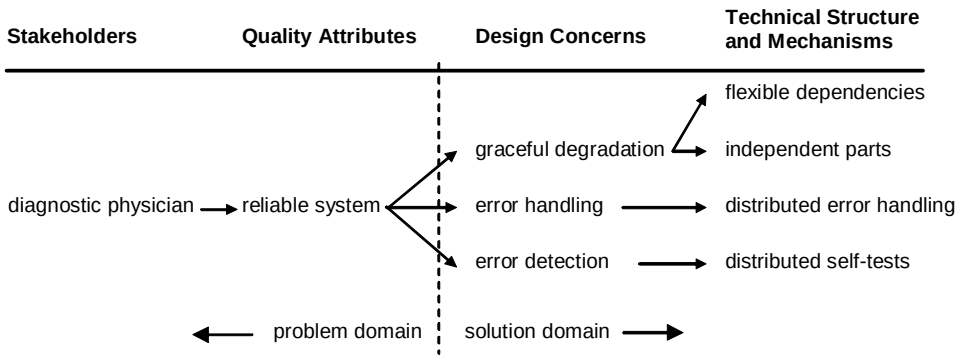

Figure 6-1 – Example of a Thread of Reasoning

Medical imaging systems are used to acquire and display images for interventional support or diagnostics. When used for diagnostics, a diagnostic physician makes images of the patient. For this physician it is very important

that he can rely on the system so that he can keep to his tight schedule. The reliability of the system is supported by a number of design concerns. Amongst other things, the system supports graceful degradation, i.e. when one part of the systems fails then other parts continue as well as they possibly can. The system is built from a number of relatively independent hardware and software parts, limiting the consequences of a failure in one of these parts. Furthermore, the dependencies between these parts are dealt with in a flexible way, i.e. when one part has to be reset due to failure, other parts depending on it continue with reduced functionality. Another design concern is error handling. Each unit is made responsible for its own error handling, in order to avoid cascading errors. The concern of detecting errors is addressed by providing self-tests, which help with the early identification of possible problem sources.

6.2.2 Concerns and Views

In the IEEE standard on Architectural Description [35], the concepts of concerns and views are modeled in the context of system architecture. Part of the model is illustrated in Figure 6-2. A system has a number of stakeholders, each of them having their own concerns. The system has an architecture as a basis for its realization. This architecture is described by an architectural description, consisting of views. A view is a representation of a whole system from the perspective of a related set of concerns. A viewpoint can be seen as a template for a view. Concerns and views thus play an important role in the description of an architecture. The standard gives a number of examples of views/viewpoints. For example, the Reference Model of Open Distributed Processing is mentioned, which has the enterprise viewpoint (concern: purpose of the system and roles played by the system), and the computational viewpoint (concern: functional decomposition of the system). This illustrates the relevance of viewpoints to the various stages of development.

In this paper we want to focus on two kinds of views, i.e. views that play a role in the problem domain, and views that play a role in the solution domain. The views in the problem domain are related to concerns of various stakeholders, both externally and internally. The views in the solution domain are related to concerns of internal stakeholders, mainly the system architect.

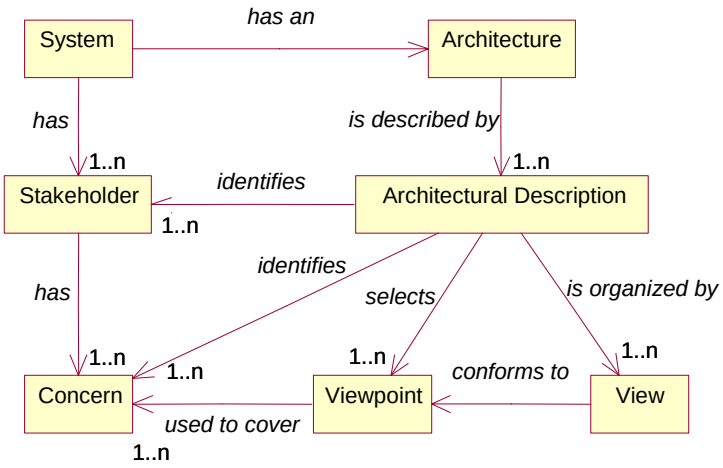

Figure 6-2 – Partial Model of IEEE 1471 Standard

6.2.3 Quality Attributes and Design Aspects

The quality attributes and design aspects discussed in this paper are defined as follows:

- A *quality attribute* of a system is an observable property of that system. A quality attribute can be observable at development-time (including testing and maintenance) or at run-time. Important quality attributes for our product family are reliability, safety, and modifiability.
- A *design aspect* of a system is a coherent part of the functionality realized by the artifacts[9] within the system that crosscuts the decomposition of the system into artifacts, i.e. a design aspect is relevant to each of the artifacts. Important design aspects for our software system include initialization, error handling, and testing.

An important difference between quality attributes and design aspects is that quality attributes deal with observable properties (problem space), whereas design aspects relate to functionality that must be realized inside the artifacts of the system (solution space). Quality attributes capture concerns that play a role in the problem domain. For example, how reliable should the system be? These concerns originate from different stakeholders, such as the end-used, the

[9] In our medical imaging case, the artifacts considered are units, components and classes (see section 6.3.6).

manufacturing department, the marketing manager, etc. In this paper we will restrict ourselves to the architecturally relevant issues of quality attributes. Views can be defined for the system based on these concerns. For the reliability concern, for example, a viewpoint can be defined that focuses on the reliability of the system behavior, consisting of:

- a detailed description of the architecturally relevant requirements, e.g. fluoroscopy must always be possible during an intervention;
- a quantification of the requirements, which can also be used to verify the system afterwards.

Similar views can be defined related to other quality attributes, covering the whole range of architecturally-relevant requirements. Quality attributes express different concerns. Dependencies can exist, however, such as the increased testability of the system which is also beneficial to the reliability in the example in section 6.2.1.

Design aspects are concerns that play a role in the solution domain. As shown in Figure 6-1, the concerns related to the quality attributes lead to a range of design concerns. Design aspects are a special kind of design concern. The following viewpoint is defined as an example for the error handling aspect:

- a general policy for error handling to be implemented by each component, e.g. localizing the error, logging the error, trying to repair the error, etc.;
- a format of how to log the error in the system log service;
- required and provided interfaces related to error handling for each component;
- a definition of supporting facilities for realizing the design aspects, e.g. the logging service, predefined interfaces, base classes.

In principle, a design aspect affects every artifact in the system, for example, every component contains some lines of code that deal with initialization or error handling. However, some artifacts may deal with a subset of the design aspects. Furthermore, a design aspect can also be relevant for a subset of the artifacts in a system, for example, the self-test aspect is related to hardware devices. A design aspect can require artifact-specific implementations, for example, the error handling of hardware piece X might be different from hardware piece Y. The design aspects must be defined in such a way that they form a complete decomposition of the functionality of an artifact. We should note here that the domain functionality, amongst others viewing and printing images, is also considered to be a design aspect in its own right.

Other approaches exist in which views are applied. For example, Kruchten [46] defines 4+1 views, namely logical, development, process, physical, and use case views. Each view deals with a number of concerns, for example, the process view deals with the performance, scalability and throughput concerns. The views as defined in our paper are more 'lightweight'. For example, we propose separate views for important quality attributes. Of course, as in the IEEE standard, models may be shared between these views. The views related to the design aspects are orthogonal to Kruchten's four views. For example, the initialization view has development elements (each component must deal with initialization), process elements (initialization sequence and parallelism), and physical elements (standards for coding the initialization in the components).

6.2.4 From Problem to Solution Domain

As described above, quality attributes are relevant in the problem domain. In the solution domain, measures must be taken to realize these quality attributes. Figure 6-3 shows three possibilities of realizing quality attributes in a system, namely:

- **System structure.** The structure of a system can support a number of quality attributes. For example, the modifiability quality attribute can be addressed using the model-view-controller pattern.
- **Design aspects.** Design aspects are also an important means of realizing quality attributes. For example, to support the reliability of the system each software component has a piece of functionality that deals with the errors locally, to limit their consequences. So, design aspects represent specific pieces of functionality for each component.
- **Rules & guidelines.** Another way of realizing certain quality attributes is to prescribe rules & guidelines for the development of artifacts. For example, it is important to minimize the dependency on external libraries to support the portability of the system. It must be prescribed which libraries may be used and which may not.

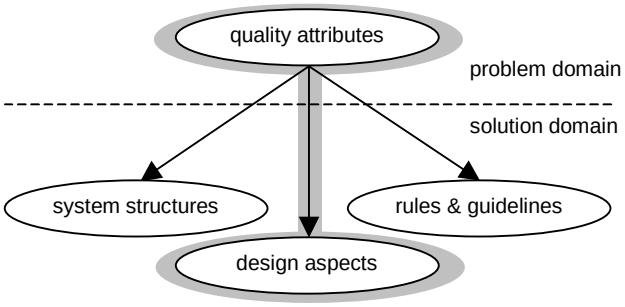

Figure 6-3 – From Problem to Solution Domain

Structures decompose the system into artifacts that form the primary decomposition of the system at a certain level. For example, the software part of our medical imaging system is decomposed into software units, which in turn consist of software components. The application of rules & guidelines is interwoven in the system; they do not lead to clearly identifiable pieces of functionality. Design aspects do lead to clearly identifiable pieces of functionality and are in some sense related to both structures and rules & guidelines: design aspects define a microstructure (see section 6.3.7) for the individual artifacts in the system, and rules & guidelines define how they should be applied within an artifact.

6.3 Designing with Aspects; a Case Study

In this section we will use the medical imaging system to discuss the development using design aspects and how they are related to quality attributes; we will focus on the shaded area in Figure 6-3. More on the system structures and rules & guidelines can be found in [114]. In section 6.3.1 we will briefly discuss the relevant quality attributes, followed in section 6.3.2 by an explanation of the primary system decomposition as a context for the crosscutting design aspects, and the most important design concepts in section 6.3.3. An example of applying design aspects is provided in section 6.3.4. The remaining sections (6.3.5 to 6.3.11) deal with the role of design aspects in the various development phases.

6.3.1 Quality Attributes

The most important quality attributes for the medical imaging system are reliability, safety, functionality, portability, modifiability (configurability, extensibility) and testability. More on these quality attributes can be found in [114]. The so-called System Requirements Specification (SRS) contains the key

requirements for the system. Here, the required functionality is described from the perspective of different stakeholders. For example, for the manufacturer it is important that he can easily install and uninstall options. Attention is also paid to the other quality attributes. They each introduce a separate view of the system. It is important to explicitly quantify the quality attributes to be able to verify later whether they have been realized.

6.3.2 Primary System Decomposition

From a physical point of view, the medical imaging system consists of a number of peripheral devices and a central system controller that combines the behavior of these devices and adds application functionality. The peripheral devices include the image processor, the geometry (which controls the major moving parts in our medical imaging system, such as the table on which the patient is lying), etc. From a functional point of view, the main blocks of the medical imaging system include image acquisition, image viewing and patient administration. Using these physical and functional views as input, among other things, a product family software architecture was defined for the system controller, as illustrated in Figure 6-4. This architecture consists of three main layers or subsystems, each containing a number of software units (about 30 units in total). The application layer represents the application knowledge and contains units that correspond to the main functional blocks of a medical imaging system, such as image acquisition. It is built on top of the technical layer, which contains units that provide abstractions of the various peripheral devices, such as the image processor or the geometry. The infrastructure layer provides support for the other two layers with facilities such as logging. Since the architecture is a product family architecture, it applies to all products within the family. We should note here that the primary decomposition of the system is based on the operational functionality in the domain, i.e. acquiring, processing and viewing images. This is because we expect the diversity in the product family, now and in the future, to mainly be in the operational functionality. We expect the other quality attributes and design aspects to remain fairly stable. The system architect identifies them up-front, using his experience and domain knowledge. Changing quality attributes and design aspects is then no longer a straightforward activity, since they would not match the primary decomposition of the system.

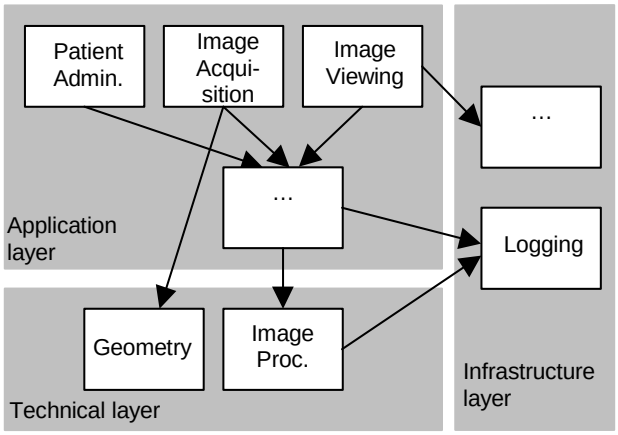

Figure 6-4 – Layer and Software Unit Structure

6.3.3 Design Aspects

The design aspects are also part of the architectural description. For the medical imaging product family, the software-related design aspects include:

- **Self-test aspect.** Various parts of the system, especially those related to the peripheral hardware, must perform self-tests in order to check whether they still function correctly. Most self-tests are activated during the start-up of the system.
- **Graceful degradation aspect.** Graceful degradation means that if there is a problem in one part of the system, the rest of the system continues to function as well as the circumstances allow.
- **Error handling aspect.** This design aspect deals with the strategy for handling errors that occur during system run-time.
- **Operational aspect.** The main operational functionality contributes to the functionality of the system for which it was intended. In our case, this is acquiring, processing, and viewing medical images.
- **Initialization and reset aspect.** This design aspect deals with starting-up and shutting down the system and initializing the components in the system. It is relevant in almost every type of system. Furthermore, restarts of an individual unit must be supported after the occurrence of a problem in that unit.
- **Wrapping aspect.** In our case, COM has been selected as middleware for the realization of the system. To limit the dependencies on this

technology, each component must have wrappers, shielding specific COM issues from the internal implementation.

- **Field-service aspect.** The field-service functionality deals with facilities like calibration of the hardware, performing tests, changing the configuration of the system, etc, each of which can be considered as design aspects in their own right. These facilities are for use by field-service engineers only, and are provided by units in the technical layer.
- **Software keys and licenses aspect.** Almost every system in our family is different due to the high level of configurability. Licensing functionality supports this by enabling additional options.
- **Logging aspect.** Each unit must do its own logging. A strategy has been defined for logging, which describes what data should be logged. The logged data will be used for off-line analysis, and can be used to increase the reliability of the system. For example, the usage of the system is registered, thus supporting pro-active maintenance of the system.
- **Debugging aspect.** This design aspect helps to find problems more easily and more quickly. It is not executed during normal use by the end-user. Tracing, or interfaces that allow easy fault introduction and examination of the intermediate states, may be added to the system.

The relationships between the quality attributes and design aspects are visualized in Figure 6-5. These relationships are part of various threads of reasoning as explained in section 6.2.1.

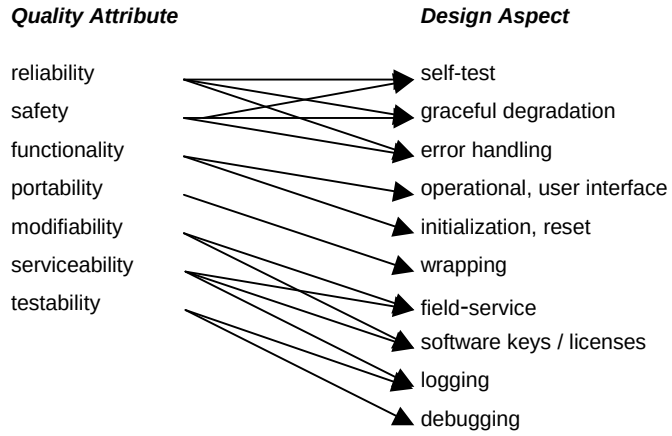

Figure 6-5 – From Quality Attributes to Design Aspects

6.3.4 Example of Design Aspects within a Unit

To help you understand what we mean by design aspects, we will now present an example based on the geometry unit from the medical imaging family. Within the product family there is variation in the mechanics of the geometry and the movements supported by these mechanics, such as the table on which the patient is lying. This is why we chose to structure this unit as a component framework into which plug-in components can be inserted which, amongst other things, provide the specific movements, see [113].

Object-orientation is used for the design and realization of the component framework and its plug-ins. The design is modeled in UML, and the classes are implemented in C++. Figure 6-6 illustrates part of the initialization aspect using a sequence diagram. The component framework is responsible for the initialization of the geometry unit. It registers itself with the Restart Service, which can restart units in case of failure. The plug-ins are also initialized, after which each plug-in is requested to provide its movements to the component framework, which manages them. The documentation provides more sequence diagrams on the initialization and other design aspects such as error handling. The design documentation also contains class diagrams. These can also be made for the various design aspects. Figure 6-7 gives an example for the user interface aspect. Every unit, including the geometry unit, can generate messages that are displayed on the graphical user interface. For this purpose the geometry unit has a UI message list, which contains messages. Objects within the geometry unit can add messages. These messages are read by classes related to the GUI. Class diagrams for other design aspects, such as the operational or the field service aspect, can also be found in the documentation. The design aspects also return in the code. For example, special interfaces are defined that are related to the design aspects. All interfaces related to field service functionality start with '`IFS`'. Special interfaces are also defined for design aspects such as initialization, reset, user interface, etc.

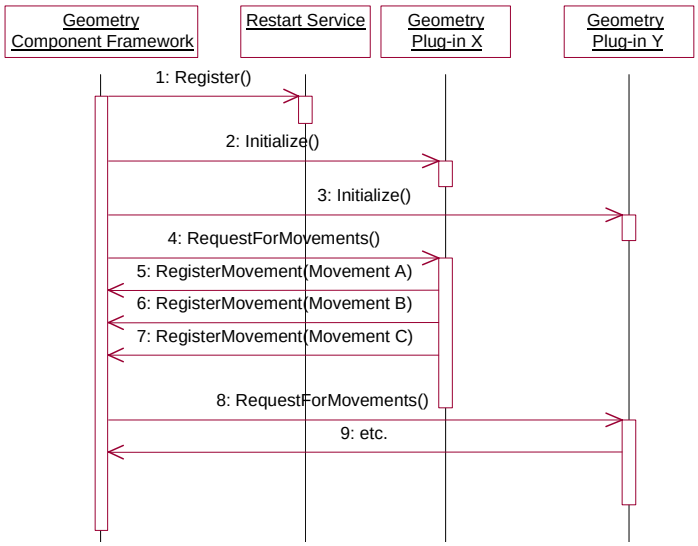

Figure 6-6 – Part of the Initialization Aspect

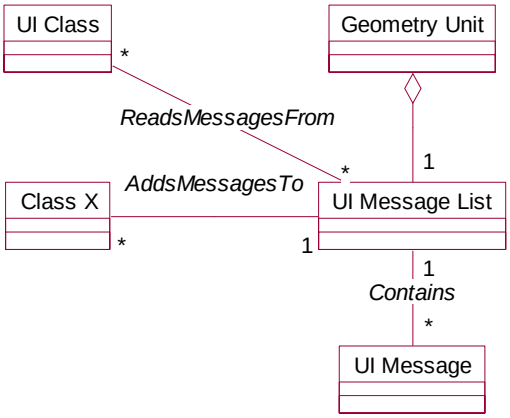

Figure 6-7 – Part of the User Interface Aspect

We should note here that in the architectural specification a design aspect is declared independently from the artifacts and the other design aspects. However, when getting to the design of the units of the system, these design aspects have to be made concrete, i.e. we have to specify what each design aspect means for a unit. This can result in unit-specific implementations of design aspects. From some examples given by papers on AOP (Aspect Oriented Programming, see [42]) one might get the impression that a generic implementation for each particular aspect suffices, and that this can be 'woven'

into all classes, by adding trace functionality to all classes, for example. However, in our situation we often need a unit-specific part for the design aspects. For example, an error in the geometry hardware should be handled differently from an error in the image processing hardware. In our case, the various design aspects are already integrated into the unit at design/implementation time. It is also possible to weave the unit-specific design aspect of a unit in at a later stage using an AOP language. However, one must realize that the unit with all its design aspects forms a logical whole; they are added as a complete entity to the system. It makes no sense to add the geometry unit without the corresponding error handling, for example.

6.3.5 Design Aspects as Part of the Architecture Description

As mentioned in the previous section, the design aspects also play a role in the documentation of the system. The overall architecture of the system is described in an SDS (System Design Specification). Several design concerns are addressed in the SDS. For example, it describes the decomposition (see section 6.3.2) as a means to deal with the complexity and to enable future evolution. However, not all requirements related to the architecture can be met with system structures. This is why separate sections are also devoted to the various design aspects. Here, the system architect describes the purpose of the design aspect. A description is also given of how the individual units and components of the system have to deal with the design aspect. It is important to stress here that design aspects are architectural concerns, because if each unit developer were to make different decisions on design aspects, it would be impossible to integrate the units. The system architect has defined the main principles for each of the design aspects; the unit designer has to make this specific for his unit. To support this, a documentation template has been made which covers the various design aspects for a unit. This helps the unit developer to deal with all relevant design aspects.

When going from requirements to design, it is very valuable to be able to trace design decisions back to the requirements; what is the reason for this design decision and what is the issue that needs to be solved? For the most important requirements this traceability is implemented via tables. The use of design aspects helps the traceability of requirements; we have one location with a description of which design aspects are needed and why. These design aspects are then applied in the individual units, knowing the purpose of the design aspects.

6.3.6 Design Aspects at Various Decomposition Levels

The software decomposition described in section 6.3.2 only deals with layers and software units on the central controller. More decomposition levels can be found within the medical imaging family, as illustrated in Figure 6-8. The system is decomposed into a number of units. A unit may require various disciplines for its realization. For example, the units shown in the lower part of Figure 6-4 require software in the central controller and peripheral devices containing dedicated hardware, embedded software on this hardware, mechanical parts (e.g. of the geometry), etc. The software part of the units on the central controller are again decomposed into software components, realized as separate DLLs or EXEs. These software components can be deployed separately. The software components themselves also consist of classes with methods and attributes. The design aspects can cover various realization disciplines, for example, the initialization aspect of the system not only concerns the software on the central controller, but also the peripheral devices. When a design aspect has been identified as being relevant for the software, the system architect has to decide on which decomposition level it will be dealt with, e.g. each unit has to take care of the restart aspect, each component has to take care of the field-service aspect, and certain classes need to take the persistency aspect into account. More on this topic can be found in [114].

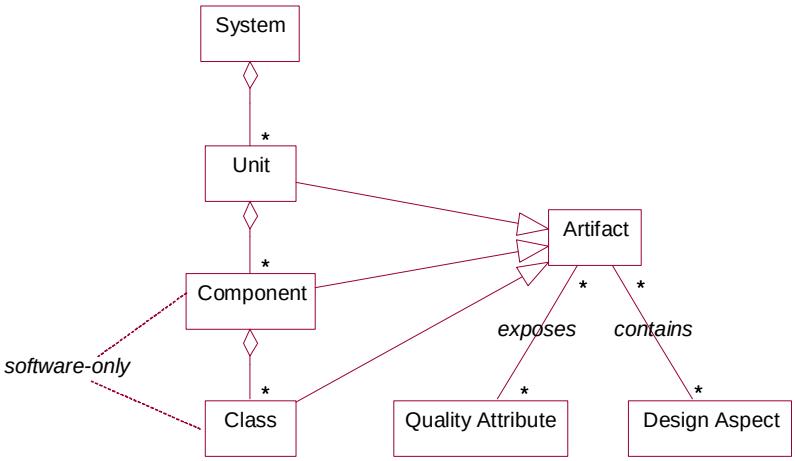

Figure 6-8 – System Decomposition Levels

6.3.7 Categories of Design Aspects

In addition to the primary decomposition of the system into units, as described in section 6.3.2, the system can also be decomposed according to the design

aspects. This decomposition is orthogonal to the primary one. We call it a secondary decomposition, not because it is less important, but because it does not match the organization of the development. As a consequence, it is not straightforward to see the system in this secondary decomposition; in section 6.3.10 we describe how we use tooling to view this decomposition in the realization. The secondary decomposition into aspects is illustrated in Figure 6-9. The design aspects are grouped into five categories, namely:

- **Supporting aspects.** These design aspects support the realization of the other design aspects. For example, the error handling aspect needs to log error messages in a log-file, or field service needs to store its settings in a persistent way.
- **Operational aspects.** These design aspects realized the main functionality of the system, which is related to the end-user (in our case the physician). In section 6.3.3, the operational functionality was presented as one aspect. However, if needed it can be divided into more specific aspects. For example, we can divide the operational functionality into an UI aspect, which is responsible for presenting the relevant information to the end-user, and an aspect that performs the requested operational functionality.
- **Configuration aspects.** These design aspects realize functionality for users other than the end-user of the system. This user group includes the field-service engineers, who have to perform maintenance, and the factory personnel who have to install options on the system, for example. A result of using these configuration aspects is that the operational functionality is configured, using the persistent data, for example. Since this functionality must be provided to different stakeholders of the system, the UI aspect may also be introduced here.
- **Controlling aspects.** These design aspects control the operational (and configuration) aspects at run-time. These aspects try to maintain an environment in which the external functionality works optimally. For example, self-testing is used to detect possible problems at an early stage and handle them via error handling. The initialization aspect brings the system into the correct initial state.
- **Additional aspects.** The previous categories are in principle sufficient, but additional design aspects may be needed to observe certain system behavior, for instance. In our case, there is an additional debugging aspect, which is not used in the field.

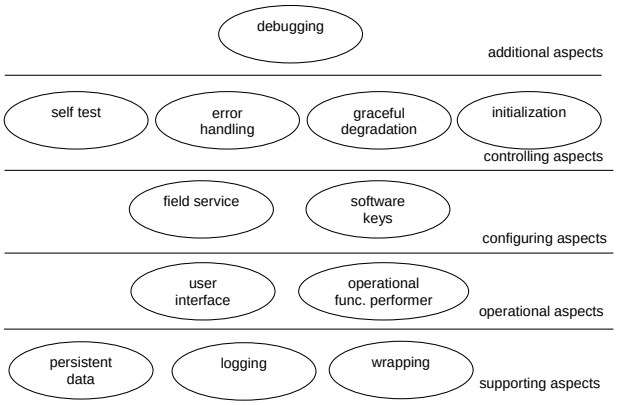

Figure 6-9 – Design Aspects Classification

A design aspect can have specific implementations in several or all artifacts. The internal structure of these artifacts can be considered to be a microstructure. This microstructure can be defined as a recurring pattern for all artifacts in the decomposition, and means that the pattern shown in Figure 6-9 reoccurs in principle in every artifact. However, some artifacts only have to deal with a subset of the design aspects.

The supporting aspects may be used by all other design aspects. The operational aspects are not allowed to invoke the configuration aspects directly, instead, the operation aspects are allowed to read the settings that are made by the configuration aspects. The controlling aspects above the operation and configuration aspects determine the environment in which they execute, by changing state variables, for example. The additional aspects like debugging may affect all other design aspects. The architect can define several rules about the dependencies between these design aspects, as will be discussed in more detail in section 6.3.10.

6.3.8 Architectural Pattern for Design Aspects

In this section, we will describe how a particular architectural pattern helps in realizing design aspects on a unit and component level within the medical imaging product family. For our medical imaging system family, we based the primary decomposition on the domain functionality. Such a decomposition can handle changes and extensions in the main functionality of the system, by applying component frameworks, for example. For some design aspects it is very useful to have supporting functionality in separate units/components, so that the specific design aspect functionality in the units/component is as small as possible.

To capture this generic design aspect functionality, component frameworks have been introduced for several design aspects, as illustrated in Figure 6-10. Such a system decomposition primarily contains artifacts (units and components) based on the domain functionality, but also has component frameworks for certain design aspects, such as initialization or field-service. The other artifacts are then plug-ins to these aspect component frameworks (see the discussion on infrastructure component frameworks in [113]). These component frameworks are located in the infrastructure layer as depicted in Figure 6-4. They provide a central point of control for some design aspects and contain generic functionality for the various design aspects. The architectural pattern that is introduced here thus consists of aspect functionality that is distributed across the units, plus an additional component framework, which contains the generic functionality for the design aspect. An example of such an infrastructure component framework is a component framework that deals with all field-service related functions, such as hardware calibrations, setting of configuration parameters, etc. Components with such field-service functions act as plug-ins and provide these functions as services to the field-service component framework. These services are made available to the field-service engineer via the component framework that serves as a central access point. This example also illustrates the different interfaces needed for the various stakeholders as discussed in section 6.3.7. Another example is the component framework that deals with restarts of units. During initialization, each unit must register itself with the restart framework, so that this component framework can restart a unit when needed. We stated earlier that the selection of the design aspects was considered to be fairly static; it has to be carefully considered by the system architect. But, since the generic part of the design aspects mentioned in these two examples is handled at one location, changes to these design aspects can be made relatively easily when needed, as long as it is only the generic part that is affected.

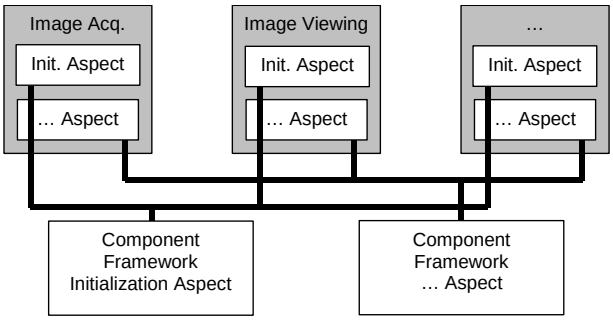

Figure 6-10 – Component Frameworks for Design Aspects

6.3.9 Interfaces and Classes

The interfaces of components are described using IDL. They are defined according to design aspects, i.e. for each relevant design aspect there are one or more interfaces and each interface deals with only one design aspect. In this way, separate concerns are handled by separate interfaces. Figure 6-11 illustrates how the context of a component can already be defined at an architectural level, without going into the specific functionality to be provided by the component. The figure shows a component with a number of interfaces provided. Some of these interfaces are mandatory, others are optional. In addition, services are defined that the component must or may use. The instantiations of such components are determined by the domain functionality. Defining these interfaces and services on an architectural level gives a better guarantee of interoperability. As discussed in section 6.3.7, there are relationships defined between design aspects. In the case of component interfaces, for example, each component may provide interfaces with field-service functionality. These interfaces may only be used by the field-service component framework, not by the operational design aspect, for example. Similarly, the interfaces dealing with initialization and reseting of components may only be used by specific components dealing with initialization and reset aspects. The correct application of these rules can be checked using architecture verification (see section 6.3.10).

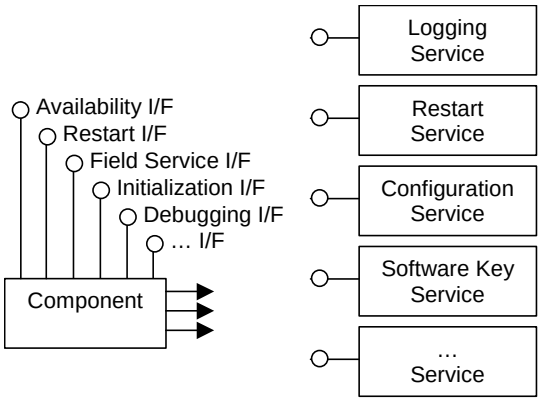

Figure 6-11 – Context of a Component

Components are part of a unit and contain objects. Figure 6-12 shows how design aspects can affect the internal structuring of a component. On the left-hand side we see the situation in which each object within the component deals with all relevant aspects, i.e. each object has a number of methods (possibly zero) for each design aspect. The definition of these objects is based on the domain and the operational functionality of the component, such as acquiring

images. On the right-hand side, we see the situation in which each object deals with a single design aspect, such as error handling or initialization. However, just as for the architectural pattern discussed in section 6.3.8, neither of these extremes is usually the best solution. A hybrid approach is chosen instead. This approach consists of identifying a number of classes based on the external functionality of the system, and supporting classes related to the other design aspects. As shown in Figure 6-6, the initialization aspect usually crosscuts a number of classes. Figure 6-7 shows that user interface functionality (or other external functionality) lends itself to identifying separate classes.

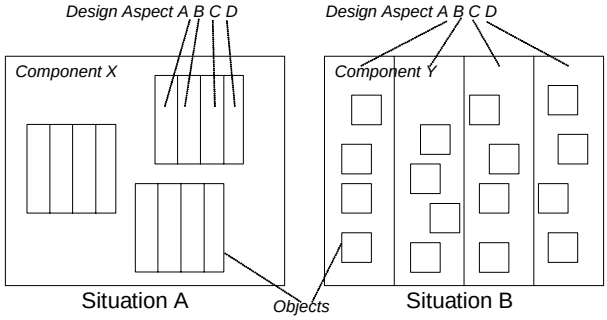

Figure 6-12 – Design Aspects and Component Structure

Section 6.3.8 discusses support for design aspects in terms of component frameworks. Other support is also provided for realizing design aspects. Support is given for some design aspects in the form of standard interfaces that must be provided by a component. For example, standard interfaces are provided for field-service functions like calibrations. Other support consists of base classes that form a basis for design and implementation. For example, for the operational aspect it was decided that each unit would act as a server and provide its functionality in the form of resources. Base classes are provided for the concepts of servers and resources. For design aspects where such concrete support is not possible, the only alternative is to provide more abstract rules & guidelines. Although not present in our development environment, one might also consider tool support for certain design aspects.

6.3.10 Testing and Verification

We have described how explicitly applying quality attributes and design aspects in the various development phases adds structure to the development process.

When considering the V-model[10], the use of quality attributes and design aspects can aid the testing at the various levels by adding structure and achieving completeness. Two examples of how design aspects help us are given below.

When considering testing, it is important to understand the architectural and design concepts, as this usually leads to a more effective test process. This also holds for the concept of design aspects, which helps to structure tests to achieve a complete test of a component, without missing any design aspect. Since design aspects are treated more or less uniformly across the various components, it is also possible to define a number of basic tests for some design aspects that must be performed for each component. For example, a number of tests can be defined to cover component initialization, or the handling of errors (e.g. the crashing of a used component).

Automated architecture verification is applied in our medical imaging product family, which verifies in an automated way whether the implicit architecture extracted from the implementation of a system is consistent with its specified architecture. This helps to maintain the conceptual integrity of the system. A tool has been developed that can automatically check (e.g. every night) whether the implementation (source code) is constructed according to the architectural rules, concerning design aspects amongst others. After the tool has been executed, an overview is available of the violations of the architectural rules in the source code that need to be repaired. Examples of design aspect-related rules that are checked are: 'Does each component provide the obligatory aspect-related interfaces', or 'Is the field-service interface not called by any component other than the field-service component framework'. What the verification tool does is to generate the relevant view and then automatically check the specific rule on that view. This is illustrated in Figure 6-13. The white part is the restart aspect in this example; the rest of the functionality is gray. Two important rules exist for the restart aspect, namely 'each unit must provide a restart interface', and 'the restart interface may only be called by the restart service'. A restart view is generated based on the situation on the left-hand side. In this generated view, it is easy to see that there are two violations of these rules, namely an illegal relation and a missing interface.

---

[10] The V-model originates from the work of Myers [69]. The model consists of one line with the activities ranging from requirements gathering to coding, and a second line with the corresponding test activities, together forming a V-shape.

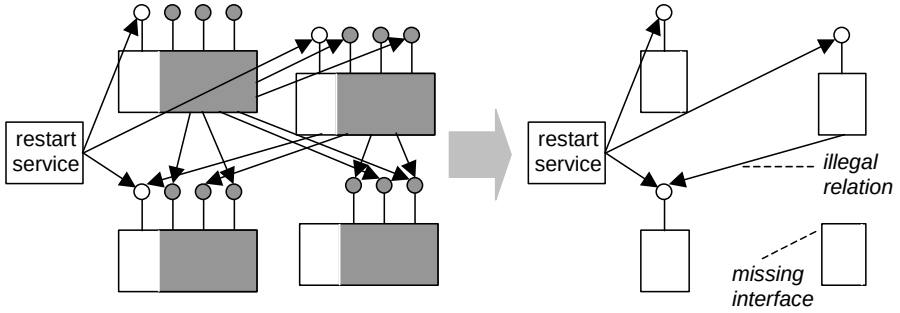

Figure 6-13 – Generating a Design Aspect View

6.3.11 Constructing Family Members

Our product family development is based on a component-based platform, see [87] and [113]. A family member is constructed by selecting the right components from the platform and adding additional specific components to realize the product specific behavior. Since each of the components in the platform deals with all relevant design aspects, these components are so-called 'aspect-complete'.

Why is aspect-completeness of components important in a product family approach? It should also be easy to build several family members with different features from the same platform at any moment in time. The main decomposition of the system is formed by units such as acquisition, image processing and geometry, each consisting of smaller components, such that localized adding, removing, and modification of new operational functionality is supported. It is important to have the changes localized in the system for several reasons, for example, it is easier to track the consequences of a change, it is easier to make an estimation of the effort needed to add a new feature, it is easier to distribute the work over the developers, it is easier to maintain the system, it is easier to configure the system, etc. However, applying a decomposition of the system that matches the direction of the diversity of the system is not enough; the components in the decomposition also have to be aspect-complete. For example, consider the case in which a new type of table can be used in the geometry of the system. This will require modification in the software of the geometry unit. This unit has been designed as a component framework in which the elements of the geometry, such as a table, can be plugged in. In this case, the plug-in is responsible for controlling the hardware of the table and sending commands to move the table, for example. By plugging this plug-in into the family member, the family member should be ready to use the new functionality. This is only possible if the plug-in component also deals with all relevant design aspects like error handling, initialization, etc. If this is

not the case, other parts of the system will have to be updated to deal with these design aspects related to the new table in the family member, introducing configuration-specific code. So, if not all design aspects are covered by this plug-in component itself, the localization of change can still not be achieved. This is why components have to be aspect-complete.

6.4 Related Work

Various publications use the terms quality attributes, aspects, views, or other related terms. The definitions of these terms have a number of properties in common. Firstly, there is the notion that different concerns have to be addressed in an explicit way. Secondly, the concerns are also handled independently wherever possible. Some of the related approaches are described in the following paragraphs.

A number of quality attributes like performance and modifiability are discussed in [4]. It illustrates the way in which architectural styles can be used to meet the requirements imposed by the quality attributes. In our approach, similar ideas have been shown to translate quality attributes into structures, design aspects and rules & guidelines. In [43] the concept of attribute-based architectural styles is described. The idea is that certain architectural styles are selected based on quality attributes. The work in our paper can be seen as a special kind of architectural style or pattern; various design aspects are defined to support quality attributes. In Figure 6-5 we already indicated how design aspects can support particular quality attributes. This could be worked out further into an overview of design aspects together with information on which quality attributes they support and how.

The first part of [10] deals with quality attributes and how they can be supported in the architecture of the system. He discusses an architecture transformation process to fulfill the quality attributes. Four transformations are described: imposing an architectural style, imposing an architectural pattern, imposing a design pattern, or conversion to functionality. The approach presented in our paper falls into the last category; based on quality attributes the design aspects are introduced which represent distributed functionality. But as stated earlier, applying distributed design aspects with supporting component frameworks can also be seen as an architectural pattern. In [67] design aspects are described in the context of the Building Block method. Our approach is related to ideas presented in that paper.

Kiczales and others describe the aspect-oriented programming approach in [42]. They identify the problem that some concerns, such as exception handling policies, are difficult to modularize. Such crosscutting concerns do not match

object-oriented decomposition, resulting in logically coherent functionality being spread over classes. The solution is to capture the crosscutting concerns in separate actions that can be woven into the rest of the program. The subject-oriented programming approach [33] is similar. The main difference with our approach is that these approaches focus on the classes at programming level, and deal with weaving aspects into programs via tool support. Our approach affects the various development phases, and various artifacts like units, components and classes; the design aspects are considered to be reasonably static.

6.5 Conclusions

In this paper, we have described our way of working with quality attributes and design aspects in the development of a medical imaging product family. Quality attributes are used in the problem space; design aspects are used in the solution space. They lead to important additional views, in addition to the decompositions of the specification and design into features and components. It is important to have multiple views in the development process to be able to manage the complexity. An example has been presented which illustrated the use of design aspects as a means to deal with quality attributes. This illustrated several benefits of using design aspects:

- Conceptual integrity and completeness is increased, since each artifact in the system must deal with each design aspect in a similar way, relevant design aspects are handled explicitly, and architectural rules for design aspects can be checked.
- The testability is increased, since standard tests can be used for certain design aspects and structuring is added.
- The traceability is increased, since the design aspects are handled explicitly and can be identified, both in the documentation (via templates) and in the code (via coding standards).
- The similarity of the realization of design aspects for the artifacts can lead to standardization of the usage of design aspects in artifacts. In addition, certain parts of a design aspect can be handled in a generic way, for example, the writing to log-files, making the specific part of the design aspect within each artifact smaller. The standardization leads to implementation support for design aspects.
- The development processes are supported by adding structuring to the architecture description, the design and its documentation, the code and testing.

- In the context of a product family, aspect-completeness of components supports the construction of family members from individual components.

So, the individual design aspects not only contribute to the various quality attributes; the fact that design aspects are used during development also contributes to various quality attributes, like traceability, testability and conceptual integrity. When applying such an approach, a number of issues must be taken into account. First of all, the use of quality attributes and design aspects requires awareness that several views of the system are needed. The most important concerns and threads of reasoning that apply to the system must also be known. These threads must be made explicit and contain the important quality attributes and the various solution mechanisms, including design aspects. For this, domain knowledge and a thorough analysis of how the design aspect can contribute to the realization of the quality attributes is needed. Once the design aspects have been identified, they must be further defined, by amongst other things:

- specifying each design aspect in an architecture document;
- specifying the dependencies between the design aspects and the rules that can be checked with architecture verification;
- specifying the level at which they are relevant, e.g. unit, component, or class;
- specifying the structure of the design documentation according to the design aspects, e.g. sections which must be provided, or the types of diagrams that can be provided (sequence diagrams, class diagrams, etc.);
- specifying coding conventions related to design aspects (e.g. naming conventions);
- specifying component frameworks, services, base classes, interface descriptions and other generic functionality that support the design aspects;
- specifying generic design aspect test cases.

The approach presented in this paper can be used as an addition to existing development methods. It is not a complete method as such. What it adds is its focus on additional views in different development phases, and its support in different areas like structuring the design and testing.