

Quality Attributes and Aspects of a Medical Product Family

Jan Gerben Wijnstra

Philips Research Laboratories

Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands

JanGerben.Wijnstra@philips.com

Abstract

In this paper we describe our experience with quality attributes and aspects in the development of a medical imaging product family. Quality attributes play a role in the problem space, whereas aspects play a role in the solution space. The usage of quality attributes and aspects introduces multiple views, dealing with separate concerns. For example, one can view a system from a safety or portability point of view, or one can focus on the initialisation or error handling aspects of components.

Quality attributes and aspects are used to add structure to the various phases of the development process. They form a supporting means for achieving completeness, i.e. have all relevant concerns been taken into account? In a product family context where the family members are constructed from a component-based platform, it is especially useful to achieve aspect-completeness of components, allowing system composition without worrying about individual aspects such as initialisation.

1. Introduction

Products in the various embedded system markets are becoming more complex and more diverse, and must be easily extendible with new features. Important influencing factors are a short time-to-market, low development costs and high demands on the quality of the software. In order to meet such requirements for a range of medical imaging products, a product family has been defined. The requirements specification for this family comprises a domain model expressed in UML and a structured collection of use cases (see [1]). The use cases describe the behaviour that is exposed to the user of the products within the family. The development of the individual products is supported by a shared family architecture and a component-based platform.

In order to master the growing complexity of systems, it is important to decompose product families, or systems in general, into smaller artefacts. Decompositions have to be defined in such a way that they support reuse and maintenance of artefacts, and a straightforward configuration of the product family members. Most development methods deal with a single decomposition of the system, and the tools support only the single

decomposition. For today's complex systems, however, a single decomposition is not enough to master the complexity. Systems exist in which the operational functionality, i.e. the functionality realising the purpose of the system, only accounts for twenty percent of the system realisation. In our case, the operational functionality comprises acquiring, processing, and viewing images. The remainder of the system deals with issues such as error handling or initialisation.

In the area of requirements specification, one can organise the specification in terms of use cases (functions, features), which usually focus on the operational functionality of the system, and therefore mostly deal with the functionality quality attribute (see [2]). The other quality attributes are then handled in different ways: one or two quality attributes drive the architectural decisions, while the other quality attributes are left to chance or are handled by over-dimensioning, e.g. additional memory. When considering the architecture and design of a system, the system is decomposed into smaller parts, primarily based on the operational functionality. But next to that, issues like initialisation, error handling, testing, and logging play a role. These issues, or aspects, are techniques in the solution space. They are just as important as the operational functionality of the system, since the system is only complete when all aspects are properly dealt with. Such additional views (or secondary decompositions) introduced by quality attributes and aspects, next to the primary decompositions into use cases and components, support the managing of the complexity by separating and localising concerns, as will be illustrated in this paper.

The remainder of this paper is structured as follows. In section 2, the need for quality attributes and aspects at multiple system decomposition levels is explained. Section 3 discusses the usage of quality attributes and aspects in the architectural phase; the subsequent phases are discussed in section 4. Related work is described in section 5, followed by concluding remarks in section 6.

2. Multiple views at multiple levels

This section describes the primary decomposition levels identified for the medical imaging family. After

that, quality attributes and aspects are introduced as additional views, providing a secondary structuring mechanism.

2.1. Multiple levels

The system decomposition levels for the medical imaging family are illustrated in Figure 1. The system is first decomposed into layers (separated by the dashed line in Figure 1), each with a number of units (the grey ovals). Next to the logical decomposition into units, a physical decomposition into one central controller (a PC with Windows NT[®]) and a number of peripheral devices is made. A unit may require different disciplines for its realisation. For example, the units shown in the lower part of Figure 1 require software in the central controller and peripheral devices containing dedicated hardware, embedded software on this hardware, mechanics, etc. The software of the units on the central controller (the white squares in Figure 1) is again decomposed into software components (the black squares). These software components can be deployed separately. The software components again consist of classes with methods and attributes. More on the system structure of the product family can be found in section 3.

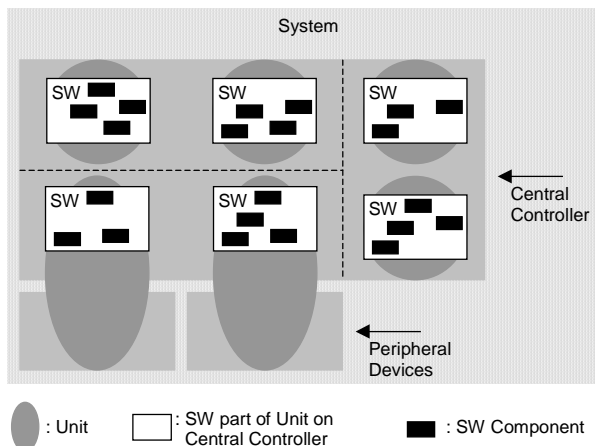


Figure 1 – Medical imaging system decomposition

In this paper, we will focus on the software part that is located on the central controller. So, the following levels of the primary system decomposition are relevant here: system, units (as parts of layers), software components, classes, and methods within the classes.

2.2. Multiple views

It is useful to have different views on an artefact, such as a system or a software component within a system. These views are based on the different concerns which are relevant for that artefact. We distinguish between quality attributes and aspects as views. A *quality attribute* of an artefact is an observable property of the artefact. 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, modifiability, and portability. An *aspect* is a coherent part of the functionality realised by the artefact that cross-cuts the primary decomposition. Relevant aspects for our software components include initialisation, error handling, and testing. The difference between quality attributes and aspects is that quality attributes deal with observable properties of a certain artefact (problem space), whereas aspects focus on the functionality that must be realised inside an artefact (solution space), as schematically shown in Figure 2. Aspects fully decompose the artefact, without overlaps, and can be used to realise quality attributes.

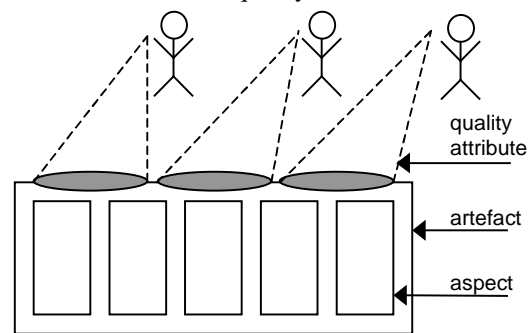


Figure 2 – Quality attributes and aspects

Figure 3 depicts the usage of quality attributes and aspects for the medical imaging family. The top of Figure 3 gives the quality attributes for the system as a whole. These quality attributes include configurability, portability and reliability. The following different means exist to realise these and other quality attributes:

- system structure
The modifiability quality attribute, for example, is handled by using component frameworks and plug-ins to separate generic functionality from specific functionality.
- rules & guidelines
Another way to realise certain quality attributes is to prescribe rules & guidelines for the development artefacts in the decomposition of the system. For example, to support the portability of the system, it is important to limit the dependency on external libraries. It must be prescribed which libraries may be used and which not.
- aspects
Finally, aspects can be used to support quality attributes. For example, to support the reliability of the system, a piece of functionality is included in each software component that handles the errors locally to limit the consequences of occurring errors.

Structures decompose the system into artefacts that fit the primary decomposition of the system at a certain level,

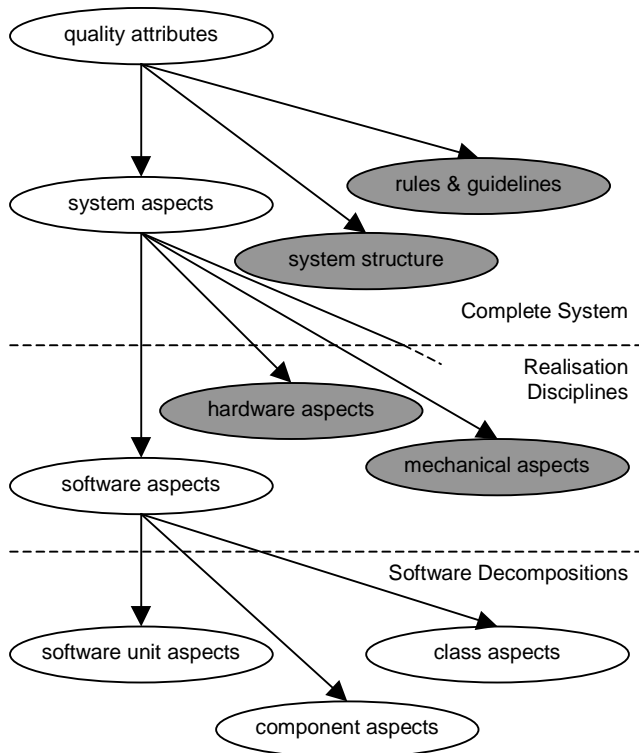


Figure 3 – From quality attributes to aspects

e.g. the units of the system are decomposed into software components. The application of rules & guidelines is interwoven in the system; they usually do not lead to clearly identifiable pieces of functionality. Aspects do lead to clearly identifiable pieces of functionality and are in some sense related to both structures and rules & guidelines: aspects define a microstructure for the individual artefacts in the system and rules & guidelines define how they should be applied within each artefact. This microstructure does not coincide with the primary decomposition, however, as will be discussed in section 2.3.

The aspects described at system level can cover various realisation disciplines. For example, the initialisation aspect of the system not only concerns the software on the central controller, but also the peripheral hardware. An integral approach must be applied for this, so that each unit will deal with this aspect in a uniform way. Of course, aspects can be specified in more detail for the individual realisation disciplines. When an aspect has been identified as being relevant for the software, one has to decide on which decomposition level it will be dealt with. For example, one can decide that:

- each *unit* takes care of the restart aspect, among other things by subscribing itself to a restart service which will restart the unit in the case of a fault,

- each *component* takes care of the field-service aspect, meaning that all field-service functions related to the component are provided by it, and
- each *class* takes care of the persistency aspect by providing methods so that the relevant attributes can be persistently stored and retrieved later on.

As shown in Figure 3, the quality attributes for the system as a whole form the basis for the various design decisions. From this figure, one might conclude that the quality attributes at the highest level can be translated into aspects, structures and rules & guidelines. However, a quality attribute (or a derivative of it) can also apply to an artefact at some lower level of the decomposition, and the decision of how to deal with it must be taken at that level. For example, the top-level architecture consists of units, and it shows no diversity. At a lower level, the designers of the units translate the modifiability into the application of component frameworks. Thus, the way in which a particular quality attribute is supported can be decided at various levels of decomposition. This means that additional structures, aspects and rules & guidelines are also introduced at lower levels.

The system and software architects make the most important decisions when defining the architecture for the system. They may decide, for example, that each unit must contain an error handling aspect. The architect does not necessarily have to restrict himself to the level of decomposition into units; important decisions for lower levels can also be made.

In [2], a distinction is made between quality attributes which are discernible at run-time or not discernible at run-time. One might expect that quality attributes that are not discernible at run-time are not handled with aspects, but with structure and rules & guidelines, since an aspect deals with a part of the functionality of an artefact. For example, the modifiability (not discernible at run-time) of a system can be supported by applying the structure of component frameworks with plug-ins in the architecture, and the reliability (discernible at run-time) is supported by the error handling aspect. However, some aspects can support a quality attribute that is not discernible at run-time. An example is the portability of software components, which are realised in our medical imaging family with the Component Object Model (COM) from Microsoft. Their portability is increased by adding a layer of wrapper classes (the wrapper aspect), shielding the specifics of COM. Conversely, a quality attribute that is discernible at run-time does not always lead to aspects. For example, as a consequence of reliability requirements, it has been decided to let each unit have its own process, thus limiting the consequences when one unit does not behave according to specification.

2.3. Applying aspects

Applying aspects within a system consists of two parts. The first part is a description of the aspect, containing information on the purpose of the aspect and which mechanisms must be used to realise the aspect in the system. This first part can be considered as a design specification. The second part is the actual implementation of the aspect. This implementation is spread across the various artefacts within the system. An aspect will have a specific implementation in each artefact. The internal structure of these artefacts can be considered as a *microstructure* (or texture), as depicted in Figure 4. The microstructure is defined as a recurring pattern for all artefacts in the decomposition, or for a selected subset of the artefacts. This pattern is recurring since the aspects are relevant for all (or most) artefacts.

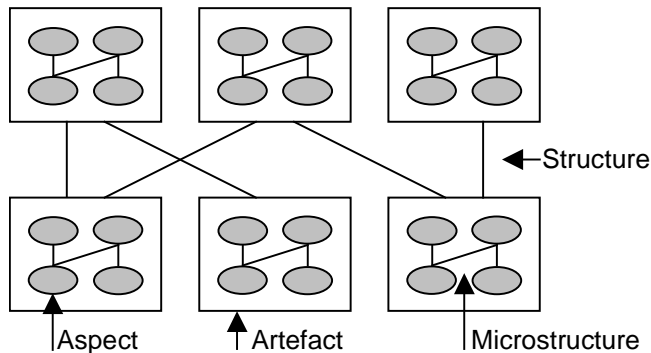


Figure 4 – Structure and microstructure

It should be noted that the decomposition of an artefact into aspects, as shown in Figure 4, does not have to coincide with the primary decomposition levels of the system. In our case, these primary decomposition levels do not match the aspects. The reason is that the domain objects, such as the image processor or patient administration, play a leading role for the primary decomposition, not the aspects such as error handling or initialisation. This difference is illustrated in section 3.4. As a consequence, the decomposition into aspects forms a secondary decomposition.

The microstructure introduced by the aspects is beneficial for mastering the complexity of the system. Aspects also have a positive effect on the conceptual integrity, since each artefact must deal with each aspect in a similar way. Furthermore, the testability is increased (see section 4.2). Also the traceability is increased, since the aspects are handled explicitly and can easily be identified.

The similarity of the realisation of aspects for the artefacts can lead to standardisation of the usage of aspects in artefacts. In addition, certain parts of an aspect can be handled in a generic way, e.g. the writing to log-files. In this way, the specific part of the aspect within

each artefact becomes smaller. The standardisation leads to implementation support for aspects. At the different levels of decomposition different support can be given, for example component frameworks can be defined (section 3.4), or base classes can be defined from which one can inherit (section 4.1).

It should be noted that some aspects are relevant for every unit, component or class in the system, while other aspects are only relevant for a subset of the artefacts. For example, the aspects of calibration and hardware testing are only relevant for the units that deal with peripheral devices. Initialisation, on the other hand, is relevant for every unit in the system. In [6], Van Ommering describes how certain architectural decisions only deal with a part of the architecture, forming a regional architecture.

3. Architecting

In [2], a software architecture is defined as “the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them”. The architecture must accommodate the quality attributes as imposed on the system, which can be handled via structures in the high-level architecture, aspects, or rules & guidelines.

3.1. Quality attributes

A short overview of the most characteristic and important quality attributes for the medical imaging product family is given below:

- Reliability

Reliability is the ability of the system to keep operating over time. It is important that the system keeps functioning when it is in use, because it is for example not allowed that the system fails during a clinical intervention.
- Safety

If a medical imaging system does not operate according to its specification, it is potentially dangerous to patients and personnel. More particularly, the FDA (Food and Drug Administration) must approve medical imaging products, especially in the area of safety, before they are allowed on the U.S. market, amongst others.
- Functionality

The domain and the requirements from the various stakeholders determine the operational functionality which the system must expose. Important functionality comprises acquiring, processing and viewing images.
- Portability

The delivered medical imaging systems must be supported for a long time (10 to 15 years). This means that one does not want strong dependencies on the

selected realisation technologies, as they may change during that period. Portability is thus an important quality attribute.

- Modifiability
In [2], modifiability is described as the ability to make changes quickly and cost effectively. In the context of the medical imaging product family, the following issues are relevant:
 - Configurability
At any given time, it must be possible to make several product configurations based on the product family architecture and shared assets, using minimal effort. For this reason component frameworks are introduced, for example.
 - Extensibility and Evolvability
Extensibility can be defined as the ability to quickly add new functionality with minimal effort. This is needed, as the family will continue to exist for a large number of years. The shared family architecture and component-based platform must also be evolvable in case changes are needed.
 - Testability
Testing is a very important activity, and is related to the safety and reliability of medical imaging systems. Since testing takes a considerable amount of effort, it requires special attention. Design concepts that support the testability are therefore applied.
 - Serviceability
The systems installed in the field have a long lifetime. During this lifetime, it must be easy for the field-service engineers to localise and repair defects and to upgrade these complex systems with new features.

3.2. High-level architecture

The primary decomposition of our system is based on the main concepts in the medical imaging system domain. The two main architectural styles guiding this decomposition are:

- layering
The system is decomposed into an application, technical, and infrastructure layer. The application layer contains the application knowledge, e.g. the procedures to acquire images and the functions to analyse images. The technical layer provides an abstraction of the peripheral devices, e.g. image processing functions and movements of the geometry, which controls the major moving parts in our medical imaging system. The infrastructure layer provides basic facilities to the other two layers, such as logging and field-service (calibration, configuration, etc.) facilities. These three layers are internally further decomposed into units.

- independence of units
The family architecture defines a decomposition into a number of units. A unit contains a coherent set of functionalities, and covers a sub-domain of the overall identified family domain, e.g. acquiring images or processing images. In order to avoid a monolithic design, units should be self-contained and de-coupled. This is supported by letting each unit deal with its own relevant aspects (aspect-completeness), using event notification and blackboard-like information exchange mechanisms, etc.

Based on these principles and the analysis of the domain, the system is decomposed into a number of units. The image acquisition chain plays an important role within the medical imaging domain. This chain contains the various peripheral devices, ranging from the generation of images to the processing and storage of images. Based on the elements in the chain, a number of units are identified in the technical layer, each providing an abstraction of the underlying peripheral device. The workflow using (and controlling) these devices comprises a number of phases, such as patient administration, image acquisition, image viewing, image handling (including printing, archiving, communication via a network); see [9]. Based on the phases in this workflow, units are identified. Since these units contain application knowledge, they are located in the application layer. The decomposition into units is schematically depicted in Figure 5.

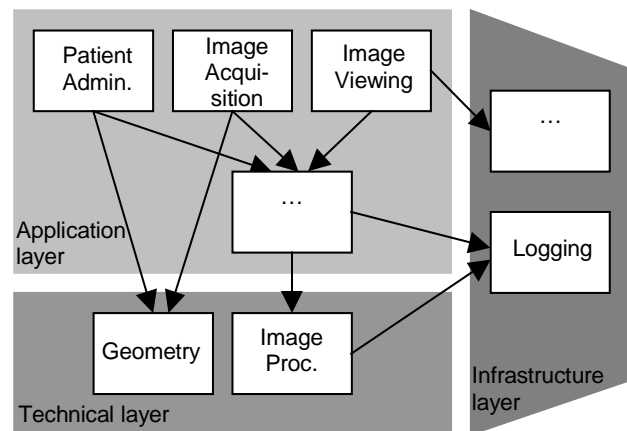


Figure 5 – Unit view of the product family architecture

The units shown in the technical layer in Figure 5 can be considered as complete units (i.e. including possible hardware), or as software units (i.e. the part of each unit that is inside the central controller). In the former case, a view on the system architecture is shown, in the latter case a view on the software architecture of the central controller is shown.

It should be noted that the primary decomposition of the system is based on the main functionality in the

domain, i.e. acquiring, processing and viewing images. The reason for this is that we expect the diversity in the product family, now and in the future, mainly to be in the operational functionality. We expect the other quality attributes and aspects to remain stable. They are identified up-front by the system architect, and go through the various decomposition levels of design in a top-down way. Changing quality attributes and aspects is then not a straightforward activity, since they do not match the primary decomposition of the system. Figure 5 does not show the required support for diversity in terms of configurability and extensibility. These qualities will be handled inside the individual units. Important architectural elements for this are component frameworks (see [10], [11]). Each unit may contain one or more component frameworks and results in one or more separately deployable components, which are realised as COM servers. For example, the different procedures for acquiring images are provided within plug-ins, supporting specific system configuration and extension of existing systems with new procedures.

The decomposition of the system into a structure with a number of units is only one part of the family architecture, of course. In addition to the decomposition into units and the specification of the relationships between units, other structures must also be defined based on quality attributes. For example, the process structure is influenced by the reliability of the system; each unit has a separate address space and process, so that an error does not immediately affect other units.

3.3. System aspects

Not all design decisions can be expressed in terms of structures and relationships. This is why the definition of architecture at the beginning of this section also mentions externally visible properties of components, which can be captured by aspects and rules & guidelines, and quality attributes that apply to the artefacts at lower levels.

Part of the document describing the system architecture deals with the various structures that are defined. Specific properties or responsibilities of the identified components are also described there. The aspects that cut across the various structures and that cannot be assigned to one or a few artefacts are discussed in separate chapters. For the medical imaging product family, these (software-related) aspects include amongst others:

- operational

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 images. This aspect is of course related to the functionality quality attribute.
- field-service

The field-service functionality deals with facilities like calibration of the hardware, performing tests, changing the configuration of the system, etc. These facilities are for use by field-service engineers only, and are particularly provided by units in the technical layer. This is one of the larger aspects that can be split into a number of sub-aspects. The field-service aspect is related to the serviceability, modifiability and the long lifetime of the systems.
- software keys and licenses

Almost every system is different due to high configurability. Licensing functionality supports this by enabling additional options. This aspect is also related to the modifiability/configurability.
- self-tests

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. This aspect is related to the safety and reliability quality attributes.
- graceful degradation

Graceful degradation is related to the safety and reliability quality attributes. It 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.
- start-up, shutdown, initialisation and reset

This aspect deals with starting-up and shutting down the system and the consequences this has for the components in the system. This aspect 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.
- error handling

This aspect deals with the strategy for handling errors that occur during system run-time. It is related to the safety and reliability quality attributes.
- logging

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.
- wrapping

COM has been selected as middleware for the realisation of the system. To limit the dependencies on this technology, each component must have wrappers, shielding specific COM issues from the internal implementation. This aspect is related to the portability of the system.

In addition to the aspects, rules & guidelines are formulated in order to meet the requirements imposed by the quality attributes. Two examples are given below to give an impression of these rules & guidelines:

- resource budgets
As in most systems, performance is an important issue. Timing constraints are given for a number of typical and critical use cases of the system. Budgets are also assigned to units for memory usage and disk-space, amongst other things. This is related to the performance quality attribute.
- software libraries
The software on the central controller executes on WindowsNT. A number of libraries exist that can be used during implementation. To limit the dependencies on these libraries, only a small selection may be used. This rule is related to the portability of the system.

The difference between the two lists, with aspects on the one hand and rules & guidelines on the other, is that the aspects result in identifiable pieces of functionality in the artefacts of the decomposition. For example, an artefact has an initialisation aspect, i.e. a piece of code dealing with its initialisation. Aspects define a microstructure for the artefacts which decomposes these artefacts. The rules & guidelines, on the other hand, are not easily identifiable. For example, it is hard to state which piece of the code is dealing with performance. This means that the rules & guidelines are interwoven with the functionality of the artefact.

The aspects listed here are handled by a large number of units in the system. For a particular system, these aspects are therefore realised by combining the aspect parts handled by each unit. An aspect for a particular system can be considered as an emergent property from the aspect implementations of the artefacts used in the system. It should be noted that some aspects are simply the sum of the aspect parts provided by each unit, for example each technical unit can provide one or more calibration functions. For other aspects, however, the quality of that aspect depends on the weakest aspect part provided by a unit, for example the quality of the graceful degradation of the system is determined by the weakest part in the chain. One could say that the first kind of aspects are individual and that the second kind of aspects are collective.

Architecting is all about taking early design decisions. To get these emergent properties right, it is important to take design decisions about aspects in advance. Sometimes, aspects are described briefly at a high level, but have a considerable impact on the implementation. Choices that have a considerable impact must be documented well to avoid situations where a certain aspect is solved in each artefact in a different way, thus

leading to a system which does not have conceptual integrity.

3.4. Software aspects and architectural styles

All system aspects mentioned in section 3.3 resulted in aspects on the software level. For example, the initialisation of the software is part of the initialisation of the complete system. The realisation of some software aspects can be supported by an architectural style as illustrated below.

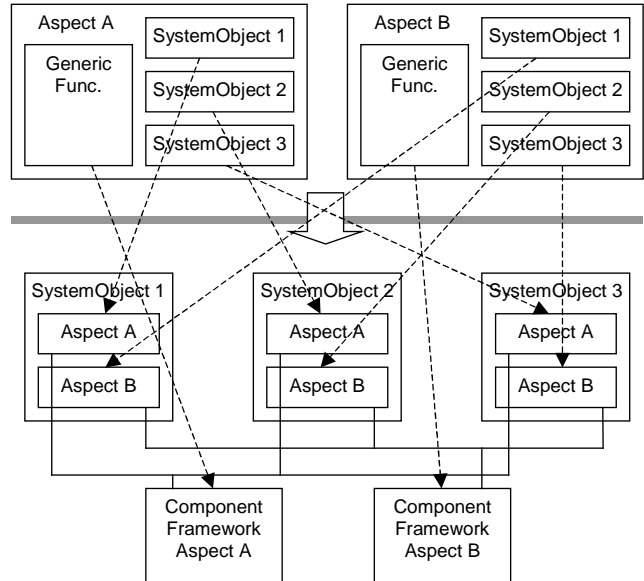


Figure 6 – Towards a hybrid approach

As stated earlier, current software development methods and tools usually allow a single decomposition. For some software systems, the primary decomposition is done according to the system aspects such as error handling and initialisation. The components in this decomposition each contain some generic functionality, plus aspect functionality that is related to the system objects, e.g. image processor or image acquisition. As a consequence, functionality that logically belongs to a system object is distributed over a number of components. This is the situation in Figure 6 above the thick grey line. For our type of systems, however, such a decomposition is not desirable. The diversity in our systems is in the direction of the system objects, not in the direction of aspects like error handling. This leads to a situation where the decomposition is based on the domain, as explained in section 3.2. Such a decomposition can handle changes and extensions in the main functionality of the system, e.g. by applying component frameworks. For some aspects it is very useful to have supporting functionality in separate components. This means that for our type of systems the situation below the thick grey line is desirable. It is then easier to add new system objects. The situation below the

line contains both system objects, and support via component frameworks for other functionalities (aspects), such as initialisation or error handling. The system objects are then plug-ins to these component frameworks in the infrastructure (see the infrastructure component frameworks in [11]). These component frameworks are located in the infrastructure layer as depicted in Figure 5. The combination of these two kinds of components in the primary decomposition of the architecture is identified in [7] as a hybrid approach.

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 provide these functions as services to the field-service component framework. These services are made available to the field-service engineer. Another example is the component framework that takes care of restarts of units. During initialisation, each unit must announce itself, so that the component framework can restart a unit when needed. We stated earlier that the selection of the aspects was considered to be fairly static; it has to be carefully considered by the system architect. But, since the generic part of the aspects mentioned in these two examples is handled at one location, changes to these aspects can be made relatively easily when needed, as long as only the generic part is affected; the specific part is distributed across the system.

4. Subsequent phases

In the previous section, the software system as a whole and its first level of decomposition into units was considered. In this section, we consider further levels of decomposition, the testing and verification of the system, and the construction of family members from a platform.

4.1. Design and implementation

A unit consists of a number of components. The software architecture has already defined aspects and rules & guidelines that apply to the components. The unit designer may add additional aspects and rules & guidelines, based on certain quality attributes. One example is that the initialisation aspect may be extended. At the highest level, initialisation for all software units in the system is defined. This includes, for example, that each unit in the system has to register itself in the registry and is started during power-up. During the initialisation of each unit, it must register itself with a component framework in the infrastructure layer that is responsible for monitoring the units and restarting them when faults have occurred. The designer who decomposes the unit may decide to introduce a component framework with

plug-ins. There will be additional initialisation requirements for this structure, for example the component framework must start the plug-ins, and the plug-ins must provide lists of services which they provide (see [10] and [11]). In this case, the initialisation aspect is extended for a specific unit.

Components are part of a unit and consist of classes themselves. Figure 7 shows how aspects can affect the internal structuring of a component. On the left-hand side we see the situation in which each class within the component deals with all relevant aspects. The selection of these classes is based on the domain and the operational functionality of the component, e.g. processing images. On the right-hand side, we see the situation in which the classes are assigned to a single aspect, e.g. error handling or initialisation. However, just as for the architectural style discussed in section 3.4, neither of these extremes is usually the best solution. Some hybrid approach is usually chosen instead. This approach consists of identifying a number of operational classes and supporting classes related to the other aspects. The operational classes then also deal with the other aspects. The aspects provide a second order decomposition (another view) of the components, in addition to the decomposition into classes.

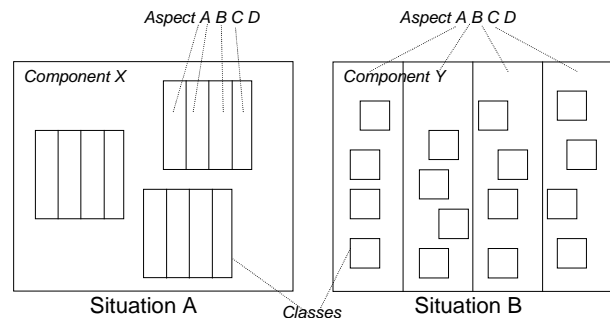


Figure 7 – Aspects and component structure

When considering the aspects inside a component, we can identify usage-relationships between them. For example, functionality related to the initialisation of a component may only be used during start-up, and it may not be called from normal operational functionality. Conversely, the initialisation aspect is allowed to make calls to normal operational functionality.

The interfaces of components are grouped according to aspects, i.e. each interface deals with one aspect. In this way, separate concerns are handled by separate interfaces. Just as there are allowed relationships between aspects within a component, the system architect defines the allowed relationships between aspects of different components. 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 any other component. Similarly, the

interfaces dealing with initialisation and reset of components may only be used by specific components. The correct application of these rules can be checked using architecture verification (section 4.2).

Section 3.4 discusses support for aspects in terms of component frameworks. Other support is also provided for realising certain aspects. For some aspects, support is given 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 as a basis for design and implementation. For example, for the operational aspect it has been decided that each unit will act as a server and provide its functionality via resources. Base classes are provided for these concepts of servers and resources. For aspects where such concrete support is not possible, one has to content oneself with more abstract rules & guidelines. Although not present in our development environment, one might also consider tool support for certain aspects.

Aspects provide another view (or secondary decomposition) of the system. However, the tooling only supports the primary decomposition. Additional measures must be taken to find out to which aspect an interface, a class or a method of a class belongs to. For example, for the interfaces, we have applied naming conventions in order to obtain views according to aspects.

Aspects do not only lead to a more structured view on the interfaces and the code, but they are also used for structuring the accompanying documentation. A template is used for component documentation. This template includes a number of sections that deal with the various aspects that are relevant for most, if not all, components.

4.2. Testing and verification

We have described how explicitly applying quality attributes and aspects in the various development phases adds structure to the development process. When considering the V-model, the use of quality attributes and aspects can aid the testing at the various levels by adding structure and achieving completeness. Two examples of how aspects help us are given below.

When considering testing, it is important to understand the architecture and design concepts behind a development method, as it usually leads to a more effective test process. The important testing concepts can be identified based on the design concepts in the development phases. This also holds for the concept of aspects, which helps when structuring tests to achieve a complete test of a component, without missing any aspect. Since aspects are treated more or less uniformly across the various components, it is also possible to define a number of basic tests for some aspects that must be performed for each component. For example, a number of tests can be

defined for component initialisation, or the handling of errors (e.g. when a component that is being used crashes).

For our medical imaging product family, automated architecture verification [8] is applied 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. It can also be used to check the applied aspects. Examples of aspect-related rules that are being 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 actually does is generating the relevant view and then checking the specific rule on that view.

4.3. Constructing family members

The product family development is based on a component-based platform. A family member is then constructed by selecting the right components from the platform and adding additional specific components to realise the system specific behaviour. In such a situation it is very important that each component is *aspect-complete*, i.e. each component deals with the functionality that is relevant for itself, e.g. its own initialisation, error handling, etc. When the components are aspect-complete, then the dependencies between the components are decreased. For example, no component is needed to take care of the specific initialisation of all other components, and that therefore depends on the specific system configuration. This localisation of concerns with the individual components is beneficial to the composition of individual family members out of separate components.

5. Related work

Various publications use the terms aspect, quality attributes, 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.

Kiczales and others describe the aspect-oriented programming approach in [4]. They identify the problem that some concerns are difficult to modularise, e.g. exception handling policies. 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 [3] is similar. The main difference when compared to aspect-oriented programming is that subject-oriented

programming deals with additional features that can be added onto objects, whereas aspect-oriented programming focuses on properties that affect the performance or semantics of objects. The main difference with our approach is that these approaches focus on the programming level and deal with adding aspects into programs via tool support. Our approach affects the various development phases, and the aspects are considered fairly static.

A number of quality attributes like performance and modifiability are discussed in [2]. These quality attributes affect the architecture to be defined. 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, aspects and rules & guidelines.

In [5], the method of applying aspects is described in the context of the Building Block method. Our approach is closely related to the approach described in that paper. They also describe how aspects can be used to meet certain requirements related to quality attributes.

6. Conclusions

In this paper, we have described our experience with quality attributes and aspects in the development of a medical imaging product family. Quality attributes are used in the problem space; aspects are used in the solution space. They provide useful additional views or secondary decompositions, 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. Using them explicitly adds additional structure to the process and they provide a means to achieve completeness.

The focus in this article is on the architecting phase. During this phase, it is important to explicitly list the quality attributes which affect the architecture, and it should be described how these will be met by applying aspects, defining structures, or prescribing rules & guidelines. Various examples have been given on the specific quality attributes, aspects, structures and rules & guidelines. In the architectural phase, a hybrid approach is applied in which the decomposition of the system is based on the domain of the system, and each unit in the system must deal with the various aspects such as initialisation and error handling. Infrastructure component frameworks have been introduced as a supporting means for some aspects. In contrast to some related methods, the quality attributes and aspects are considered to be fixed, i.e. the main diversity in the product family is expected in the operational functionality only.

In subsequent phases, the use of aspects is relevant for the units, components, and classes. For components,

which are used for configuring specific family members, we have identified that it is important that they are aspect-complete. Aspects provide a way of structuring the code and the accompanying documentation. In the test and verification phase, aspects again prove to be a useful means for structuring and achieving completeness.

Acknowledgements

I would like to thank the chief architect Ben Pronk and the people involved in the medical imaging product family development. I would also like to thank my colleagues Frank van der Linden, Gerrit Muller, André Postma, and Tobias Röttschke for comments on earlier versions of this paper. This research has been partially funded by ESAPS, project 99005 in ITEA, the Eureka Σ! 2023 Programme.

References

- [1] Pierre America, Jan van Wijgerden, *Requirements Modeling for Families of Complex Systems*, Proceedings of the IW-SAPF-3, March 2000.
- [2] Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.
- [3] William Harrison, Harold Ossher, *Subject-Oriented Programming (A Critique of Pure Objects)*, Proceedings of the OOPSLA '93, pages 411-428, September 1993.
- [4] Gregor Kiczales, John Lamping, Anurag Menhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, John Irwin, *Aspect-Oriented Programming*, Proceedings of the ECOOP '97, June 1997.
- [5] Jürgen K. Müller, *Aspect Design with the Building Block Method*, Proceedings of the First Working IFIP Conference on Software Architecture, February 1999.
- [6] Rob van Ommering, *Beyond Product Families: Building a Product Population?*, Proceedings of the IW-SAPF-3, March 2000.
- [7] Dewayne E. Perry, *A Product Line Architecture for a Network Product*, Proceedings of the IW-SAPF-3, March 2000.
- [8] André Postma, René Krikhaar, Marc Stroucken, *A Method for Module Architecture Verification and its Application on a Large Component-Based System*, submitted for publication to ICSE 2001.
- [9] Ben J. Pronk, *Medical Product Line Architectures – 12 years of experience*, Proceedings of the First Working IFIP Conference on Software Architecture, February 1999.
- [10] Jan Gerben Wijnstra, *Component Frameworks for a Medical Imaging Product Family*, Proceedings of the IW-SAPF-3, March 2000.
- [11] Jan Gerben Wijnstra, *Supporting Diversity with Component Frameworks as Architectural Elements*, Proceedings of the ICSE 2000, June 2000.