Variation Mechanisms and Multi-view Architecting in Platform-based Product Family Development

Wijnstra, Jan Gerben

Chapter 3
Supporting Diversity with Component Frameworks as Architectural Entities

Abstract

In this paper, we describe our experience with component frameworks within a family architecture for a medical imaging product family. The component frameworks are handled as an integral part of the architectural approach and are an important means to support diversity in the functionality provided by the individual family members.

This paper focuses on a particular kind of component framework that has been applied throughout the medical imaging product family. This kind of framework is useful when the various family members are based on the same concepts and the diversity is formed by the differences in the specific instances of these concepts that are present in the family members. These component frameworks have a number of similarities, allowing a standardised approach to their development. They support the division of the system into a generic architectural skeleton, which can be extended with plug-ins to realise specific family members, each with their own set of features.

3.1 Introduction

Products in various embedded system markets are becoming more complex and more diverse, and they must support easy extension with new features. Also important factors are a short time-to-market and low development costs. These

requirements should be met with a product family that covers a large part of the market. The development of a product family and its individual members (i.e. single products) can be supported by a shared family architecture. Like any other architecture, such an architecture must cover a number of quality attributes like performance, security, and testability (see chapter 4 of [4]). An important additional quality attribute of such an architecture is support of diversity (related to modifiability). This paper focuses on component frameworks as a means to deal with diversity.

The work presented here has been carried out in the context of the development of a medical imaging product family. The main characteristics of such a product family are:

- only a relatively small number of systems are delivered in the field, and almost every system is different due to high configurability and customizability;
- the delivered systems must be supported for a long time (10 to 15 years), and updates of mechanical, hardware, and software components in the field must be supported by field-service engineers;
- new features must have a short time-to-market, and the fact that the product family deals with a relatively mature market implies that customers will have high expectations and will request specific features;
- high demands are imposed on the systems' safety and reliability: if a system does not operate according to specification, it may be potentially dangerous to patients and personnel;
- there are different development groups at different sites, each of which is responsible for the development of a sub-range of the total product family range.

These characteristics must be tackled by applying the principle of reuse among the family members. This is achieved by developing a common component-based platform for all the family members, and using component frameworks as a means to handle diversity. The following topics are discussed in this paper:

- the high-level product family architecture in which the component frameworks are applied;
- how component frameworks help to handle the diversity;
- architectural issues of the component framework itself;
- embedding component frameworks in the overall family architecture.

These issues are described in sections 3.2 to 3.5. Experiences are described in section 3.6, followed by pointers to some related work in section 3.7. Conclusions can be found in section 3.8. More information on our work can be found in [112]. There, we focus on the relationship between the domain model and the component frameworks, and on the process for framework development. In this paper, we focus on the architectural issues.

3.2 Product Family Architecture

A product family architecture has been set up based on the identified domain concepts and the scoping of the domain (more on domain modelling can be found in [1]). The main architectural concepts used within this product family architecture are:

- **Layering.** The system is decomposed into a number layers. The main layers are the application, technical and infrastructure layers. The application layer contains the application knowledge, such as the procedures to acquire images and the functions to analyse images. The technical layer provides an abstraction of the underlying hardware to the application layer, such as image processing functions and movements of the geometry[7]. The infrastructure layer provides basic facilities to the other two layers, such as logging and field-service facilities (calibration, configuration, etc.). These three layers are further decomposed internally.
- **Self-contained units.** The system is decomposed into a number of units. A unit contains a coherent set of functionality and deals with a sub-domain of the complete identified family domain, such as acquiring images or processing images. Ideally, when such a unit is added to the system, no adaptations of other parts of the system are required. This means, among other things, that each unit must carry out its own error handling and logging and must provide its own field-service functionality. These kinds of functionalities are called *aspects* [67]. Architectural rules concerning these aspects are formulated, which apply to the units. The infrastructure layer provides support to realise aspects.
- **Independent units.** The independence of units is related to the previous point. To avoid a monolithic structure, the units should have no direct

[7] The geometry controls the major moving parts in the medical imaging family and determines the part of the patient to be examined and its projection on the image.

knowledge of each other unless this is required. A number of concepts are applied to achieve this, such as event notification and facilities based on the blackboard pattern [13]. Various units are connected to such a blackboard facility, which enables these units to achieve combined behaviour without direct interaction.

Applying these concepts results in a system decomposition into units, divided over three layers, as is schematically shown in Figure 3-1 (the arrows indicate the usage relationship). The decomposition of the technical layer is based on the various hardware devices, such as image processing hardware and geometry hardware. In the application layer, the decomposition is based on the functional areas in the application workflow, such as acquiring images and analysing images. The decomposition of the infrastructure layer is based on the infrastructure facilities, such as logging and field-service. The actual product family architecture contains about thirty units. The architectural concepts are further elaborated in [86].

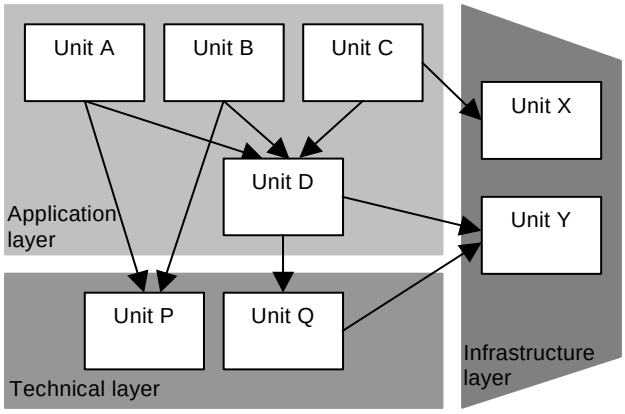

Figure 3-1 – Unit View of the Product Family Architecture

According to [81], in which different styles for generic architectures are discussed, the architecture schematically shown in Figure 3-1 can be classified as a *generic variance free architecture*. This means that the variance is not made explicit yet. The units are identified and responsibilities are assigned to them. Each unit is relevant to each family member (although some units are optional). When designing a unit, it must be decided how the diversity will be handled internally. In that sense, the architecture presented in this section serves as a starting point or context for applying component frameworks. Each unit may contain one or more component frameworks and results in one or more separately deployable components [102], which are realised as COM servers.

3.3 Component Frameworks as a Means to Handle Diversity

In this section, we describe which kinds of diversity exist in the medical imaging product family and how component frameworks help to support this diversity.

3.3.1 Diversity in the Medical Imaging Product Family

One consequence of the diversity amongst the various family members is that the family architecture must support a variety of functions at the same time, which requires configurability. Furthermore, the family must be extensible over time. Diversity within a product family can stem from several sources. The main sources that can be identified are changes in the set of features that must be supported by the product family (relating to user requirements; see [66], which deals with a feature-oriented approach for software structuring), and changes in the realisation technologies, such as new hardware. When changes in the realisation technologies are noticeable to the end-user, they also affect the features. In our product family, examples of diversity in features are different procedures for acquiring images and different ways of analysing images. Examples of technology-related diversity are new implementations of image processing hardware, larger and faster disks for image storage, and new geometries for moving patients.

3.3.2 Realising Diversity

The product family architect has defined two principles for the family architecture which are relevant when selecting techniques for realising diversity, see [86]:

- binary reuse of components;
- division of the system into a generic part and member-specific parts.

Component frameworks support these two principles, as will be explained below. Generally, a framework forms a skeleton of an application that can be customised by an application developer. A framework therefore supports the division of functionality into a generic part and a specific customisation part, covering the second principle.

A range from white-box frameworks to black-box frameworks is described in [26]. The white-box frameworks rely heavily on object-oriented language features such as inheritance and dynamic binding. Black-box frameworks

support extendibility by defining interfaces for components that can be plugged into the framework via object composition. These black-box frameworks agree with Szyperski's view [102] on component frameworks as being sets of interfaces and rules of interaction that govern how components 'plugged into' the framework may interact. We will refer to the white-box frameworks as *class frameworks*. From an abstract viewpoint, the different framework types are based on some common properties. They all capture some reusable design, improve on quality, reduce development costs, etc. Identification of variation points [36], extension points [48], hot spots [94], or plug points [22] is also important. If you look at them more closely, however, they are different. In contrast to the class frameworks, the component frameworks are easier to use and to understand, since no knowledge of the internal structure of the framework is needed, and the simpler mechanism to extend the framework permits binary reuse. This ease of use is especially of importance in a situation where component frameworks are applied at several distributed development sites.

In addition to the component frameworks, which realise diversity on an architectural level, another applied mechanism for realising diversity is the configuration of a component via data. Component frameworks are applied when there is significant different functionality needed in different family members, for example, a different geometry offering different movements. The configuration data is used when the differences are smaller, such as for country specific settings.

3.4 Service Component Frameworks

This section introduces a particular kind of component framework, the *service component framework*, and discusses issues concerning the development of such a framework.

3.4.1 Introduction to Service Component Frameworks

Two kinds of component frameworks used in the medical imaging product family have been identified in [112]: *structure component frameworks* and *service component frameworks*. The structure component frameworks define a co-operation structure for a fixed number of components, of which some have fixed implementations and others have selectable implementations. In contrast, the service component frameworks support a variable number of plug-ins for various family member configurations, and they are components themselves. The rest of this paper focuses on service component frameworks.

Service component frameworks are especially useful when the various family members are based on the same concepts (from the domain model) and the diversity is formed by the different instances (with different behaviour) of these concepts that are present in the family members. These concepts can range from a software representation of a scarce hardware resource, such as a movement of the geometry, to some piece of logical functionality, such as a function to analyse images. The concept instances are called *services*. There may be more than one service present for one concept in a particular family member.

An example of where such a component framework has been applied is the geometry unit. Amongst other things, this unit as a whole provides a number of movements for positioning a patient, such as TableHeight and TiltTable. Various geometry hardware modules exist, each providing a subset of all the possible movements. To deal with this diversity, the hardware model is mirrored in software by one component framework, which provides the generic functionality, and a number of plug-ins, each providing a number of movements to the component framework. In this example, the movements are modelled as the services that are provided by the plug-ins. The various clients of this component framework can use these movements without worrying about the internal component framework structure. In addition to the movements, other kinds of concepts have also been identified in the geometry unit for which services can also be provided.

Figure 3-2 shows a schematic overview of a service component framework and its participating plug-ins. A plug-in is simply a container of one or more services, where the services represent the actual functionality. The services related to one particular concept all support the same interface as prescribed by the component framework, but may manifest different behaviour. Generally speaking, a component framework defines one or more roles for participating plug-ins. In the case of the service component framework, one can define one role for each concept (and its related services) that prescribes the interface for the services of that concept. Service component frameworks allow multiple plug-ins to be connected to the same role, for example, multiple plug-ins can contain movements of the geometry. This kind of framework has also been applied to various other units of the product family, for example, in acquisition for adding acquisition procedures, in reviewing for adding analytical functions, and in the field-service domain for field-service functions (calibration, configuration, etc.).

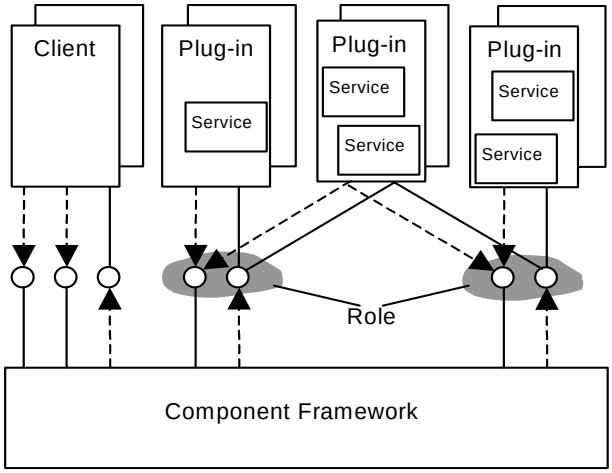

Figure 3-2 – Dynamic Component Framework

Clients use the functionality provided by the component framework and plug-ins as a whole. The component framework provides the services from the various plug-ins in a generic way. A service component framework is a component itself, and is active in connecting interfaces and providing the functionality of the services to clients. A single unit in the product family architecture can contain one or more of these component frameworks.

3.4.2 Service Component Framework Development Issues

The main responsibility of a service component framework is to create an environment in which the services provided by the plug-ins are optimally used. More on these responsibilities can be found in [112]. When developing such a framework, a number of architectural issues must be taken into account, including the ones listed below.

- **Aspects.** The functionality of a component (or the system as a whole) can be divided into the main functionality of the component, for example, moving the patient to acquire the required images, and additional supporting functionality, for example, initialising the component, handling errors, etc. These various types of functionality are called *aspects* [67]. The component framework together with its plug-ins must cover, next to the main functionality aspect, all relevant supporting aspects. For the initialisation aspect, the component framework prescribes how the initialisation is carried out, and what the plug-ins should do for their correct initialisation (see next bullet). Similarly, in the case of an error in a plug-in, the component framework

decides what to do, for example, to continue without the plug-in or to stop completely. Similar decisions must be made for the other aspects. Design patterns can be identified for handling certain aspects, but that is not elaborated on here.

- **Connecting the plug-ins to the component framework.** During initialisation, the plug-ins must connect to the component framework to make their services available. At development time, the component framework does not yet know which plug-ins will be present at run-time. The plug-ins have to register themselves in some way. For our system on a Microsoft Windows NT®/COM computing platform, we use COM's category mechanism. One category is defined for each kind of plug-in, prescribing a particular set of interfaces. During the system installation, the plug-ins register themselves at the relevant category. During initialisation, the component framework then starts the registered plug-ins and obtains their services.
- **Prescribed interfaces and roles.** A generic architectural skeleton is developed as the basis for the development of the individual family members (see section 3.5). This architectural skeleton contains the component frameworks. Plug-ins are developed by application groups to realise particular features. This approach needs clearly defined interfaces between the component frameworks and the plug-ins. These interfaces are prescribed by the component frameworks and are provided to the plug-in developers. The roles related to these interfaces can be classified in two groups: the *service related roles* (as introduced in section 3.3) and the *infrastructure related roles*. The service related roles deal with the main functionality aspect of the component framework, and the infrastructure related roles deal with supporting infrastructure aspects, such as initialisation and error handling. A plug-in must handle both the service related roles and the infrastructure related roles.
- **Separate entry points.** A component framework interfaces with several plug-ins with different roles and it also interfaces with the clients of the component framework. Each of the plug-ins and clients interacts with the component framework for a specific purpose; the complete functionality of the component framework does not have to be exposed to each of the interacting components. Instead, façades should be defined that offer the right amount of functionality to the plug-ins and the clients, see Figure 3-3.

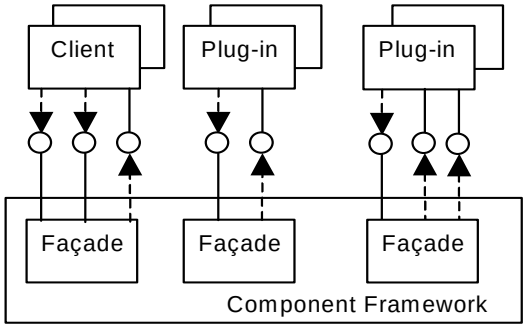

Figure 3-3 – Façades in a Component Framework

- **Distributed set.** Various plug-ins can provide various services for a specific role, for example different movements of the geometry. Although these services have the same interface, they manifest different behaviour as identified by some identifier, such as TableHeight or TiltTable. When a client of the component framework wants to use a movement, it can select one out of the set of movements that is distributed over the various plug-ins. We can establish the complete distributed set with all defined services based on the complete set of plug-ins, see Figure 3-4. The developer responsible for the component framework has to document the complete distributed set at a central point. The idea is that when two plug-ins each provide a service with the same behaviour, then the same identifier should be used. Conversely, two services with different behaviour should not have the same identifier. In this way, a client can work with a service such as TableHeight, without knowing whether it is provided by plug-in A or B.

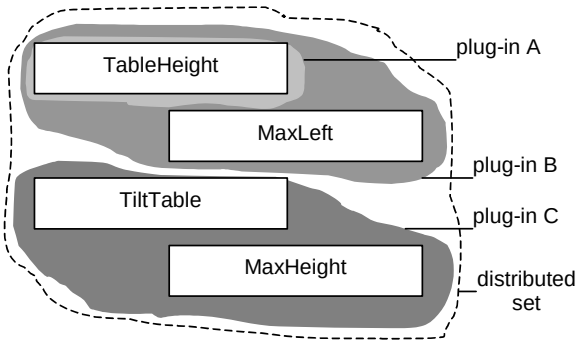

Figure 3-4 – Distributed Set for Movements of the Geometry

In addition to the complete distributed set, each family member has its actual distributed set which is established at initialisation time. In order

to implement the actual distributed set, the component framework contains some class that keeps track of the services that are provided by the installed plug-ins. This class is comparable to the Context class in the Strategy pattern [29], however, where the Context class deals only with a single strategy, the framework class must deal with one or more services in the distributed set.

- **Availability interface.** The component framework supports different configurations, resulting in different functionality being present in the various family members. It must be possible for the clients of the component framework to find out which functionality is actually present. In order to do this, a component framework has an availability interface for each service concept, via which the actual supported distributed set can be obtained. Using this information, the client can determine which functionality it can realise itself. For example, it is possible to query which movement instances are available in a specific configuration, after which an operation like `GetMovement(…)` can be invoked to get a particular movement.
- **Adding functionality to the services.** One of the tasks of the component framework is to establish connections so that the clients can use the functionality of the provided services. Depending on the situation, the framework may add additional functionality to the services. For example, when scarce services such as the geometry movements are concerned, a resource managing mechanism with priorities can be introduced. This requires functionality to be added to each movement so that it can be locked and unlocked, for instance. Since this information is added inside the component framework, proxies [29] must be made for each movement provided by the plug-ins. In this way, a resource manager inside the geometry component framework can use the proxies with the additional functionality.
- **Dynamic model.** In addition to the structural and interface issues, the component framework developer must also deal with the dynamic model of the component framework and its plug-ins. Problems like inversion of control [26] must be taken into account. For this reason, the framework developer must clearly describe which kinds of threads are present, what they do, and how they interact. For example, when a client invokes an operation on a service provided by a plug-in, this operation can be performed on the client thread, or a new thread can be used. When a service can generate events, a decision must also be taken which thread is used to inform the client.

3.4.3 Roles and Services

A component framework defines one or more roles that must be filled by services from the plug-ins. In some systems, a particular role may not require a service to be provided during run-time, that is, the role is optional. For other roles, multiple services may be provided, that is, the role is cumulative. This leads to the following table:

	not cumulative	cumulative
not optional	1	1..*
optional	0..1	0..*

- **Optional or not.** If a role is optional, the framework itself contains a complete set of functionality that operates correctly without additions. However, additions are possible when required. The component framework implements the default services that are relevant to all members of the product family, and special services must be added by plug-ins. Optional roles may be used to override default behaviour that is provided by the component framework. If a role is not optional, the framework provides some basic functionality, but it cannot operate without one or more services from plug-ins. This applies to the geometry example, where the plug-ins have to supply movements.
- **Cumulative or not.** When only one service is allowed per role, this means that no diversity exists in that functionality within one specific family member. For example, a word processor (normally) uses only one spelling checker at the same time. In the geometry example, however, multiple services, such as TableHeight and TiltTable, are provided by multiple plug-ins.

The discussion above concerns the number of services that must be provided during run-time for a specific role. Another classification deals with the number of services *with the same identifier* that may be present either during development-time or during run-time.

- **Development-time.** Depending on the kind of service, only one service or multiple services with the same identifier may be defined at design-time. In the geometry case, for example, multiple plug-ins define the same movement, such as TableHeight, since different hardware modules support this movement. This allows the clients of the framework to use the TableHeight movement, without worrying about the underlying hardware. In some cases, however, a service with a specific identifier can only be defined once. This is the case when

services represent some specific hardware calibration functionality, for example, and each of these services has a one-to-one relationship with a specific hardware module.

- **Run-time.** Even if at development-time multiple services are present with the same identifier, this may not be allowed at run-time. For example, the geometry framework does not allow multiple TableHeight movements to be present, since this movement is realised only once in a particular hardware configuration. In other cases, it may be useful to have multiple services with the same identifier, such as when a service represents a printer and multiple printers are connected to the system; the component framework can select one that is free.

The nature of the applied service concept thus determines the optional and cumulative properties of a role, and whether multiple services may exist with the same identifier, both at development-time and at run-time. The discussion above is about roles and services. Since plug-ins contain one or more services, this discussion is also related to the number of plug-ins that are allowed per component framework.

3.5 Service Component Frameworks in a Family Architecture

In this section, we focus on applying service component frameworks in a larger architectural context. One can distinguish two main views on the architecture, as listed below:

- **Top-down (decomposition) view.** In this view, the focus lies on the system as a whole and partitioning the system into smaller parts (units), in order to master the complexity and to understand the system.
- **Bottom-up (composition) view.** In this view, the focus lies on the components which can be used to construct a family member. The configurability plays an important role here.

The top-down view is initially applied to define the product family architecture. The bottom-up view is also applied; it shows how the family members can be composed from individual components, and how diversity can be achieved.

3.5.1 Configurable Product Family Platform

The product family architecture defines a structure of units and applies it to all members of the family (although some units are optional). The two main architectural diversity mechanisms for the medical imaging family are:

- different implementations for (parts of) a unit are available as separate components, of which one must be selected (i.e. structural component frameworks);
- a unit is separated into one or more service component frameworks with plug-ins, and the plug-ins with the right services must be selected.

Not all units exhibit diversity; they have a fixed implementation that can be reused in every family member. These units and the units containing fixed parts (component frameworks) form the generic architectural skeleton [50] which enables component reuse. Such a skeleton supports architectural configurability in two ways:

- selecting an implementation component for each (part of a) unit that has no fixed implementation;
- selecting the right plug-ins for the service component frameworks.

Well-designed generic functionality, based on the domain model, is stable for a whole system family. This means that a product family based on such a generic skeleton will ideally be stable for a long time. The generic skeleton defines the interfaces to which the extensions must adhere. This means that you can clearly see which extensions are anticipated, and also which extensions are not supported. Figure 3-5, shows an example skeleton consisting of units with complete or partly fixed implementations depicted in dark grey, and the diversity arising from freely selectable parts in light grey.

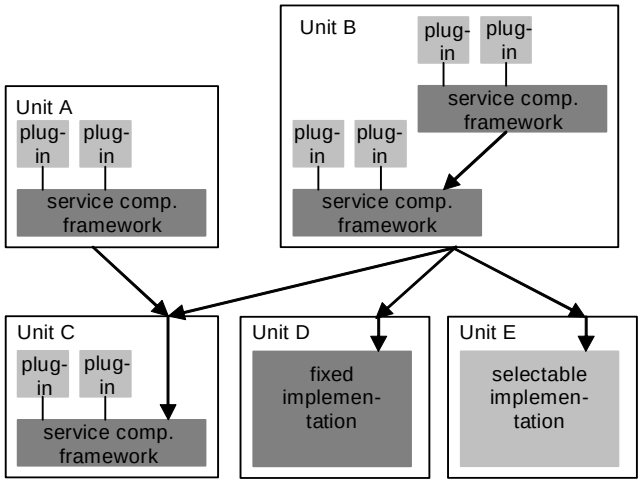

Figure 3-5 – Generic Architectural Skeleton

3.5.2 Types of Frameworks

Service component frameworks are applied in three areas in the medical imaging family architecture, coinciding with the three main layers of the architecture:

- **Application frameworks.** These frameworks deal with diversity in the application domain. For example, several procedures exist for acquiring images. Each family member supports a subset of all of these procedures. The various procedures are provided as services to a component framework.
- **Technical frameworks.** The technical layer provides functionality on which the application layer is built. The component frameworks in this layer are closely related to the underlying hardware. For example, the geometry functionality is realised by combining a number of hardware modules. This structure is mirrored in software using a component framework with various plug-ins.
- **Infrastructure frameworks.** In addition to the domain specific frameworks, the infrastructure contains some computing platform related frameworks. For example, there is a component framework present 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. In the same way

that a component framework provides a local infrastructure for its plug-ins, covering all relevant aspects, the infrastructure layer provides infrastructure frameworks to support aspects for all components in the system.

From the discussion above, it is clear that a plug-in in may be related to multiple component frameworks, for example, a geometry plug-in provides its movements to the geometry framework and its calibration functions to the field-service framework. It has to play multiple roles for multiple component frameworks.

3.5.3 Related Component Frameworks

The product family must support the diversity of features that are offered to the end-user and the changes in the realisation technology. As mentioned earlier, component frameworks are an important means to achieve this. The realisation of a particular feature is not always confined to a single component framework, however. In fact, plug-ins from several component frameworks may be needed to realise a single feature.

As a consequence, relationships can exist between component frameworks, and services for these frameworks can be grouped together. These groupings can range from loose to strict groupings. An example is the framework responsible for the procedures to acquire images in the application layer and the geometry framework in the technical layer. Some acquisition procedures can be performed on several geometry configurations and, conversely, a specific geometry configuration can support several acquisition procedures via its movements. This means that there is an m-to-n relationship between these geometry movement services and acquisition procedure services. There are some very specific acquisition procedures, however, which can only be realised by one specific movement, which in turn is realised by one specific geometry hardware module. In this case, there is a one-to-one relationship between the services of the two frameworks.

It is important to distribute the services correctly over the plug-ins. Two frameworks are depicted in Figure 3-6 with three and two plug-ins respectively. The grey areas indicate that plug-in X provides functionality that can be combined with plug-ins A and B. Similarly, plug-in Y provides functionality that can be combined with plug-ins B and C. For example, if plug-in B contains one service that is only related to plug-in X and one service that is only related to plug-in Y, it would be worthwhile distributing the services of plug-in B over A and C. The grouping of services in plug-ins is influenced by the required configurability of the features and the realisation techniques. For example, a

plug-in of the geometry framework contains the services that are supported by one geometry hardware module. In this way, adding a geometry hardware module requires adding one software plug-in.

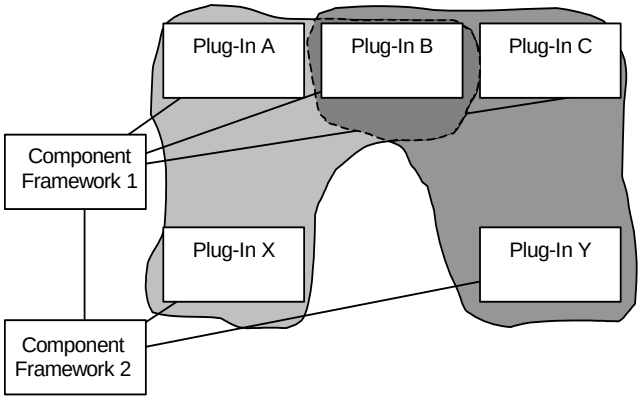

Figure 3-6 – Relations between Plug-ins

Such layered component frameworks are a result of the layered architecture, where different levels of abstraction are identified, such as technical and application. The functionality related to a feature can run across these layers. Various kinds of services can be constructed in a higher layer based on the services provided in a lower layer. Of course, no plug-ins are needed when a higher layer can handle all services in a generic way.

3.5.4 Some Special Situations

Some simplifications have been made in the above description. For example, we assumed that no relationships exist between plug-ins for the same framework, and that only generic framework interfaces are used between plug-ins for different frameworks. This is not always the case, however, as will be explained in the following paragraphs.

Until now, we assumed that multiple features could be realised by simply combining a number of orthogonal software components. In some cases, however, the combination of two features not only introduces the sum of the behaviour of the individual features, but also some combined behaviour. For example, a geometry configuration is composed of a number of hardware modules. These modules cannot simply be combined, since they also have to exhibit combined behaviour: some combined movements are supported, and collisions of the mechanical parts in the new combined configuration must be avoided. This requires additional plug-ins in software to model the combined

behaviour. It may be necessary for such an additional plug-in to override functionality of the existing plug-ins, however, this is not elaborated on here.

In principle, functionality realised by plug-ins is provided to the clients via a generic interface defined by the component framework. In this way, the clients can remain generic, since they know the meaning of the various interfaces. In some cases, it may be necessary to introduce services that each have a specific interface, in order to provide required functionality to the clients. The clients must know the specific interfaces precisely in order to use them. We encountered such a situation with the geometry. There, a service concept representing very specific properties of geometry hardware modules could not be captured in a generic interface. The geometry component framework provides functionality to obtain the services belonging to this concept. A component framework is introduced in a higher layer, that allows services to deal with the specific geometry services. These services in the higher layer have a one-to-one correspondence with the geometry services with specific interfaces.

3.5.5 Using the Family Platform

Component frameworks are applied as a means to support diversity. They support the definition of a family platform consisting of those software components[8] that are relevant to several (but not necessarily all) family members. In addition, a generic architectural skeleton has been defined that is formed by those software components that are relevant to all family members. It is possible to construct such a family skeleton since the various units are relevant to all family members and the (expected) variation in these units is limited. Each unit may contain one or more component frameworks.

Application groups use the family platform to build their specific family members. The generic architectural skeleton is used as a basis. Diversity is achieved by selecting the right specific extensions to this skeleton. When building a specific family member, the features that have to be supported must be identified. If these features are supported by plug-ins of component frameworks, the related plug-ins must be selected, if they already exist, or new plug-ins must be developed. As described earlier, a feature may be related to

---

[8] In addition to the software, the platform also comprises requirements and design documentation, hardware, interface specifications, architectural rules and guidelines, tools, test environments, etc.

several component frameworks. It is therefore important to keep track of the relationship between plug-ins and the features they support.

3.6 Experiences

The approach described in this paper is currently being applied at Philips Medical Systems, and involves a large development crew. A major part of the development and testing can be shared for the various family members. The first members of the medical imaging product family are being developed at different development sites; one site for the platform development, and several sites for family member development. This approach is very promising as far as the first release of the family members is concerned. Further releases must validate this approach, of course. Some experiences with this approach are described below.

- As stated earlier, the product family architecture contains a little over thirty units. About twenty of these units have fixed implementations and are present in each family member, a few units have a selectable implementation. About ten units have one or more service component frameworks, for example geometry, acquisition of images, viewing of images, field-service, etc. The size and complexity of the individual frameworks differ. The development effort can range from one to several man-years.
- The service component frameworks proved to be very suitable for the diversity that needs to be modelled in this specific family architecture. Since the various service component frameworks have a lot in common, as identified in section 3.4, various parts of their design, documentation and testing can be done in a standardised way. They are also suitable for the organisational context of multi-site development for two reasons: ease of use and independence. Plug-ins for service component frameworks only have to interact with the framework via a predefined interface; the internal structure of the framework does not have to be known. This increases the ease of use and requires less support when compared to class frameworks. The interfaces also separate the component framework from the plug-ins; when the component framework is modified, this has no effect on the plug-ins (except when the interfaces are involved).
- It is very important that the domain, as far as relevant for setting up the product family, is modelled, and not only a subset of the products to be supported. As a means to realise diversity, the architecture as a whole, and the frameworks in particular, are based on the domain model,

including the expected variation. The right concepts must be chosen to deal with variation. If these concepts are not based on the (expected) domain, the wrong concepts might have been chosen, leading to rework in subsequent releases of the frameworks and their plug-ins.

- The separation of the functionality into a generic and a specific part introduces a new way of working. In our case, the platform activities deal with the generic part, and the application activities with the specific parts. This means that the main architectural choices are already made within the platform activities, and that the application activities use the provided components and interfaces, and extend the functionality with additional components to realise a specific family member. It is important to involve the application groups in defining the common platform, especially the interfaces to the specific parts.

3.7 Related Work

This paper describes a specific kind of component framework and how it can be applied to support diversity in a family architecture. The development of the service component frameworks is based on a domain model and a product family architecture, see [112]. In the paper, we focus on the relationship between the domain model and the component framework design model and the process involved.

There are several publications which concern experiences with frameworks in different domains. In [61], for example, Meekel and others describe their experiences with frameworks in the portable wireless communication devices domain. A process is described in which, on the basis of domain analysis and an architecture defining a high-level decomposition into components, frameworks can be identified within these components. They use class frameworks for the product family. No further details are given on these class frameworks.

In [28], Fregonese and others describe experiences with frameworks in a telecommunication domain. The relationship between domain analysis and framework development is also stressed here, and class frameworks are used. These class frameworks are developed using design patterns, and object-orientation is used to provide hooks for the features that are likely to change. Since each class framework can use different design patterns, no general structure and way of using them is given.

The component frameworks described in this paper are related to the generic building blocks used in the Building Block Method [50]. This method also identifies the differences between class and component frameworks. It is argued

that the self-contained nature of components results in better system structures. A number of generic building block types are described that were encountered in a telecommunication infrastructure product family.

There are various other publications which describe frameworks in general. For example, [26] and [48] consider a range of white-box to black-box frameworks. In [48], a number of tasks are described to design and implement a framework. Three ways are discussed to extend a framework, that is, via interfaces, via inheritance, and via aggregation. A black-box framework (component framework) can only be extended by implementing interfaces specified by the component framework. In [26], it is argued that many framework experts favour black-box frameworks over white-box frameworks, since they are easier to extend and reconfigure.

3.8 Conclusions

In this paper, we have described an approach to handle diversity in a medical imaging product family, using component frameworks. Two important starting points for developing these component frameworks are the domain model, which defines the relevant concepts, and the product family architecture in which the component frameworks are integrated.

This approach deals with the requirement that the medical imaging product family must deal with diversity in different areas. Various features and realisation technologies are supported, and the family must be extendible in the future. A generic architectural skeleton has been defined which is reused for all family members. As part of the architectural skeleton, service component frameworks are applied in areas where the various family members are based on the same concepts and the diversity is formed by the different instances of these concepts that are present in the specific family members. These component frameworks are applied in several units of the family architecture. They form a relatively simple, but important means to deal with this diversity. They are simple in the sense that they only use interfaces as a means of extension (the component framework is a black-box), and similarities between the service frameworks allow a standardised approach for developing such a framework. The services provided by plug-ins are related to the features that need to be supported in the product family.