

# Implementing a Reuse Strategy: Architecture, Process and Organization Aspects of a Medical Imaging Product Family

Eelco Rommes  
Philips Research  
Eelco.Rommes@Philips.com

Jan Gerben Wijnstra  
Philips Semiconductors  
JanGerben.Wijnstra@Philips.com

## Abstract

*Product family engineering is one form of strategic reuse that can be particularly effective. However, establishing a product family is not a simple matter. Technical and non-technical aspects have to be dealt with to reap the benefits of organization-wide software reuse.*

*Especially if the scope surpasses that of a single product domain, this can be challenging. The family may then span several product groups, each of which might have its own product family installed. Product variability and the number of people involved will be larger, calling for appropriate measures.*

*We present a strategy for software reuse in complex, software-intensive medical systems. We describe its implementation in a platform architecture, an organization and its processes, as well as its evolution in response to strategic changes. The case presented here is especially interesting because it deals with a hierarchical product family for a wide range of complex systems, in a large organization.*

## 1. Introduction

Good software engineers will try to avoid duplication by reusing existing code, and by writing clear code that is easy to reuse, thus making life easier for others or their future selves. In many organizations, however, software written in one project or group will never cross the boundaries to other projects or groups, limiting the potential impact of reuse for the organization as a whole. When such software does find a broader audience, it is often found unsatisfactory by its receivers, because it solves a problem that is too different from theirs and hence it is not reused.

Product family engineering is a way to overcome these problems, and to bring the benefits of reuse to the organization as a whole [3,17]. Among the benefits reported are shorter time to market, increased efficiency and higher quality [12]. These benefits have clear business value, making product families a means to achieve strategic business goals.

Adopting a product family approach is not a simple matter. The task becomes even more complex when the

scope spans multiple product groups, each of which may have its own product family installed.

We describe the implementation and evolution of such a product family for medical imaging systems. Over the years, we have worked in several (research) projects in both platform and product groups within the family. Recently, we have interviewed some fifteen system and software architects working in the product family to gather their experiences. This paper is based on the results of these interviews and on our own experience.

Our contribution is that we relate architecture, process and organization aspects of a product family to strategic business goals. We analyse the design of these aspects in this specific case, describe risks that follow from it and present solutions.

We use the term ‘product family’ and related terms as defined within the ITEA FAMILIES project [8]. A product family is a collection of more or less similar products built upon the same technology. Some authors prefer ‘product line’ or ‘software product line’ to denote similar things.

We have structured the paper as follows. We describe the product family in section 2, paying attention to its scope, the strategy behind it and the implementation of this strategy in architecture, process and organization. Evolution of the product family is the topic of section 3. We analyse the responses to major changes in the product family and its context. Section 4 contains three case studies of products derived from the product family. These cases illustrate successes and challenges of the product family strategy. Section 5 gives an overview of related work and we present our conclusion in section 6.

## 2. The Garden product family of medical imaging systems

Here we introduce the product family. The first subsection introduces the systems within the family. The next subsection deals with the business strategy behind the product family. The implementation of this strategy in terms of architecture, process and organization is the topic of the last two subsections.

## 2.1 Scope

The systems in the Garden product family are medical image acquisition systems and medical IT systems. Examples of acquisition systems are ultrasound, X-ray and magnetic resonance imaging. The application of these systems depends on their acquisition technology. For example, ultrasound is used among other things for prenatal screening because of its minimal invasiveness. X-ray can be used for interventional cardiology because of its real-time performance and unsurpassed image quality. Examples of medical IT systems are picture archiving systems and information systems for radiology and cardiology departments.



Figure 1- Two examples of systems in the product family: MRI and Ultrasound

All of these are software-intensive systems with stringent requirements in the areas of performance, safety and security.

Medical imaging data is typically very large, both in number of images and in the size of these images. Depending on their acquisition mechanism, they can be two- or three-dimensional. Some modalities acquire sequences of images (movies). Some acquire information at many different levels and use it to reconstruct images. Special hardware is often needed for acquiring, processing or storing images.

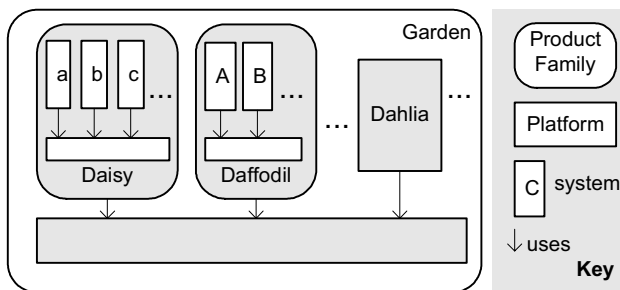


Figure 2 - Schematic Representation of the Garden Product family

Many of the systems in the product family are life-critical: errors in the system can cause serious physical harm to people, and wrong information given by such

systems could cause clinicians to take wrong decisions regarding the treatment of their patients. All systems handle sensitive patient data, making specific demands on areas like privacy and integrity.

The way in which this derivation is done varies per product family. We take a closer look at system derivation from the Garden platform in section 2.3.

## 2.2 Strategy

Despite their diversity, many of the medical systems in our scope share common functionality, for instance printing and archiving of medical images. Without Garden, each product group would develop and maintained such functionality by itself. This imposes high maintenance costs on the organization as a whole.

The following trends played an important role in the domain of medical imaging systems when Garden was initiated, and still do [13]:

- *Software growth*  
The role of software in establishing functionality is growing. This leads to more and more complex software in these systems. At the same time, the quality requirements for attributes such as safety and reliability also grow stronger.
- *Market pressure*  
Competition in the market puts pressure on the time-to-market for new system releases. For example, the number of imaging applications is rapidly increasing.
- *Need for integration*  
The number of digital systems in hospitals grows rapidly. As a result, customers demand better integration of these systems, both on the level of sharing data and of appearance to the end user.

The following goals can be formulated in response to these trends:

- *Increase efficiency*  
To increase the software development efficiency. This is needed to deal with the larger role of software and to shorten the time-to-market.
- *Feature propagation*  
Particular features in one system should quickly become available in other systems too. Propagated features improve the integration of systems at the presentation level, and make it easier to bring new releases to the market.
- *Improve interoperability*  
To improve the interoperability of the systems between each other and with systems from other vendors. Interoperability is important to achieve forms of integration such as data sharing and workflow.

The strategy that is followed to achieve these goals can be expressed as:

*To develop and maintain software solutions for common cross-product group functionality once and only once, in a cost-effective way.*

Here, ‘common cross-product group functionality’ means functionality that is needed by at least two product groups. The idea is that such software is applicable in at least four systems because each product group covers more than one system. This is in accordance with the rule of thumb that ‘a component has to be reused three to five times to recover the initial cost of creating it and the on-going cost of supporting it’ [11].

‘Once and only once’ is a design rule to avoid code duplication [2]. In this context, it means that all common functionality should have a single implementation that is available to all product groups. Within a single system, it may be worthwhile to scrupulously follow this rule, but in our context of many systems distributed across the organization, the overhead of reuse can be large enough to favour duplication over reuse. In this context, the term ‘in a cost-effective way’ means that reuse should only be pursued when it makes sense: it should not become an end in itself.

Let us see how this reuse strategy allows us to achieve the stated goals. Basically, the strategy is about fighting the well-known software problem of code duplication, but on a larger scale than usual. Instead of duplication within a project, it attacks duplication within an organization of multiple projects. By removing such duplication, the efficiency of the organization as a whole improves. It also allows features to spread across systems in different product groups. Interoperability is one of the areas of common functionality needed by many product groups. Developing such functionality once for many systems should improve interoperability with respect to each other. Moreover, the problem of multi-vendor interoperability can be addressed once, after which it is available to many systems.

The Garden product family is an implementation of this strategy. It establishes a platform of components that delivers common functionality to all product groups. A shared architecture facilitates that the components can be deployed in all products. But before theory can be put to practice, some challenges have to be dealt with.

The most important challenges have to do with scope, which is large in many ways. Some of the direct consequences of this large scope are:

- *Many groups involved*  
There are about ten target product groups. Having so many groups involved means a multitude of stakeholders with interests to defend.
- *High degree of variation*  
The variation addressed is much larger than for most product families. The systems vary in scope

from department IT products to Ultrasound modalities to MRI scanners. These systems differ substantially in such aspects as the applications they support, their architectures and technologies.

- *Massive code size*

The target systems have code sizes in the range of 100-1,000 KLOC. To make impact on the development effort of these systems means that a significant part of their code base should be reused from the platform. Unsurprisingly, the platform code is quite large too. Currently, the platform size is in the order of 100 KLOC. Since this code is designed to be reusable, it is inherently more complex.

We describe how Garden deals with these issues in the following sections.

### 2.3 Platform architecture

The scope of the product family comprises too many products and groups to allow a single comprehensive architecture. Instead, we need an architecture that safeguards the ability to reuse, while restraining the freedom of the product groups as little as possible (Malan and Brede Meyer call this ‘minimal architecture’ [14].) As we will see in section 4.2, even a minimal platform architecture can be too much in some cases. The architecture was described comprehensively in our earlier work [22]. Here we discuss the basic concepts needed to understand the rest of the paper. (See also Wang and Fung [21] for an overview of architecture paradigms in component-based systems.)

In reply to the high level of variation in system demands, the Garden platform provides a reference architecture that allows maximum room for product specific architectures. Yet it is powerful enough to enable reuse across the product family. This flexibility is achieved by combining three elements:

- (1) a suite of reusable components,
- (2) stable, generic interfaces and
- (3) a set of evolving information models providing semantics to the interfaces.

We will now discuss each of these elements, starting with the component suite. We use a definition of components by Szyperski [20]: a software component can be deployed independently and is subject to composition by third parties.

As illustrated by Figure 3, the Garden components are developed by the platform group and composed by the product groups to be used in their systems.

This figure shows that product architectures are composed of both product specific and platform components. It must be noted that these components are quite large in code size as well as in the scope of functionality that they offer, unlike for example GUI components common in some development environments. To give an impression,

a single component might take tens of thousands of lines of code.

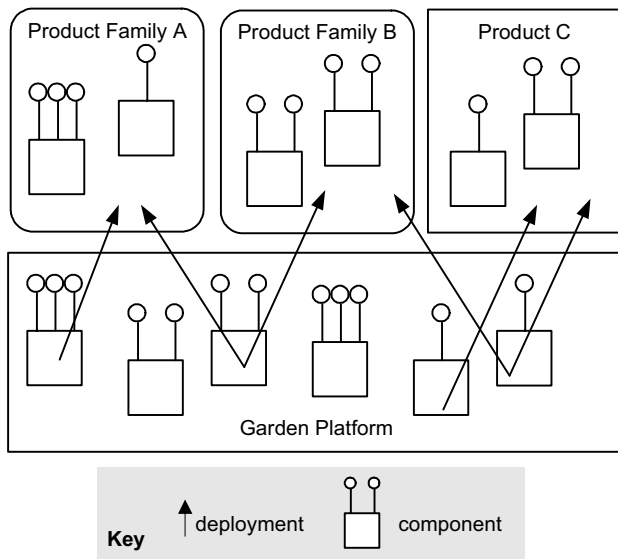


Figure 3 - Component Suite Usage

Each platform component provides one or more interfaces, and may also require one or more interfaces. A system that uses a component can access its functionality through the provided interfaces, and must offer an implementation for the required interfaces. Sometimes, using another Garden component can do this. Alternatively, a system-specific component may provide an implementation for this interface.

The interfaces are defined in a generic way: their semantics are defined separately in concrete information models. These models describe domain entities such as patient or image. This approach enables the separation of syntax (the interfaces) and semantics (the information models). The main advantage is encapsulation of knowledge, allowing the interfaces to remain stable over time. The information models are used in many components. An image, for example, can be displayed, printed, stored, etc., requiring interaction with different components. When the image information model evolves, the changes are captured in a single place: the information model. The interfaces that use the image information model may be left unchanged.

## 2.4 Organization

The product family organization comprises a platform group and (originally) about ten product groups, located in a number of geographically distributed sites. Some product groups have established their own product families. This situation is close to what Bosch calls a hierar-

chical domain engineering unit model [3,4]. In this case, the platform groups are the domain-engineering units.

The evolution of the shared architecture is guided by a cross-organizational board of product and platform architects.

The platform group takes care of development and maintenance of the reusable components. This group receives funding from the product groups before the start of a financial year. The contribution that a product group pays is partly based on its predicted use of platform components in the coming year. Another part is independent of the use of the platform: a product group will always contribute, even when no components are used in its products.

It is left to the product groups to decide what, if any, components they will use: reuse is not enforced by upper management, so product groups are free to create their own implementation of functions provided by Garden if they deem it necessary to do so.

Inevitably, some product groups will reuse more components than others. Some groups have started earlier, others may have an architecture that is better suited to reuse. But whatever the reasons behind it, this imbalance in reuse implies a risk that the platform group will focus its efforts too much on a few product groups. This would mean that it would become harder for the other groups to start reusing or to expand their existing reuse efforts. This risk is mitigated somewhat in the financing scheme: every product group supports the platform group financially, regardless of whether they reuse or not. Therefore, each product group has a stake in the platform to back up its needs, and cannot be neglected.

On the other hand, there is the risk that the platform group will not pay enough attention to its customer needs. Pursuing the holy grail of the ultimate generic platform is not a completely hypothetical situation for software experts, especially in a company with a strongly technology-centred culture. Since at least part of the platform group's funding depends on the reuse of their components, this risk is reduced. The shared architecture board is another means to mitigate this risk.

## 2.5 Process

A new release of the complete component suite is offered at a steady pace of two releases per year. Components in each release are thoroughly tested against each other. Releasing the full suite of components at once prevents the need for continuous retesting of components against many different versions.

Each first release adds new functionality to the component suite. Each second release is a stabilized version of the previous one, mainly dealing with quality by fixing reported problems. Both releases are developed in parallel.

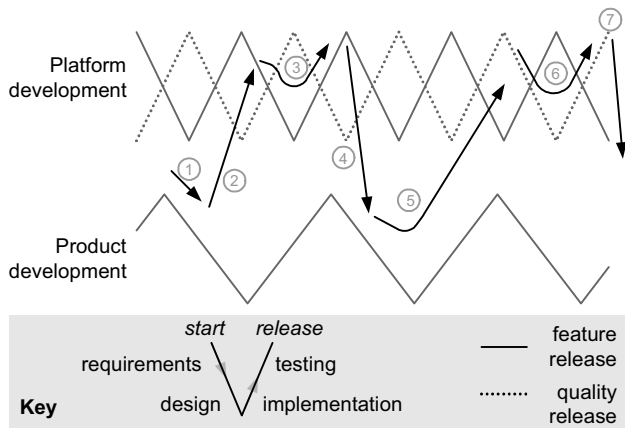


Figure 4 - Platform vs. product heartbeats

Figure 4 schematically shows this two-release development cycle against the hypothetical heartbeat of a product. Even though the platform release rate is almost twice as high as that of the product, it takes a number of product releases before a stable version of the platform is available that takes new product requirements into account. The numbers in the figure refer to the following activities:

- 1) A new product development cycle is started.
- 2) After some time, platform requirements for this release are available and communicated to the platform group.
- 3) Some of these product requirements are implemented in the next feature release of the platform.
- 4) The next version of the platform is released.
- 5) The product group evaluates and tests this new version.
- 6) The results of this evaluation are taken as input for the next stable platform release.
- 7) Release of the first stabilized platform that takes the requirements from step 2 into account.

When the product shown here is itself a reusable platform within a product family, the path from product requirements to platform use becomes still longer.

One way to shorten this path is by increasing the frequency of platform releases. This requires abandoning the suite-at-once release policy, which brings on aforementioned testing problems. We discuss a possible solution to this problem in section 3.1.

### 3. Evolution

Here we describe three major changes in the product family and its context: the growing number of components, the increase in the number of product groups and the broadening of the functional scope of the component suite. To some extent, each of these changes was foreseen from the beginning and thus planned for. Still, the practical effects are hard to predict and have to be experienced

in order to really appreciate them and find suitable responses. We present the effects of these changes on the strategic goals and the tactical responses to each of these changes.

### 3.1 More components

The first Garden component suite contained just a handful of components. The suite has gradually grown, and currently offers about twenty large components.

This steady growth has the following side effects:

- *Many assets*  
With the components come the requirements, designs, test cases, documentation, etc. All these artefacts have to be created and subsequently maintained. This is of course true for many large-scale software projects. But in this case the burden for creating and maintaining these assets is on the product groups as well as the platform group. They are the primary stakeholders for many of these artefacts and as such have to invest time and effort to ensure their quality. One product group architect pointed out that he had to review thirty thick requirements documents for a single new platform release. Multiply this effort by the number of product groups involved and a lot of manpower goes into reviewing documents, contradicting the goal of increased efficiency.
- *Mixed component maturity*  
Although the original plan was to allow a unique heartbeat for each component, in practice the components were soon released as a single suite. This decision was taken to avoid too much testing overhead: for each new release of a component, all other components that interface with it have to be retested. By releasing all components at once, the context of each is known in advance, which eases the testing pain. The drawbacks of this suite-at-once release strategy become more significant as the number of components grows. Some of the components in the platform have been there from the beginning. They have been used (and tested) in a broad range of products and contexts and are relatively feature complete. These components flourish in a steady, slow release rhythm because they are very stable. Other components were developed just yesterday. These youngsters are very dynamic and crave for short feedback cycles to chase out the bugs and to find the most valuable feature mix. With the suite-at-once release rate, these younger components suffer from a long feedback loop, leading to a slow advance in both functionality and quality. It is hard to get a component right the first time, it usually takes another two releases to get a really useful component. With the slow release rate, this means

that it may take years before a component is mature enough to be released in a product. This conflicts with the goal of rapid feature propagation.

To solve these problems, we need an approach that decreases the number of components, while fully maintaining the functional coverage of the platform. One option to reach this is to bundle existing components into a handful of *integrated components*, each covering a large area of functionality. Currently, we are investigating ways to achieve this.

### 3.2 More product groups

The number of product groups in the product family has grown substantially, and with it the number of (sub)organizations and stakeholders. This growth was mainly driven by acquisitions of (parts of) existing companies, but some of it comes from completely new products, developed from scratch.

Inevitably, the newer product groups did not take part in the early development of the shared architecture. Rather than a common achievement, this architecture is a fact of life to them. On the other hand, these product groups have no history with the platform. Therefore, they have a more objective view of it than the original product groups who suffered all the early growing pains and teething troubles that are inevitable in an undertaking of this size and complexity.

The steady adoption of reusable components by the new product groups indicates that the shared architecture is generic enough to support these new domains. This brings the strategic goals of the product family within reach of these groups too. Relying on the platform components for connectivity increases product interoperability. Although the goal of feature propagation also benefits from these product groups using the existing components, a significant contribution to this goal could be realized if parts of existing products were made reusable. This opportunity increases with the number of products, each of which has the potential of sharing features with other products. The product family organization and processes do not explicitly support this, but section 4.3 gives an example of how such reuse is realised in an ad hoc way.

### 3.3 Larger scope

Early releases of the Garden component suite dealt with basic medical imaging functionality, like support for industry standards in the area of connectivity. Such functionality is especially well suited to be developed in a generic way because virtually all products need and use it in similar ways. To further exploit the benefits of planned reuse, the scope of the product family was enlarged to address less generic areas of functionality as well. Thus, the platform entered the problem domains of the various

product groups more deeply. Consequently, the variation that these newer components face was larger. One source of such variation is the application context of the products. For instance, a viewing application will be used differently in a static review workstation on a doctor's desk, than in an operating room during a complex intervention. Professional culture plays a role too: different medical disciplines tend to do similar things in different ways. The imaging techniques that the products supported are another source of variation: viewing a handful of X-ray movies is very different from viewing hundreds of MRI images.

The main consequence of this change in scope is that more specific domain knowledge is needed to develop successful components. Sharing information through the architecture board is not sufficient. Instead, more intensive ways of cooperation is needed. This has led to projects in which the platform group closely cooperates with product groups to jointly develop domain knowledge intensive components. Since the number of product groups is large, it is hardly practical to form teams spanning all product groups. Instead, the goal is to have at least two product groups on board of such projects, although on occasions only a single product group will take part.

This close cooperation with product groups has important benefits. Of course, the developers of the product groups bring in the necessary clinical domain knowledge. With the domain knowledge comes the focus on a specific problem, which helps to avoid the genericity trap. Third, this cooperation stimulates the early use of generic code in end products, leading to a shorter feedback cycle.

On the other hand, the cooperation with just a few product groups gives these groups a lot of influence. There is a clear risk that the components thus developed will be less generic and more specific to the needs of the cooperating product groups. An indication that this has indeed happened comes from the fact that, for some components, adoption by the other product groups is slow.

Given the product family organization, it is the responsibility of the platform group to ensure that the components are reusable by more than just a few product groups. But for some components, generalizing them may require just as much domain-specific knowledge as was needed to develop them in the first place. Since the platform group has little or no direct contact with customers (hospitals) or end users (clinicians), it is difficult for them to build up the intensive domain knowledge needed to create more application-specific components. The product groups do have this knowledge, but mostly for their specific sub-domains. There are too many product groups to involve all of them in the development of each component. In practice, these problems are overcome by (informal) communication between platform and product groups.

## 4. Case studies

Here we present three case studies of product groups within the product family. These cases serve to give concrete examples of reuse in the family. We have chosen three very different cases. The first illustrates a product group with an architecture that is well aligned to obtain maximum benefit from platform reuse. The second shows the opposite case, with an incompatible architecture. The third shows a case of ad hoc reuse across product groups.

### 4.1 Daisy

The Daisy product family supports a single product group developing medical imaging modalities for one specific imaging acquisition technique. The Daisy architecture has steadily evolved. Daisy architects have worked closely together with the Garden group to develop some of the higher-level components and their concepts. Unsurprisingly, the Daisy product family architecture is quite well aligned with the shared Garden architecture, making reuse relatively easy. This alignment enables Daisy to incorporate a large number of Garden components into its architecture, allowing it to focus its own development efforts on the problems and features specific to their sub-domain. However, there are also some drawbacks.

One of the consequences of this wide-scale adoption of reusable components is a strong dependency on the platform, which is felt by both the Daisy and Garden groups.

On the one hand, Daisy is especially sensitive to the Garden lifecycle: schedule slips and scope changes in a Garden release have serious consequences for Daisy. To mitigate this risk, Daisy does not use the latest Garden release but rather an older, stabilized version. This has the drawback of not having access to the newest features, but it does allow for some breathing space.

On the other hand, there is the risk that Daisy could overwhelm the Garden group with its requirements and change requests, overshadowing the needs of other product groups. As explained in section 2.4, the platform financing scheme obliges the Garden group to pay attention to all of its customers.

### 4.2 Daffodil

Like Daisy, the Daffodil product family spans a number of imaging modalities that are based on common acquisition technology. Unlike it, this family replaced a number of more or less independent products in a revolutionary way some years ago. The development of the Daffodil product family overlapped in time with the initiation of Garden. Some of Daffodil's architecture decisions were taken using the unfinished Garden shared architecture as input, after which the shared architecture was evolved further. Unfortunately, this caused significant differences

between the Daffodil architecture and the final Garden's shared architecture in some key areas. As a consequence, reusing the Garden components within Daffodil is not so easy (see also Garlan et al. [9].)

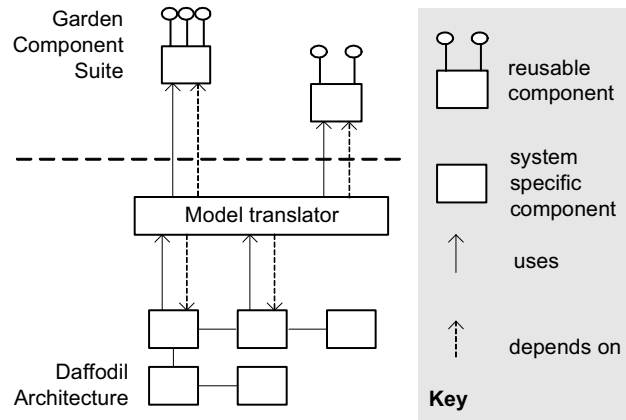


Figure 5 – Schematic Use of a Model Translator in Daffodil

Most notably, the Daffodil architecture has a domain model at its core that strongly combines semantics and syntax, whereas Garden uses *generic* interfaces in combination with its information models (as explained in section 2.3). To incorporate Garden components, the Daffodil concepts need to be translated to Garden concepts and vice versa. Because of the different approaches of the two, this is no simple mapping and therefore Daffodil uses a model translator as shown in Figure 5. This translator is dependent on changes in the information models of both Daffodil and Garden, making it a change-sensitive and maintenance-intensive component.

Daffodil's architecture mismatch makes it hard to reuse Garden components in the Daffodil products. Whenever a new Garden component is introduced, or an existing component is upgraded to a new release, the model translator must be adapted to accommodate for these components.

We identify two major responses by Daffodil to this situation. First, Daffodil uses only a few components from the Garden component suite. These are the older, more stable components at the infrastructure level. Reusing stable components means that the chance of large changes is less. This is important, since such changes mean that high costs are incurred to adapt the model translator. Furthermore, using infrastructure level components makes fewer demands on integration than using components from higher levels such as the middleware and application levels. The reluctance to reuse Garden components is also evident in the fact that there are Daffodil components for which a reusable counterpart is available, yet these are not used. Rather, the Daffodil group sticks with its homemade components at the cost of maintaining these itself, which

is estimated to be cheaper than the costs of integrating a new reusable component.

The second response is a slow evolution of the Daffodil architecture to the shared Garden architecture. Possible roadmaps to evolve towards a Garden-compliant architecture have been investigated, but have been neither embraced nor fully rejected. The main reason for moving towards such an architecture is to make it easier and cheaper to reuse Garden components by getting rid of the model translator component. The obvious reason for hesitation is that the current Daffodil architecture represents a considerable investment and several products depend on it: the architecture is not changed lightly. As said, the information model is at the core of its architecture. Changing it would mean drastically changing the Daffodil software architecture.

This architecture mismatch prevents Daffodil from getting optimum reuse results and thus from achieving Garden's strategic goals as formulated in section 2.2.

### 4.3 Dahlia

The Dahlia case study illustrates how an ad hoc form of reuse occurred in parallel to the planned reuse of the Garden product family.

The Dahlia product is a workstation, which runs several medical imaging applications. Some of these applications handle images and data from many different imaging modalities. Some other product groups that have a need for similar functionality have taken Dahlia applications and reused them in their products, mostly medical imaging modalities. This reuse is different from the managed reuse of Garden components in several ways. First, the Dahlia applications were not designed to be reusable. Although they incorporate Garden components, the applications as such have no variation mechanisms to support reuse in different modalities. Second, entire Dahlia applications are reused, whereas Garden offers components, which need to be integrated and extended by the product groups in order to form applications.

Since the Dahlia applications were not meant to be reused, no reuse process is in place, for instance to handle requirements from other product groups. The Dahlia group has a primary responsibility for its own product and it has no reasons to support reuse by other groups, other than being a good corporate citizen. This means that the reusing product groups have very little influence on how the applications will evolve, with no guarantee that future versions will remain useful to them.

Despite this risk, there are compelling reasons for modalities to reuse Dahlia applications. Dahlia is a workstation based on common commercial hardware and software. This makes it relatively easy for modalities to run its applications, for example by adding a separate system next to the core system and using simple hardware solutions to establish integration at the user interface level.

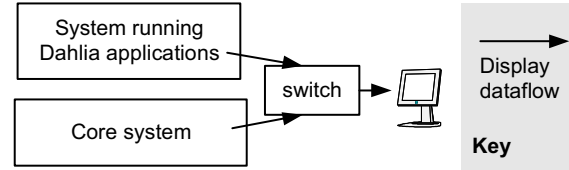


Figure 6 - Dahlia Application Reuse

There are several advantages to this reuse scheme.

First, since the applications are run on separate hardware, they do not interfere with the core functionality of the modality. This greatly decreases testing and integration efforts. For example, subtle interactions through shared resources such as disk space, memory and processor power are excluded. Such effects may impact the safety and reliability of these systems, and should be carefully considered.

Second, the dependencies on the Dahlia lifecycle are minimized. When new versions of applications become available, the old versions can be replaced relatively simply on the extra system, without touching the rest of the system. Feature propagation is thus established in an efficient way.

One disadvantage of this reuse scheme is that more sophisticated forms of integration are hard to achieve. Currently, there is no demand for integration beyond the current level, but users will probably demand better integration in time. A second disadvantage is the lack of support for reusing these applications. There is no guarantee that future versions will be equally useful to the product groups. There are no procedures to handle change requests or bug reports.

Despite these problems, there is clearly a need within the product family for this kind of application reuse across product groups. For as long as it holds, it serves the goals of feature propagation, efficiency and interoperability well.

## 5. Related work

Several overviews of product family engineering approaches including technical, process and organization aspects have been published, for instance by Jacobson et al. [11], Bosch [3] and SEI [17].

Many case studies on software reuse through a product family approach are available, for example by Clements and Northrop [6], Jaaksi [10], Ebert and Smouts [7], Bass, Clements and Kazman [1], and our own work [23]. A classification of product family evolution is given by Svahnberg and Bosch [19].

Less attention has been paid to reuse across multiple product families. Van Ommering coins the term product population [15] and proposes composition as a means to achieve this [16]. Bosch includes hierarchies of families in his discussion of product family organizations [4].

Sherrill et al. discuss a case comprising three product families in the printing domain that share a core architecture [18]. Bratthall et al. describe a common architecture for reuse across many, very diverse product families in a large organization [5].

## 6. Conclusion

We have described an approach to product family engineering for a large family of diverse medical imaging systems. The approach involves a minimal architecture and a platform comprising a suite of components, with interfaces and information models. The platform is developed and maintained by a platform group, in close cooperation with the product groups that use the platform in their own products and product families. The platform group is financed by these product groups. The contribution paid by a product group is partly based on its (planned) usage of platform components.

The main risks that we have encountered are:

- *Imbalance of power*  
A few product groups may gain too much impact on the platform because they are the platform group's most important customers.
- *Genericity trap*  
The platform may become too generic because the platform group is independent from any end product.
- *Missing domain knowledge*  
The platform group does not have the knowledge needed to develop certain components.
- *Architecture mismatch*  
The architecture of a product group may prevent it from optimally using the platform.
- *Heartbeat mismatch*  
Differences in the release schedules of the platform and product groups cause late use of components in end products and thus a slow feedback loop. Also, more mature components benefit from a slow release cycle where younger, dynamic components thrive on a short feedback loop. Releasing the full suite as a whole inevitably leads to a suboptimal cycle for some components.
- *Increasingly dependent*  
The more a product group uses the platform, the more it suffers from its dependencies on the platform group. As the product group tries to regain control, it may affect the imbalance of power.

Some techniques used to overcome these risks are:

- *Financing scheme*  
Since every product group contributes to the platform group, every group has a direct financial stake and thus influence. The usage-dependent part of the contribution serves to avoid the genericity trap by

providing direct, financial feedback on the usefulness of the platform components.

- *Cross-group cooperation*  
A board of product and platform architects guides the shared architecture to ensure its usefulness and to obtain buy-in from all groups. Early involvement of product groups on platform component projects yields domain knowledge, focus, and shorter feedback loops, but it may also affect the imbalance of power, in a positive or a negative way.
- *Minimal architecture*  
The shared architecture must be specific enough to allow reuse while restraining the product groups as little as possible. This helps prevent architecture mismatch and lessens the dependencies of product groups on the platform group.
- *Informal communication*  
With so many people involved, official procedures easily get cluttered to become bottlenecks. Informal communication between platform and product groups is therefore indispensable.

The following are potential solutions that need further investigation:

- *Decoupled component heartbeats*  
Decoupling the release schedules of components should lead to a higher release frequency of more immature components, thus shortening their feedback loop and making them ready for use sooner. At the same time, more mature components can remain a slower development pace that does justice to their size and complexity. It is feared that decoupling components will lead to many versions of many components, making testing a lot more complicated.
- *Integrated components*  
The Dahlia case study shows that reusing whole applications can be beneficial within a product family. By integrating platform components into even larger chunks that are close to applications, decoupling component heartbeats should become easier.

In conclusion, the case described here shows that architecture, process and organization are all aspects that influence each other. They should be balanced to optimize the strategic results of product family engineering.

## Acknowledgements

We thank our colleagues Frank van der Linden, Eugene Ivanov, André Postma, Auke Jilderda and Pierre America for discussions and feedback on this work, as well as numerous Philips Medical Systems architects who were kind enough to let us interview them, as well as those with whom we have cooperated.

This work was carried out in the Eureka Σ! 2023 Programme, ITEA project ip02009 Families.

## References

- [1] L. Bass, P. Clements, and R. Kazman: *Software Architecture in Practice - Second Edition*. Addison-Wesley, 2003.
- [2] K. Beck: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
- [3] J. Bosch: *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*. ACM Press/Addison-Wesley, 2000.
- [4] J. Bosch: Software Product Lines: Organization Alternatives. In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Ontario, Canada, May 2001, IEEE Computer Society, p. 91-100.
- [5] L. G. Bratthall, R. v. d. Geest, H. Hofmann, E. Jellum, Z. Korendo, R. Martinez, M. Orkisz, C. Zeidler, and J. S. Andersson: Integrating Hundred's of Products through One Architecture - The Industrial IT architecture. In: *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, Orlando, Florida, USA, ACM Press, p. 604-614.
- [6] P. Clements and L. Northrop: *Software Product Lines - Practices and Patterns*. Addison Wesley Professional, August 2001.
- [7] C. Ebert and M. Smouts: Tricks and Traps of Initiating a Product Line Concept in Existing Products. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, Portland, Oregon, USA, IEEE, p. 520-525.
- [8] European Software Institute: Families Project. <http://www.esi.es/en/Projects/Families/famMain.html>, European Software Institute (ESI), April 2002.
- [9] D. Garlan, R. Allen, and R. Ockerbloom: Architectural Mismatch or Why it's hard to build systems out of existing parts. In: *Proceedings of the 17th International Conference on Software Engineering (ICSE '95)*, Seattle WA, USA, April 1995, ACM Press.
- [10] A. Jaaksi: Developing Mobile Browsers in a Product Line. *IEEE Software*, Vol. 19, No. 4, p. 73-80, IEEE, August 2002.
- [11] I. Jacobson, M. Griss, and P. Jonson: *Software Reuse - Architecture, Process and Organization for Business Success*. ACM Press Books, 1997.
- [12] C. W. Krueger: Benefits of Software Product Lines. *Software Product Lines*, May 2004. <http://www.softwareproductlines.com/benefits/benefits.html>
- [13] F. v. d. Linden and J. G. Wijnstra: Platform Engineering for the Medical Domain. In: Frank van der Linden, ed.: *Software Product-Family Engineering, 4th International Workshop (PFE 2001)*, Bilbao, Spain, October 2001, Springer LNCS, p. 224-237.
- [14] R. Malan and D. Bredemeyer: Less is More with Minimalist Architecture. *IT Professional*, Vol. 4, No. 5, p. 48-47, IEEE, September 2002.
- [15] R. v. Ommering: Beyond Product Families: Building a Product Population? In: *Proceedings of the 3rd international workshop on the development and evolution of software architectures*, Las Palmas de Gran Canaria, March 2000, p. 78-85.
- [16] R. v. Ommering and J. Bosch: Widening the Scope of Software Product Lines - From Variation to Composition. In: *Proceedings of the 2nd International Conference on Software Product Lines (SPLC-2)*, San Diego, California, USA, Springer Verlag.
- [17] SEI: A Framework for Software Product Line Practice - version 4.2. Carnegie Mellon Software Engineering Institute (SEI), May 2004. <http://www.sei.cmu.edu/plp/framework.html>
- [18] J. Sherrill, J. Averett, and G. Humphrey: Implementing a Product Line-Based Architecture in Ada. In: *Proceedings of the 2001 annual ACM SIGAda international conference on Ada*, Bloomington, MN, USA, ACM Press, p. 39-46.
- [19] M. Svahnberg and J. Bosch: Characterizing Evolution in Product Line Architectures. In: *Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications*, Scottsdale, Arizona, USA, p. 92-97.
- [20] C. Szyperski: *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
- [21] G. Wang and C. K. Fung: Architecture Paradigms and Their Influences and Impacts on Component-Based Software Systems. In: *Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS '04)*, IEEE Computer Society.
- [22] J. G. Wijnstra: Components, Interfaces and Information Models within a Platform Architecture. In: *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE '01)*, Erfurt, Germany.
- [23] J. G. Wijnstra: Quality Attributes and Aspects of a Medical Product Family. In: *Proceedings of the Hawaii International Conference on System Sciences (HICSS-34)*, Island of Maui, Hawaii, IEEE Computer Society.