Cover page

Title of paper:
“Characterizing Evolution in Product Line Architectures”

Type of Paper:
Research paper and experience paper

Authors:
Mikael Svahnberg, Professor Jan Bosch

Abstract:
Product-line architectures present an important approach to increasing software reuse and reducing development cost by sharing an architecture and set of reusable components among a family of products. However, evolution in product-line architectures is more complex than in traditional software development since new, possibly conflicting, requirements originate from the existing products in the product-line and new products that are to be incorporated. In this paper, we present a case study of product-line architecture evolution. Based on the case study, we develop categorizations for the evolution of requirements, the product-line architecture and product-line architecture components. Subsequently, we analyze and present the relations between these categorizations.

Subjects:
frameworks, software engineering practices, experience with object-oriented applications and systems

Contact information:
email: [Mikael.Svahnberg|Jan.Bosch]@ipd.hk-r.se
WWW: http://www.ipd.hk-r.se/[~msv|~bosch]

postal address:
University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
Soft Center, S-372 25 Ronneby
Sweden

Telephone:
Mikael Svahnberg: +46 457-28618
Jan Bosch: +46 457-78726
Fax:
+46 457-27125

Characterizing Evolution in Product Line Architectures

Mikael Svahnberg & Jan Bosch
University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
S-372 25 Ronneby, Sweden
e-mail: [Mikael.Svahnberg|Jan.Bosch]@ipd.hk-r.se
URL: http://www.ide.hk-r.se/[~msv/|~bosch/]

Abstract
Product-line architectures present an important approach to increasing software reuse and reducing development cost by sharing an architecture and set of reusable components among a family of products. However, evolution in product-line architectures is more complex than in traditional software development since new, possibly conflicting, requirements originate from the existing products in the product-line and new products that are to be incorporated. In this paper, we present a case study of product-line architecture evolution. Based on the case study, we develop categorizations for the evolution of requirements, the product-line architecture and product-line architecture components. Subsequently, we analyze and present the relations between these categorizations.

1 Introduction

Product-line architectures, i.e. a software architecture and set of reusable components shared by a family of products, present an important approach to increasing software reuse and reducing the cost of software development and maintenance for software companies. Different from the popular view on component-oriented programming, the components in the product-line architectures that we have studied [Bosch 98a][Bosch 98b] are large pieces of software containing up to 100 KLOC. These components are typically not modeled as black-box components, but rather as object-oriented frameworks that cover functionality common for the products in the product-line and support the variability required for the various products.

Software in product-line architectures, once developed, is subject to considerable evolution, since a constant flow of new requirements is present. These requirements originate from the feedback received from customers concerning existing products that are out on the market and from new products that are to be integrated into the product-line. Since so many new requirements need to be handled that, potentially, are conflicting, the typical way to dealing with this is to create two independent evolution cycles, i.e. for each product, incorporating new requirements that are product-specific, and for the product-line architecture as a whole, incorporating new requirements that affect all or most of the products in the product-line.

Although evolution in product-line architectures is a complex and difficult to manage process, it is not studied much by the research community. To address this, we have performed a case study of a product-line architecture at Axis Communication AB, a swedish company selling a wide range of printer, storage, scanner and camera server products worldwide. The company has employed product-line architecture based software development since the beginning of the 1990s and uses object-oriented frameworks as the components in the product-line. We study one product-line architecture component in detail, i.e. a filesystem framework component. This component has evolved through two generations of four releases each and contains many relevant examples.

Based on the case study, we present categorizations of the evolution of the requirements, the product-line architecture and the product-line architecture components. Using the categorizations, we relate the three levels of evolution to each other. The result is a first version of a taxonomy of product-line architecture evolution that, we hope, improves understanding of the evolution process and the relations between the types of evolution at the different levels.

The remainder of the paper is organized as follows. In the next section, the notion of product-line architectures and the evolution process are described. Section 3 discusses the case study method, the company, the product-line architecture and the evolution of the file-system framework component through eight releases. Based on the case, we define categorizations of requirements evolution, product-line architecture evolution and component evolution in section 4. Related work is discussed in section 5 and the paper is concluded in section 6.

2 Product Line Evolution

In this section we present our view of how a product line evolves. As we see it, the process has a static part, and a dynamic. Furthermore, we see that there are a small number of changes that a product line as a whole can be subject to.

Static and Dynamic

The evolution of a product line is driven by changes in the requirements. These requirement changes can come from a number of sources, such as the market, future needs in the company, or a desire to introduce new products into the product line. The evolution consists of two parts. One is the static picture of how the company has organized itself around the product line, and the other is the actual evolution, as a requirement propagates through the static organization.

Take, for example, the organization in Figure 1, where a particular business unit is driven by a number of requirements. These requirements are divided between the products that this business unit is responsible for and the general requirements on the product line as a whole. The requirements on a particular product are naturally implemented in the product-specific code, and the more general requirements influence the product line as a whole. The product-line requirements can affect either the architecture of a particular framework, thus creating a change in its interface, or it can cause a change in one or more of the concrete implementations of the framework. In some cases, the requirements may even cause a split of one component into two components, or the introduction of a completely new component into the product line architecture.

The product-line architecture with a number of concrete framework implementations is then instantiated into a set of products. The products give input as to how they should evolve, which is feedbacked into the business unit requirements, thus completing the cycle.

Definition of framework

We would at this point like to give our definition of a framework, in order to alleviate continued reading. As is hinted in Figure 1, we define a framework to consist of a framework architecture, and a number of concrete framework implementations. As an example, a file system framework has an abstract architecture where the conceptual entities and their relations are identified. The framework implementations are the specific implementations for each concrete file system to be implemented, such as FAT, UFS, and ISO9660.

This interpretation holds well in a comparison to [RJ 96], in which a white box framework can be mapped to our framework architecture, and a black box framework consists of several of our framework implementations in addition to the framework architecture.

It should be noted that this definition is a somewhat modified terminology compared to the one we found during our study. In the company studied, the term ‘framework’ is only used to refer to the framework architecture. The framework implementations are merely referred to as ‘implementations’.

Product Line Architecture Changes

The top-level changes that a product line architecture can be subject to can be divided into three categories; *new components* are introduced, *new relations* between components are added, and existing *relations are changed*.

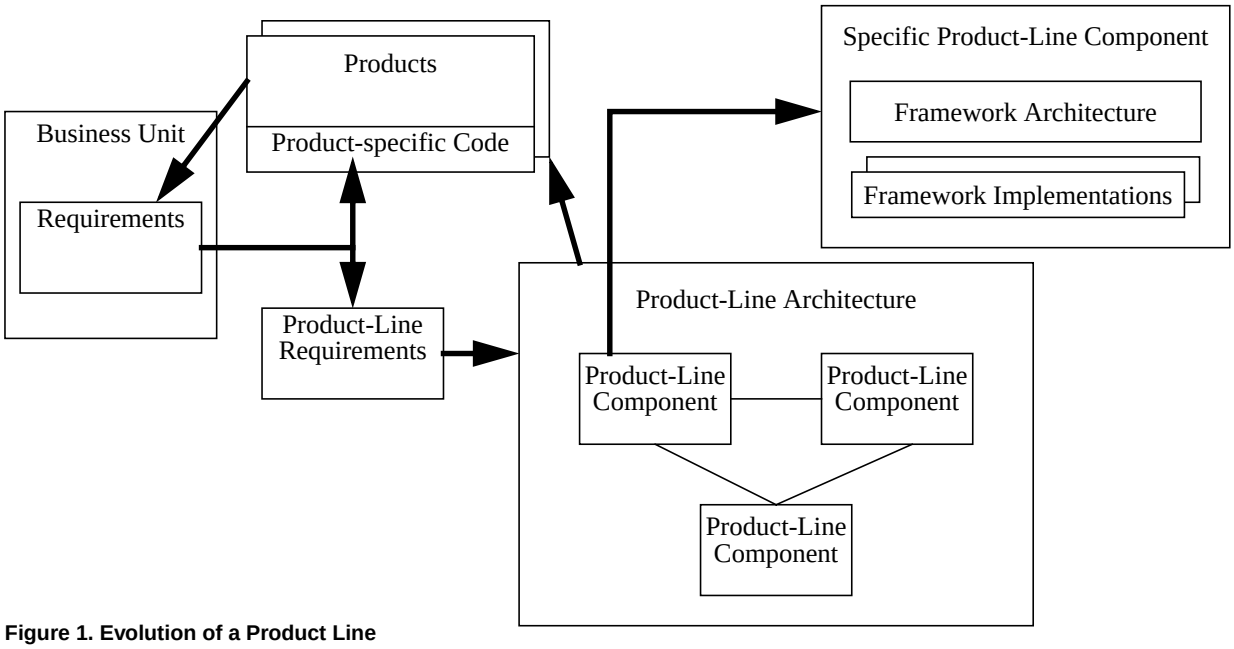

Figure 1. Evolution of a Product Line

When new functionality is requested, say for a new product, it is obvious that this may lead to addition of frameworks into the product line architecture. This regards when the functionality is of a kind that does not fit into the existing structure as a concrete implementation of some other component in the product line architecture. Another case when the product line changes in this way is when existing functionality is broken out from one framework to become a stand-alone component in the architecture.

Previously unrelated components may be connected by some requirements. When a new component is added to the architecture, this is usually connected to the existing components in some way. It may also be that some implementation of a component may require a binding to a framework that was not there before. From this follows that new relations between components are introduced.

Almost as a corollary to the splitting of one component into several new components, the relations that were concerned with that part of the framework will have to follow to the new component. This leads to the third category, the changing of relations. The other major contributor of this category is when the interface of a framework is changed, and the components depending on this interface have to adjust to this. Furthermore, one could argue that the strenghtening or weakening of a relation, i.e. that a component binds itself further or detaches itself from another component, is also part of this third category.

3 The Case

The case that we studied is a product-line architecture at Axis Communications AB. In this section, we present the case study method that we used, the company and the product-line architecture. The major part of this section is concerned with the evolution of a major component in the architecture, i.e. an object-oriented framework for file systems. We describe two generations of this component, consisting of four releases each. This case forms the primary input for the categorization that we present in section 4.

Case study method

The main goal in our study was to find out how a framework evolves when it is part of a product line. To achieve this goal, we conducted interviews with key personnel that have been involved in the studied system for quite some time. Furthermore, we foraged the intranet for more information. To our fortune, their intranet is a veritable bonanza of information, containing everything from requirement specifications to design decisions and protocols from project meetings.

The selection of the company was perhaps not as random as one would wish, we chose the company for a number of reasons. First, Axis is a large software company in the region. Second, they are part of a government-sponsored research project on software architectures, of which our university is one of the other partners. Third, and this we appreciate very much, they have a philosophy of openness, making it very easy to extract information from them. Despite this “convenience sampling”, we believe that the company is representative for a larger category of software development organizations.

As a preparation, the intranet was studied extensively before the interviews took place. The interviews were unstructured, and consisted mostly of encouragements to the interviewed people to help them remember how and in what order things happened. The interview findings were correlated to what can be found on the intranet, and some further questions were later asked to the interviewed people, to explain some things further.

The company

Axis Communications is a relatively large swedish software and hardware company that develops networked equipments. Starting from a single product, an IBM print-server, the product line has now grown to include a wide variety of products such as camera servers, scanner servers, CD-ROM servers, Jaz-servers, and other storage servers.

As mentioned earlier, the company has been using a product-line approach since the beginning of the ‘90s. Their software product line consists of reusable assets in the form of object-oriented frameworks. Currently, the assets are formed by a set of 13 object-oriented frameworks, although the size of the frameworks differs considerably.

The layout of the product line is a hierarchy of specialized products, of which some are product lines in themselves. At the topmost level, elements that are common for all products in the product line are kept, and below this level those things that are common for all products in a certain product group are kept. Examples of product groups are the storage servers, including for example the CD-ROM server and the Jaz server, and the camera servers. Under each prod-

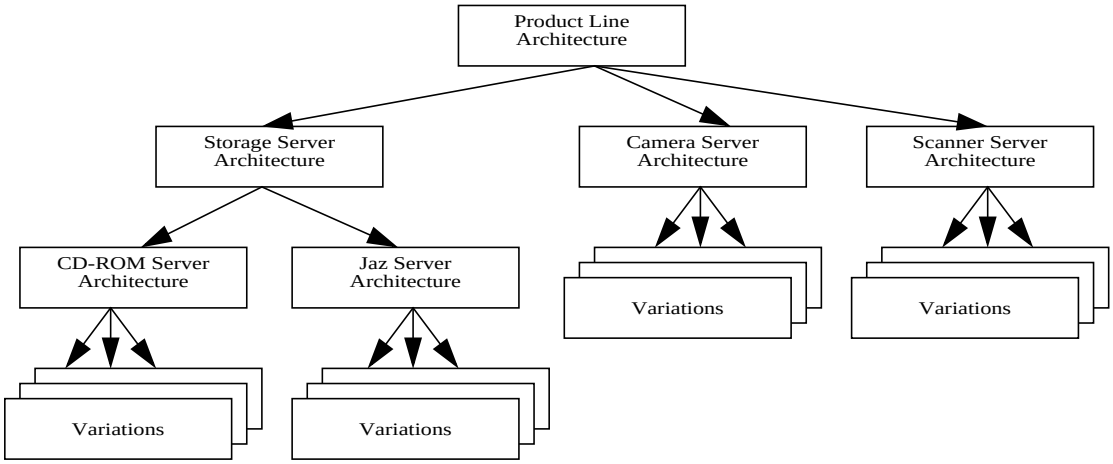

Figure 2. Product Line hierarchy

uct group, there are a number of products, and under this are the variations of each product. Figure 2 illustrates the layout of the product-line and the variations under each product group.

Each product group is maintained by a particular business unit that is responsible for the products that come out of the product line branch, and also the evolution and maintenance of the frameworks that are used in the products. Business units may perform maintenance or evolution on a software asset that belongs to another business unit, but this must be done with the consensus of the business unit in charge of the asset.

Further info on how the product line is maintained, and the issues concerning this is presented [Bosch 98a] and [Bosch 98b].

The studied product line

During this study, we have focused on a particular product, the storage server. This is a product that initially allowed you to plug in CD-ROM devices onto the network. This initial product have later been evolved to several products, of which one is still a CD-ROM server, but you also have a Jaz server and recently an HardDisk server was added to the collection of storage servers. Central for all these products is a file system framework that allows for uniform access to all types of storage devices. In addition to being used in the storage servers, the file system framework is also used in most of the other products since these in general include a virtual file system for configuration and access to what ever the products may interface. However, since most of the development on the file system framework naturally comes from the storage business unit, this unit is also responsible for it.

The file system framework have existed in two distinct generations, of which the first was only intended to be included in the CD-ROM server, and was hence developed as a read-only file system. The second generation was from the beginning intended to support read-write file systems, and was implemented accordingly. The first generation consists of a framework interface, under which there are concrete implementations of various file systems like ISO9660 (for CD-ROM disks), Pseudo (for virtual files), UFS (for Solaris-disks), and FAT (for MS-DOS and MS-Windows disks). These concrete file system implementations interface to a block device unit, that in turn interface a SCSI-interface. Parallel with the abstract framework is an access control framework. On top of the framework inter-

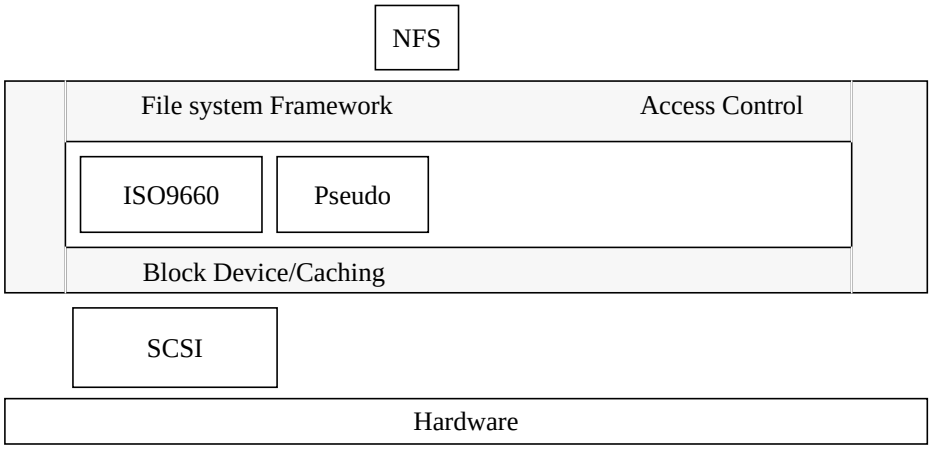

Figure 3. Generation one of the file system framework

face, various network protocols are added such as NFS (UNIX), SMB (MS-Windows), and Novell Netware. This is further illustrated in Figure 3, where we see that the file system framework consists of both the framework interface, and the access control that interfaces various network protocols like NFS. We also see that the block device unit and the caching towards device drivers such as the SCSI unit is part of the file system framework, and that the concrete implementations of file system protocols are thus surrounded by the file system framework.

The second generation was rather similar to the first generation, but things were more modularized, and some of the future enhancements were foreseen from the beginning. Like the first generation, the first release of this framework only had support for NFS, with SMB being added relatively soon. An experimental i-node-based file system was developed, and was later replaced by a FAT-16 file system implementation. The second generation file system framework is depicted in Figure 4. As can be seen, the framework was split into several smaller and more specialized frameworks. Notably, the Access Control-part was “promoted” into a separate framework.

Generations and Releases

Below, the major releases of each generation are presented in further detail, with focus on a particular product line component, i.e. the file system framework and its concrete implementations. As mentioned earlier, the two generations had differing overall requirements in that the first generation was only intended to be a read-only file system framework for usage in the CD-ROM server. The second generation was intended to be used in other products as well, and these products required read and write capabilities. The read-only functionality was embedded so deep into the code that it is simply too costly to merely convert the first generation into a read-write file system. This is the reason for why there are two generations.

Generation one, Release 1

The requirements on this, the very first release of the product, were to develop a CD-ROM server. A CD-ROM server is, as described earlier, a device that distributes the contents of a CD-ROM over the network. This implies support for network communication, network file system support, CD-ROM file system support, and access to the CD-ROM

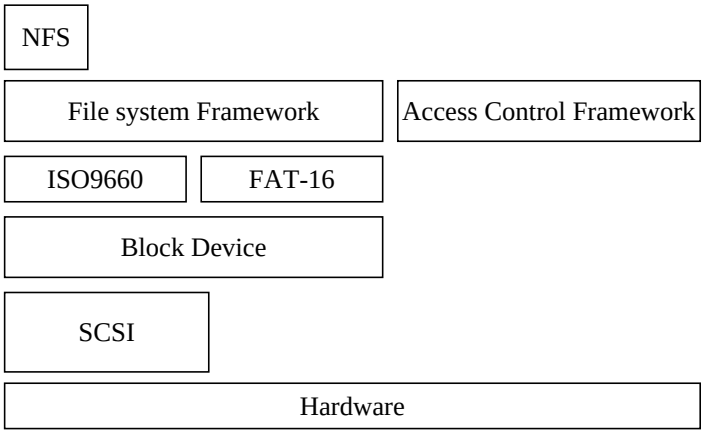

Figure 4. Generation two of the file system framework

hardware. The birth of the CD-ROM product started this product line. The product line were later to branch in three different sub-trees with the introduction of the camera servers and the scanner servers.

Requirements on the file system framework were to support the CD-ROM file system, i.e. the ISO9660 file system, and a Virtual-file pseudo file system for control and configuration purposes. In release one, it was only required that the framework supported the NFS protocol.

In Figure 3 we see a subsection of the product line architecture where the NFS protocol, which is an example of a network file system protocol, interfaces the file system framework. The file system framework in turn accesses a hardware interface, in this case a SCSI interface. Not shown in the picture is how the network file system protocol connects to the network protocol components to communicate using for example TCP/IP, and how these protocol stacks in turn connects to the network hardware interfaces.

Since, in this release, the product was only to support NFS, the file system framework contained many NFS-specific things. Later in the project, SMB was added as well.

The two framework implementations, the ISO9660 and the Pseudo file system implementations differed of course in how they worked, but the most interesting difference was in how access rights were handled differently in the two subsystems. In the ISO9660 implementation access rights were handed out on a per-volume basis, whereas in the Pseudo file system access rights could be assigned to each file. This was done with the use of the two classes NFSUser and NFSAccessRight.

The addition of the SMB network file system protocol later in the project did not change much, since the system had been designed to work with SMB from the beginning, and SMB is not that different from NFS. Code was added to handle access control for SMB, specifically the classes SMBUser and SMBAccessRights were added.

In retrospect, it is hard to agree upon a cause for why the code was bound to a particular network file system, i.e. NFS. Inadequate analysis of future requirements on the file system framework could be one reason. Shortage of time could be another reason; there might not have been enough tome to do it flexible enough to support future needs.

Generation one, Release 2

The goal for this, the second product line release was, among other things, to create a new product. To this end, a shift from Ethernet to Token Ring was conducted. In addition, support for the Netware network file system was added, plus some extensions to the SMB protocol. Furthermore, the SCSI-module was redesigned, support for multisession CD's was added, as well as a number of other minor changes.

The effect this had on the product line architecture was that Netware was added as a concrete implementation in the network file system framework, modules for Token Ring was added, and a number of framework implementations were changed, as discussed below. Figure 5 summarizes how the product line architecture was changed by the new product's requirements.

The addition of the Netware protocol could have had severe implications on the file system framework, but it the requirements were reformulated to avoid these problems. Instead of putting requirements on the framework that was known to be troublesome to fit into the existing architecture, the specification of the Netware protocol was changed to suit what could be done within the specified time frame for the project. This meant that all that happened to the file system framework was the addition of NWUser and NWAccessRight-classes. Basically, it was this access right model that could not support the Netware model for setting and viewing access rights.

The NWUser and NWAccessRight-classes were added to the implementations of the Pseudo file system and the ISO9660 module. Furthermore, the ISO9660-module was modified to support multisession CD's.

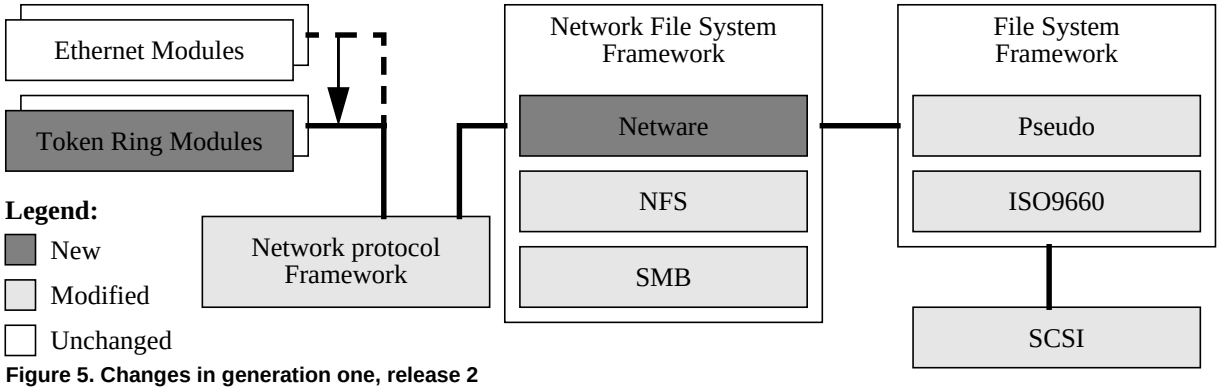

Figure 5. Changes in generation one, release 2

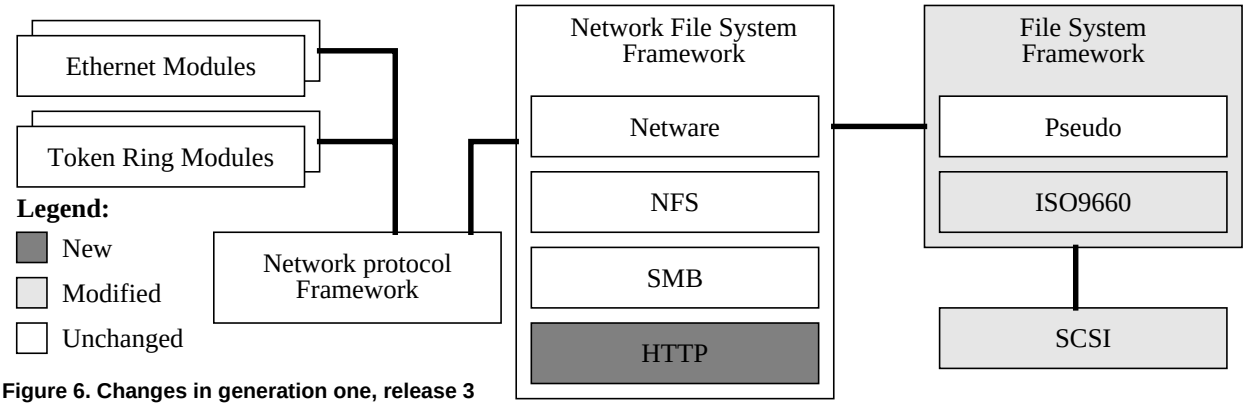

Figure 6. Changes in generation one, release 3

Late in this project a discrepancy was found between how Netware handles file-ID's compared to how NFS and SMB handle them. Instead of going to the bottom of the problem and fixing this in the ISO9660-module, a filename cache was added to the Netware module. Again, the file system framework interface managed to survive without a change.

Generation one, Release 3

Release three of the product line architecture was more of a clean-up and bug-fixing project. The SCSI-driver was modified to work with the new version of the hardware, and a web interface was incorporated to browse details mainly in the pseudo file system that contains all the configuration-details. Support for long filenames was to be implemented, as was support for PCNFS clients.

The effects on the product line architecture were, consequently, small. A new framework implementation was added to the network file system framework to support the web-interface, and some other framework implementations, in particular the SCSI-driver, were modified to support the new functionality and hardware. Figure 6 shows the modifications in release 3.

As mentioned, the SCSI-driver was modified to better utilize the new version of the hardware, that supported more of the SCSI-functionality on-chip instead of in software. Long filename support was added in the ISO9660-module. This change had impact on the framework architecture as well, since the DirectoryRecord class had to support the long filenames. This implies that the interface of the file system component changed, but this merely caused a recompilation of the dependent parts, and no other changes.

Generation one, Release 4

This release were to be the last one in this generation of the file system framework, and was used as part of new versions of the two aforementioned CD-ROM products until they switched to the second generation two years later. Requirements for this release were to support NDS, which is another Netware protocol, and to generally improve the support for the Netware protocol.

As in the previous release, the product line architecture did not change in this project but, as we will see, several framework implementations were changed to support the requirements. NDS was added as a new module in the network file system framework, and were the cause for many of the other changes presented below.

NDS has a, to the file system framework, totally new way of acquiring the access rights for a file. Instead of having the access rights for a file inside the file only, the access right to the file is calculated by hierarchically adding the access rights for all the directories in the path down to the file to the access rights that are unique for the file itself. This change of algorithm was effectuated in the access control part of the file system implementations. Unfortunately, the access control parts for the other network file system protocols had to be modified as well, to keep a unified interface to them.

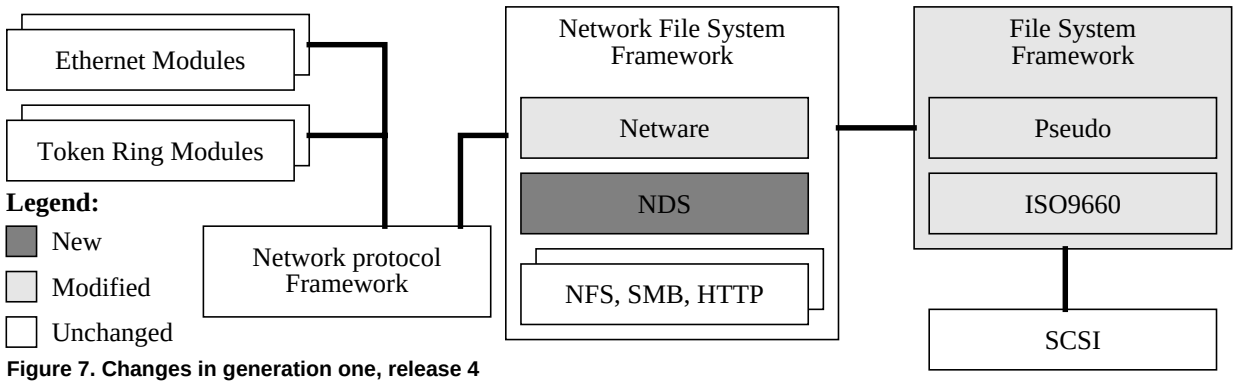

Figure 7. Changes in generation one, release 4

The namespace cache that was introduced in release two was removed, and the requirement was now implemented in the right place, namely in the ISO9660-subsystem. This meant that the namespace cache could be removed from the Netware module. Figure 7 summarizes the changes in release four.

Generation two, Release 1

Work on generation two of the file system framework was done in parallel to generation one for quite some time. As can be seen in Figure 8, the two generations existed in parallel for approximately four years. However, after the fourth release of generation one all resources, in particular the staff, were transferred into the second generation, so parallel development was only conducted for two-odd years. The transfer of resources is signified by the arrow between generation one and two in the picture.

Requirements on release one of this second generation were very much the same as those on release one of the first generation, with the exception that one now aimed at making a generic read-write file system from the start, something that it had been realized from the start that the previous generation was unfit for. The experiences from generation one were put into use here, and as can be seen when comparing Figure 4 with Figure 3 the second generation was much more modularized from the start. As before, only NFS and SMB were to be supported in this first release, but with both read and write functionality. A proprietary file system, the MUPP file system was developed in order to understand the basic of an i-node based file system.

Some framework implementations could be re-used from generation one, modified to support read and write functionality, such as the SMB protocol.

Generation two, Release 2

Release two came under the same project frame as the first release, since the project scope was to keep on developing until there was a product to show. This product, a JAZ-drive server, used major release 2 of the file system framework. Because of the unusual project plan, it is hard to find out what the actual requirements for this particular development step were, but it resulted in the addition of a new framework implementation in the file system, the FAT-16 subsystem, and the removal of the MUPP-fs developed for release one. The NFS protocol was removed, leaving the product only supporting SMB. Some changes were made in the SCSI-module and the BlockDevice module.

Figure 9 illustrates the changes conducted for this release on the product line architecture. As can be seen, the actual file system framework architecture remained unchanged in this release, mainly because the volatile parts have been broken out to separate frameworks in this generation. As identified among the general deltas of this release, the implementation of FAT-16 was added to the collection of concrete file system implementations, and the MUPP-fs was removed.

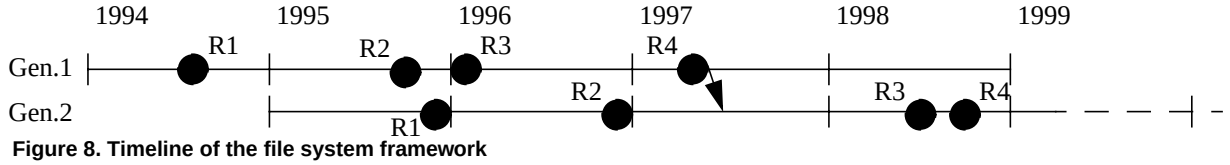

Figure 8. Timeline of the file system framework

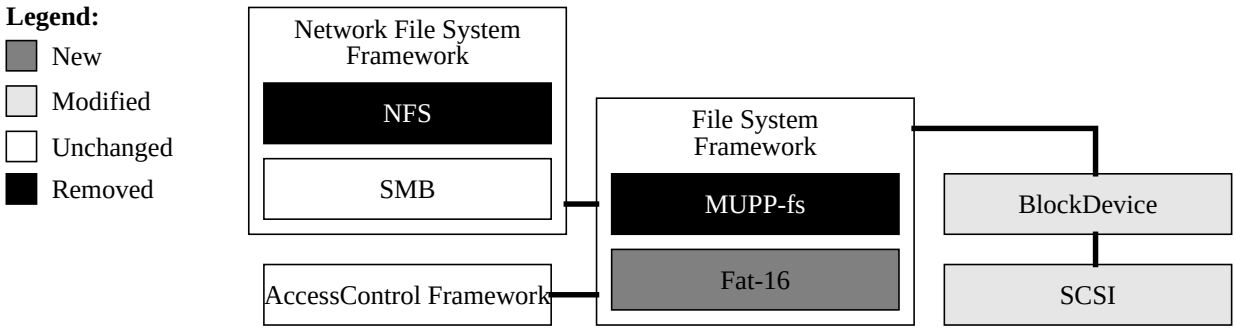

Figure 9. Changes in generation two, release 2

The product intended to use this release was a JAZ-server, and this is what caused the changes to the SCSI and the BlockDevice-module. Apparently, JAZ-drives uses some not so common SCSI-dialect, and the SCSI-driver needed to be adjusted to use this. The BlockDevice module changed to use support the new commands in the SCSI-driver.

Generation two, Release 3

The project requirements for release 3 were to develop a hard disk server with backup and RAID support. This, in itself, would not yield a new version of the product-line architecture, but in order to support the tape station, a new file system implementation had to be added. Furthermore, it was decided to support the I-node based file system UDF as well as the FAT-16 system from the previous release.

The backup was supposed to be done from the SCSI connected hard disks to a likewise SCSI connected backup tape station. The file system to use on the tapes was MTF. SMB and Netware were the supported network file systems, and the web-interface remained unchanged as did the access control management. Moreover, the product was to support an additional network management protocol, called SNMP. RAID support was delegated to RAID controllers on the SCSI-bus.

Once again, the overall product line architecture remained more or less unchanged for this release; most of the additions happened inside of the product line components, with the addition of new framework implementations. Furthermore, the BlockDevice changed name to StorageInterface, and adding Netware caused modifications in the access control framework. Figure 10 depicts the changes that took place in release 3.

This was the first time that Netware was supported in generation two, and it was taken from generation one and modified to support write functionality. Likewise, the web-interface was copied from generation one. The access control framework was modified to work with the new network file systems[1], Netware, HTTP, and SNMP.

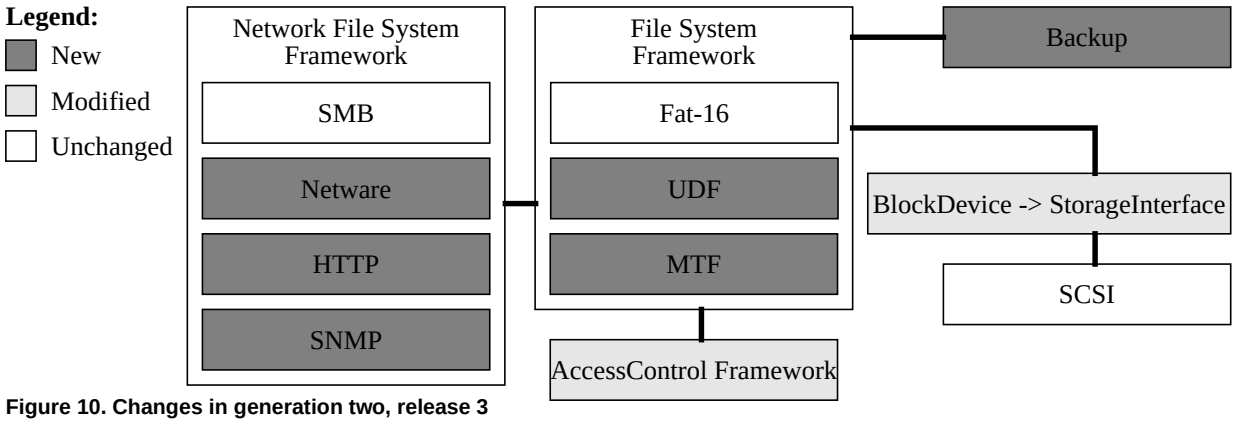

Figure 10. Changes in generation two, release 3

[1]We have consistently used the term network file systems for all the units that interface the file system component. In the actual product line, this component is called ‘storage_appl’, which is referring to that this is the application logic of all the storage products. The HTTP and SNMP interfaces should thus not be considered network file systems, but rather means of managing and monitoring the product parameters.

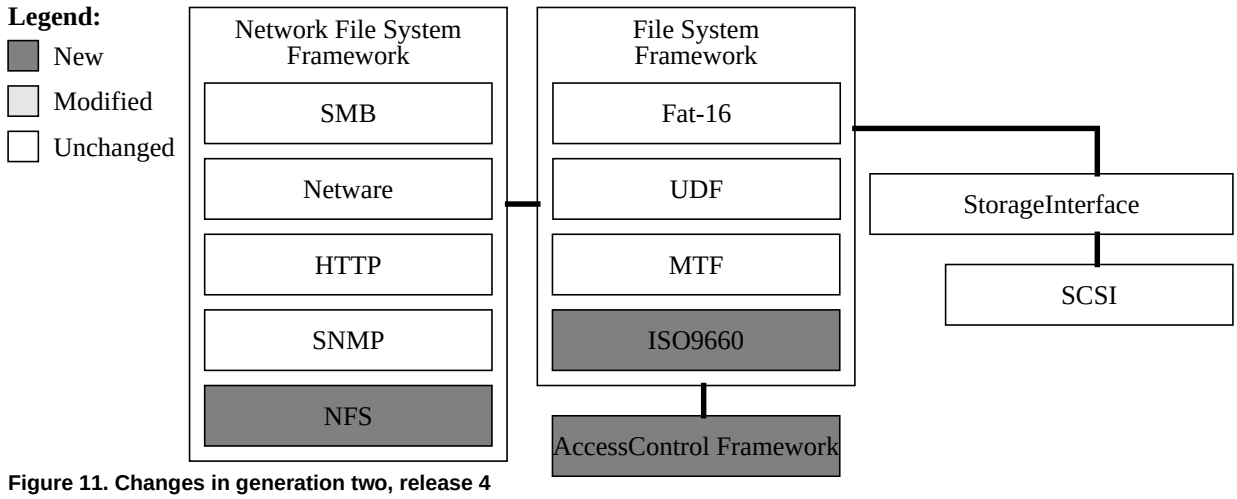

Figure 11. Changes in generation two, release 4

Generation two, Release 4

The product requirements for this release was to make a CD-server working with a CD-changer. This is the first CD-product developed using generation two of the file system framework. The requirements were inherited from the last CD-product developed using generation one, and the last project using the second generation. To these requirements some minor adjustments were made; one group of requirements dealt with how CD's should be locked and unlocked, another group concerned the physical and logical disk formats, and a third group concerned caching on various levels.

The product line architecture again survived relatively unchanged by all these requirements. An implementation of ISO9660, supporting two ways of handling long filenames (Rockridge and Joliet) was developed under the file system component, and the NFS protocol was reintroduced to the set of network file systems. As we can see, there are once again additions of framework implementations, but the product line architecture does not change in itself. Figure 11 depicts the changes made in release 4.

As can be seen in Figure 11, the AccessControl framework was not only modified to work with Netware, it was completely rewritten, and the old version discarded. The version existing in release two was a quick fix that couldn't support the required functionality.

4 Categorization

Evolution in product-line architectures is a complex and difficult to manage issue. It is important to increase our understanding of PLA evolution to improve the way we incorporate new requirements in the product-line architecture and in its components. In this section, we present categorizations of the evolution of requirements, the product-line architecture and the PLA components. Each category is presented by a name, a short explanation and an example taken from the case. Finally, based on the categorizations, we present a first version of an overall taxonomy of product-line architecture evolution.

Categories of Requirements

In this section, we present six categories of requirements that we found evidence of in the studied product line. To some extent these categories relate to the three maintenance categories adaptive, perfective, and corrective, but, as can be expected since they are requirements and not activities, the mapping is not one-to-one. In particular, category three below can be related to all three maintenance categories, and category four and five can be considered both perfective and adaptive maintenance.

1. **Construct new product family:** Requirements of this kind comes when there is a need on the market for a new type of product. This generally leads to creation of new business units, new product lines, etc.

   This has happened several times at the company of our study. There are business units managing print servers, others managing storage servers, and others again managing camera servers. Each of these business units are created out of the requirement to construct a new product family.
2. **Add product to product line:** Once you have a product line in place, most of the work goes in to improving the functionality, and to customize the support given to the end users. This is done by adding products to the product line that are tailored for certain needs.

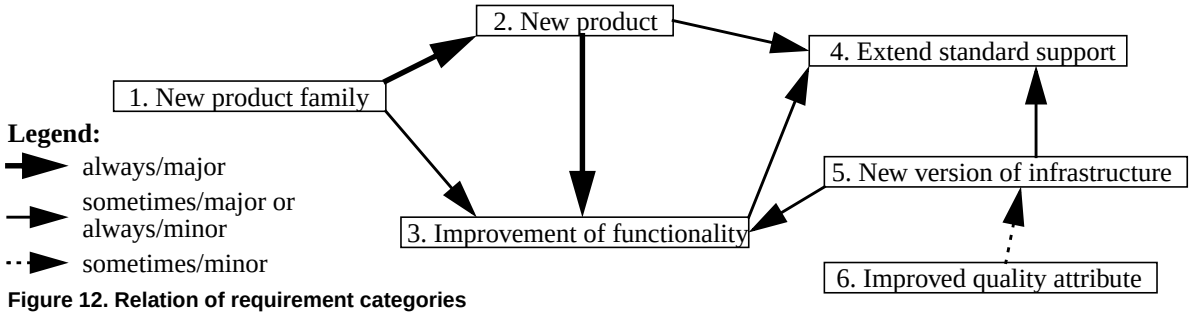

Figure 12. Relation of requirement categories

As described in the case above, it was at one time decided to create a CD-ROM server supporting TokenRing instead of Ethernet. This is a typical example of customizing the product line to the needs of the market.

3. **Improve existing functionality:** It is not enough to only create new products that fill a niche in the market, you must also improve the products you have. This features supporting new standards, adding user-requested features, etc.

   This is an ongoing activity in a product line. For example, in the case study above, there are constantly requirements of supporting new network file system standards, such as the NFS, SMB, and Netware protocols.
4. **Extend standard support:** Standards tend to be rather elaborate and, typically, only part of the standard is implemented in new products. A typical requirement for subsequent versions is to incorporate additional parts of standards.

   Related to the case, requirements fall into this category when they result in improvements of a particular framework implementation. So does, for example, the incorporation of long filenames into the ISO9660-module, imply an extension of the aspects covered of the implementation. Another example can be found in release two of generation one, where it was required to implement some extensions to the SMB protocol.
5. **New version of hardware, operating system, or third party component covers more functionality:** The above categories are all examples of that the software system boundaries expand to support more of the tasks that users expect of them [ML 94]. Naturally this is also true on the levels below your system. Should a new version of the layers of functionality under your product line architecture appear that supports more of the functionality you use, you will naturally try to use these lower-level libraries, resulting in functionality being removed from the product-line architecture components.

   The company in our study develops its own hardware, and have an in-house developed CPU. As new versions of this CPU emerge, the lower layers of the product line architecture often find that part of their functionality has been embedded in subsequent versions of the CPU. Naturally, this leads to rewrites of the concerned modules to use the hardware-supported functionality rather than continuing to support the same functionality in software.
6. **Improve quality attribute of framework/framework implementation:** First versions of products often focus on providing the required functionality and, consequently, spend less effort on the quality attributes of the system. As a result, the performance, reliability and maintainability of the product may not be optimal.

   Although not explicitly described in section 3, several efforts for improving quality attributes took place at the case company. For instance, breaking out the access control functionality from the file system framework was performed to improve maintainability and, thus, simplify future evolution. At other occasions, performance improving changes were made.

If we relate these categories to each other, we see that the establishment of a new product family (category 1) always leads to the addition of at least one product into a product line (category 2). Indirectly, this implies that a new product family improves on the set of existing functionality by adding support for new things (category 3). Likewise, adding a new product always gives rise to improvements on existing functionality, but may also include additional aspects of a standard (category 4). Furthermore, improvements of existing functionality may require incorporation of additional aspects, as may new versions of the supporting infrastructure (category 5). Improving quality attributes (category 6), finally, is somewhat on a different level, but can in some cases put requirements on the underlying infrastructure. Figure 12 illustrates the relations between the requirement categories.

Categories of Product Line Architecture Evolution

New requirements that should be incorporated for all products have an impact on the product-line architecture. We have identified eight categories of product-line architecture evolution that may occur. The fist and second category are

related in that both are concerned with creating a new product-line architecture, but describe alternative ways to achieving this.

1. **Split of product line architecture:** When it is decided that a new set of product should be developed, a decision must be taken of whether it can be kept in the same product line, or whether it is necessary to split the product line architecture into two, meaning that one uses the existing product line as a template for creating another. The product families, and hence the product line architecture, then evolve into different directions, but may keep similar traits.

   At the studied company, this type of product line split has occurred once. The company was originally built up around its print server product, and expanded into the storage area much later. The seed for the storage product line architecture was taken from the print server product line architecture. This can still be seen in some places, even if there is ongoing work to make all the original c-code that came from the print server into object-oriented C++ code that can more easily fit with the rest of the storage product line architecture.
2. **Derived product line architecture:** A second approach to incorporating a new set of products is to define a derived product-line, analogous to a subclass in object-oriented programming. Instead of copying the product line and continue evolution of both product-lines independently, the product line is branched of to a sub-product line. This way, functionality that is common for the complete product-line is still shared, and other products can directly benefit from bug fixes or other improvements.

   Deriving a product line architecture is nowadays the desired form when creating product families at the company in our study. One example of this is when the read-write file system framework was developed and used in a number of products while still keeping the CD-ROM server separate using the first generation of the file system framework. At this stage, one set of products in the product line architecture used generation one of the file system framework, and the other products used generation two. In figure 2, the hierarchical organization of the product-lines is presented.
3. **New product line architecture component:** Some requirements are such that they cannot easily be solved in any of the existing product line architecture components. It may, for example, be a large block of functionality that does not logically fit into any single framework implementation, but is rather used by all of them. In that case, the product-line architecture needs to be extended with a new component and the relations between the components need to be adjusted accordingly.

   On the third release of generation two of the storage product line architecture, there was a requirement to support backup utilities. Backup is called by several network management protocols, and in turn uses at least two of the file system framework implementations. This lead to the addition of a backup component in the product line architecture.
4. **Changed product line architecture component:** This is perhaps the most common category. Every time a new implementation is added or changed, it is an instance of this category. We will discuss this category in further detail in the next section.

   As an example, every time a new file system protocol is added, there are changes to the file system component. When a file system implementation is modified, the component as a whole is changed as well. Adding ISO9660 and adding support for long filenames in the ISO9660 module are examples of this category.
5. **Split product line architecture component:** Another way of creating new components in the product line architecture, as opposed to simply introducing new components, is by splitting out functionality from one framework into a separate unit. Arguments for splitting a component in two may simply to get rid of some cumbersome configuration management problems, but often the evolving component covers more than one logical type of functionality, requiring a split even from a conceptual point of view.

   Looking at the two generations of the file system framework, we see that in the first generation access control was part of the framework, which meant that it had to be implemented for each framework implementation. In generation two, the access control functionality have been broken out into a separate framework, and this framework is generic for all the file system implementations. Access control is conceptually different from file system functionality and one can thus conclude that this was a logical step, at the point where sufficient access control functionality was present in the system.
6. **Replaced product line architecture component:** In some cases it is not sufficient to merely modify a framework's interface to incorporate new functionality, it may be necessary to rewrite the component from scratch. This leads to the discardment of the old component, and the introduction of a new component.

   The developers of the file system framework was well aware that the first generation would never be anything more than a read-only file system framework. To make it into a read-write file system would simply cost too much, and it would not have the required performance. It was decided to develop a new generation, one that supported

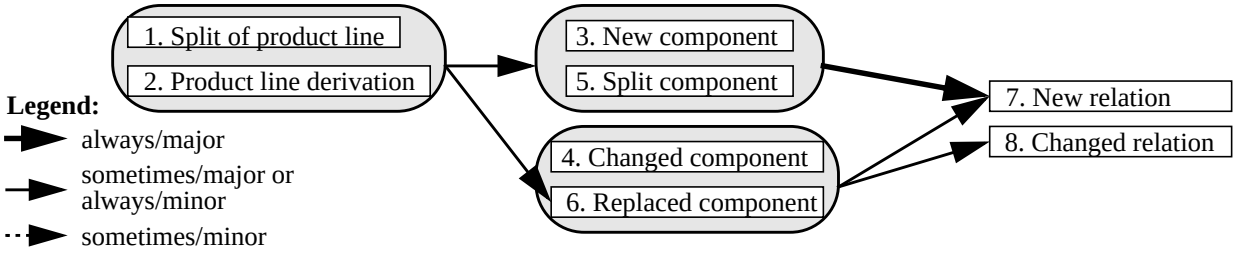

Figure 13. Relation of Product Line Architecture categories

both read and write functionality, and to plug this component into the product line architecture instead. Another example is the access control framework in the second generation, which was replaced in release 4, because it would cost too much to adjust the existing version to future needs.

7. **New relation between components:** Since the components in the architecture are related in various ways, it is natural to be able to connect previously unrelated components. New requirements may require one component to inquire at another component, consequently leading to a new relation between the components. Particularly obvious is this when you introduce a new component, since this had better communicate with some of the previously existing components.

   There are many examples in our case of new relations being added as a result of the architecture being extended with a new component. For example, the backup component was connected to the file system framework. Another example is the breaking out of the AccessControl framework, and the consequent relation to the file system framework. One should, however, avoid adding relations since this increase the coupling and hinders reuse of individual components. Due to this fact, there are no examples of new relations being added between previously existing components.
8. **Changed relation between components:** This category basically includes two forms of changes. The first is that the provides-interface of the called framework changes, and this alters the relation logically, if not visibly. The other is the simple removal of a relation, meaning that two previously connected components become decoupled.

   Although no example of this category was explicitly presented in the case description, changed relations between components typically occur. For instance, the introduction of the token ring CD-ROM server caused the network component to be changed from ethernet to token ring, which lead to adjustments in the network protocol framework. As a result, the relation between the network module and the network protocol module changed.

Figure 13 illustrates the relations between the different categories of product-line architecture evolution. The shaded boxes in the figure signifies that the categories enclosed have some common traits. For example, the first and second category are both concerned with the creation of a new product line architecture, be it totally cut off or merely a branch of the main product line architecture. Likewise, category three and five are both concerned with the creation of new elements in the architecture. Both of these lead to new relations (category 7). A replaced component (category 6) may lead to a change in the relations (category 8), such as when the second generation of the file system provided write functionality as well as read functionality, but the interface can also be left unchanged, as happened when the access control framework was replaced. A changed component (category 4), finally, can lead to changed relations, or even new relations.

Categories of Product Line Architecture Component Evolution

In the previous section we discussed how the product line architecture can change. In our experience, category four, i.e. changes to a product-line architecture component, is the most common type of evolution. In this section, we discuss the different types of evolution that a component can be subject to.

1. **New framework implementation:** Given a framework architecture, the traditional way of extending the family of supported functionality is by adding implementations of the framework. This is also the evolution presented by Roberts and Johnson [RJ 96], where concrete implementations are added to the library of implementations.

   There are many examples of this, for example adding the FAT-16 file system to the set of supported file systems. Nothing in the framework architecture changes, and the addition is, in the best case, transparent to the calling components.
2. **Changed framework implementation:** Evolution of this type may occur when new functionality is added, or when existing functionality is rewritten to, for example, better support some quality attribute. This category basically has the same effects as the first one, but on a framework implementation level.

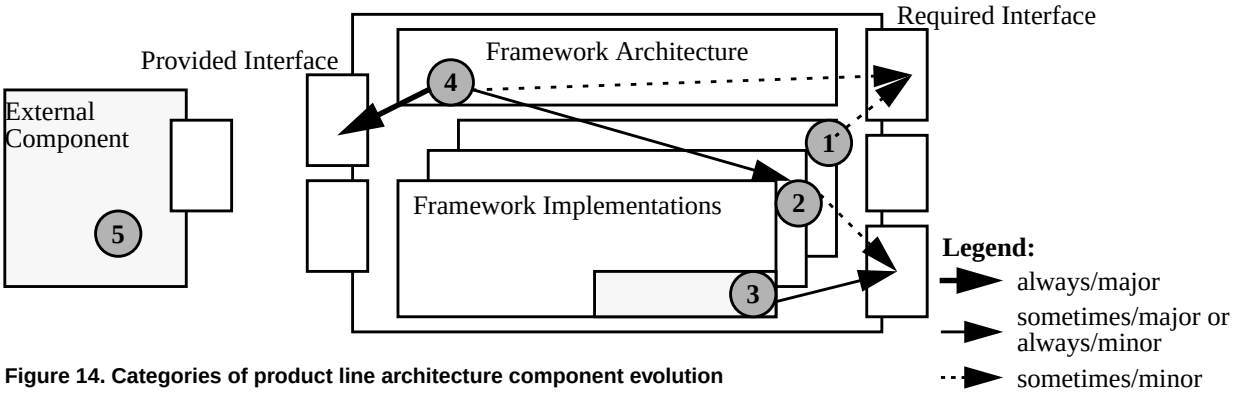

Figure 14. Categories of product line architecture component evolution

Adding support for long filenames in the ISO9660 implementation is one example of a changed framework implementation. When the name cache was removed from the Netware module in generation one is another example. Interesting about this example is that removing the name cache changed the 'required' interface, which meant that an internal change in the ISO9660 file system needed to be done to change the 'provided' interface accordingly.

3. **Decreased functionality in framework implementation:** As a direct consequence of the requirement category concerning new versions of hardware, operating systems and other third party components, the functionality in a framework implementation can decrease, since part of its functionality is now handled elsewhere.

   An example of this is when a new version of the CPU came that supported SCSI on-chip, the SCSI-driver was rewritten to merely act as a hardware interface wherever this was possible. The functionality of the SCSI-software driver decreased (internally), since it could now leave much of its previous functionality to the hardware instead.
4. **Increased framework functionality:** This may seem similar to the first category, but they are not quite on the same level. Whereas category one adds a new implementation that supports the same functionality as its peers, this category adds functionality to all of the framework implementations. This occurs when there is a framework gap [MB], i.e. that the desired functionality is not covered by any of the composed frameworks.

   Adding the network file system protocol NDS required a new strategy for managing access rights. The Access Control framework was extended accordingly, so that all network protocols can use this new strategy. This is an example of increased functionality of the framework, in that the service level provided by the framework was enhanced.
5. **Add external component to add functionality with otherwise framework architectural impact:** This is an interesting category, which we have found evidence of not only in our case, but also in other companies. As a project begins to draw towards its end, there is a growing reluctance to go in and meddle with the underlying frameworks, since this yields more tests, and a higher risk for introducing bugs that have impact on a large part of the system. Then a new requirement is added (or an existing requirement is reinterpreted), leading to changes with architectural impact, if implemented normally. To avoid the excessive amounts of effort required to incorporate the architectural changes, one instead develops a solution that implements the requirement outside of the framework. This has both the benefit of localizing possible problems to a particular module, as well as not being forced to late in a project modify all of the calling components. However, being 'a hack' it does, of course, violate the conceptual integrity [Brooks 95] of the component architecture.

   The example of this is when support for Netware was added to the network file systems. Late in the project exactly this situation occurred, when it was discovered how Netware handles file names in relation to their id's, and that this differed from the way that NFS and SMB worked. Instead of changing this in the ISO9660 module, where the root of the problem lay, a name cache was added into the Netware module, that took care of the problem. In a subsequent release, the name cache was removed and the problem was solved in the ISO9660 module. This had impact on the NFS and SMB protocols, because they needed to incorporate the change of the interface.

If we look at a product line architecture component, we see that adding a framework implementation (category 1) may have impact on the required interface, as can a decrease of functionality (category 3). The former may add requirements on the required interface, and the latter may remove requirements on this interface. Likewise, to change a framework implementation (category 2) can have impact on the required interface, but also on the provided interface. An increase of the framework's functionality (category 4) will have impact on the provided interface, and on the framework implementations. Figure 14 depicts how the product line architecture component evolution categories relate to the component itself.

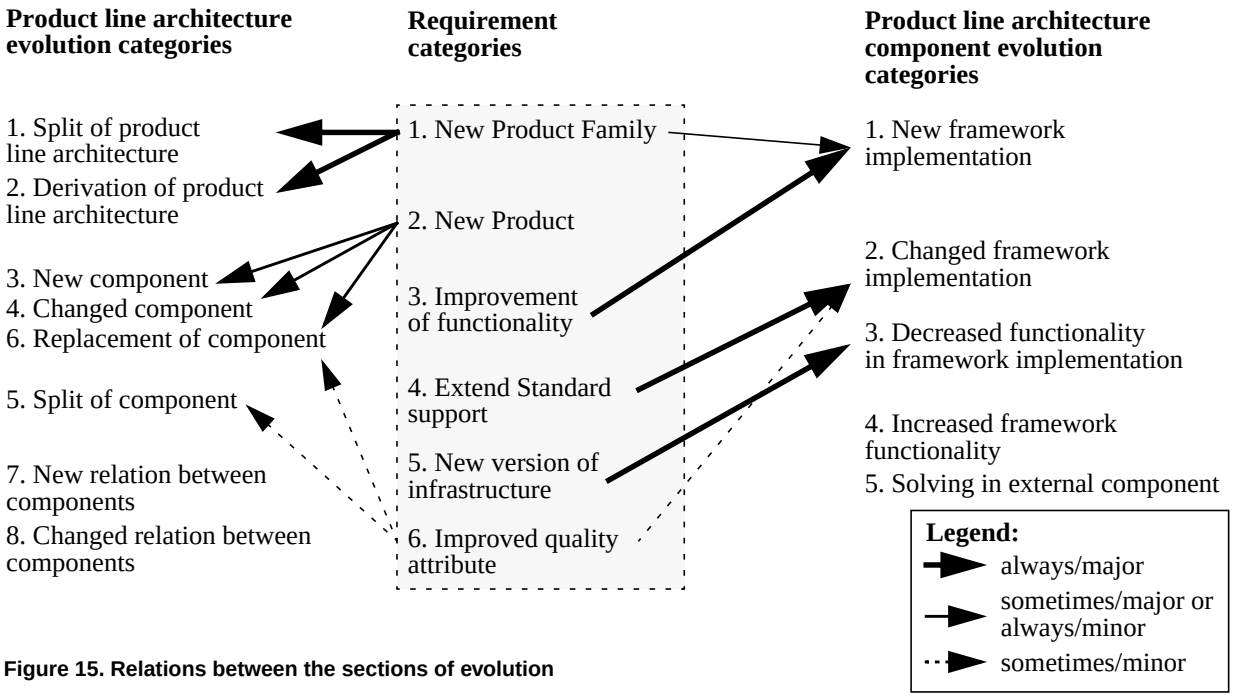

Figure 15. Relations between the sections of evolution

Relation of evolution categorizations

In the previous sections, we have presented categorizations for the evolution of requirements, the product-line architecture and product-line architecture components. However, these are not independent, but can be related to each other. We have already related categories within their logical section, to show how requirements of one type relates to requirements of another, and how a certain type of changes to the product line architecture naturally leads to another. In this section we relate the categories across the sections, to see how a certain type of requirement results in a subset of the product line architecture evolution categories. Figure 15 shows how the requirement categories lead to different types of evolution both on the product line architecture and on the product line architecture components.

As can be seen, many of the evolution types are not directly caused by requirements, but are rather indirect effects of the requirements, in that one change triggers another change of a different type. This second change may be on the same level, so that one change on the product line architecture triggers another change on the product line architecture. It may also be of the type that it causes a change on another level. For example, a new framework implementation may cause a change in another framework's implementation, which in turn may change the relation between the two concerned components.

5 Related work

Most categorizations of requirements found are rather high-level. For example, Tracz discusses reference requirements [Tracz 95] on domain-specific software architectures. Reference requirements consists of the set of functional requirements, the non-functional requirements, the design requirements and the implementation requirements. Prieto-Diaz simply divides requirements into the categories of stable requirements, i.e. those who are invariant over a set of applications, and variable requirements, meaning those that differ or may differ between products [PD 91].

With respect to characterizing evolution, some related work exists. Siy & Perry describes the management of a large product line architecture [SP 98], but their focus is primarily on configuration management rather than on the requirement and framework evolution. [Lehman 88] states a number of laws concerning evolution of software, but these are too fundamental, and also too high-level to be applicable to our work.

Basili *et al.* presents a case study that charts the evolution of a software system over a number of releases [BBCKMV 96]. The focus of this study is on the activities performed during maintenance, and the effort put into the activities. The study gives understanding of how and why effort is spent on subsequent releases, and also helps in process improvement issues.

A paper by Mattson and Bosch describes how frameworks evolve, and the types of changes that a framework can become subject to [MB98]. They state that a framework can undergo (a) internal reorganization, (b) change in functionality, (c) extension of functionality, (d) reducement of functionality.

6 Conclusions

Product-line architectures present an important approach to increasing software reuse and reducing development cost by sharing an architecture and set of reusable components among a family of products. However, evolution in product-line architectures is more complex than in traditional software development since new, possibly conflicting, requirements originate from the existing products in the product-line and new products that are to be incorporated.

Although evolution in product-line architectures is a complex and difficult to manage process, it is not studied much by the research community. To address this, we have performed a case study of a product-line architecture at a swedish company selling a wide range of printer, storage, scanner and camera server products worldwide. The company has employed product-line architecture based software development since the beginning of the 1990s and uses object-oriented frameworks as the components in the product-line. We study one product-line architecture component in detail, i.e. a filesystem framework component. This component has evolved through two generations of four releases each and contains many relevant examples. For each release we present the requirements, the impact this had on the product line architecture as a whole, and what the effects were on the studied component

From this we have constructed a set of categories concerning the product line requirements, the product line architecture evolution, and the product line architecture component evolution. These categories have, after being presented and exemplified, been related to each other.

Based on the case study, we present categorizations of the evolution of the requirements, the product-line architecture and the product-line architecture components. Using the categorizations, we relate the three levels of evolution to each other. The result is a first version of a taxonomy of product-line architecture evolution that, we hope, improves understanding of the evolution process and the relations between the types of evolution at the different levels.

By creating this taxonomy, we hope to have improved understanding of how a framework inside a product line architecture evolve as a consequence of direct requirements on the framework, and indirect requirements on other parts of the product line architecture. This improved understanding helps, we believe, in being able to on forehand understand the impact of a new requirement. This, in turn, helps us to avoid an overly rapid aging product line architecture, i.e. architecture erosion.

We intend to continue working on the taxonomy by conducting more case studies to confirm the findings, and to further investigate the relations between the categories. In so doing, we hope to achieve a prediction model presenting the changes one can expect as a consequence of a particular requirement.

References

[BBCKMV 96] Basili, V., Briand, L., Condon, S., Kim, Y.M., Melo, W.L., Valett, J., *"Understanding and predicting the Process of Software Maintenance Releases"*, Proc. of the 18th Int'l Conf. on Software Engineering, Berlin, Germany, IEEE, March 1996, pp. 464-474.

[Brooks 95] Brooks, F.P., *The Mythical Man Month* (anniversary edition), Addison-Wesley, 1995.

[Bosch 98a] Bosch, J., *"Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study"*, Accepted for the First Working IFIP Conference on Software Architecture, October 1998.

[Bosch 98b] Bosch, J., *"Product-Line Architectures in Industry: A Case Study"*, Accepted for the 21st International Conference on Software Engineering, November 1998.

[Lehman 88] Lehman, M.M., *"On understanding laws, evolution and conservation in the large program life cycle"*, Journal of Systems and Software, 1(3):213-221, 1980.

[ML 94] Marciniak, J.L. (Editor), *"Encyclopedia of Software Engineering"*, Lehman, M.M., on "Software Evolution", John Wiley & Sons, 1994.

[MB] Mattsson, M., Bosch, J., "Framework Composition: Problems, Causes and Solutions", In *"Building Application Frameworks: Object Oriented Foundations of Framework Design"* Eds: M.E. Fayad, D.C. Schmidt, R. E. Johnson, Wiley & Sons, ISBN#: 0-471-24875-4, Forthcoming.

[MB98] Mattsson, M., Bosch, J., *"Frameworks as Components: A Classification of Framework Evolution"*, Proceedings of NWPER´98 Nordic Workshop on Programming Environment Research, Ronneby, Sweden, August 1998, pp. 163-174

[PD 91] Prieto-Diaz, R., *"Reuse Library Process Model"*, Technical Report AD-B157091, IBM CDRL 03041-002, STARS, July 1991.

[RJ 96] Roberts, D., Johnson, R.E., *"Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks"*, Proceedings of PLoP - 3, 1996.

[SP 98] Siy, H.P., Perry, D.E. *"Challenges in Evolving a Large Scale Software Product"*. Principles of Software Evolution Workshop. 1998 International Software Engineering Conference (ICSE98), Kyoto Japan, April 1998.

[Tracz 95] Tracz, W., *"DSSA (domain-specific software architecture): pedagogical example"*, SIGSOFT Software-Engineering Notes. vol.20, no.3; July 1995; p.49-62, 1995.