Research Report 4/00

HÖGSKOLAN KARLSKRONA RONNEBY

A Study of Evolution Impact in Software Product Lines

by

Mikael Svahnberg, Jan Bosch

Department of
Software Engineering and Computer Science
University of Karlskrona/Ronneby
S-372 25 Ronneby
Sweden

ISSN 1103-1581
ISRN HK/R-RES—00/4—SE

A Study of Evolution Impact in Software Product Lines

by Mikael Svahnberg, Jan Bosch

ISSN 1103-1581
ISRN HK/R-RES—00/4—SE

Printed by Psilander Grafiska, Karlskrona 2000

A Study of Evolution Impact in Software Product Lines

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.ipd.hk-r.se/[msv/ | bosch/]

Abstract
Product-line architectures, i.e. a software architecture and component set shared by a family of products, represents a promising approach to achieving reuse of software. Several companies are initiating or have recently adopted a product-line architecture. However, little experience is available with respect to the evolution of the products, the software components and the software architecture. Due to the higher level of interdependency between the various software assets, software evolution is a more complex process. In this paper, we discuss the results of two case studies concentrating on the evolution of software assets in two swedish organizations that have employed the product-line architecture approach for several years. Based on these two cases, we discuss the commonalities, presented as categorizations of the evolution of the requirements, the software architecture and the software components, and also the differences between the two cases.

1 Introduction

Wide-scale reuse of software has been a long standing ambition of the software engineering industry. The cost and quality of much of the currently developed software is far from satisfactory. In the situation where a software development organization markets a family of products with overlapping, but not identical functionality, the notion of a software product-line is a feasible approach to decrease cost and increase the quality of software. The software product-line defines a software architecture shared by the products and a set of reusable components that, combined, make up a considerable part of the functionality of the products. Especially in the swedish industry, this is a logical development since software is an increasingly large part of products that traditionally were considered to be primarily mechanical or electronic. In addition, software often defines the competitive advantage. When moving from a marginal to a major part of products, the required effort for software development also becomes a major issue and industry explores alternatives to increase reuse of existing software to minimize product-specific development and to increase the quality of software.

A number of authors have reported on industrial experiences with product-line architectures. In [Bass et al. 97] and [Bass et al. 98a], results from two workshops on product line architectures are presented. Also, [Macala et al. 97] and [Dikel et al. 97] describe experiences from using product-line architectures in an industrial context. The aforementioned work reports primarily from large, american software companies, often defence-related, but also in Europe considerable effort has been invested in investigating software product lines. Most notably, the 4th framework EU projects ARES [Linden 98] and PRAISE involving several major european industrial companies.

Although the notion of software products lines is studied by several authors, the primary effort is directed towards the conversion towards and the initiation of a software product line and, consequently, its evolution is studied considerably less. In this paper, we address this by presenting the results of two case studies of software product line evolution. Two software assets are studied, i.e. an object-oriented framework for file-systems, which is one of the primary components in the software product-line of Axis Communications AB, and the Billing Gateway software product line, one of the flagship product families of Ericsson Software Technology. For each asset, we have studied the evolution over several versions, i.e. eight and four releases respectively, and investigated what changes were made to the requirements, to the software architecture and to the components and their implementation. Based on the cases, we discuss the commonalities, presented as categories of evolution for each of these aspects, i.e. requirements, architecture and components, and the differences between the two cases.

The remainder of this paper is organized as follows. In the next section, the notion of software product-line evolution is discussed in more detail and definitions for some concepts are presented. Section 3 discusses the research method that was used and its concrete application to this study. The cases are presented in section 4 and 5, respectively. The cases are compared and discussed in section 6. The paper is concluded in section 7.

2 Software Product Line Evolution

A software product line, as any piece of software evolves over time. The process can be viewed from an organizational and from a process perspective. The evolution of a software product line is driven by changes in the requirements on the products in the family. These new and changed requirements originate from a number of sources, such as the customers using the products, future needs predicted by the company and the introduction of new products into the product line.

In Figure 1 an overview of the evolution process is presented. Each business unit is responsible for one product or a small set of highly related products and is interested in supporting a set of requirements. These requirements are divided into the requirements specific for the product that this business unit is responsible for and the requirements common for all or most products in the software product line. The requirements on a particular product are naturally implemented in the product-specific code, but the common requirements are supported by the software product line architecture and component set. Changes to the latter category, i.e. the product-line requirements, can affect the software architecture of the product line, the architecture of a particular component, thus creating a change in its interface, or it can cause a change in one or more of the concrete implementations of the component. 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 software product-line architecture and its component set is subsequently used to construct new releases or versions of the products in the family. These new versions of the products lead to new requirements, which are fed back into the business unit requirements, thus completing the cycle.

Our view of Products, Architecture, and Software Product Lines

We define a software product line as consisting of a software product line architecture, a set of reusable components and a number of software products. The products can be organized in many ways, one example is the hierarchical organization presented in Figure 2, and the architectural variations are organized accordingly.

A software product line architecture is a standard architecture, consisting of components, connectors, and additional constraints, in conformance to the definition given by [Bass et al. 98b]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. [Bass et al. 98b]

The role of the software product line architecture is to describe the commonalities and variabilities of the products contained in the software product line and, as such, to provide a common overall structure.

A component in the architecture implements a particular domain of functionality, for example, the file system domain, or the network communication domain. Components in software product lines, in our experience, are often imple-

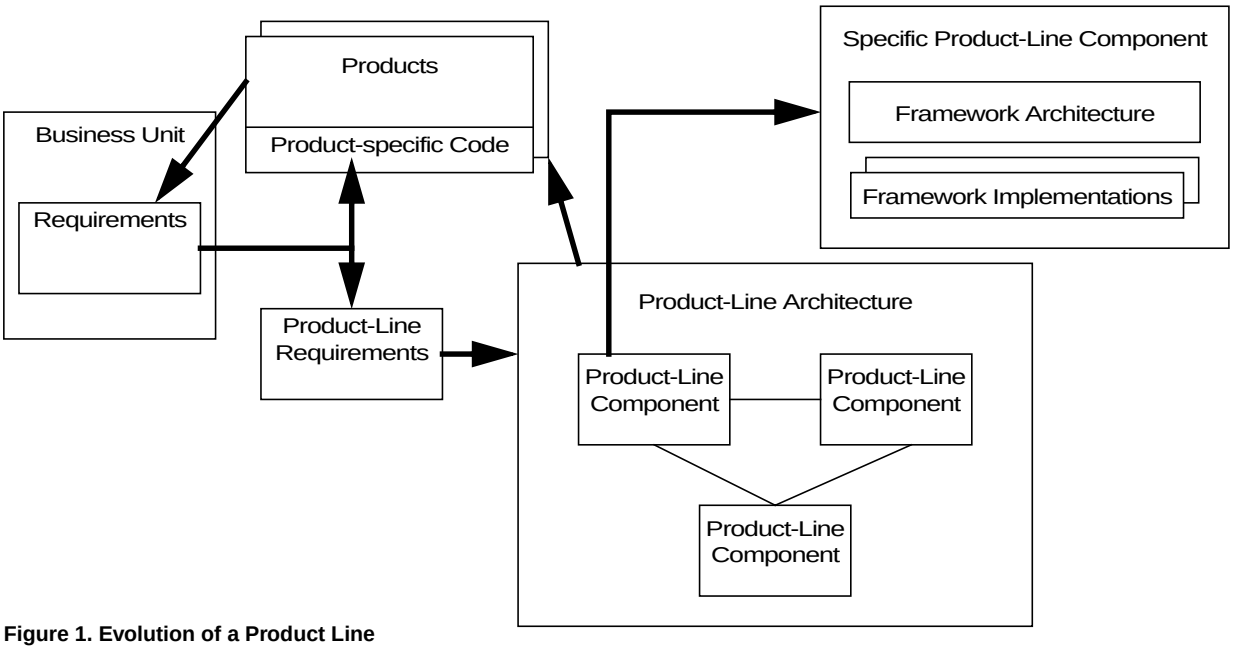

Figure 1. Evolution of a Product Line

mented as object-oriented frameworks. A product is, thus, constructed by composing the frameworks that represent the component in the architecture.

As shown in Figure 1, we define a framework to consist of a framework architecture, and one or more concrete framework implementations. As an example, one of the components used by Axis Communications, i.e. the 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. Another example is the communications component in the Billing Gateway case discussed below, where there is a common API, or framework interface that is used by all implementations of network protocols such as TCP/IP and X.25. This interpretation of a framework holds well in a comparison to [Roberts & Johnson 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.

The products in the software product line are instantiations of the product line architecture and of the components in the architecture. There may also be product-specific component extensions and, generally, product-specific code is present to capture the product-specific requirements.

3 Case Study Method

The main goal in our study was to find out how a framework evolves when it is part of a software product line. To achieve this goal, we conducted interviews with key personnel that have been involved in the studied systems for quite some time. Furthermore, we studied requirement specifications, design documents, and functional specifications that we got either from the companies intra-net, or from other sources.

Before the interviews, much time was spent on studying the available documentation, in order to get a good background understanding. The interviews were open, 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 was found in the documentation, and some further questions were later asked to the interviewed people, to further explain discrepancies and things that were unclear.

The two case sites were selected in a non-random manner. The first case, at Axis Communications, was selected because Axis is a large software company in the region, Axis is part of a government-sponsored research project on software architectures of which our university is one of the other partners, and thirdly Axis has a philosophy of openness, making it relatively easy to conduct studies within this company. The second case, at Ericsson Software Technology, was chosen because it resides close to the campus, making it easy to interact with them, and because our university has had a number of fruitful cooperations together with Ericsson in the past. Despite this “convenience sampling”, we believe that the companies are representative for a larger category of software development organizations. The companies are presented in further detail below.

4 Case 1: StorageServer

In this first case we studied a software product line at Axis Communications AB. This section present the company and the software product line. 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.

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.

Axis communications 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 software 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 assets 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 product group, there are a number of products, and below this are the variations of each product. Figure 2 illustrates the layout of the software product line and the variations under each product group.

Each product group is maintained by a business unit that is responsible for the products that come out of the product line branch, and also for 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.

The company uses a standard view on how to work with their software product line, even if their terminology might not be the same as is used by academia. A product line architecture, in Axis' eyes, consists of components and relations. The components are frameworks that contain the component functionality. We feel that this is a useful usage of the framework notion, since each component covers a particular domain and can be of considerable size, i.e. up to 100 KLoc. However, when comparing it to the traditional view [Roberts & Johnson 96], frameworks are usually used in isolation, i.e. a product or system uses only one framework. In the Axis case, a framework consists of an abstract architecture and one or more concrete implementations. Instantiation of the framework is thus made by creating a concrete implementation by inheriting from the abstract classes, and plug this into place in the product.

Further info on how the product line is maintained, and the issues concerning this is presented [Bosch 99a] and [Bosch 99b].

The studied product line

During this study, we have focused on a particular product, the storage server. This is a product initially designed 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 there is also a Jaz server and recently a HardDisk server was added to the collection of storage servers. Central for all these products is a file system framework that allows uniform access to all the supported types of storage devices. In addition to being used in the storage servers, the file system framework is also used in most other products since these products generally include a virtual file system for configuration and for accessing the hardware device that the product provides network support for. 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 developed to support read and 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 interface, various network protocols are added such as NFS (UNIX), SMB (MS-Windows), and Novell Netware. This is further illustrated in Figure 3, where it is shown 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 similar to the first generation, but the modularity of the system was improved. In addition, the framework was prepared for incorporating some likely future enhancements based on the experience from the first generation. 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

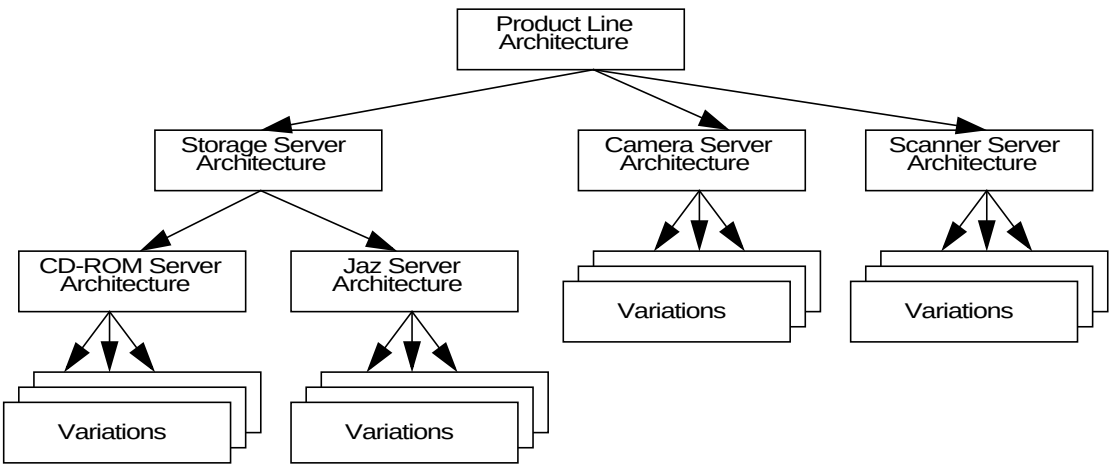

Figure 2. Product Line hierarchy

file system implementation. The second generation file system framework is depicted in Figure 4. As can be seen, the framework is split into several smaller and more specialized frameworks. Notably, the Access Control-part has been “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.

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 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.

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

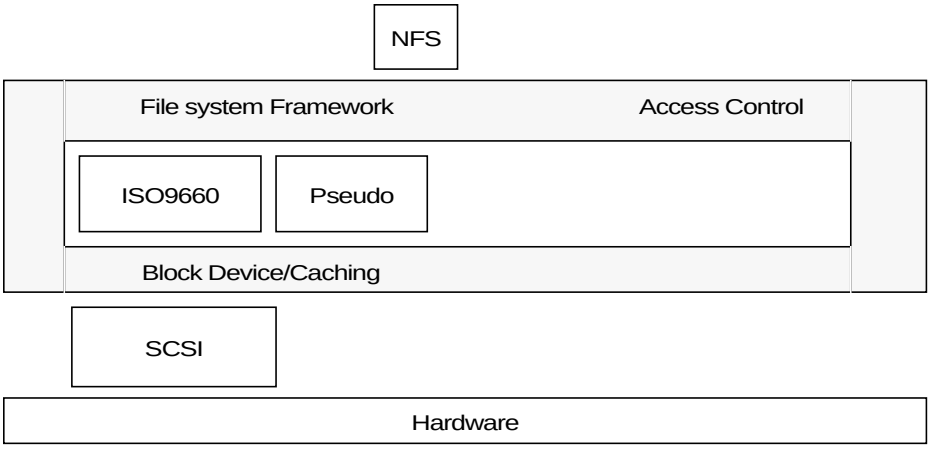

Figure 3. Generation one of the file system framework

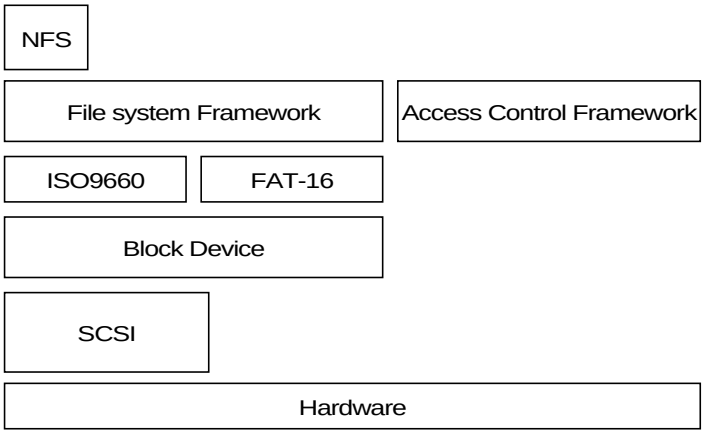

Figure 4. Generation two of the file system framework

Pseudo file system access rights could be assigned to each file. This was done with the use of the two classes NFSUser and NFSAccessRight.

Later in the project, SMB was added as well, which did not change much, since SMB is not that different from NFS. Code was added to handle access control for SMB, specifically the classes SMBUser and SMBAccessRight 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 time to do it flexible enough to support future needs.

Generation one, Release 2

The goal for 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.

This had the effect on the product line architecture that Netware was added as a concrete implementation in the network file system framework, modules for Token Ring were 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 products requirements.

The addition of the Netware protocol could have had severe implications on the file system framework, but the requirements were reformulated to avoid these problems. Instead of putting the requirements that were known to be hard to implement in the existing architecture into the framework, 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 rights 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.

Late in this project a discrepancy was found between how Netware handles file-ID's compared to NFS and SMB. 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 file system framework was primarily concerned with improving the structure of the framework and with fixing bugs. The SCSI-driver was modified to work with a 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 also 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. Figure 6 shows the modifications in release 3.

On component level, 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 interface as well, since the DirectoryRecord class had to support

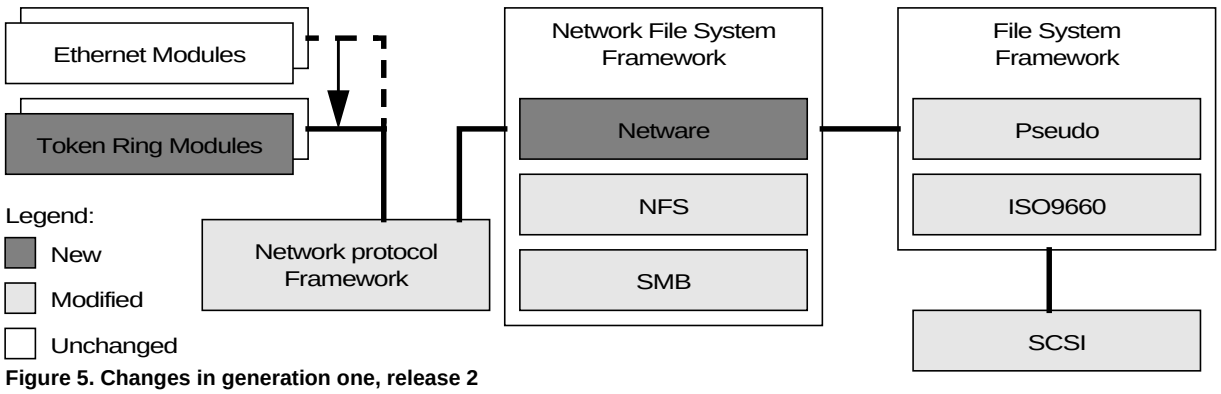

Figure 5. Changes in generation one, release 2

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 had 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.

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 development of the second gen-

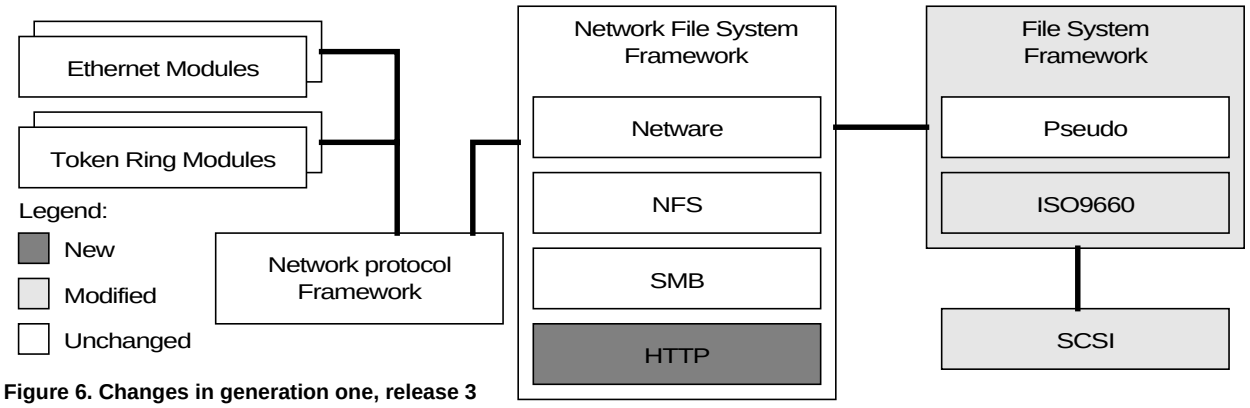

Figure 6. Changes in generation one, release 3

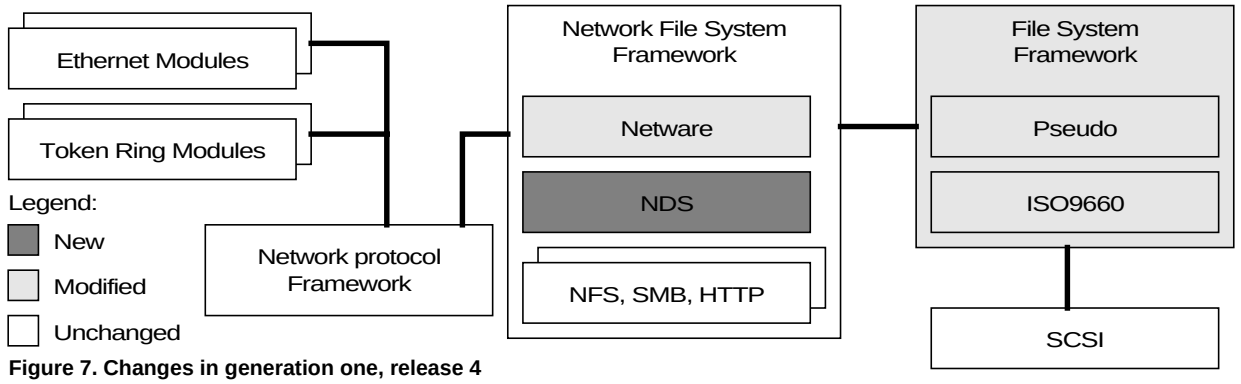

Figure 7. Changes in generation one, release 4

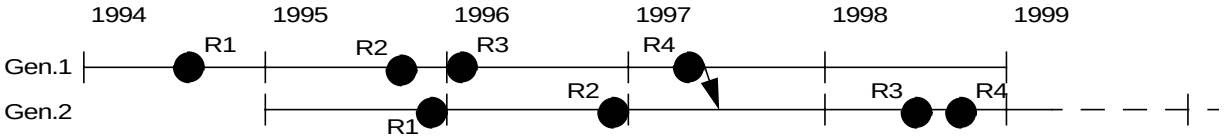

Figure 8. Timeline of the file system framework

eration, 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 had been realized early 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 first release of the second generation was much more modularized. 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. An example of this is the SMB protocol implementation.

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.

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. In order to support the backup tape station, a new file system implementation had to be added. Furthermore, it was decided to include support for 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. 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. The only major change was the addition of the Backup component, that was connected to the file system framework. 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.

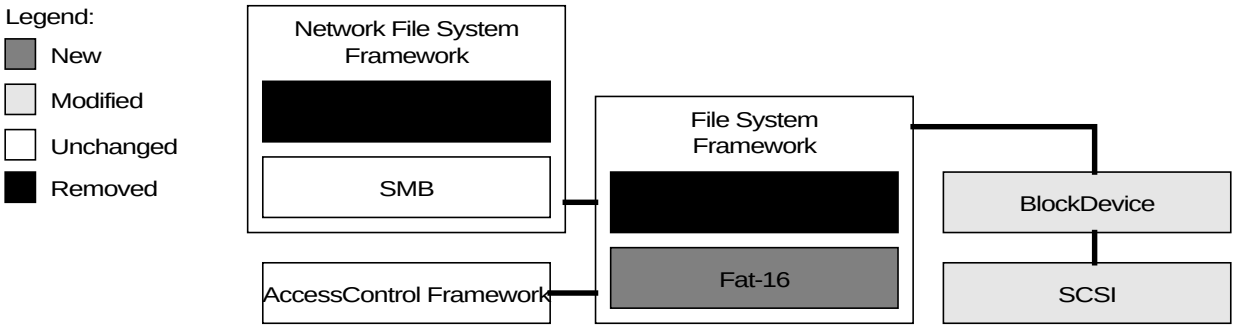

Figure 9. Changes in generation two, release 2

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

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, and 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 re-introduced to the set of network file systems. As we can see, there are, once again, additions of framework implementations, but the 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.

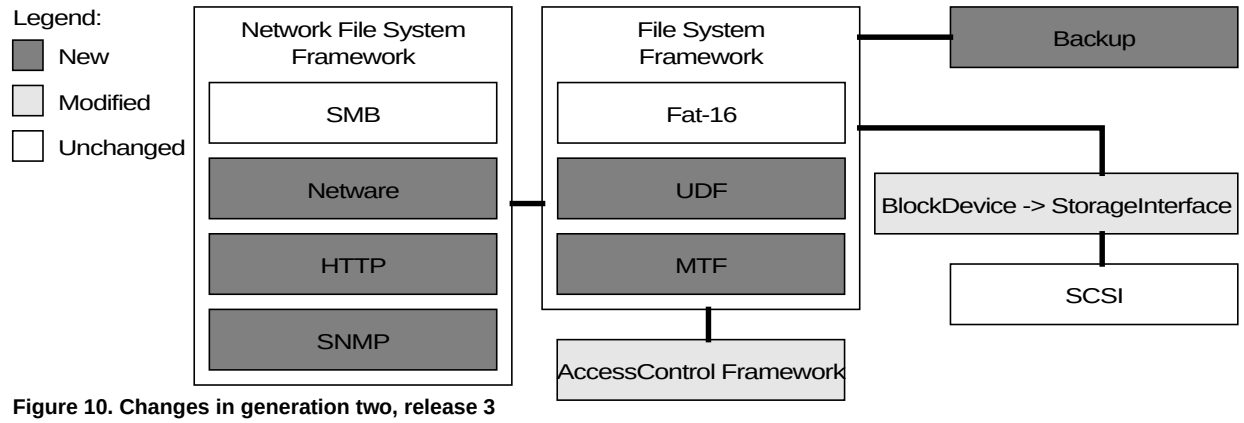

Figure 10. Changes in generation two, release 3

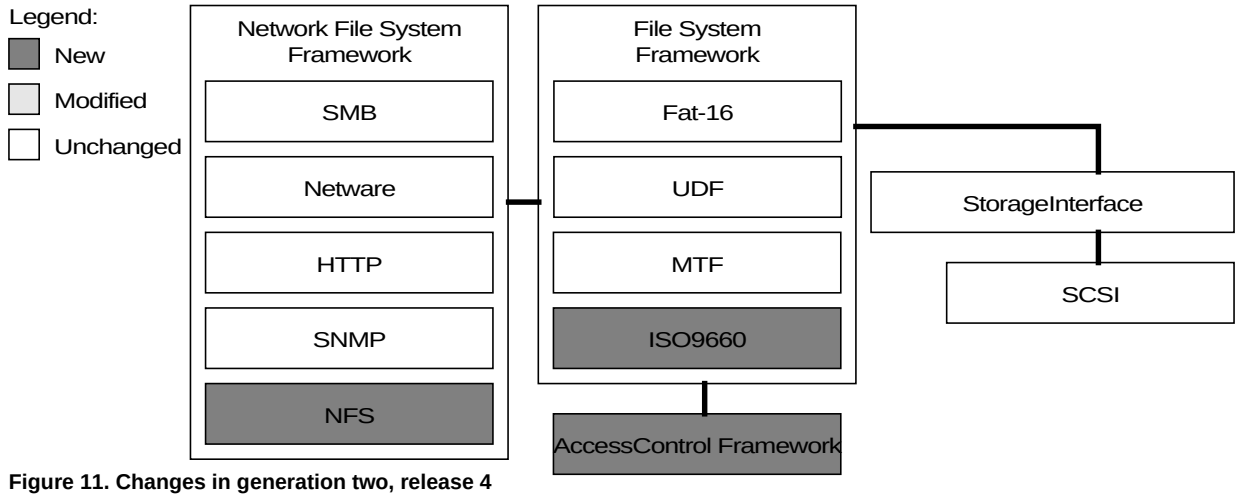

Figure 11. Changes in generation two, release 4

[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.

5 Case 2: Billing Gateway System

The second case studied in this article was Ericsson Software Technology, in particular, the software product line for the Billing Gateway system. This section presents the company, the system, and the four available releases of the system.

The Company

Ericsson Software Technology is a leading software company within the telecom industry. The company is located in southern Sweden, and employs 800 people. Our study was conducted at the site in Ronneby, which is closest to the University. This section of the company has been developing a software product line for controlling and handling billing information from telecommunication switching stations since the beginning of this decade. Whereas the software product line in Axis Communications is centered around products for the consumer market, the product line in Ericsson Software Technology is focused on developing products for large customers, so there is typically one customer for each product released from the software product line. [Mattsson & Bosch 99a], [Mattsson & Bosch 99b], and [Mattsson & Bosch 98c] discuss the company and the Billing Gateway product further.

The System

The Billing Gateway (BGw for short) is a mediating device between telephone switching stations and various post-processing systems, such as billing systems, fraud control systems, etc. The task of the Billing Gateway is thus to collect call data from the switching stations, process the data in various ways, and distribute it to the postprocessing systems.

The processing that is done inside the BGw includes formatting, i.e. reshaping the data to suit the postprocessing systems, filtering, i.e. to filter out the relevant records, splitting, i.e. to clone a record to be able to send it to more than one destination, encoding & decoding, i.e. to read and write call data records in various formats, and routing, i.e. to send them to postprocessing systems.

One of the main thoughts with the system architecture is to transfer asynchronous data from and to the network elements and the post-processing systems into a controlled, synchronous processing unit. To achieve this, the system was divided into three major parts; collection, processing, and distribution, as can be seen in Figure 13. Inside the processing node, there are four separate components, handling encoding, processing, decoding, and data abstraction, as illustrated in Figure 12.

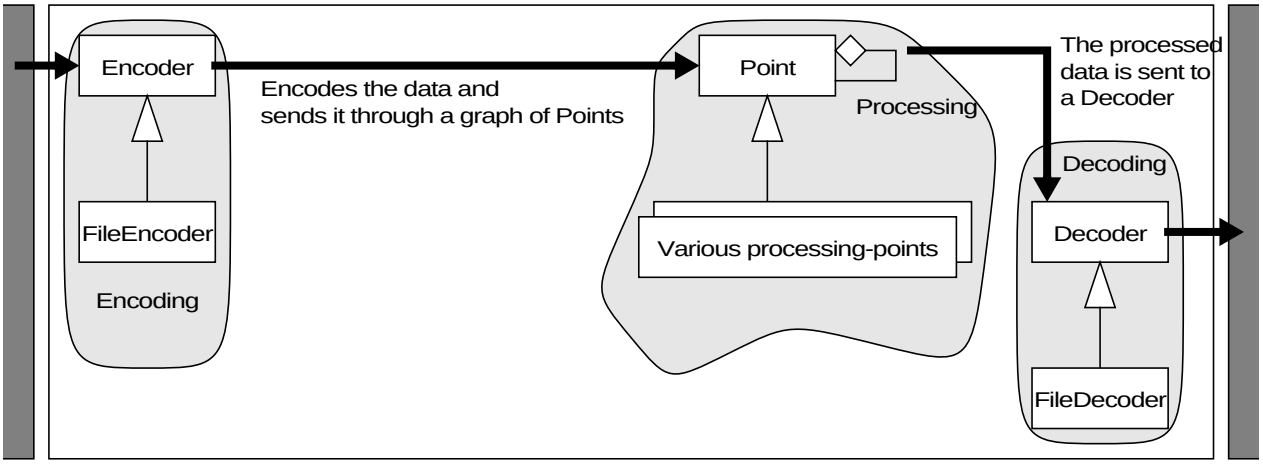

Figure 12. Architecture of the BGw Processing node

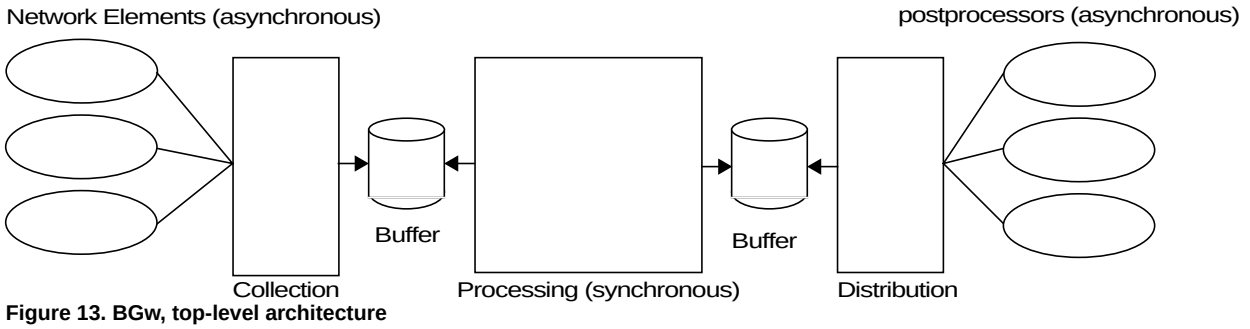

Figure 13. BGw, top-level architecture

Collection and Distribution is solved using a Communication API, CAPI. This CAPI is file-based, using the primitives connect, transfer, commit, close, and abort. The call data is put into the file buffer, where it is picked up by the processing node once this is ready to accept a new file.

Inside the processing node, the files are first encoded into an internal format which is then sent through a graph of processing points, such as formatters and filters, that each process the data in some way. Last in the graph the data is coded into an output format, after which is put into the outgoing buffer for further distribution to post-processing systems.

As opposed to the StorageServer case, Ericsson Software Technology uses the traditional framework view, in that the entire Billing Gateway product is considered a framework, that is instantiated for each customer. According to [Mattsson & Bosch 99a], over 30 instantiations of the Billing Gateway have been delivered to customers. However, to view the Billing Gateway as one framework yields, in our view, unnecessary complexity, and we find it better to consider the components in the product, and to regard these from a framework perspective, as we do in the StorageServer case. If we adjust our view accordingly, we see that the product is instantiated in much the same way as the StorageServer, in that concrete implementations are inherited down from abstract interfaces, and that they are in the running product instantiated according to each product's configuration.

Releases

The BGw has, as yet, been released in four different releases, each of which is described below. For each release, we present the requirements on the system, and the impact these requirements had on the architecture. Some of the terms used are specific for the telecom domain, and unless they are of relevance to the paper, they will not be explained further.

Release 1

Ericsson Software Technology had previously developed several systems in the Billing Gateway domain, and was interested in creating a software product-line to capitalize on the commonalities between these systems. At the next opportunity when a customer ordered such a system they took the chance and generalized this system into the first release of the Billing Gateway. The requirements on this release were, as described above, to collect call data from network elements, process this data, and distribute the data to post-processing systems. Two types of network elements were supported, communication to these was done using X.25, and communication is done using a standard language known by the switching-stations, or in other forms that some switches uses, for example a packed version of the switch language. Moreover, the BGw was required to support an interpreted language used in the processing nodes to perform conversions and other functions on the call data. Distribution of the processed data was done over NFS, FTP, or to local disk. In addition to this, a number of requirements were aimed at improving fault tolerance, performance, and scalability.

Since this release is a first, the entire product line architecture is new, as are the implemented components. Figure 12 illustrates the overall architecture inside the processing node, and Figure 14 illustrates what instantiations of the components in the architecture that were made. The protocols that were implemented were chosen from a pool of protocols planned for implementation. As is presented below, the remaining protocols were implemented in release two.

When developing release one, few concrete thoughts were dedicated towards the requirements that might appear in the future. Rather, the focus was on making as good a design as possible, using sound design principles to create a generic, flexible, and reusable system.

Release 2

Release two can be seen as a conclusion of release one, and the requirements were invented or found by the developers, covering what they thought was of importance. In this release, the development team completed the generaliza-

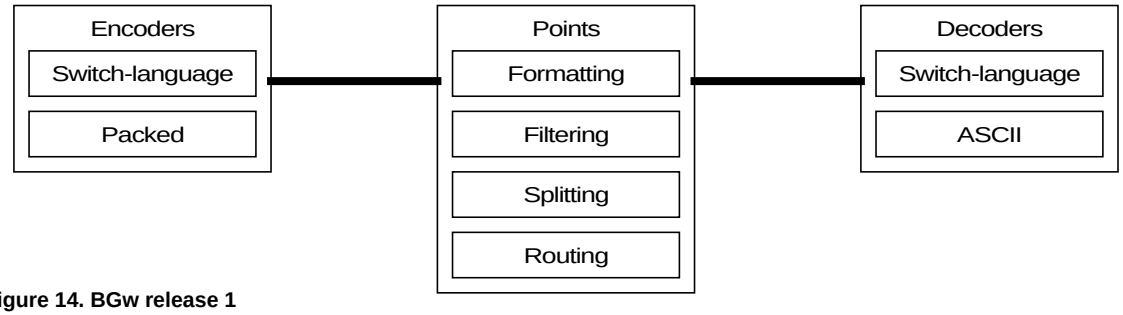

Figure 14. BGw release 1

tions that had begun in release 1. The concepts from release one were cultivated further, and most of the work took place inside of the product. Some new network protocols were added, and the data transfer was done using memory instead of files, as had been the case in release one. Most notably, the inheritance hierarchies in release one had been very deep, and these were now broken up into aggregate relationships instead. This because it was realized that it is easier to add classes to an aggregate relationship than to a class hierarchy. Figure 15 illustrates how the inheritance hierarchies were transformed. In effect, what was done was to apply the state and strategy patterns [Gamma et al. 95].

According to the developers interviewed, what was done in release two was merely to conclude release one by implementing the rest of what was already planned, including network protocols and class generalizations. In this sense, they say, release 2 would be better named release 1.1.

Consequently, the effects this release had on the product line architecture were relatively small. In addition to the continued work to generalize, a number of new encoders and decoders were supported. Furthermore, moving to memory-based data transfer changed the encoders, in that you now had a MemoryEncoder in addition to the FileEncoder. Figure 16 illustrates the changes in release 2. As can be seen, the encoding and decoding frameworks change because of the memory-based transfer form, and the formatting unit is extended by supporting new default formatters. In fact, the formatting unit does not change because new default formatters are added, since these are implemented in the BGw internal language, and are interpreted during run-time.

Release 3

When trying to ship release two on the market, it was realized that the product didn't quite satisfy the potential customers' needs; small things were missing, which made it harder to compete with the product on the market. This lead to release three, where the demands of the market were implemented. Specifically, the interpreted language used in the processing nodes was too simple for the needs of many customers, and the file-based transaction was too coarse. The changes in this release were thus to introduce a new, more powerful language, and to shift from file-based transactions to block-based.

The introduction of a new language affected the point-nodes, since these use the language module. The introduction of the new language went fairly painless, and the impact was of a relatively local scope. However, to move to a block-based transaction scheme turned out to be more troublesome. The entire system had been built up for processing large files relatively seldom, say a 4 MB file every four minutes, and the new block-based transaction assumed that the system would process a large number of small files often, say 8 blocks each 2 KB in size per second. A new Communications-API (CAPI), a Block-CAPI was developed, as was a new processing node. The old CAPI could still be used for data of the old type, but had to be adapted to the new system. Figure 17 depicts the top-level changes that were made to the product line architecture in release 3. The new nodes, both the collection and distribution nodes as well as the processing node, were constructed by scavenging the old code.

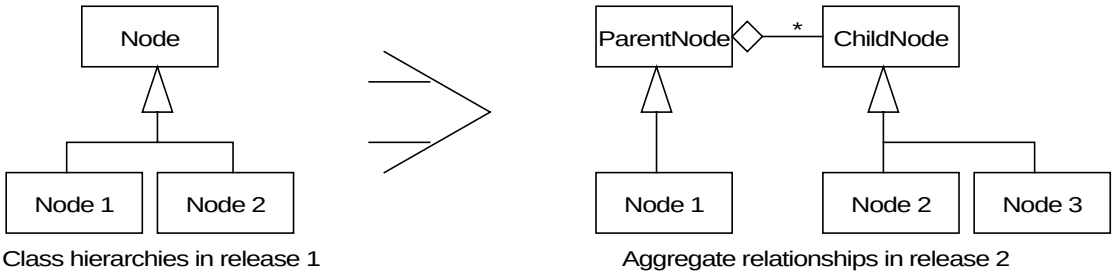

Figure 15. Breaking up of class hierarchies

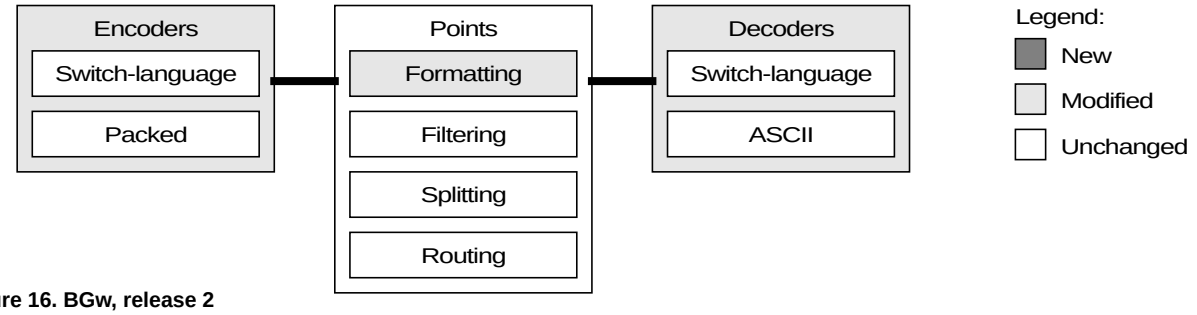

Figure 16. BGw, release 2

In addition to these larger changes, new network protocols were supported, and some new point-nodes were added. As in release two, new default formatters were added, and new encoders and decoders were implemented. On the distribution side some new network protocols were also added. Figure 18 depicts the changes in release 3.

Release 4

Almost as soon as release three was completed, work started on finding the requirements for release four. For various reasons this release is referred to as BGwR7. What had happened since release three was that the customers had run the BGw application for a while, and now required additional functionality. The major changes were that the BGw was ported from SUN to Hewlett-Packard, a database was added as a post-processing system, the set of encoders and decoders were now dynamically configurable, and verifications were made to ensure that all the data was received from the network elements and that nothing was lost during the transfer. In addition, some new point-nodes were implemented, a number of new library functions were added to the interpreted language in the formatters, and lookup tables for temporary storage of data in tables inside the BGw was added. Furthermore, some new network standards and data formats were added.

Architecture-wise, a module for the lookup-tables was added, and the addition of a database as a post-processing system caused some design changes. The database was supposed to store the call data in an unformatted and uncached way, which meant that the steps in the BGw where the data records are decoded to a post-processing format and written to disk for further distribution had to be skipped in the database case. Instead, the data should be sent directly from the penultimate point in the point-graph to the database, since the last point in the graph codes the data to an output format.

Except for this, the product line architecture remained the same. Inside the components, more things happened. HP handles communication differently than SUN, so the port to HP lead to new classes in many class hierarchies, as is illustrated in Figure 19. To dynamically configure the set of coders did in fact not cause as much trouble as could be expected. The functionality already existed in another component, namely the one loading library functions for the interpreted language, and this solution could be reused for the coders as well. The requirements on verification that all data was received forced changes in all the in-points in the point-graph, since this is the first place where the data can be examined inside the BGw. The remaining changes; the new points, the library functions, the lookup table, and the new standards, did not cause any trouble, and were implemented by creating sub-classes in inheritance hierarchies, and by plugging in functions according to existing guidelines. Figure 20 illustrates the changes done in release 4. The changes caused by the port to HP are left out for clarity. Also, several new encoding formats were supported in release

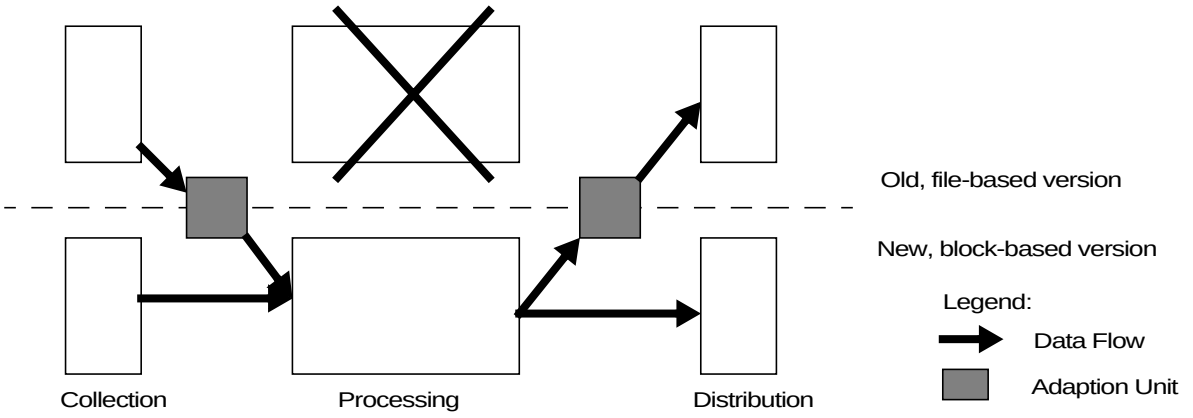

Figure 17. Introducing block-based transactions

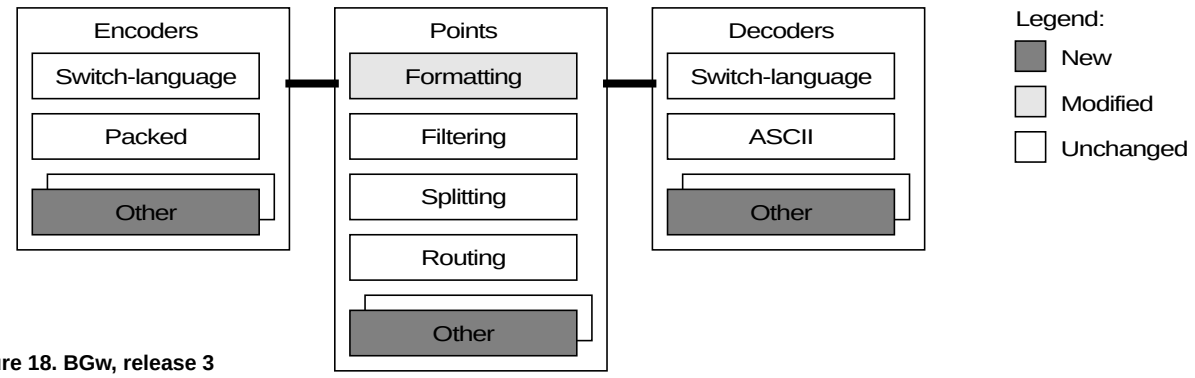

Figure 18. BGw, release 3

four, but as in release three, these are getting quite technical and we feel that they are not so relevant to mention by name.

6 Discussion

Comparing the cases

If we study the differences, we see that the terminology used in the two companies is different. Whereas Ericsson Software Technology considers the entire Billing Gateway to be a framework, Axis Communications talks about frameworks on the subsystem level. Two cases are, of course, not enough to associate significant importance to this difference, but at Ericsson Software Technology the BGw is the flagship of the company, whereas at Axis the StorageServer is merely one product among many others, and the actual flagship of Axis are the 13 reusable assets that the products are built up by. This could be what causes the difference in terminology.

Another difference is how Axis allows experimentation, in that software can be developed for the sole purpose of learning. Consider, for example, the in-house developed MUPP file system, that was present in the system for one product release, and then removed and replaced with other file systems. We also see that networked file system protocols like NFS have been developed, put into one or more products, and then removed for some reason. Ericsson, on the other hand, operates much more goal-determined. Everything that is put into the product stays, be it used or not.

Evolution of the two cases

Examining the evolution of the two cases, we see that the requirements, the evolution of the product line architecture, and of the product line architecture components neatly falls in to a set of categories, that are common to both cases. Figure 21 presents these categories in short, and also illustrates the relations between the sections. Below, we discuss the categories further, from the perspective of the requirements categories.

- **New product family:** When a new product family is introduced to the software product line, one is faced with the decision to either clone an existing product line architecture (split the product line), or to create a branch (derive a product line) in the software product line. At Axis, both has happened, whereas in the Ericsson case branching is preferred. New products and product families generally implies that some components are added, changed, or removed. Indirectly, this also leads to new or changed relations between the components.
- **Introduction of a new product:** A new product differs from the previous members in the product family by supporting more functionality, or the same functionality in a different way. This is achieved by either adding a new component, or by changing an existing component in some way. In some cases, the changes required are so vast

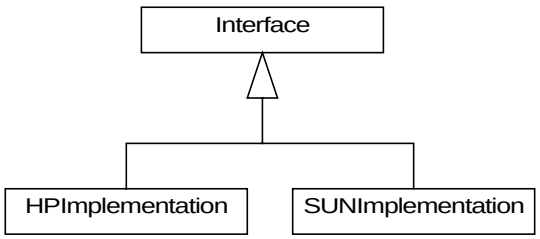

Figure 19. Example of adding classes for HP support

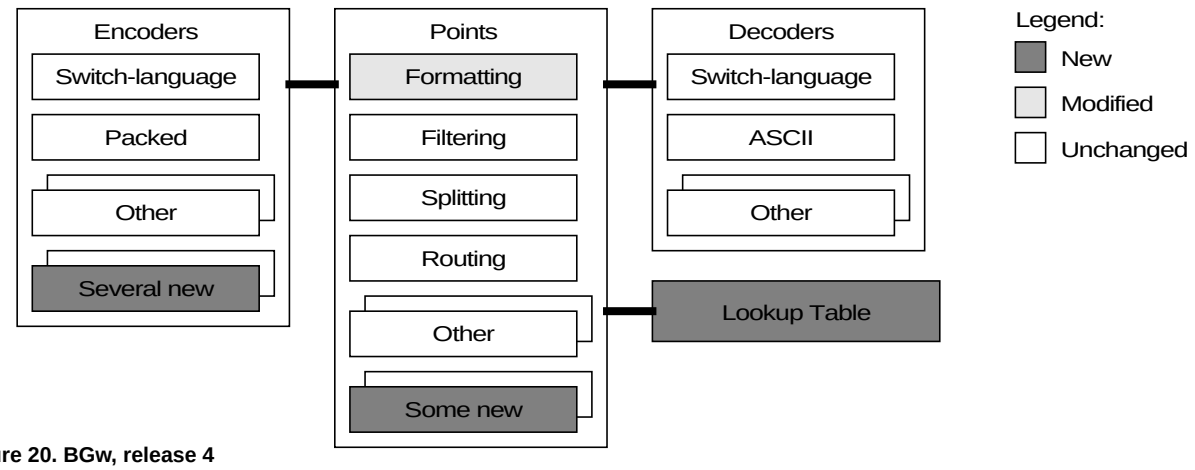

Figure 20. BGw, release 4

that it is more cost-effective to simply rewrite the component from scratch and replace the existing component, drawing from the experiences from the previous version, but designing to incorporate the new desired functionality.

- **Improvement of functionality:** This is the common evolution both at Axis and at Ericsson, where the driving requirement is to keep up with the plethora of standards that are available. Commonly, what this will lead to is a new implementation of an existing framework architecture. The architecture generally does not change, nor do the other framework implementations, even if the framework interface sometimes needs to be adjusted. This is also the evolution presented by [Roberts & Johnson 96], where concrete implementations are added to the library of implementations.
- **Extend standard support:** For various reasons, it has in the two cases sometimes been decided to not support a standard completely in the first release, but rather support a subset of a standard. In subsequent product releases, the support is extended until the entire standard is supported. To extend the standard support is most often concerned with changing an existing framework implementation, in order to incorporate the additional functionality. An example of this is the SMB networked file system protocol in the StorageServer, that has been implemented in turns.
- **New version of infrastructure:** As [Lehman 94] states; functionality has a tendency to move from the perimeter of a system towards the centre, as the innermost layers extend and support more new functionality. This category is more unique to the Axis case, because of the in-house developed CPU; of which frequently new versions are released. Since the Billing Gateway runs on a standard operating system, this system is much less prone to experience changes in the infrastructure. What this leads to is normally that the functionality implemented in a certain framework implementation decreases.
- **Improvement of quality attribute:** At Axis, the quality attribute to improve is mainly maintainability, whereas at Ericsson, most of the efforts are directed at performance. However, it is hard to say what effect the requirement will have on the architecture or the components in the software product line, because every type of quality attribute, as described by [McCall 94], will have a unique set of relations to the product line architecture and the product line architecture components. However, most of the effects are covered by the arrows in Figure 21, i.e. that improvement of a quality attribute results in new, changed, or replaced components. Specifically, a changed component implies in this case that a particular framework implementation has been changed to better meet for example performance requirements. Maintainability requirements can in some cases be met by splitting a component, in order to break out functionality that does not fit in to the original component description anymore.
- **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 cases, but also in other companies. As a project begins to draw towards its end, there is a growing reluctance to alter 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,

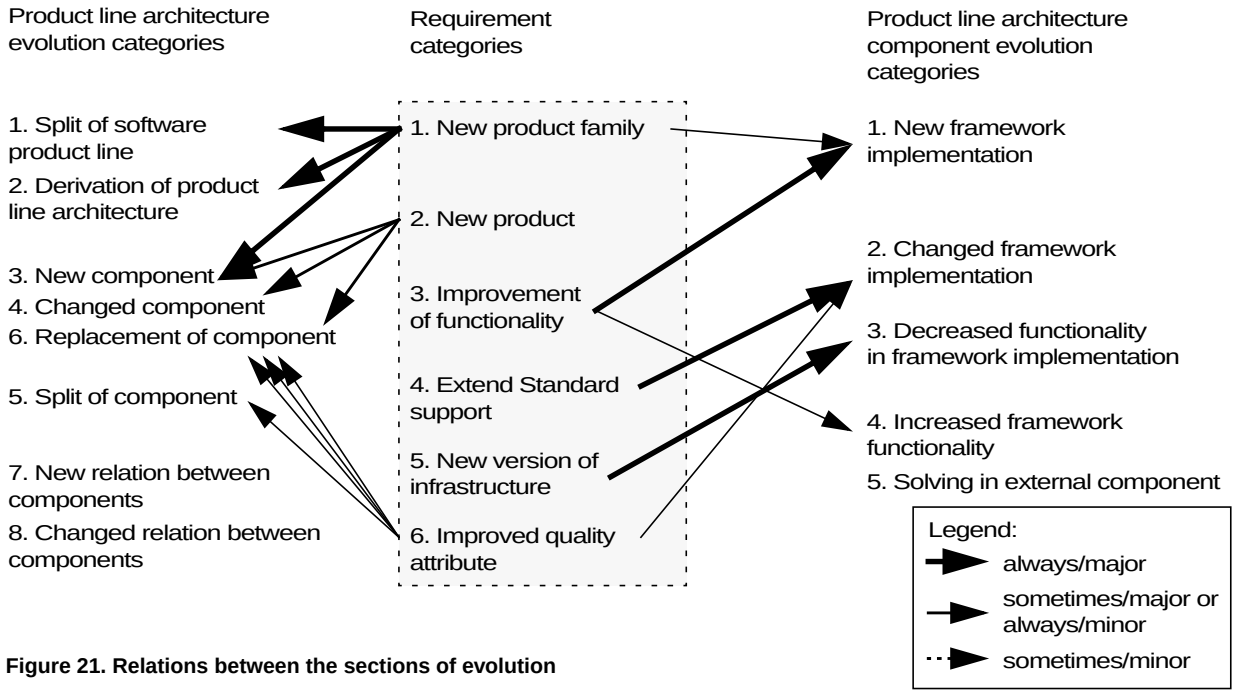

Figure 21. Relations between the sections of evolution

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.

Requirements with Component Impact

In both of our cases, we see that at some point in time a large section of the product is thrown away and replaced by a newly implemented section. The section replaces the old one completely, and does not add anything really revolutionary functionality-wise. In the Axis-case, it was to support write-functionality in the file system framework, and in the Ericsson-case, the transaction model did not work together with the block-based data units. In both cases, it is a single new requirement on the product that forces the developers to discard a whole section and rewrite it.

Also in both cases, it was decided that instead of trying to adjust the old component to work with the new requirements, it is better to start from scratch and scavenge what code can be reused to put into the new component. We also see that it is not the code in itself that is the problem, nor is it the components, the architecture, or the overall concepts. The system still work under the same “work description”, and it still uses the same entities to solve its task.

Why some requirements have such an impact on one or a set of components is not quite clear. We believe that the reason is that some design decisions and assumptions do not take on a first class representation in the final architecture, and, consequently, become embedded into the source code. Active design decisions are thus reformed into implicit assumptions in the source code. When such a design decision alters because of some new or changed requirement, there is no easy way to find all the places where this design decision is used, and it is hence easier, or perceived easier, to rewrite the component from scratch.

In release three of the Billing Gateway case, the 2k-block-based transaction model was introduced. The BGw thus change from a batch-sequential system to a pipes and filters system [Shaw & Garlan 96], meaning that data could flow through the system instead of being buffered between every processing station. The original design decision to process large files relatively seldom was hence changed to process small files often. In the first model, there were no definite time constraints on how long it should take to process one data record, whereas in the second model, the data records could not be buffered and processed as a batch job.

In the StorageServer, the requirement to support write as well as read functionality caused the new generation of the file system framework. While reading is a very passive thing to do in most file systems (Unix file systems may require an update on the ‘last read’-variable), writing to a file system is on the other hand a completely different matter. That files and directories can change gives many new requirements on, for example, caches and access control. Rather than patch the first generation, a new version was created that could handle all the new requirements. Here, we see that the initial design decision to only support read functionality changed, and that write functionality could not be solved parallel to the previous functionality, since the read functionality depends on certain aspects of the write functionality. An active decision to only support read functionality was thus assumed as every line of code was written, and any attempts to code for a future change in this was proven insufficient.

Related work

[Jacobson et al. 97] addresses many issues regarding reuse, but in particular discusses application family engineering, component system engineering and application system engineering. Covered is the requirements elicitation from use-cases, robustness analysis, design and implementation, and also testing of the application family. However, the evolution of the architecture and the components is addressed only marginally. Reuse is assumed to be out-of-box, and the particular problems that arise when other products depend on the evolution of a common asset are, as far as we understand, not covered. Our work does not discuss the aforementioned topics that are covered by Jacobson et al. Instead we study the area of evolution from a technical perspective, and the guidelines presented are more focused towards the later stages (traditionally called post-delivery) of the life cycle.

The Product Line Practice initiative from the Software Engineering Institute resulted in a number of workshops on the topic of product line architectures and the development of these. The reports from these workshops provide much insight in the state of the art regarding product lines. The first workshop [Bass et al. 97] focused very much on charting the situation; how the participants of the workshop worked with their product line with regards to a number of topics. The second workshop [Bass et al. 98a] focused more on giving general insight on some topics, in effect the same as the previous workshop covered. Both of these reports are very broad, and do not provide any in-depth study of the topics. Positioning our work with relation to this, we are focused on the technical aspects of evolution, and provide more detailed analysis of evolution, in addition to the wide, but shallow, surveys in the two SEI workshop reports.

Another, similar, initiative is ARES, which is focused on methods, techniques, and tools to support the development of software for embedded systems in product families. The end goal is to find ways to design software for software product lines to help design reliable systems with various quality attributes that also evolves gracefully. See [Linden 98] for further information on this ESPRIT project.

In 1997, Dikel et al. conducted a study at Nortel [Dikel et al. 97], examining the success factors for Nortel's newly created product line architecture. They identified six factors which they considered critical and evaluated the product line architecture with respect to these six factors. The six factors are: focusing on simplification, adapting to future needs, establishing an architectural rhythm, partnering with stakeholders, maintaining a vision and managing risk and opportunities. For each of these, a rationale is presented, and an example of how the company of the study works with respect to that factor. The product lines in our study are, we believe, quite similar to that of Nortel, and the traps described have also been encountered by Axis and Ericsson. The focus of the Nortel study differs from ours, but we believe they complement each other; the Nortel study from a management perspective, and ours from a technical perspective.

[Macala et al. 97] reports on product line development in the domain of air vehicle training systems. In this project, four key elements were enforced: the process, the domain-specificness, technology support, and the architecture. From the experiences gained, seven lessons and 13 recommendations are given, which are mostly concerned with project management. Most of these lessons and recommendations are not specific for software product-line thinking, and there is little emphasis in the paper about the actual software product line. Compared to [Dikel et al. 97], this study is on a more detailed level, presenting guidelines for the micro-processes whereas Dikel et al. presents more top-level management advice. Similar to the Dikel-case, we see that this study is not focused on predicting changes, other than that they recommend to have at least a five-year plan over the products to release from the product line.

Extending the scope, [Basili et al 96] presents a case study at NASA to build up an estimation model for predicting maintenance effort. Whereas this study is not done on a product line architecture, the data presented covers the activities in maintenance project life cycles, which is in our experience not so different from that of a product line project. In both cases there is an existing code base which must be considered, and the project phases are also the same. The study covers as many as 25 releases, and measures things like effort per activity (analysis, isolation, design, code & test, and inspection) and effort per type of change (adaptive, corrective, perfective). The study assumes that the different types of changes are evenly spread, meaning that taken all changes, both costly and cheap, the effort estimations hold, but the model cannot be used to effectively predict the impact of a particular change request. The taxonomy presented in our work provides a more detailed view of the evolution, and Basili's model can be used per requirement category and hence give more accurate estimations per change request.

Another example of an evolution paper is [Siy & Perry 98], which describes an approach for managing parallel evolution in a large product, i.e. the 5ESS switching system from Lucent Technologies. This study provides valuable insight regarding configuration management. In the Axis case, the two generations of the file system framework coexisted for some years, which makes some of the configuration management issues relevant, even if the two companies in our study does not have as rigorous an approach to parallel development as Lucent Technologies seems to have.

[Lehman 80] presents a set of laws of evolution. While at a high level, they are nevertheless applicable to our study as well. In particular, the laws of continuing change and increasing complexity are extremely important to consider during development using a software product line. If we relate this to our work, we see that our work provides better insight in the dynamics of Lehman's laws by describing the evolution and providing some basic steps one can take to prevent deterioration.

When it comes to categorizing evolution, [Mattsson & Bosch 98c] describe how frameworks evolve, and the types of changes that a framework can become subject to. They state that a framework can undergo (a) internal reorganization, (b) change in functionality, (c) extension of functionality, and (d) reduction of functionality. These categories of change are also found and confirmed in our study.

[Lindvall & Runesson 98] presents data from a case study regarding changes between two versions of a system. This system is, as far as we know, not part of a product line. From the presented figures, one can deduce that the evolution has been markedly different to that of our study, in that many more classes have been changed rather than, as in our case, added. However, we still find some evidence of our guideline regarding general component interfaces; in their study 44.4% of the changes are visible, but the union of these changes and internal class body changes consists of as much as 79.4% of the changes, which indicates that many changes can be kept out of the interfaces. However, it should be noted that the figures from their study are on a class level and not on component, or framework, level as in our case.

A topic that is related to software product lines is domain-specific software architectures, see for example [Tracz 95]. A domain-specific software architecture is a set of components that are specialized for a particular type of task (domain). The relation to a software product line is thus that a product line architecture component contains a domain-specific software architecture. [Tracz 95] is particularly interesting in our case, since it also discusses requirements, especially reference requirements, which are requirements that drives the design of the abstract domain-specific software architecture.

FODA [Kang 90] is a method for domain analysis. Part of FODA is concerned with constructing graphs of features, in order to spot commonalities and benefit from these commonalities when planning reuse and adding new features. Much effort is put into the analysis model, to foresee future enhancements on the features or components. Comparing FODA to our work, we see that FODA focuses on the spotting of commonalities by constructing domain-specific feature categories, and the relations between the categories. The categories we have presented are not based on particular features, but rather on possible steps of evolution. In that sense, FODA is orthogonal to our work. Related to FODA is FeatureRSEB [Griss et al 98], which extends the use-case modeling of RSEB [Jacobson et al. 97] with the feature model of FODA.

The Billing Gateway has been studied in earlier publications by our research group. The papers most closely related to our subject are [Mattsson & Bosch 99a] and [Mattsson & Bosch 99b]. In [Mattsson & Bosch 99a] metrics data from the four Billing Gateway releases are presented. The data presented include the number of classes, the number of added classes per release, and change rates for four of the major subsystems. Based on this data, modules prone for restructuring are identified using an adapted version of a prediction approach proposed by [Gall et al. 97]. The second paper, [Mattsson & Bosch 99b], presents another set of metrics on the releases of the Billing Gateway, where the purpose is to replicate a case study by Bansiya [Bansiya 98][Bansiya 99], which presents an approach to assess framework maturity using a set of design metrics. Whereas the main focus of these two papers may not be exactly in line with our research, the data presented for the four releases of the Billing Gateway is a valuable contribution for further research.

Axis communications have as yet not been researched as much as the Billing Gateway. There are two initial studies conducted on their product line and the development process; [Bosch 99a] and [Bosch 99b]. These papers present an overall view of the architecture and the development work. Problems are identified and discussed, and a number of research issues are presented.

7 Conclusions

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

To address a lack of existing study of software product line evolution, we have performed case studies of software product line evolution at two Swedish companies; Axis Communications, producing a wide range of printer, storage, scanner and camera server products worldwide, and Ericsson Software Technology, a leading company within the telecom industry. The companies have employed software product line based software development for a considerable time and use object-oriented frameworks as the components in the product line. We have studied the evolution of their software product line over a number of releases, i.e. eight and four, respectively. For each release, we presented the requirements and the impact this had on the product line architecture.

Lessons Learned

In the two cases, there is one type of evolution that is predominant, discussed below as “Common evolution“ and we believe that this type of evolution is also predominant in the general case. Furthermore, some other evolution steps also occur more frequently than others, discussed as “Other significant evolution”. Lastly, we have found a number of guidelines that apply to the development and evolution of software product lines.

Common evolution

There are three categories that occur much more frequently than others, namely *Improvement of Functionality*, *Changed Component*, and *New Framework Implementation*. This is the most common way that a product line evolves, namely improving on the functionality of the components by supporting new standards.

Granted that this type of evolution will not cost nearly as much as when the architecture changes as a consequence of the evolution, we still feel that the data is strong enough to suggest that it is very important to spend time in the design phase to ensure that future functionality fits into the existing interfaces, so that architectural change can be minimized.

The data is also strong enough to suggest that if requirements of this type is detected in the requirements specification, a fairly accurate estimation of the impact of this requirement can be made, which helps in planning a project. Furthermore, by examining earlier requirements on a software product line, one can better estimate the current state of erosion that the product line is in, by examining how many requirements of this type the product line has been subjected to.

Other significant evolution

In addition to the standard way of evolution presented above, there are some evolution steps that, because of their high rate of occurrence, are worth mentioning further. These are to introduce a new component, to replace a component, and to change a framework implementation.

1. **Changed Framework Implementation.** During the development of the StorageServer, it happened frequently that only part of a standard is implemented in the first release, and that additional parts are added in later projects. Likewise, in the Billing Gateway, the work done for release two was mostly concerned with restructuring the object-oriented design, which caused changes of structural nature to many of the components. Concluding, it is an ongoing activity to restructure the software in order to prevent premature aging, and to extend the support for various standards in order to better compete on the market.
2. **New Component.** The main contributors of this category are the first releases of the StorageServer and the Billing Gateway. Since this is where the software studied started, all components are naturally new.

   Whereas this category occurs fairly frequent in our cases, we have seen that the occurrences are mostly concerned with the creation of the first version of the software product line. Once the product line is in place and is used in new products, it is very rare that new components are introduced. This is, we argue, also true in the general case; that to add new components to the architecture after the initial version is created is not done very often.
3. **Replaced Component.** Predominant in this category is the replacement of the processing-node in the third release of the Billing Gateway, that forced a replacement of all of the components inside it. Consequently, adding new component implementations, as described above is a common evolution, whereas replacing both the framework interface and all of the implementations is much more rare.

To conclude these two sections, based on the two cases we have presented one case of common evolution, namely that software product lines evolve by the addition of new component implementations of the existing component interfaces without touching these interfaces, and one case of less common but still frequent case of evolution where the component implementations are modified for various reasons. In addition, two frequent occurrences of what at a quick glance seems to be evolution categories have been shown to be more concerned with the initiation of the software product line than the actual evolution of it. In our opinion, these findings are strong enough to be considered further when designing and maintaining a software product line, and focus design and work efforts accordingly.

Guidelines for Software Product Line Evolution

We have evaluated the two cases with respect to the evolution categories, and discuss some of the categories that occur more frequently further. Out of this analysis we present a set of guidelines focused on the technical aspects of software product line evolution.

These guidelines are useful when creating a new software product line, but are even more important if there already exist a product line that is evolving. Each of the guidelines present an area for future research, in order to further help the evolution work by proposing methods and techniques to avoid the common pitfalls of software product line development.

1. **Avoid adjusting component interfaces.** We have found that sooner or later the interfaces of components will need to be changed because a new implementation requires more of the interface than was planned from the beginning. The trick is to delay this moment for as long as possible. There are two ways in which an interface can be altered; it can grow, i.e. provide more functionality, and it can shrink, i.e. provide less functionality. The former is the result of an external component requiring more of the component than it can currently cope with. To add functionality to a component is laborious, but it is easy to find where the change needs to be implemented; in all the implementations of the component. There are also cases where the calling components need to be found and modified as well. A shrinking interface may be caused by the introduction of a new component, that is specialized on a type of functionality previously handled by another component. This change will have impact on the component implementations, in order to reduce the space required by the component and also to use the new component wherever the functionality is used internally in the component. Furthermore, all external components that use the functionality will also need to be found and modified to use the new component.

There may exist some ways to solve growing interfaces without touching the existing implementations. If the new functionality is well behaved, it can be added as a small module next to the previous implementations, and connect to them using the public interface. In most other cases, there is no solution but modifying the original source code. We find this a very interesting area for future research; how to design components for growing interfaces.

2. **Focus on making component interfaces general.** During our studies, we have found that the prevalent form of evolution is that of adding new implementations to the existing components and thus increase the set of supported standards. When designing the architecture and the component interfaces, focus on how new implementations in components are added. Issues concerning this are where to keep functionality that at the design stage is perceived to be common to all implementations, and what to actually include in the interface. This guideline is highly related to the previous one, which discusses this issue under the evolution of the software product line, whereas this is more focused on the initial design of the architecture.

   To the best of our knowledge, the way to find out if an interface is general enough is to make a thorough domain analysis. [Bengtsson & Bosch 99a] and [Bengtsson & Bosch 99b] present a somewhat alternative view of how to evaluate the architecture and the interfaces for evolvability, based on maintenance scenarios prepared by developers with more or less experience.
3. **Separate domain and application behavior.** It is common that a component implements not only the domain functionality that it is supposed to support, but also a number of facilitating functionalities that are more focused on providing application structure and support. This can be things like reference counting for memory management or dependency graphs to start up the components in the correct sequence. The problem is that such functionality has a tendency to evolve more than the domain functionality, in that frequently new features are requested to provide even better control. If this functionality is embedded in the same component implementations as the domain functionality, not only must all component implementations be altered for the new control functionality, but the developers must also manually find their way between domain and control functionality to find out where to make the changes.

   To this end, we suggest that application functionality is separated from the domain functionality, and that aggregate relationships are used as the only connector between the two. This will not help alleviating changes to the application functionality, but it will help in making the domain functionality more reusable, and less prone to require modifications is the application functionality changes.
4. **Keep the software product line intact.** The term “software product line” suggests that there is one product line per type of product, and if a company develops more than one type of product, it also has more than one product line. This is normally not the case. In many cases, the term “Company Software Architecture” better depicts the reality. The term “product” denotes, in such cases, only variations on a theme, whereas “product family” is used to refer to the product areas in general. As an example, Axis Communications has a storage product family, a camera product family, and a scanner product family. The architectures in these systems are however very similar, and many assets can be reused in more than one product family.

   A new product is normally created by exchanging some component implementation, for example ethernet to token ring. With this view, different products are mostly created for marketing reasons. Technically, products are managed by configuration management, i.e. different make-files. Product families, on the other hand, are created by exchanging parts of the architecture, i.e. replacing components. Unless there are some very compelling reasons, for example that the new product family is written in a new language, one should strive to keep as much as possible from the other product families, i.e. to only create a branch of the product line.

   This seems to be common sense, but there are often forces that advocates a split up of the software product line as well, or that more than is really necessary is divided into two versions; the old one, and a new one for the new product family. Often, however, even if a component does not provide all of the functionality one may wish, it will be cheaper to enhance the existing component to supply the new functionality and adopt the previous products to this, than it is to maintain two parallel versions of the same component, with all that this brings of bug fixes and synchronization problems. Sooner or later, a decision will be taken to join the two versions anyhow.
5. **Detect and exploit common functionality in component implementations.** Developing a new framework implementation is often done by finding the most similar implementation and write the new code using the old one as a template. This leads to very similar code, code that could have been reused instead. We suggest that every domain component should have at least one library component in its tow, where common functionality between component implementations can be placed as soon as it is identified as shared between implementations. Development of a new implementation then also becomes a “reuse inspection” for the implementation used as a template. Trouble with this approach arises when it is realized that the functionality put into the library component was in fact just common for two of the implementations, and that all the other implementations use another functionality. Also, the impact of changing in the shared functionality may be hard to estimate.

6. **Be open to rewriting components and implementations.** As we have shown, there are situations when a particular implementation, or maybe even an entire component simply cannot cope with a desired change. Rather than forcing the change into the component, which results in a patchy code and shortens the life of it even further, we suggest that the option to rewrite the component from scratch is considered. Even if the rewritten code does not get better than the original code it will certainly be better than the code produced by patching the previous implementation. Our two case studies suggest that it is common practise in industry to rewrite components like this.
7. **Avoid hidden assumptions and design decisions in the source code.** Twice in our study we have encountered a situation in which an entire component or even a set of components have been discarded and rewritten from scratch. The reason for this has been that some design decisions do not take on a first class representation in the architecture, and becomes hidden in the source code. When such assumptions or design decisions change, it is easier to rewrite the entire component, according to guideline 6, rather than to change the assumption in the existing code.

   Our solution proposal to this is to try to give all design decisions a first class representation in the form of components or design patterns so that the effects of changing these decisions are minimized.

Acknowledgments

We would like to thank the companies involved in the two case studies, Axis Communications AB, Lund, and Ericsson Software Technology, Ronneby. We would also like to thank our colleagues PerOlof Bengtsson and Michael Mattsson for valuable comments and encouragements.

References

[Bansiya 98] Bansiya, J., "Assessment of Application Framework MAturity Using Design Metrics", Accepted for publication in *ACM Computing Surveys - Symposia on Object-Oriented Application Frameworks*, 1998, http://indus.cs.uah.edu/research-papers.htm

[Bansiya 99] Bansiya, J., "Assessing Maturity of Object-Oriented Application Frameworks", Accepted for publication in *Object-Oriented Application Frameworks; Problems & Perspectives*, M.E. Fayad, D.C. Schmidt, R.E. Johnson (eds.), Wiley & Sons (Forthcoming), http://indus.cs.uah.edu/research-papers.htm

[Basili et al 96] Basili, V., Briand, L., Condon, S., Kim, Y.M., Melo, W.L., Valett, J., "Understanding and predicting the Process of Software Maintenance Releases", *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, IEEE, March 1996, pp. 464-474.

[Bass et al. 97] Bass, L., Clements, P., Cohen, S., Northrop, L., Withey, J., *"Product Line Practice Workshop Report"*, CMU/SEI-97-TR-003, Pittsburg, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Bass et al. 98a] Bass, L., Chastek, G., Clements, P., Northrop, L., Smith, D., Withey, J., *"Second Product Line Practice Workshop Report"*, CMU/SEI-98-TR-015, Pittsburg, PA: Software Engineering Institute, Carnegie Mellon University, 1998.

[Bass et al. 98b] Bass, L., Clements, P., Kazman, R., *"Software Architecture in Practice"*, Addison-Wesley, 1998.

[Bengtsson & Bosch 99a] Bengtsson, P-O., Bosch, J., "Architecture Level Prediction of Software Maintenance", *Proceedings of CSMR'3, 3rd European COnference on Software Maintenance and Reengineering*, Amsterdam, March 1999.

[Bengtsson & Bosch 99b] Bengtsson, P-O., Bosch, J., "An Experiment on Creating Scenario Profiles for Software Change", Accepted with revisions for *Annals of Software Engineering*, June 1999.

[Bosch 99a] Bosch, J., "Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study", *Proceedings of the First Working IFIP Conference on Software Architecture*, February 1999.

[Bosch 99b] Bosch, J., "Product-Line Architectures in Industry: A Case Study", *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

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

[Dikel et al. 97] Dikel D., Kane D., Ornburn S., Loftus W., Wilson J., "Applying Software Product-Line Architecture", IEEE Computer 30(8), pp. 49-55, August 1997.

[Gall et al. 97] Gall, H., Jazayeri, M., Klösch, R.G., Trausmuth, G., "Software Evolution Observations Based on Product Release History", *Proceedings of the Conference on Software Maintenance - 1997*, pp. 160-166, 1997.

[Gamma et al. 95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *"Design Patterns, Elements of Reusable Object-Oriented Software"*, Addison-Wesley, 1995.

[Griss et al 98] Griss, M.L., Favaro, J., d'Alessandro, M., *"Integrating feature modeling with the RSEB"*, Proceedings of the Fifth International Conference on Software Reuse, IEEE Computer Society, Los Alamitos, CA, USA, p.76-85., 1998.

[Jacobson et al. 97] Jacobson, I., Griss, M., Jonsson, P., *"Software Reuse: Architecture, Process, and Organization for Business Success"*, Addison-Wesley-Longman, May 1997.

[Kang 90] Kang, K., et al, "Feature-Oriented Domain Analysis Feasibility Study", SEI Technical Report CMU/SEI-90-TR-21, November 1990.

[Lehman 80] 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.

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

[Linden 98] v.d. Linden, F. (Editor), *"Development and Evolution of Software Architectures for Product Families"*, Proceedings of the Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, LNCS 1429, Springer Verlag, February 1998.

[Lindvall & Runesson 98] Lindvall, M., Runesson, M. *"The Visibility of Maintenance in Object Models: An Empirical Study"*, Accepted to IEEE conference in Software Maintenance 1998 (ICSM'98), Washington, USA, November 1998.

[Macala et al. 97] Macala, R.R., Stuckey, L.D. Jr., Gross, D.C., Managing domain-specific, product-line development, IEEE Software 13 3 May 1996 IEEE p 57-67 0740-7459, 1996.

[Mattsson & Bosch] 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.

[Mattsson & Bosch 99a] Mattsson, M., Bosch, J., "Evolution Observations of an Industry Object-Oriented Framework", Accepted for the *International Conference on Software Maintenance (ICSM'99)*, May 1999.

[Mattsson & Bosch 99b] Mattsson, M., Bosch, J., "Characterizing Stability in Evolving Frameworks", *Proceedings of TOOLS Europe 99*, pp. 118-130, March 1999.

[Mattsson & Bosch 98c] 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.

[McCall 94] McCall, J.A., "Quality Factors", *Encyclopedia of Software Engineering*, Marciniak, J.L. (Editor), John Wiley & Sons, 1994.

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

[Shaw & Garlan 96] Shaw, M., Garlan, D., *"Software Architecture - Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.

[Siy & Perry 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.