A Taxonomy of Variability Realization Techniques

Mikael Svahnberg
Blekinge Institute of Technology
Department of Systems and Software Engineering
Box 520
SE-372 25 Ronneby
Sweden
Mikael.Svahnberg@bth.se

Jilles van Gurp
GX creative online development B.V.
Wijchenseweg 111
NL-6538 SW Nijmegen
The Netherlands
jilles@gx.nl

Jan Bosch
University of Groningen
Department of Mathematics & Computing Science
P.O.Box 800
NL-9700 AV Groningen
The Netherlands
bosch@cs.rug.nl

Abstract. Development of software product families relies heavily on the use of variability to manage the differences between products by delaying design decisions to later stages of the development and usage of the constructed software systems. Implementation of variability is not a trivial task, and is governed by a number of factors to consider. In this paper, we describe the factors that are relevant to determine how to implement variability, and present a taxonomy of variability realization techniques.

Keywords: Variability, Software Product Families, Development Process, Software Architecture, Variability Realization Techniques

1. Introduction

Over the last decades, the software systems that we use and build require and exhibit increasing variability, i.e. the ability of a software artifact to vary its behavior at some point in its lifecycle. We can identify two underlying forces that drive this development. First, we see that variability in systems is moved from mechanics and hardware to the software. Second, because of the cost of reversing design decisions once these are taken, software engineers typically try to delay such decisions to the latest phase in a system's lifecycle that is economically defendable. An example of the first trend is car engine controllers. Most car manufacturers now offer engines with different characteristics for a particular car model. A new development is that frequently these engines are the same from a mechanical perspective and differ only in the software of the car engine controller. Thus, earlier the variation between different engine models was first incorporated through the mechanics and hardware. However, due to economies of scale that exist for these artifacts, car developers have moved the variation to the software.

An earlier version of this article is available as a technical report: M. Svahnberg, J. van Gurp, J. Bosch, "On the Notion of Variability in Software Product Lines", Technical Report BTH-RES--02/01--SE, ISSN 1103-1581, Blekinge Institute of Technology, Sweden, 2001.

The second trend, i.e. delayed design decisions, can be illustrated through software product families [1][2][3] and the increasing configurability of software products. Over the last decade many organizations have identified a conflict in their software development. On the one hand, the amount of software necessary for individual products is constantly increasing. On the other hand, there is a constant pressure to increase the number of software products put out on the market in order to better service the various market segments. For many organizations, the only feasible way forward has been to exploit the commonality between different products and to implement the differences between the products as variability in the software artifacts. The product family architecture and shared product family components must be designed in such a way that the different products can be supported, whether the products require replaced components, extensions to the architecture or particular configurations of the software components. Additionally, the software product family must also incorporate variability to support likely future changes in requirements and future generations of products. This means that when designing the commonalities of a software product line, not all decisions can be taken. Instead, design decisions are left open and determined at a later stage, e.g. when constructing a particular product or during runtime of a particular product. This is achieved through variability.

Since this article discusses variability, it is important to provide a definition of the term: *software variability is the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context*. This article is about techniques that enable software developers to improve variability of software artefacts.

Based on our case studies [4][5][6], we have found that it is not a trivial task to introduce variability into a software product family. We also see that engineers are seeking variation mechanisms beyond those shipped with their development tools or that are not supported by used software systems. The adoption of mechanisms such as aspect oriented programming [7] and the popularity of generative and reflective techniques (see e.g. Reference [8]) in e.g. the Java and .Net programming communities are evidence of this.

Essentially, by supporting variability, design decisions are pushed to a later stage in the development. Rather than making specific design choices, the design choice is made to allow for variability at a later stage. For example, by allowing users to choose between different plug-ins for a media player, the media player designers can avoid hard wiring the playback feature to a particular playback functionality (by enabling the system to use plug-ins). Thus they can support new file formats after the media player has been shipped to the end user.

Many factors influence the choices of how design decisions can be delayed. Influencing factors include for example the type of the software entity for which variability is required, how long the design decision can be delayed, the cost of delaying a design decision and the intended run-time environment. Another factor to consider is that variability does not need to be represented only in the architecture (i.e. the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution [9]) or the source code of a system. It can also be represented as procedures during the development process, making use of various tools outside of the actual system being built.

Although the use of variability techniques is increasing, research, both by others, e.g. Reference [2][3][10][11], and by ourselves, e.g. Reference [12][13][14], shows that several problems exist. A major cause for these problems is that software architects typically lack a good overview of the available variability techniques as well as the pros and cons of these techniques. In addition they tend to apply them in an ad-hoc fashion without properly considering the various variability related constraints imposed by requirements and the road map of the system.

This paper discusses the factors that need to be considered for selecting an appropriate method or technique for implementing variability. We also provide a taxonomy of techniques that can be used to implement variability. The contribution of this is, we believe, that the notion of variability and its qualities is better understood and that more informed decisions concerning variability and variation points can be made during software development. The provided toolbox of available realization tech-

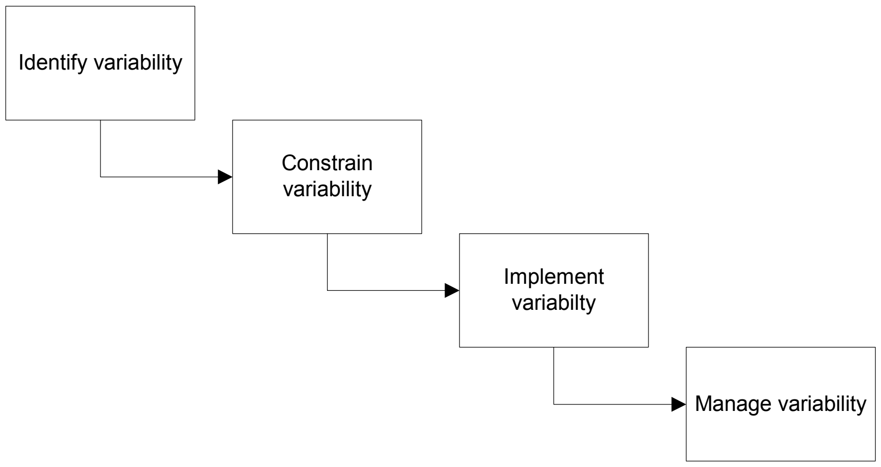


Figure 1. Steps for introducing variability

niques facilitates the development process since the consequences of a particular choice can be seen at an early stage. This is similar to the way Design Patterns [15] also present developers with consequences of a particular design decision.

It should be noted that this paper focuses on implementing variability in architecture and implementation artifacts, such as the software architecture design, the components and classes of a software system. We do not address issues related to how to handle for example variability of requirements, managing variations of design documents or test specifications, structure of the development organization, etc. While these are important subjects and need to be addressed to properly manage variability in a software product family, the goal of this paper is to cover the area of how to technically achieve variability in the software system. This paper should thus be seen as one piece in the large puzzle that is software product family variability. For a description of many of the other key areas to consider, please see e.g. Reference [3] or [4].

2. Introducing variability in software product families

While introducing variability into a software product family there are a number of steps to take along the way in order to get the required variability in place, and to take care of it once it is in place. In this section we present the steps (see Figure 1) that in our experience are minimally necessary to take. These steps are an extension of the process presented in our earlier article on software variability [12].

We use the term *variability* to refer to the whole area of how to manage the parts of a software development process and its resulting artifacts that are made to differ between products or, in certain situations, within a single product. Variability is concerned with many topics, ranging from the development process itself to the various artifacts created during the development process, such as requirements specifications, design documents, source code, and executable binaries (to mention a few). In this paper, however, we focus on the software artifacts, involving software architecture design, detailed design, components, classes, source code, and executable binaries. Variability in this context, refers to the ability to select between these artifacts at different times during a product's lifecycle.

2.1 Identification of Variability

The first step is to identify where variability is needed in the software product family. The identification of variability is a rather large field of research (see for example Reference [3] or [4]), but it is outside of the scope of this paper to investigate it in detail. However, there appears to be some consensus that there is a relation between features and variability, in that variability can be more easily identified if the system is modeled using the concept of *features* (see e.g. Reference [16][17][18][19], as well as FODA [20] and FORM [21]). A major advantage of discussing a system in terms of features is that they bridge the gap between requirements and technical design decisions since software components rarely addresses a single requirement but rather an entire set of requirements. As discussed below, this is similar to our perception of a feature.

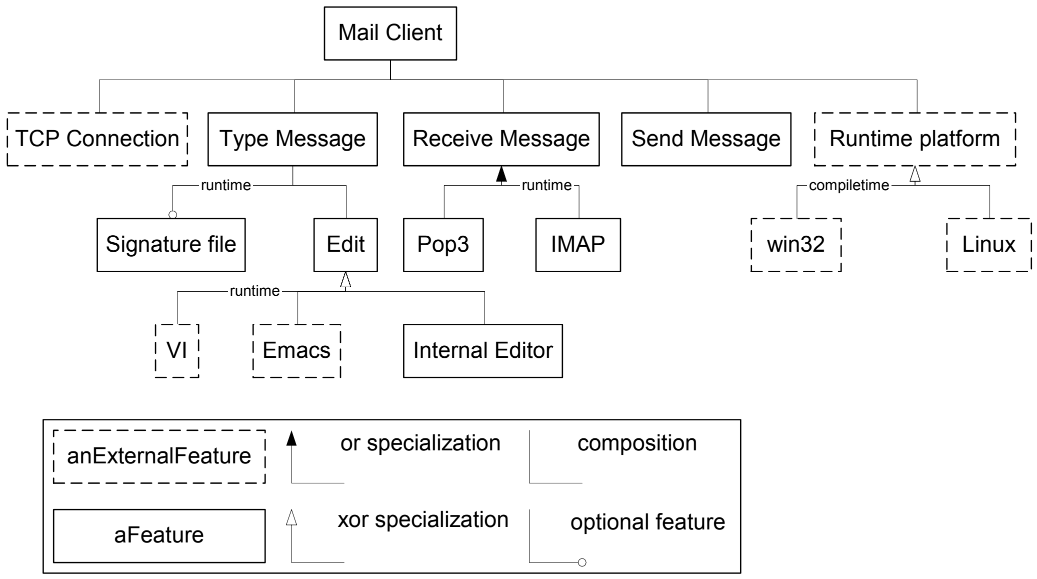


Figure 2. Example feature graph

In Reference [4], we defined features as follows: "*a logical unit of behavior that is specified by a set of functional and quality requirements*". The point of view taken there, and in this article, is that a feature is a construct used to group related requirements ("*there should at least be an order of magnitude difference between the number of features and the number of requirements for a product family member*" [4]). In other words, features are an abstraction from requirements. In our view, constructing a feature set is the first step of interpreting and ordering the requirements. In the process of constructing a feature set, the first design decisions about the future system are already taken. It is important to realize that there is an *n*-to-*m* relation between features and requirements. This means that a particular requirement (e.g. a performance requirement) may apply to several features and that a particular feature typically meets more than one requirement.

A *software product family* provides a central architecture that can be evolved and specialized into concrete products. In the context of software product families, features are used to differentiate various products (with respect to marketing, management of artifacts, etc.), i.e. variation between products of a software product family is usually expressed in terms of features. Consequently, a software product family must support variability for those features that tend to differ from product to product.

The process of identifying variability consists of listing the features that may vary between products. These so called *variant features* will need to be implemented in such a way that the resulting software artifact can easily be adapted to accommodate the different variants that are associated with the variant feature.

In order to identify variant features, we suggest that features are organized into so called feature diagrams. Over time, various feature diagram notations have been proposed, for example: FODA [20][21], RSEB [10] and FeatureRSEB [22]. In addition, our notation, which is derived from FeatureRSEB, is introduced in Reference [12]. An example of our notation is provided in Figure 2. The example models some of the features of an email client application. Each of the notations makes a distinction between:

- **Mandatory features.** These are the features that identify a product. For example, the ability type in a message and send it to the smtp server is essential for an email client application.
- **Variant features.** A variant feature is an abstraction for a set of related features (optional or mandatory). An example of a variant feature for the email client might be the editor used for typing in messages. Some email clients, for example, offer the feature of having a user configurable editor. We make a distinction between XOR (only one of the variants can be selected) and OR (more than one of the variants may be selected) variation.
- **Optional features.** These are features that, when enabled, add some value to the core features of a product. A good example of an optional feature for an email client is the ability to add a signature to each message. It is in no way an essential

feature and not all users will use it but it is nice to have it in the product. Optional features can be seen as a special case of variant features.

Our own notation adds a fourth category:

- **External features.** These are features offered by the target platform of the system. While not directly part of the system, they are important because the system uses them and depends on them. E.g. in an email client, the ability to make TCP connections to another computer is essential but not part of the client. Instead the functionality for TCP connections is typically part of the OS on which the client runs. Differences in external features may motivate inclusion of parts in the software to manage such variability. Requirements with respect to deployment platform have consequences for the amount and type of the available external features. Similarly depending on certain external features limits the amount of deployment platforms. Organizing external features under variant/optional features therefore may help improve platform independence. In any case, the decision on how to handle external features requires an informed approach, such as our variation management process. Our choice of introducing external features is further motivated by Zave and Jackson [23]. In this article it is argued that requirements should not reflect on implementation details (such as platform specific features). Since features are abstractions from requirements, we need external features to map requirements that can be met by using functionality external to the system to features.

The main difference between our notation and other notations is this notion of an external feature. In addition, our notation communicates the notion that the relation between a variant feature and its variants is a specialization relation. Consequently, we have adopted a UML like notation. Finally, our notation includes the notion of binding time, which is one of the constraints discussed in Section 2.2. However, for the purpose of identifying variant features, either of the notations is adequate.

2.2 Constraining Variability

Once a variant feature has been identified, it needs to be constrained. After all, the purpose is not to provide limitless flexibility but to provide just enough flexibility to suit the current and future needs of the system in a cost effective way. By constraining the variability we enable an informed decision on how to implement the variant feature in the software product family, as is described in Section 2.3.

The aspects to consider when selecting an appropriate technique for implementing a variant feature can be identified by considering the lifecycle of the variant feature. During the lifecycle the variant feature is transformed in several ways during different phases until there is a decision on which variant to use in a given moment. In this section we briefly introduce these transformations and then discuss them further in the subsequent sections.

- **Identified**. As described above, in Section 2.1, the first step is to identify a variant feature and the available variants.
- **Implicit**. When a variant feature is first identified it is said to be *implicit*, as it is not yet realized in the software product family. An implicit variant feature exists only as a concept and is not yet implemented. Software designers and developers are aware that they eventually will need to consider the variant feature, but defer its implementation until a later stage.
- **Introduced**. A variant feature ceases to be implicit when it is *introduced* into the software product family. When a variant feature is introduced, it has already been decided how to implement it in the software product family. After a variant feature is introduced it has a representation in the design or implementation of the software product family. This representation takes the form of a set of *variation points*, i.e. places in the design or implementation that together provide the mechanisms necessary to make a feature variable. Note that the variants of the variant feature need not be present at this time.
- **Populated**. After the introduction of a variant feature the next step is to *populate* the variant feature with its variants. What this means is that software entities are created for each of the variants in such a way that they fit together with the

variation points that were previously introduced, and then the variation points are instrumented so that they can use the new variant.

- **Bound**. At some stage a decision must be taken which variant of a variant feature to use, and at this stage the software product family or software system is *bound* to one of the variants for a particular variant feature. This means that the variation points related to the variant feature are committed to the software entities representing the variant decided upon.

For constraining a variant feature the last three of these transformations is of importance, and are described further in Section 2.2.1, Section 2.2.2 and Section 2.2.3, respectively.

During each of the transformations, decisions are made about the variant feature. Various stakeholders may be involved in these decisions and specifying who takes what decisions can be considered to be a part of the constraints. We have identified three groups of stakeholders that need to be considered in the context of a software product-line (we only consider stakeholders that directly interact with the variant feature):

- **Domain Engineers.** Domain engineers are all people (e.g. designers, programmers, etc.) that are participating in the development of a software product family (i.e. a set of reusable software artifacts).
- **Application Engineers.** Application engineers (e.g. product managers, programmers), reuse the software artifacts from the software product family to create specific products.
- **End Users.** The products created by the application engineers are eventually used by an end user. At present we make no distinction between the real end users of the product and the “end users” of the variation points, i.e. the system or service engineers. The system engineers are the active users of variation points in software (e.g. setting environment variables, configuring infrastructure, upgrading components etc.). The border between an end user and a system engineer is not always clear and distinct. At present we see few reasons to differ between end users and service engineers from a development perspective as both categories are users of a delivered system, albeit using the system differently.

2.2.1 Introducing a Variant Feature

When a variant feature is introduced into the software product family (by a domain engineer) it takes the form of a set of variation points. The variation points are used to connect the software entities constituting the variants with the rest of the software product family. The decision on when to introduce a variant feature is governed by a number of things, such as:

- The size of the involved software entities
- The number of resulting variation points
- The cost of maintaining the variant feature

Size of Software Entities. A variant can be implemented in many ways, for example as a component, a set of classes, a single class, a few lines of code, or a combination of all of these. We refer to these different implementation artifacts as *software entities*. As is presented in Table 1 different software entities are most likely in focus during the different stages architecture design, detailed design, implementation, compilation and linking. While the size of the involved software entities does not have any direct connection to how the variation points are implemented, it determines the overall strategy of how to implement the variant feature. This is further discussed in Section 2.3.

Table 1: Entities most likely in focus during the different development activities

Development Activities	Software Entities in Focus
Architecture Design	Components Frameworks
Detailed Design	Framework Implementations Sets of Classes
Implementation	Individual Classes Lines of Code
Compilation	Lines of Code
Linking	Binary components

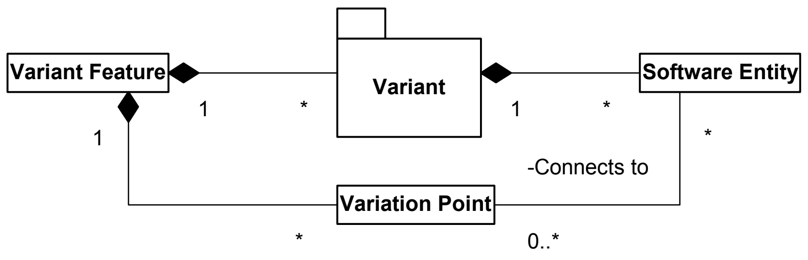


Figure 3. Relations between terms

We would also like to stress the point that a variant of a variant feature can be implemented by one or more cooperating software entities. Each of these software entities are connected to the rest of the software system with one or more variation points. Figure 3 presents the relations between a variant feature, variants, variation points and software entities further. The introduction of a variant feature is the process of adding the entire set of variation points necessary to implement all of the variants for the variant feature.

Number of Variation Points. In our experience, a variant feature usually maps to a set of software entities. These software entities need not be of the same type. Hence a variant feature is typically implemented by a set of collaborating components, classes and even lines of code. These collaborating software entities can be scattered throughout the software system, even if it is usually possible to keep them gathered in a few places. Consequently, a single variant feature typically manifests itself as a set of variation points in the software system, working with different types of software entities. These variation points may be introduced into the software system during different development phases.

To increase the understandability of the source code, facilitate maintenance and reduce the risk of introducing bugs, it is desirable to keep the set of variation points as small as possible. This means that rather than using a mechanism that works with lines of code, these lines of code should be gathered into a class, or a set of classes. This would enable us to use a single variation point for the entire variant feature, rather than many variation points for each individual part of the variant feature.

Cost of Maintaining Variant Feature. Whenever something is added to the software system - e.g. a component, a design element, a class or a line of code - it needs to be maintained during subsequent development phases. For example, if a software entity is added during architecture design, it needs to be taken into consideration during architecture design, detailed design, implementation, compilation and linking. If, however, a software entity is added during implementation, it only needs to be considered during implementation, compilation and linking. Hence, there is a cost associated with introducing a variant feature at an early stage during development. On the other hand, it is typically more expensive (computationally and in terms of complexity) to have variant features that work during run-time. Hence, care should be taken to add a variant feature (i.e. the software entities of the variants as well as the variation points) not too early, but not later than is needed either.

Example of Introducing a Variant Feature. Consider the example in Figure 2 of an e-mail application. This feature graph is created by the domain engineers to identify the features, and in particular identify the features that may vary. One example of a variant feature is which protocol to use when receiving messages (Figure 2 lists the two variants POP3 and IMAP). It is decided to implement the two variants as two implementations of a component in the architecture. Thus, this sets the size of the involved software entities to components, or component implementations. By ensuring that both variants have the same interface, i.e. the same set of possible operations, the number of variation points can be reduced to only the point where it is decided which variant to use. This also reduces the cost of maintaining the variation points, as there is only one place to consider. Should, however, the interface to the variants change, this will impose a larger cost since not all places where the variant feature is used are gathered in one place.

2.2.2 Populating a Variant Feature with Variants

During this step the software entities of the variants are created such that they fit together with the variation points introduced in the previous step. After this the variation points are instrumented to be made aware of each new variant. There are three things to consider when instrumenting the variation points, namely *when*, *how* and *who*.

When it is possible to Populate. A variation point is typically available for population only during a limited period of its life cycle. This depends on e.g. what type of software entities that are involved in the variant feature, and what types of software entities that are in focus during different phases of a systems lifecycle. This means that during some phases of a systems lifecycle (comprising all of the activities from product architecture design to run-time) a variation point is *open* for adding new variants or for removing old ones. During all other phases it is not possible to change the set of available variants, and then the variation point is *closed*.

The time when a variation point is open or closed for adding new variants is mainly decided by the development and run-time environments and the type of software entity that is connected to the variation point. For example, if the variation point is implemented using lines of code or classes it is typically open during detailed design and implementation, but closed during all other phases. However, if the variation point is implemented in components it can be e.g. open during architecture design and during run-time but closed during detailed design, implementation and compilation.

An important factor to consider is when linking is performed. If linking can only be done in conjunction with compilation, then this closes all variation points for population at this phase. If the system supports dynamically linked libraries, the variation points can remain open even during run-time.

When a variation point is open or closed may also be determined by which type of application the system is. For example, embedded software may have fewer means for populating a variation point once the system is delivered, and it may be undesirable to be able to populate variation points in some other types of systems once it has been delivered (e.g. for security reasons). Hence, when the population phase is open or closed is determined from both a technical perspective (i.e. the technical limitations of the chosen implementation technique for a variant feature) and as a design decision (i.e. when one *desires* the variation points to be open or closed).

How to Populate. Depending on how a variation point is implemented the population is either done *implicitly* or *explicitly.* If the population is implicit the variation point itself has no knowledge of the available variants and the list of available variants is not represented as such in the system. While the variation point may in some cases include functionality to decide which variant to use there is no explicit list of all variants available. For example, an if-statement in the source code that decides which variant to use is an implicit variation point, since there are only two variants (the if-block and the else-block). With an implicit population, if there is a list and a description of the available variants it is maintained outside of the system, e.g. in a document describing how to instrument the variation point.

With an explicit population the list of available variants is manifested in the software system. This means that the software system is aware of all of the possible variants, can add to the list of available variants and possesses the ability to discern between them and by itself select a suitable variant during run-time. In the case of the if-statement mentioned above, this is not an explicit population since new variants (e.g. new else-blocks) cannot be added by the if-statement itself and in most cases not during runtime.

Another example of an implicit population is when selecting which operating system to build for, since this is a decision made by the application engineers outside of the software system. On the other hand, selecting between different e-mail editors (as in the example in Figure 2) is a decision controlled by the software system itself and may hence be an explicit population. In the first case the system need not in itself be aware for what operating systems it can be built, whereas in the latter case the system can be aware which e-mail editors are available, how to add a new e-mail editor, how to distinguish between the editors and which editor to use at any given moment.

The differences between an implicit and explicit population is that with an explicit population the set of variants is managed inside of the system. Connected with an internal binding (as discussed below, in Section 2.2.3), this enables the system to decide without extra help from the user which variant is most suitable at any given moment. It also enables the ability to add new variants to the system and decide between these without restarting. With an implicit population, the set of variants is managed outside of the system, e.g. by application engineers. There may be a manifestation of the population in the source code, e.g. in the form of an if-statement, but the set of variants cannot be extended without help from either an application engineer or an end user. In the latter case, the end user typically also need to actively decide between the variants when it is time to bind the variation point.

The decision on when and how to add variants is governed by the business strategy and delivery model for the products in the software product family. For example if the business strategy involves supporting late addition of variants by e.g. third party vendors, this constrains the selection of implementation techniques for the variation points as they may need to be open for adding new variants after compilation or possibly even during run-time. This decision also impacts whether or not the collection of variants should be managed explicitly or implicitly, which is determined based on how the third party vendors are supposed to add their variants to the system. Likewise, if the delivery model involves updates of functionality into a running system this will also impact the choices of implementation techniques for the variation points.

In addition, the development process and the tools used by the development company influence how and when to add variants. For example if the company has a domain engineering unit developing reusable assets, more decisions may be taken during the product architecture derivation, whereas another organization may defer many such decisions until compile or link-time.

Who populates the variant feature with variants. The decision as to who is allowed/enabled to populate a variant feature with variants is key to selecting the appropriate mechanism. Domain engineers may choose to provide application engineers with a fixed set of variants to choose from. However, they may also allow application engineers to create and add their own product specific variants. Additionally, there is an increasing trend to provide variability to end users (e.g. plug-in mechanisms). However, one cannot expect end users to edit and compile source code[1] so any variability technique that requires this would be unsuitable for this kind of variant features. Similarly, domain engineers typically may want to shield application engineers from the complexity of the software product family and provide them with easy to use mechanisms instead. Depending on who needs to populate a variant feature with variants, some techniques are more suitable than others.

Example of Populating a Variant Feature. In the e-mail example, the set of protocols for receiving messages is fixed by the domain engineers. There is no requirement to be able to add new e-mail protocols in a running system. Hence, it is decided to deliberately lock the set of available variations during architecture design. This means that the variation points related to this variant feature can be closed at any point after this - the sooner the better. As the set of available variations is locked, the population can be implicit in the system. A simple if-statement can be used to decide which variant to use. For a variation point that is as static as this, i.e. where the set of variants is not expected to change much over time, and especially not during run-time, an implicit population is to prefer over an explicit population.

Another example is that many command-line email clients on UNIX offer end users the ability to select their own text editor for editing their messages. Typically such email clients include a variation point that allows end-users to select a suitable

1. Within the open source community, e.g. the Linux kernel, the border between an end user and a developer has historically not been distinct, so in many cases one has indeed expected the end users to be able to edit and compile source code. The trend in more recent releases of e.g. the Linux kernel is to move away from this method of supporting variability.

text editor program. In most cases, this will be implemented with an implicit population where the system only has an option with the path to the desired executable. If the system would implement this variant feature as an explicit population, it would include functionality to add editors, the paths of which are then maintained internally.

2.2.3 Binding to a Variant

The main purpose of introducing a variant feature is to delay a decision, but at some time there must be a choice between the variants and a single variant will be selected and used. We refer to this as *binding* the system to a particular variant. As before, this decision is concerned with *when* to bind and *how* to bind.

When to Bind. Binding can be done at several stages during the development and also as a system is being run. Decisions on binding to a particular variant can be expected during the following phases of a system's lifecycle:

- **Product Architecture Derivation.** The product family architecture typically contains many unbound variation points. The binding of some of these variation points is what generates a particular product architecture. Typically, configuration management tools are involved in this process, and most of the mechanisms are working with software entities introduced during architecture design. A good example of this is KOALA [24], an architecture description language that is used within Philips to describe the components in their product family and derive products.
- **Compilation.** The finalization of the source code is done during the compilation. This includes pruning the code according to compiler directives in the source code, but also extending the code to superimpose additional behavior (e.g. macros and aspects [7]).
- **Linking.** When the link phase begins and when it ends is very much depending on the programming and run-time environment being used. In some cases, linking is performed irrevocably just after compilation, and in some cases it is done when the system is started. In other systems again, the running system can link and re-link at will. In general there appears to be a trend to delay linking until run-time (even in embedded systems, where we have observed companies such as Axis (see Section 4.1), Symbian (see Section 4.4), and Danaher Motion Särö AB (see Section 4.5) that increasingly require binding of functionality during runtime in their embedded software). Consequently, any variability mechanism traditionally associated with linking is becoming available at run-time.
- **Run-time.** This type of binding is usually implemented by means of any standard object-oriented language. The collection of variants can be closed at run-time, i.e. it is not possible to add new variants, but it can also be open, in which case it is possible to extend the system with new variants at run-time. Such variants are normally referred to as plug-ins and these may often be developed by third party vendors. This type of run-time variability usually relies heavily on the linking mechanism on the platform. Another type of run-time binding is the interpretation of configuration files or startup parameters that determines what variant to bind to. This type of run-time binding is what is normally referred to as parameterization.

Note that binding times do not include the design and implementation phases. Variation points may well be introduced during these phases, but to the best of our knowledge a system can not be bound to a particular variant on other occasions than the ones presented above. This is because during these phases classes and variables are introduced, but there are no means available to select between them. It is possible to manually remove all but one variant in the design but this is part of the product architecture derivation. Hence, the selection is either done as the architecture for the particular system is derived from the product family architecture or when the compiler, linker or run-time system selects a particular variant e.g. based on some constant or variable.

As with the adding of variants, the time when one wants to be able to bind the system (again, as for the adding of variants, this decision may in some instances be influenced by the type of system being built) constrains the selection of possible ways to implement a variation point. For a variant feature resulting in many variation points this results in quite a few problems. The

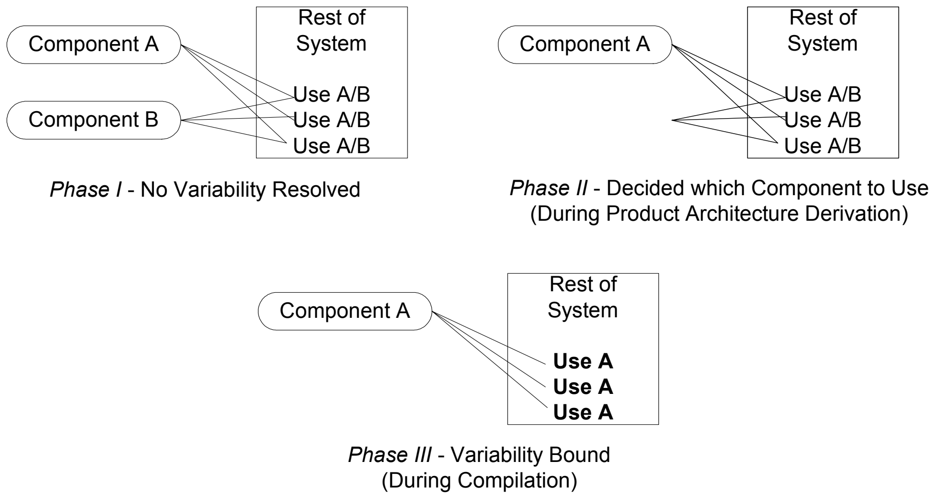

Component A
Component B
Rest of System
Use A/B
Use A/B
Use A/B
Phase I - No Variability Resolved
Component A
Rest of System
Use A/B
Use A/B
Use A/B
Phase II - Decided which Component to Use (During Product Architecture Derivation)
Component A
Rest of System
Use A
Use A
Use A
Phase III - Variability Bound (During Compilation)

Figure 4. Example of Phases in Committing to a Variant

variation points need to be bound either at the same time (as is the case if binding is required at run-time), or the binding of several variation points is synchronized so that for example a variation point that is bound during compilation binds to the same variant that related variation points have already bound to during product architecture derivation.

Consider, for example, a variant feature consisting of a component and some lines of code in other components where the variant feature is called from. Each variant has its own component implementation, and the lines of code in the other components are also different to make best use of the variant. Which component to use is decided during product architecture derivation, but which lines of code to use in the other component cannot be determined at this stage as lines of code are not in focus during this stage. Instead, this has to be deferred until compilation where the source code can be pruned from all but the chosen variant. This example is illustrated in Figure 4. We see the three phases of this particular variant feature implementation. In the first phase, both variants (Component A and B) are available, and the variability points include code to use either A or B. In the second phase, the architecture is pruned of Component B, i.e. it is decided to use Component A. In the third and last phase in this example the variability points are during compilation bound to use Component A.

When determining when to bind a variant feature to a particular variant, what needs to be considered is when binding is absolutely required. As a rule of thumb, one can in most cases say that the later the binding is done, the more costly (e.g. in terms of performance or resource consumption) it is. Deferring binding from product architecture derivation to compilation means that developers need to manage all variants during implementation, and deferring binding from compilation to run-time means that the system will have to include binding functionality. This introduces a cost in terms of e.g. performance to conduct the binding. Moreover, there are security aspects to consider when allowing run-time binding, and the memory footprint of the application also increases.

How to Bind. The other aspect of binding is to decide whether or not the binding should be done *internally* or *externally.* An internal binding implies that the system contains the functionality to bind to a particular variant. This is typically true for the binding that is done during a system's run-time. An external binding implies that there is a person or a tool external to the system that performs the actual binding. This is typically true for the binding that is done during product architecture derivation, compilation and linking, where tools such as configuration management tools, compilers and linkers perform the actual binding.

These two concepts are related but not equivalent to the implicit and explicit concepts introduced earlier. The different alternatives can be combined in four different ways:

- **Implicit population and internal binding.** Variation points of this type are typically bound during run-time. The system contains functionality to bind to a variant, but does not contain functionality to manage or extend the list of possible variants. The e-mail application and the e-mail retrieval protocols discussed earlier is an example of this. The system need not explicitly manage the set of variants, but needs to by itself be able to select one of the variants each time e-mail is retrieved.

- **Implicit population and external binding.** This combination is predominant for variation points that are bound in the development phases, i.e. all phases except for run-time and the linking performed at run-time. The resulting system need not know what different variants exist, as it is not concerned with the actual binding process. This is done by the application or domain engineers e.g. by using configuration management tools, compilers and linkers. For example, consider an application that allows compilation for both a Windows and a Unix platform. Once the compiler knows what variant is required, the system itself need never be made aware that it could have been compiled for another platform as well.
- **Explicit population and internal binding.** This is typically used for variation points that can be both populated and bound at run-time. With this combination the system is aware of what variants exist, and have sufficient knowledge to select between them. For example, consider an e-mail application that allows a user-defined text editor that is then launched automatically when needed. The set of available text editors is not fixed, and can be extended during run-time.
- **Explicit population and external binding.** This combination involves including in the running system the ability to discern between variants and determine which variant is most suitable and then let the users perform the binding manually. We do not think that this combination is very common or even likely. Normally, the end user asks the computer to perform tasks, not the other way around.

Whether to bind internally or externally is decided by many things, such as whether the binding is done by the software developers or the end users, and whether the binding should be made transparent to the end users or not. Moreover, an external binding is sometimes preferred as it does not necessarily leave any traces in the source code which may simplify the work of e.g. product engineers. When the binding is internal the system must contain functionality to bind and this may increase the complexity of the source code.

Example of Binding to a Variant. In the e-mail example, it is decided during run-time what e-mail retrieval protocol to use. This means that the binding time is set to run-time. Moreover, as the system decides by itself between the different variants, the binding is done internally.

2.2.4 Summary of Example

The running example used in this section can thus be summarized as follows:

- It is *identified* from the feature graph in Figure 2 that there is a choice between e-mail retrieval protocols. At this point, the variant feature “retrieval protocols” is *implicit*.
- As the system is designed, it is decided by the domain engineers that the e-mail retrieval protocols can be implemented as component implementations. The number of variation points can thus be reduced to one, i.e. the point where it is decided which variant to use. This need not be done for every point where the e-mail retrieval protocols are used in the system At this point, the variant feature “retrieval protocols” is said to be *introduced*.
- Also during architecture design, the component “e-mail retrieval protocol” is designed, as are the two component implementations, POP3 and IMAP. We have thus *populated* the variant feature. The population is thus done during architecture design. Moreover, it is decided that it is sufficient if the population is implicit, i.e. we need not include functionality to manage the list of available variants. The actual implementation of the two component implementations is of lesser importance from a variability perspective. They can be created during the general implementation of the system, be reused from a previous project, developed in parallel or bought as COTS components. Accordingly, this task is handed over to the application engineers.

- The last part of constraining the variant feature is to decide when it should be possible to *bind* the system to a variant. For the e-mail retrieval protocols, this cannot be decided until during run-time, for each time that e-mails are retrieved. The end users should not have to be involved in the binding, so it is decided to keep the variation point internal, i.e. it is the system that binds without any help from the outside.

The last two parts of this summary, i.e. the population and the binding, may require further discussion. Starting with the "when"-part of the decision. When to populate determines when it should be possible to add new variants, i.e. software entities implementing each variant. This is not connected to when the variants are implemented or when the variation points are introduced and implemented. When one wishes to bind determines when the variation points should be active. Again, this is not directly connected to when the variation points are introduced and implemented. Part of our argumentation is that it is the different aspects covered in this chapter that determines when and how to introduce and implement variation points, and not the other way round.

The second part of the decision, the "how"-part, is in both of the last two steps involved with the variation point but with two different perspectives. The decision on how to populate (i.e. implicitly or explicitly) determines how much the variation points need to be aware of the variants. This ranges from cases where there is no manifestation at all in the source code of the variation point, to where there is source code to decide between the variants but with no "awareness" of the different variants, and all the way up to variation points where there is source code in place to explicitly manage the set of variants and decide which to use. The decision on how to bind (i.e. internally or externally) is related to this, as described in Section 2.2.3. In the example above, the population is implicit and the binding is internal.

2.3 Implementing Variability

Based on the previous constraint of variability a suitable *variability realization technique* may be selected from our taxonomy (Section 3) for the variation points pertaining to a certain variant feature. The selected realization technique should strike the best possible balance between the constraints that have been identified in the previous step. Using our process this choice may be a reasonably informed choice. We believe that making the right choices with respect to variability is vital for the survival of a product family. Being too conservative with respect to variability may lead to a situation where new features can either not be implemented at all, or only at a very high cost. On the other hand adding too much variability makes the product family more complex which may result in maintainability problems and may also result in higher product derivation cost. Additionally, having too many variation points may lead to architecture drift [25] and design erosion [26].

Example of Implementing Variability. In the e-mail example used in the previous section, this step consists of deciding which variability realization technique to use for the variation point that decides which e-mail protocol to use when retrieving e-mails. Browsing through the taxonomy in Section 3, and especially Table 2 we see that we are in fact working with framework implementations (the framework is "e-mail retrieval protocols", and the two implementations are POP3 and IMAP), and the binding time is run-time. This means that we have two variability realization techniques available, "Run-time Variant Component Specializations" and "Variant Component Implementations". For this variation point the "Variant Component Implementations" is the variability realization technique that is the most suitable.

2.4 Managing the Variability

The last step is to manage the variability. This involves maintenance and to continue to populate variant features with new variants and pruning old, no longer used, variants. Moreover, variant features may be removed altogether, as the requirements change, new products are added and old products are removed from the product family. Management also involves the distribution of new variants to the already installed customer base, and billing models regarding how to make money off new variants. As with the identification of variability, this is outside the scope of this paper.

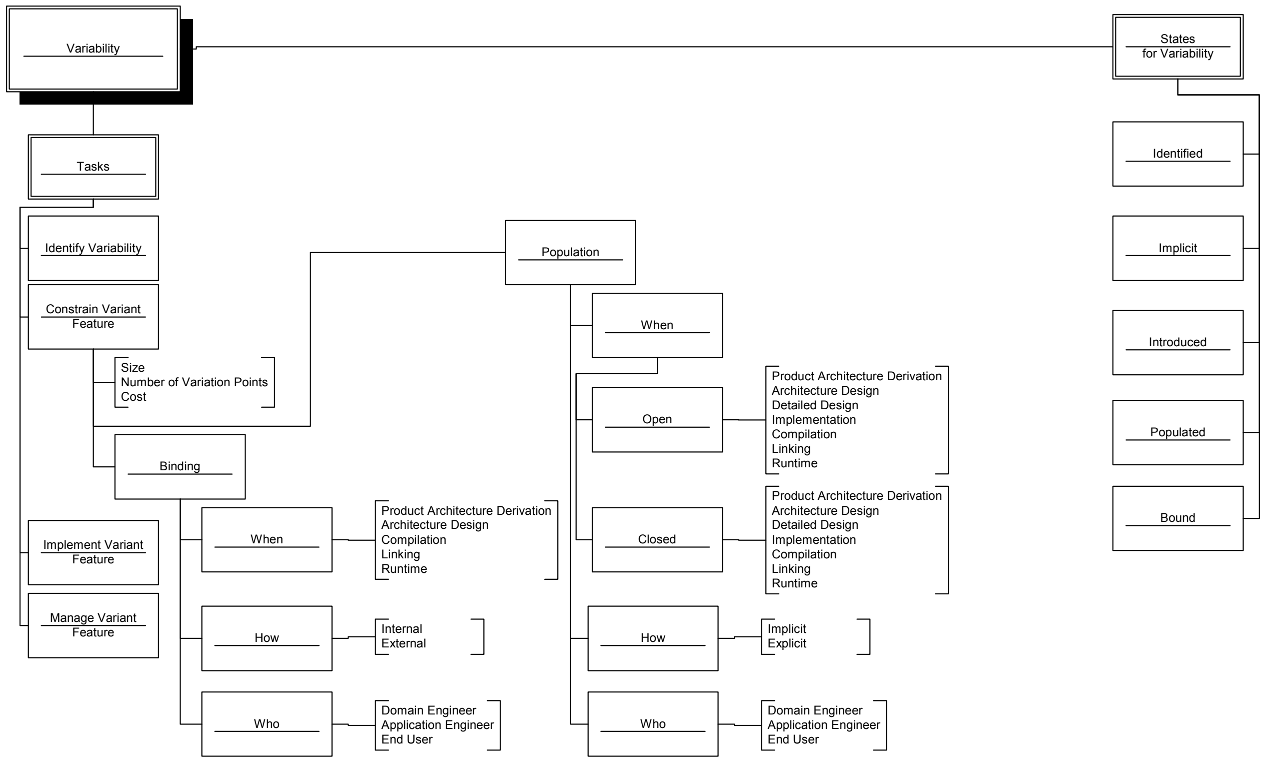


Figure 5. Summary of Discussed Concepts

2.5 Summary

Figure 5 illustrates the different tasks discussed in this section that together decide where and how to implement an identified variant feature. Especially important for this decision are the two tasks identify variability and constrain variant feature. In this paper we focus on the aspects needed to constrain a variant feature. This depends on a number of aspects, e.g. the *size* of the variant feature, the desired *number of variation points*, the *cost* for managing the variant feature, when how and by whom to *populate* the variant feature with variants, and when, how and by whom the system is *bound* to a particular variant.

During these tasks, the variant features undergo a number of transformation, also depicted in Figure 5. These transformations range from when a variant feature is first *identified*, thus being *implicit* until it is *introduced* (this follows after the task constrain variant feature and during the task implement variant feature). During the task implement variant feature the variation points are introduced into the software system, and variants for the variant feature are implemented. The variant feature is then *populated* with these variants, and eventually the variant feature is *bound* to a particular variant.

3. Variability Realization Techniques

In this section, we present a taxonomy of different ways to implement the variation points for a variant feature, which we refer to as *variability realization techniques*. If the variation points are constrained, e.g. as described in the previous section, it enables a more informed choice of how to implement them. The taxonomy in this section is structured according to the criteria set up in the previous section. The taxonomy is based on the different realization techniques we have observed in several industry cases. Section 4 presents the industry cases in which the variability realization techniques have been observed along with some additional examples. We encourage readers to extend this taxonomy with variability realization techniques that may be used in companies other than those we have studied.

The variability realization techniques in our taxonomy are summarized in Table 2. In this table the variability realization techniques are organized according to the software entity the variability realization techniques work with and when it is at the latest possible to bind them. For each variability realization technique there is also a reference to a more detailed description of the technique. These descriptions are presented below. There are some areas in this table that are shaded where we perceive that it is not interesting to have any variability realization techniques. These areas are:

- Components and Frameworks during compilation, as compilation works with smaller software entities. This type of software entities comes into play again only during linking.
- Lines of Code during Product Architecture Derivation, as we know of no tools working with product architecture derivation that also work with lines of code.
- Lines of Code during Linking, as linkers work with larger software entities.

We present the variability realization techniques using a Design Pattern like form, in the style used by e.g. Buschmann et al. [27] and Gamma et al. [15]. For each of the variability realization techniques we discuss the following topics:

- **Intent.** This is a short description of the intent of the realization technique.
- **Motivation.** A description of the problems that the realization technique address and other forces that may be at play.
- **Solution.** Known solutions to the problems presented in the motivation section.
- **Lifecycle.** A description of when the realization technique is open, when it closes, and when it allows binding to one of the variants.
- **Consequences.** The consequences and potential hazards of using the realization technique.
- **Examples.** Some examples of the realization technique in use at the companies in which we have conducted case studies.

It should be noted that the techniques described in our taxonomy have deliberately been described in an abstract fashion. Our reason for this is that we wish to abstract from specific programming languages and yet capture the commonalities between similar mechanisms in languages. The number of specific language features (and their numerous implementations) that would have to be taken into account is so large that it distracts from the essence of our taxonomy. Moreover, our taxonomy also takes into account software entities (e.g. architecture components) that are currently poorly supported in programming languages.

We would like to emphasize that we do not present a pattern collection such as Reference [15] or [27] do - we only borrow the presentation form from these pattern collections. The goal of the presented taxonomy differs since we present specific technical solutions (albeit in an abstract form) rather than the type of abstract solutions proposed in Reference [15] and [27]. Another goal of the taxonomy is to be able to reason about the specific properties of the listed solutions (e.g. binding time) and to outline benefits and liabilities of different variability realization techniques. We would also like to stress that although we only present the techniques from the perspective of variability, many of the techniques may also serve other quality attributes. Moreover, implementing support for some quality attributes (e.g. portability) requires supporting flexibility or variability in the software, which can be provided using the variability realization techniques presented in this taxonomy.

Table 2: Variability Realization Techniques

Involved Software Entities	**Binding Time**			
	Product Architecture Derivation	**Compilation**	**Linking**	**Run-time**
Components Frameworks	Architecture Reorganization (Section 3.1)	N/A	Binary Replacement - Linker Directives (Section 3.4)	Infrastructure-Centered Architecture (Section 3.6)
	Variant Architecture Component (Section 3.2)			
	Optional Architecture Component (Section 3.3)		Binary Replacement - Physical (Section 3.5)	
Component Implementations Framework Implementations Classes	Variant Component Specializations (Section 3.7)	Code Fragment Superimposition (Section 3.13)	Binary Replacement - Linker Directives (Section 3.4)	Run-time Variant Component Specializations (Section 3.9)
	Optional Component Specializations (Section 3.8)		Binary Replacement - Physical (Section 3.5)	Variant Component Implementations (Section 3.10)
Lines of Code	N/A	Condition on Constant (Section 3.11)	N/A	Condition on Variable (Section 3.12)
		Code Fragment Superimposition (Section 3.13)		

3.1 Architecture Reorganization

Intent. Support several product specific architectures by reorganizing (i.e. changing the architectural structure of components and their relations) the overall product family architecture.

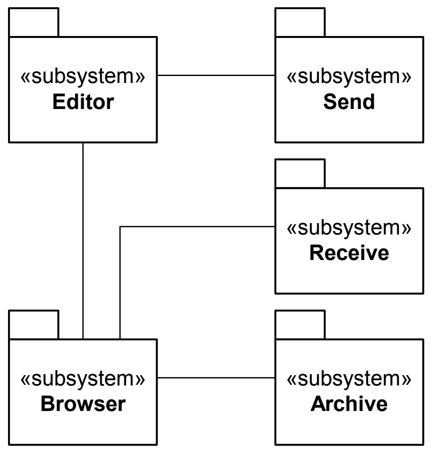
Motivation. Although products in a product family share many components, the control flow and data flow between these components need not be the same. Therefore, the product family architecture is reorganized to form the concrete product architectures. This involves mainly changes in the control flow, i.e. the order in which components are connected to each other, but may also consist of changes in how particular components are connected to each other, i.e. the provided and required interface of the components may differ from product to product.

Solution. In this realization technique, the components are represented as subsystems controlled by configuration management tools or, at best, *Architecture Description Languages (ADL)* (e.g. Koala [24], XVCL [28], or any of the ADL's mentioned in Reference [29]). Which variants are included in a system is determined by the configuration management tools. Some variation points may be resolved at this level, as the selected components may impose a certain architecture structure. Typically this technique also requires variation points that are in focus during later stages of the development cycle in order to work. These later variation points can e.g. be used to connect the components properly. This technique is implicit and external, as there is no first-class representation of the architecture in the system. For an explicit realization technique, see Infrastructure-Centered Architecture.

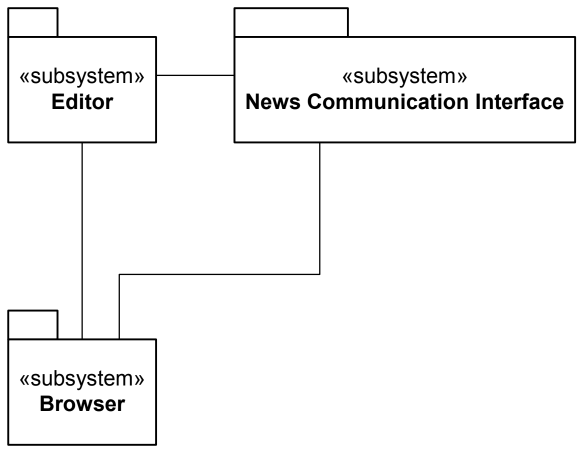
Lifecycle. This technique is open for the adding of new variants during architecture design, where the product family architecture is used as a template to create a product specific architecture, i.e. during product derivation. As detailed design commences, the architecture is no longer a first class entity, and can hence not be further reorganized. Binding time, i.e. when a particular architecture is selected, is when a particular product architecture is derived from the product family architecture. This also implies that this is not a technique for achieving dynamic architectures. If this is what is required, see Infrastructure-Centered Architecture. Moreover, as this realization technique typically relies on other variation points in focus during later stages of the development, the architecture is not entirely committed until these variation points are also bound.

Consequences. The product family architecture is kept as an abstract template, from which concrete product architectures are derived. This facilitates development of the individual products. However, this also introduces a risk that part of the development made for a particular product is not compatible with the rest of the products in the product family. Secondly, this variability realization technique may require variation points in subsequent development phases as well, in order to function properly. What we've seen at several companies (e.g. Philips [24]) that applies this technique is that the organization is changed in such a way that product engineers are not allowed to change components. If a certain change is required, product engineers have to file a change request with the domain engineers.

Examples. A design of the e-mail application used as example earlier is presented in Figure 6a. To create a USENET news-application, the architecture is reorganized to that of Figure 6b. As can be seen, some components and some connectors are removed or joined and other components and connectors are removed altogether. Most of this is due to that the communication standard for news (NNTP) provides a uniform interface for both sending and receiving messages whereas e-mail has several different protocols both for sending and for receiving messages. Another example of this technique is Axis Communications [5] where a hierarchical view of the Product Family Architecture is employed, where different products are grouped in subtrees of the main Product Family. To control the derivation of one product out of this tree, a rudimentary, in-house developed, ADL is used. Another example is Philips that uses a proprietary ADL called Koala to derive products [24].



(a) an E-mail client



(b) an USENET News application

Figure 6. Design of Example E-mail Application

Table 3: Summary of Architecture Reorganization

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

3.2 Variant Architecture Component

Intent. Support several, differing, architecture components representing the same conceptual entity.

Motivation. In some cases, an architecture component in one particular place in the architecture can be replaced with another that may have a differing interface, and sometimes also representing a different domain. This need not affect the rest of the architecture. Using the e-mail application example from the previous section, this product can quite easily be changed into a USENET News reader application. This, however, requires that some components work quite differently. By replacing the components for the e-mail application with those for the USENET news application we use this realization technique to cope with this variability.

Solution. The solution to this is to, as the title implies, support a set of components, each implementing one variant of the variant feature. The selection of which component to use any given moment is then delegated to the *configuration management tools* that select what component to include in the system. a part of the solution is also delegated to subsequent development phases, where the Variant Component Specialization will be used to call and operate the different components in the correct way. To summarize, this technique has an implicit collection, and the binding functionality is external.

Lifecycle. It is possible to add new variants, implemented as components, during architecture design, when new components can be added, and also during detailed design, where these components are concretely designed as separate components in the architecture. The architecture is bound to a particular component during the transition from a product family architecture to a product architecture, when the configuration management tool selects what architecture component to use.

Consequences. A consequence of using this pattern is that the decision of what component interface to use, and how to use it, is placed in the calling components rather than where the actual variant feature is implemented. Moreover, the handling of the differing interfaces cannot be coped with during the same development phase as the varying component, but has to be deferred until later development stages.

Examples. In the e-mail application example, there may be a need to replace the browser component with one that is specific for the news application, supporting e.g. a nested view of the messages. This can be done using this variability realization technique by providing two components: one e-mail browser and one news browser and then let the configuration management tool decide which to include in the system being built. Another example is Axis Communications [5], where there existed during a long period of time two versions of a file system component; one supporting both read and write functionality, and one supporting only read functionality. Different products used either the read-write or the read-only component. The reason for maintaining two components like this is, in Axis' case, e.g. that there is a limited amount of memory in their embedded products. Since the two components differ in the interface as well as the implementation, they are, in effect, two different architecture components.

Table 4: Summary of Variant Architecture Component

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

3.3 Optional Architecture Component

Intent. Provide support for a component that may, or may not be present in the system.

Motivation. Some architecture components may be present in some products but absent in other. For example, the e-mail application described earlier may optionally provide functionality to manage a contact list. This means that in some configurations e.g. the "compose e-mail" functionality needs to interact with the contact list, and in other configurations this interaction is not available or even possible.

Solution. There are two ways of solving this problem depending on whether it should be fixed on the calling side or the called side. If we desire to implement the solution on the calling side the solution is simply delegated to variability realization techniques introduced during later development phases. To implement the solution on the called side, which may be nicer but is less efficient, create a "null" component. This is a component that has the correct interface, but replies with dummy values. This latter approach assumes, of course, that there are predefined dummy values that the other components know to ignore. This null component is then included in the product configuration during product architecture derivation. The binding for this technique is done external to the system, using e.g. configuration management tools.

Lifecycle. This technique is open when a particular product architecture is designed based on the product family architecture, but due to the lack of architecture representation during later development phases is closed at all other times. The architecture is bound to the existence or non-existence of a component when a product architecture is selected from the product family architecture.

Consequences. Consequences of using this technique are that the components depending on the optional component must either have realization techniques to support its not being there, or have techniques to cope with dummy values. The latter technique also implies that the "plug", or the null component, will occupy space in the system, and the dummy values will consume processing power. An advantage is that should this variation point later be extended to be of the type variant architecture component, the functionality is already in place, and all that needs to be done is to add more variants for the variant feature.

Examples. In the e-mail application example and the architecture reorganization in Figure 6, the "Archive" component is removed in the news application. This is an example of where the variability is solved on the calling side, as the browser component need to be aware of whether there is an archive component or not. If we had included a null component, this would have been entirely transparent for the browser component. More examples include a Storage Server at Axis Communications which can optionally be equipped with a so-called hard disk cache. This means that in one product configuration, other components need to interact with the hard disk cache, whereas in other configurations, the same components do not interact with this architecture component. Also, in Symbian's EPOC operating system (which has since our case study been renamed SymbianOS) [4][30], the presence or absence of a network connection decides whether network drivers should be loaded or not. The web browser Mozilla comes with a null component in the plug-in directory (npnul32.dll) that is called in the absence of a plug-in to call when an embedded object is encountered. Curiously, there even exist replacements for it (e.g. DefaultPluginPro, an alternative that offers users a save option to save the object for which no plug-in could be found).

Table 5: Summary of Optional Architecture Component

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

3.4 Binary Replacement - Linker Directives

Intent. Provide the system with alternative implementations of underlying libraries.

Motivation. In some cases, all that is required to support a new platform is that an underlying system library is replaced. For example, when compiling a system for different UNIX-dialects, this is often the case. It does not need to even be a system library, it can also be a library distributed together with the system to achieve some variability. For example some games have been observed to be released with different libraries to work with different graphics environments such as the window system (e.g. X-Windows), an OpenGL graphics device or a standard SVGA graphics device.

Solution. Represent the variants as stand-alone library files, and instruct the linker which file to link with the system. If this linking is done at run-time, the binding functionality must be internal to the system, whereas it can, if the linking is done during the compile and linking phase prior to delivery, be external and managed by a traditional linker. An external binding also implies, in this case, an implicit collection. Naturally, this assumes that all the available libraries conform to a common interface. If this is not the case the switch to another library will not work as seamlessly.

Lifecycle. This technique is open for adding new variants as the system is linked. It is also bound during this phase. As the linking phase ends, this technique becomes unavailable. However, it should be noted that the linking phase does not necessarily end. In modern systems (e.g. Java based systems), linking is also available during execution.

Consequences. This is a fairly well developed variability realization technique, and the consequences of using it are relatively harmless. Security is a concern since this technique means that there's a trust issue if you make a call to a library. Advantages are that it is relatively easy to use and it allows the memory footprint of the executing application to be reduced by simply not loading any unwanted libraries.

A major disadvantage is a situation that is generally described as DLL hell. Multiple applications may require different versions of the same DLL. In the worst case, system stability may degrade because applications link to different versions of DLLs than they were tested with. Particularly older versions of MS Windows are vulnerable to this type of problems

Examples. In the e-mail application example the situation is as illustrated in Figure 7 where there are two sets of libraries to use, one for Windows and one for UNIX. Which set of libraries to use is decided as the system is linked by the application engineers before the system is delivered to customers. Another example is the Mozilla web browser (http://www.mozilla.org)

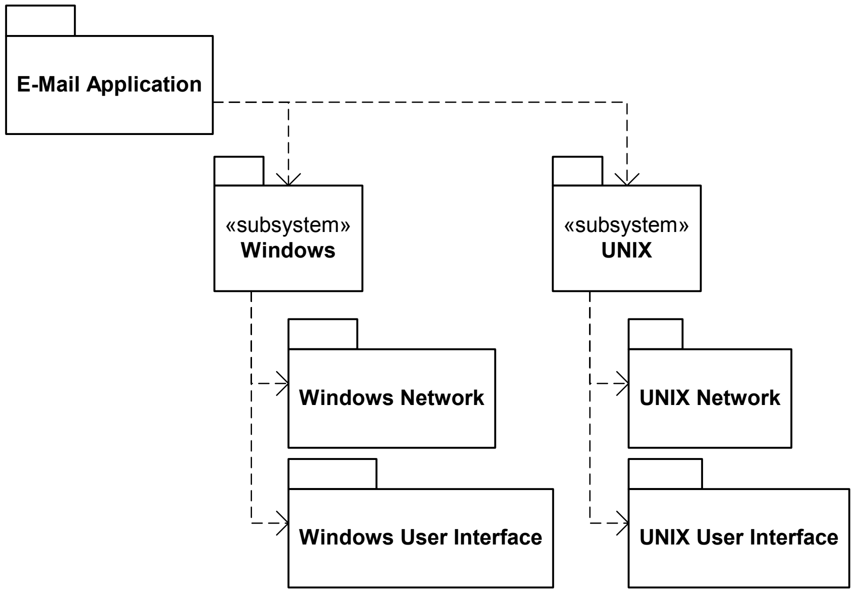


Figure 7. E-mail application with Two Different Sets of Libraries

that is available on many platforms, including Linux. Some versions of Linux can display so-called anti-aliased fonts that look smoother on LCD screens. However, this requires that programs that wish to use this link to specific system libraries. To support this in the Mozilla browser, an (at the moment of writing) experimental build option exists that creates a version of Mozilla that uses these libraries.

Table 6: Summary of Binary Replacement - Linker Directives

Introduction Times	Architecture Design
Open for Adding Variants	Linking
Collection of Variants	Implicit or Explicit
Binding Times	Linking
Functionality for Binding	External or Internal

3.5 Binary Replacement - Physical

Intent. Facilitate the modification of software after delivery.

Motivation. Unfortunately, very few software systems are released in a perfect and optimal state. This creates a need to upgrade the system after delivery. In some cases these upgrades can be done using the variation points already existing in the system, but in others the system does not currently support variability at the places needed.

Solution. In order to introduce a new variation point after delivery, the software binary must be altered. The easiest way of doing this is to replace an entire file with a new copy. To facilitate this replacement the system should thus be organized as a number of relatively small binary files, to localize the impact of replacing a file. Furthermore the system can be altered in two ways: Either the new binary completely covers the functionality of the old one or the new binary provides additional functionality in the form of, for example, a new variant feature using other variability realization techniques. In this technique the collection is implicit, and the binding is external to the system.

Lifecycle. This technique is bound before start-up (i.e. before run-time) of the system. In this technique the method for binding to a variant is also the one used to add new variants. After delivery (i.e. after compilation), the technique is always open for adding new variants.

Consequences. If the new binary does not introduce “traditional” variation points, the same technique will have to be used again the next time a new variant for the variant feature in question is detected. However if traditional variation points are introduced this facilitates future changes at this particular point in the system. Replacing binary files is normally a volatile way of upgrading a system, since the rest of the system may in some cases even be depending on e.g. software bugs in the replaced binary in order to function correctly (please note that bug fixing is not the only usage of this technique. One may just as well use it to extend the functionality of a product). Moreover, it is not trivial to maintain the release history needed to keep consistency in the system. Furthermore, there are also some trust issues to consider here, e.g. who provides the replacement component and what are the guarantees that the replacement component actually does what it is supposed to do.

Examples. The e-mail application example can be upgraded after delivery to support editing e-mails in HTML, as opposed to the default text only editor. When the end user installs the upgrade, the binary file containing the previous editor component is overwritten with a new file, containing the HTML editor component. This technique is particularly popular with hardware manufacturers such as Axis Communications that provide a possibility to upgrade the software in their devices by re-flashing the ROM [5]. This basically replaces the entire software binary with a new one. Another example is the Bios software that is located on the motherboard of a PC. Typically, manufacturers offer binary replacements for this software on their homepage to fix bugs and add compatibility with hardware that became available after the product was sold.

Table 7: Summary of Binary Replacement - Physical

Introduction Times	Architecture Design
Open for Adding Variants	After Compilation
Collection of Variants	Implicit
Binding Times	Before Run-time
Functionality for Binding	External

3.6 Infrastructure-Centered Architecture

Intent. Make the connections between components a first class entity.

Motivation. Part of the problem when connecting components (in particular components that may vary) is that the knowledge of the connections is often hard coded in the required interfaces of the components and is thus implicitly embedded into the system. A reorganization of the architecture, or indeed a replacement of a component in the architecture, would be vastly facilitated if the architecture is an explicit entity in the system, where such modifications could be performed.

Solution. Convert the connectors into first class entities, so the components are no longer connected to each other but are rather connected to the infrastructure, i.e. the connectors. This infrastructure is then responsible for matching the required interface of one component with the provided interface of one or more other components. The infrastructure can either be an existing standard, such as COM or CORBA [31], or it can be an in-house developed standard such as Koala [24]. The infrastructure may also be a *scripting language* in which the connectors are represented as snippets of code that are responsible for binding the components together in an architecture. These code snippets can either be done in the same programming language as the rest of the system, or they can be done using a specialized scripting language. Such scripting languages are, according to e.g. Reference [32], highly suitable for "gluing" components together. The collection of variants is in this realization technique either implicit or explicit and the binding functionality is internal, provided by the infrastructure.

Lifecycle. This technique can be implemented in many ways and this governs e.g. when it is open for populating with variants. In some cases the infrastructure is open for the addition of new components as late as during run-time and in other cases the infrastructure is concretized during compile and linking, and is thus open for new additions only until then. Binding can also be performed during compile and linking, even if it is more likely that this is deferred until run-time.

Consequences. When used correctly, this realization technique yields perhaps the most dynamic of all architectures. Performance is impeded slightly because the components need to abstract their connections to fit the format of the infrastructure, which then performs more processing on a connection before it is concretized as a traditional interface call again. In many

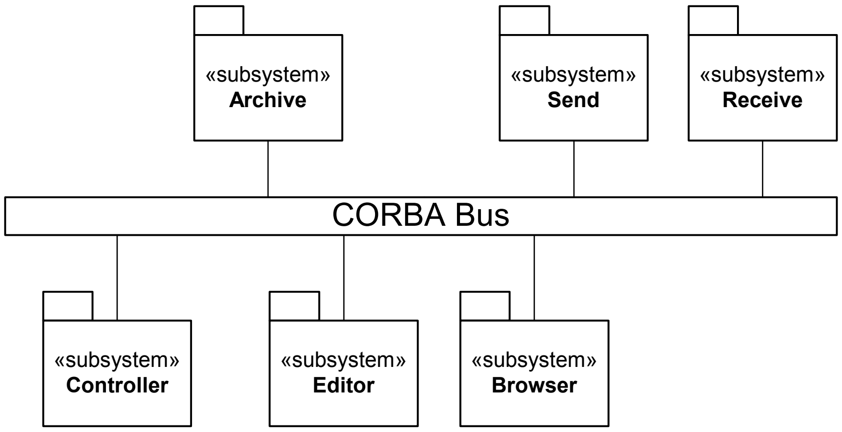


Figure 8. An E-mail application using a CORBA bus

ways, this technique is similar to the Adapter Design Pattern [15]. As complexity is moved from the components to the glue code, this may in some cases have a negative impact on maintainability. On the other hand, the components become more focused on their actual task and hence easier to maintain.

The infrastructure does not remove the need for well-defined interfaces, or the troubles with adjusting components to work in different operating environments (i.e. different architectures), but it removes part of the complexity in managing these connections.

Examples. An alternative to the design of the e-mail application in Figure 6 is to design it around a CORBA bus (http://www.omg.org), as illustrated in Figure 8. Advantages of this is that the infrastructure can take care of the connections between components, so that the components themselves need not be as dependent or aware of what other components are available in the system. Other examples are programming languages and tools such as Visual Basic, Delphi and JavaBeans that also support a component based development process where the components are supported by some underlying infrastructure. Another example is the Mozilla web browser, which makes extensive use of a scripting language, in that everything that can be varied is implemented in a scripting language and only the atomic functionality is represented as compiled components. It is our experience that most larger systems we encounter in industry either use an off the shelf component architecture or an in house developed component architecture (e.g. Philips [24] and Axis [5]).

Table 8: Summary of Infrastructure-Centered Architecture

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design Linking Run-time
Collection of Variants	Implicit or Explicit
Binding Times	Compilation Run-time
Functionality for Binding	Internal

3.7 Variant Component Specializations

Intent. Adjust a component implementation to the product architecture.

Motivation. Some variability realization techniques on the architecture design level require support in later stages. In particular those techniques where the provided interfaces vary need support from the required interface side as well. In these cases, what is required is that parts of a component implementation, namely those parts that are concerned with interfacing a component representing a variant of a variant feature, need to be replaceable as well. This technique can also be used to tweak a component to fit a particular product's needs. It is desirable that there are no traces left of the variability realization technique once

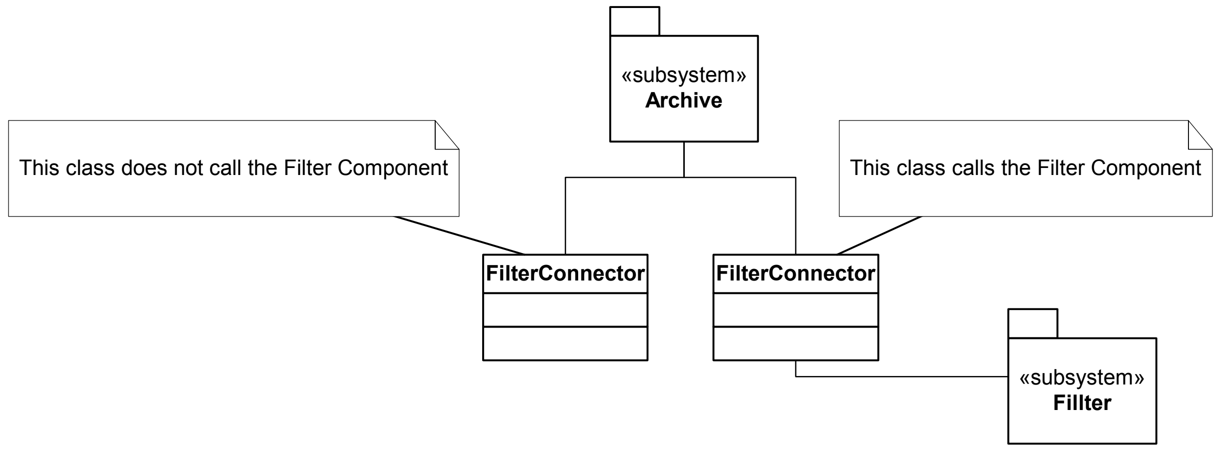

«subsystem»
Archive
This class does not call the Filter Component
This class calls the Filter Component
FilterConnector
FilterConnector
«subsystem»
Fillter

Figure 9. Example of Variant Component Specialization

it is bound, thus presenting an unobstructed view for subsequent development phases. Hence, the technique needs to work during product architecture derivation, i.e. at the same time as the variability realization techniques it works together with.

Solution. Separate the interfacing parts into separate classes that can decide the best way to interact with the other component. Let the configuration management tool decide what classes to include at the same time as it is decided what variant of the interfaced component to include in the product architecture. This technique has an implicit collection, and external binding functionality.

Lifecycle. The available variants are introduced during detailed design, when the interface classes are designed. The technique is closed during architecture design, which is unfortunate since it is here that it is decided that the variability realization technique is needed. This technique is bound when the product architecture is instantiated from the source code repository.

Consequences. A consequence of using classes is that it introduces another layer of indirection, which may consume processing power (Although today the overhead incurred by an extra layer of indirection is minimal.). It may not always be a simple task to separate the interface. Suppose that the different variants require different feedback from the common parts, then the common part will be full with method calls to the varying parts, of which only a subset is used in a particular configuration. Naturally this hinders readability of the source code. However, the use of classes like this has the advantage that the variation points are localized to one place (i.e. a single class) in the source code, which facilitates adding more variants and maintaining the existing variants.

Examples. In the example e-mail application, one version of the product is delivered with the functionality to filter (i.e. process in various ways) e-mails as they arrive. This is done in a separate component connected to the Archive component (see Figure 6). This means that the Archive component needs to be made aware whether the filter component is available. At the same time, once this decision is made it is not necessary to clutter the source code with an already bound variability realization technique. This is solved using two separate class implementations bearing the same name, and then letting the configuration management tool decide which class to include in the product to build. Figure 9 illustrates this situation. We have also observed this variability realization technique in the Storage Servers at Axis Communications [5], that can be delivered with a traditional cache or a hard disk cache. The file system component must be aware of which is present, since the calls needed for the two are slightly differing. Thus, the file system component is adjusted using this variability realization technique to work with the cache type present in the system.

3.8 Optional Component Specializations

Intent. Include or exclude parts of the behavior of a component implementation.

Table 9: Summary of Variant Component Specialization

Introduction Times	Detailed Design
Open for Adding Variants	Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

Motivation. A particular component implementation may be customized in various ways by adding or removing parts of its behavior. For example, depending on the screen size an application for a handheld device can opt not to include some features, and in the case when these features interact with others this interaction also needs to be excluded from the executing code.

Solution. Separate the optional behavior into a separate class and create a "null" class that can act as a placeholder when the behavior is to be excluded. Let the configuration management tools decide which of these two classes to include in the system. Alternatively surround the optional behavior with compile-time flags to exclude it from the compiled binary. Binding is in this technique done externally, by the configuration management tools or the compiler. Technically, this variability realization technique is similar to the Optional Architecture Component technique. The difference is just a matter of scale; the software entities involved in this technique are smaller.

Lifecycle. This technique is introduced during detailed design and is immediately closed to adding new variants, unless the variation point is transformed into a Variant Component Specialization. The system is bound to the inclusion or exclusion of the component specialization during the product architecture derivation or, if the second solution is chosen, during compilation.

Consequences. It may not be easy to separate the optional behavior into a separate class. The behavior may be such that it cannot be captured by a "null" class.

Examples. In the e-mail application example, this technique is used when the Archive component is not present, as in Figure 6. In this case, the Browser component needs to be aware that there is no Archive component present. All connections from the Browser component to the Archive component are localized into a single class, much the same as in the Variant Component Specialization example. In addition, a null class with the same name is created that simply acts as if there are no mail folders or stored messages available. Configuration management tools are used to decide which of these classes to include in the system. This variability realization technique was used when Axis Communications [5] added support for Novel Netware, some functionality required by the filesystem component was specific for Netware. This functionality was fixed external of the file system component, in the Netware component. As the functionality was later implemented in the file system component, it was removed from the Netware component. The way to implement this was in the form of an Optional Component Specialization.

Table 10: Summary of Optional Component Specialization

Introduction Times	Detailed Design
Open for Adding Variants	Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

3.9 Run-time Variant Component Specializations

Intent. Support the existence and selection between several specializations inside a component implementation.

Motivation. It is required of a component implementation that it adapts to the environment in which it is executing, i.e. that for any given moment during the execution of the system, the component implementation is able to satisfy the requirements

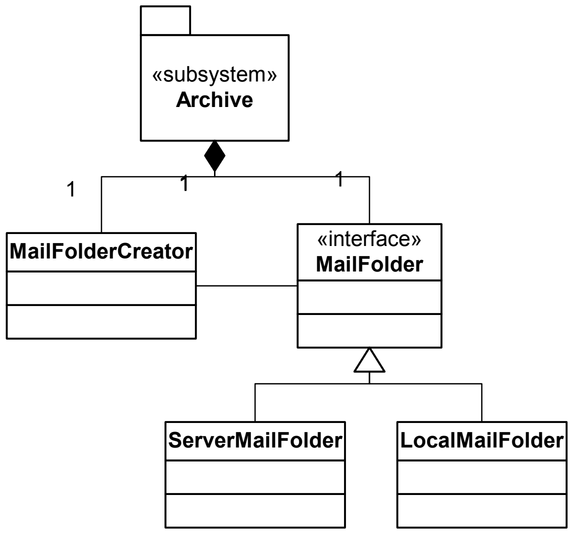


Figure 10. Example of Run-time Variant Component Specialization

from the user and the rest of the system. This implies that the component implementation is equipped with a number of alternative executions, and is able to, at run-time, select between these.

Solution. There are several *Design Patterns* [15] that are applicable here, for example Strategy, Template Method and Abstract Factory. Alternating behavior is collected into separate classes, and mechanisms are introduced to, at run-time, select between these classes. Using Design Patterns makes the collection explicit, and the binding is done internally, by the system. Design Patterns use language constructs such as *inheritance* and *polymorphism* to implement the variability. Alternatively, generative programming solutions such as e.g. C++ *templates* may also be used [8]. Essentially the mentioned design patterns work around the lack of such language features in some popular object oriented languages.

Lifecycle. This technique is open for new variations during detailed design since classes and object-oriented concepts are in focus during this phase. Because these are not in focus in any other phase this technique is not available anywhere else. The system is bound to a particular specialization at run-time, when an event occurs.

Consequences. Depending upon the ease with which the problem divides into a generic and variant part, more or less of the behavior can be kept in common. However the case is often that even common code is duplicated in the different strategies. A hypothesis is that this could stem from quirks in the programming language such as the self problem [33].

Examples. As the e-mail example application support two protocols, i.e. POP and IMAP, for receiving e-mails, the differences between these must be managed different parts of the system. For example, IMAP provides support for keeping and managing mail folders on the server, which POP does not. This means that the Archive component in Figure 6 needs functionality that is specific for IMAP and other similar protocols. This is solved by localizing these specific parts into separate classes, as illustrated in Figure 10. An abstract factory [15] class is used to instantiate the correct class when needed (Please note that this may be an oversimplification of how to design an e-mail application and how IMAP functions. The main purpose is to exemplify the variability realization technique, not to provide a complete and correct design of an IMAP e-mail client.). Another example is hand-held devices that can be attached to communication connections with differing bandwidths, such as a mobile phone or a LAN and this implies different strategies for how the EPOC operating system [30] retrieves data. Not only do the algorithms for, for example, compression differ, but on a lower bandwidth the system can also decide to retrieve less data, thus reducing the network traffic. This variant feature need not be in the magnitude of an entire component but can often be represented as strategies within the concerned components.

Table 11: Summary of Run-time Variant Component Specializations

Introduction Times	Detailed Design
Open for Adding Variants	Detailed Design
Collection of Variants	Explicit
Binding Times	Run-time
Functionality for Binding	Internal

3.10 Variant Component Implementations

Intent. Support several coexisting implementations of one architecture component so that each of the implementations can be chosen at any given moment.

Motivation. An architecture component typically represents some domain or sub-domain. These domains can be implemented using any of a number of standards and typically a system must support more than one simultaneously. For example an e-mail application may support several protocols for receiving e-mails, e.g. POP and IMAP, and is able to choose between these at run-time. Forces in this problem is that the architecture must support these different component implementations and other components in the system must be able to dynamically determine to what component implementation data and messages should be sent.

Solution. Implement several component implementations adhering to the same interface and make these component implementations tangible entities in the system architecture. There exists a number of Design Patterns [15] that facilitate this process. For example, the Strategy pattern is on a lower level a solution to the issue of having several implementations present simultaneously. Using the Broker pattern is one way of assuring that the correct implementation gets the data, as are patterns like Abstract Factory and Builder. Part of the flexibility of this variability realization technique stems from the fact that the collection is explicitly represented in the system, and the binding is done internally.

The decision on exactly what component implementations to include in a particular product can be delegated to configuration management tools.

Lifecycle. This technique is introduced during architecture design, but is not open for addition of new variants until detailed design. For the domain and application engineers, the technique is not open during other phases. They can, however, develop component implementations for the end users to add to the system during later stages. This may involve installation, before or during startup or during run-time (if the system supports dynamic linking). Binding time of this technique is at run-time. The binding is done either at start-up, where a start-up parameter decides which component implementation to use, or at run-time, when an event decides which implementation to use. If the system supports dynamic linking, the linking can be delayed until binding time, but the technique work equally well when all variants are already compiled into the system.

Consequences. A consequence of using this technique is that the system will support several implementations of a domain simultaneously, and it must be possible to choose between them either at start-up or during execution of the system. Similarities in the different domains may lead to inclusion of several similar code sections into the system, code that could have been reused if the system had been designed differently.

Examples. In the e-mail application example, this technique is used to support several protocols for fetching e-mails, e.g. POP and IMAP. Both of these component implementations share the same interface for connecting to the mail server and checking for new e-mails. In addition, the IMAP component also has other interfaces since this protocol supports more functionality. For example, Axis Communications uses this technique to select between different network communication standards. Ericsson Software Technology uses this technique to select between different filtering techniques to perform on call data in their

Billing Gateway product [5]. Some open-source web browsers, e.g. Konqueror (http://www.kde.org), can optionally use the web browsing component of Mozilla, called Gecko, instead of its internal browsing component (i.e. KHTML).

Table 12: Summary of Variant Component Implementations

Introduction Times	Architecture Design
Open for Adding Variants	Detailed Design
Collection of Variants	Explicit
Binding Times	Run-time
Functionality for Binding	Internal

3.11 Condition on Constant

Intent. Support several ways to perform an operation, of which only one will be used in any given system.

Motivation. Essentially this is a more fine-grained version of a Variant Component Specializations where the variant is not large enough to be a class in its own right. The reason for using the condition on constant technique can be for performance reasons and to help the compiler remove unused code. In the case where the variant concerns connections to other, possibly variant, components it is also a means to actually get the code through the compiler, since a method call to a nonexistent class would cause the compilation process to abort.

Solution. We can in this technique use two different types of conditional statements. One form of conditional statements is pre-processor directives such as C++ #IFDEF*s*, and the other is the traditional if-statements in a programming language. If the former is used it can be used to alter the architecture of the system, for example by opting to include one file over another or using another class or component, whereas the latter can only work within the frame of a given system structure. In both cases, the collection of variants is implicit but depending on whether traditional constants or pre-processor directives are used the binding is either internal or external, respectively. Another way to implement this variability realization technique is by means of C++ *templates* which are handled similar to pre-processor directives by most compilers we have encountered[1].

Lifecycle. This technique is introduced while implementing the components and is activated during compilation of the system when it is decided using compile-time parameters which variant to include in the compiled binary. If a constant is used instead of a compile-time parameter, this is also bound at this point. After compilation the technique is closed for adding new variants.

Consequences. Using #IFDEFs or other pre-processor directives is always a risky business, since the number of potential execution paths tends to explode when using #IFDEFs, making maintenance and bug-fixing difficult. Variation points often tend to be scattered throughout the system, because of which it gets difficult to keep track of what parts of a system is actually affected by one variant.

Examples. When the e-mail application example is reconfigured to become a USENET news reader, some menu options are also added, for example the option to connect to a news server is added. The adding of this option is implemented within an #IFDEF statement, as is the functionality connected to the menu choice. Another example is the different cache types in Axis Communications [5] different Storage Servers, that can either be a Hard Disk cache or a traditional cache, where the file system component must call the cache type present in the system in the correct way. Calls to the cache component is spread

1. A simple experiment is to construct a template class containing a text string and then instantiate it a number of times with different types for the template. Compile using e.g. the default CC compiler in Solaris and then see how many times the text string occurs in the compiled binary. By changing the number of instantiations of the class a couple of times a correlation is quickly found between the number of different instantiation types and the number of occurrences of the text string.

throughout the file system component, because of which many variability realization techniques on different levels are used, including in some cases Condition on Constant.

Table 13: Summary of Condition on Constant

Introduction Times	Implementation
Open for Adding Variants	Implementation
Collection of Variants	Implicit
Binding Times	Compilation
Functionality for Binding	Internal or External

3.12 Condition on Variable

Intent. Support several ways to perform an operation of which only one will be used at any given moment but allow the choice to be rebound during execution.

Motivation. Sometimes, the variability provided by the Condition on Constant technique needs to be extended into run-time as well. Since constants are evaluated at compilation this cannot be done, hence a variable must be used instead.

Solution. Replace the constant used in Condition on Constant with a *variable* and provide functionality for changing this variable. This technique cannot use any compiler directives but is rather a pure programming language construct. The collection of variants pertaining to the variation point does not have to be explicit, even though it can be. The process of binding to a particular variant is internal.

Lifecycle. This technique is open during implementation, where new variants can be added, and is closed during compilation. It is bound at run-time where the variable is given a value that is evaluated by the conditional statements.

Consequences. This is a very flexible realization technique. It is a relatively harmless technique but, as with Condition on Constant, if the variation points for a particular variant feature are spread throughout the code it becomes difficult to get an overview.

Examples. This technique is used in all software programs to control the execution flow. For example, the ability in the e-mail application example to browse messages sorted by name, date and subject are controlled using a variable that determines which sort order that is currently active.

Table 14: Summary of Condition on Variable

Introduction Times	Implementation
Open for Adding Variants	Implementation
Collection of Variants	Implicit or Explicit
Binding Times	Run-time
Functionality for Binding	Internal

3.13 Code Fragment Superimposition

Intent. Introduce new considerations into a system without directly affecting the source code.

Motivation. Because a component can be used in several products it is not desired to introduce product-specific considerations into the component. However, it may be required to do so in order to be able to use the component at all. Product specific behavior can be introduced in a multitude of ways but these all tend to obscure the view of the component's core functionality, i.e. what the component is really supposed to do. It is also possible to use this technique to introduce variants of other forms that need not have to do with customizing source code to fit a particular product.

Solution. The solution to this is to develop the software to function generically and then superimpose the product-specific concerns at stage where the work with the source code is completed anyway. There exists a number of tools for this, for example *Aspect Oriented Programming* [7], where different concerns are weaved into the source code just before the software is passed to the compiler and *superimposition* as proposed by Reference [34], where additional behavior is wrapped around existing behavior. The collection is, in this case, implicit, and the binding is performed externally.

Lifecycle. Depending on what kind of implementation technique is used, variation points of this type are typically bound at either compilation or linking. Some experimental java extensions exist that allow for run-time imposition of new functionality. However, such solutions typically require the use of reflection, which in most languages is not available. New variants may either be added by application or product engineers during either detailed design or compilation.

Consequences. A consequence of superimposing an algorithm is that different concerns are separated from the main functionality. This is positive as it increases the readability of what the source code is intended to do. However, this also means that it becomes harder to understand how the final code will work since the execution path is no longer obvious. When developing one must be aware that there will or may be a superimposition of additional code at a later stage. In the case where binding is deferred to run-time one must even program the system to add a concern to an object.

Examples. In the e-mail application example, this technique is used to ship the product with a context-sensitive user assistant in the shape of a paper clip. Aspects are added throughout the source code containing functionality to inform the user assistant component what is happening in the system. Advice for the end user is then dispersed based upon the gathered information. During compilation, the aspects are intertwined and compiled with the rest of the source code. To the best of our knowledge none of the case companies use this technique. This is not very surprising since at the time of writing, the only production ready technique that could be used is AspectJ. Several case studies are available from the Aspect Oriented Programming home page (http://www.aosd.net) that suggest that this technique might be used more often in the near future.

Table 15: Summary of Code Fragment Superimposition

Introduction Times	Compilation
Open for Adding Variants	Compilation
Collection of Variants	Implicit
Binding Times	Compilation Run-time
Functionality for Binding	External

3.14 Summary

In this section we present a taxonomy of variability realization techniques. The way to use this taxonomy is, as described in Section 2, to decide the size of the software entities and the binding time. This is used with the help of Table 2 to decide which variability realization techniques that are available. After this, additional characteristics are used to decide which technique to use. These characteristics concern the population, i.e. when to add the variants and how the collection of variant should be managed, as well as where the functionality for the binding should reside. These characteristics are summarized at the end of each variability realization technique and in Table 16.

4. Case Studies

In this section we briefly present a set of companies that use product families, and how these have typically implemented variability, i.e. what variability realization techniques they have mostly used in their software product families.

The cases are divided into three categories:

- Cases which we based the taxonomy of variability realization techniques on.

- Unrelated case studies conducted after the initial taxonomy was created, which were used to confirm and refine the taxonomy.
- Cases found in literature that contain information regarding how variability was typically implemented.

We provide a brief presentation of the companies within each category and how they have typically implemented variability. The cases from the first category are presented to give a further overview of the companies behind the examples in the taxonomy. The second category is presented to give further examples of which we have in-depth knowledge and have had full insight in the development process of and which have helped in refining the taxonomy. The third category is included to extend the generalizability of the taxonomy further, by providing cases which we ourselves have not had any influence in.

Observed Cases. The taxonomy of variability realization techniques and indeed the identification of the relevant characteristics to distinguish between different variability realization techniques was created using information gathered from four companies. These companies are:

- Axis Communications AB and their storage server product family [4][5][6] (presented in Section 4.1)
- Ericsson Software Technology and their Billing Gateway product [5][36][37] (presented in Section 4.2)
- The Mozilla web browser [12][38][39] (presented in Section 4.3)
- Symbian and the EPOC Operating System [4][30] (presented in Section 4.4)

Confirming Cases. In the second category we have case studies conducted by the research groups of the authors of this paper. These case studies were not conducted with the purpose of creating or refining the taxonomy of variability realization techniques. However during these studies we have had the opportunity to see and get an in-depth understanding of their software product families. Hence, we can make confident statements regarding how these companies choose implementation strategies for their variant features and what these implementation strategies are. The companies in this category are:

Table 16: Summary of Variability Realization Techniques

Section	Name	Introduction Time	Open for Adding Variants	Collection of Variants	Binding Times	Functionality for Binding
3.1	Architecture Reorganization	Architecture Design	Architecture Design	Implicit	Product Architecture Derivation	External
3.2	Variant Architecture Component	Architecture Design	Architecture Design Detailed Design	Implicit	Product Architecture Derivation	External
3.3	Optional Architecture Component	Architecture Design	Architecture Design	Implicit	Product Architecture Derivation	External
3.4	Binary Replacement - Linker Directives	Architecture Design	Linking	Implicit or Explicit	Linking	External or Internal
3.5	Binary Replacement - Physical	Architecture Design	After Compilation	Implicit	Before Run-time	External
3.6	Infrastructure-Centered Architecture	Architecture Design	Architecture Design Linking Run-time	Implicit or Explicit	Compilation Run-time	Internal
3.7	Variant Component Specializations	Detailed Design	Detailed Design	Implicit	Product Architecture Derivation	External
3.8	Optional Component Specializations	Detailed Design	Detailed Design	Implicit	Product Architecture Derivation	External
3.9	Run-time Variant Component Specializations	Detailed Design	Detailed Design	Explicit	Run-time	Internal
3.10	Variant Component Implementations	Architecture Design	Detailed Design	Explicit	Run-time	Internal
3.11	Condition on Constant	Implementation	Implementation	Implicit	Compilation	Internal or External
3.12	Condition on Variable	Implementation	Implementation	Implicit or Explicit	Run-time	Internal
3.13	Code Fragment Superimposition	Compilation	Compilation	Implicit	Compilation or Run-time	External

- Danaher Motion Särö AB [40] (presented in Section 4.5)
- Rohill Technologies BV [14] (presented in Section 4.6)

Additional Cases. In the third and final category we include examples of case studies described in literature where these descriptions are of sufficient detail to discern what types of variability realization techniques these companies typically use. The cases in this category are:

- Cummins Inc. [3] (presented in Section 4.7)
- Control Channel Toolkit [3] (presented in Section 4.8)
- Market Maker [3] (presented in Section 4.9)

4.1 Axis Communications AB

Axis Communications is a medium sized hardware and software company in the south of Sweden. They develop mass-market networked equipment, such print servers, various storage servers (CD-ROM servers, JAZ servers and Hard disk servers), camera servers and scan servers. Since the beginning of the 1990s, Axis Communications has employed a product family approach. At the time of our studies this software product family consisted of 13 reusable assets. These assets are in themselves object-oriented frameworks of differing size. Many of these assets are reused over the complete set of products, which in some cases have quite differing requirements on the assets. Moreover, because the systems are embedded systems, there are very stringent memory requirements; the application, and hence the assets, must not be larger than what is already fitted onto the motherboard. What this implies is that only the functionality used in a particular product may be compiled into the product software, and this calls for a somewhat different strategy when it comes to handling variability.

In this paper we have given several examples of how Axis implements variability in its software product family, but the variability realization technique they prefer is that of variant component implementations (Section 3.10), which is augmented with run-time variant component specializations (Section 3.9). Axis uses several other variability realization techniques as well, but this is more because of architecture decay which has occurred during the evolution of the software product family.

Further information can be found in two papers by Svahnberg & Bosch [5][6] and in our co-author's book on software product families [4].

4.2 Ericsson Software Technology

Ericsson Software Technology is a leading software company within the telecommunications industry. At their site in Ronneby, Sweden they develop (among other products) their Billing Gateway product. The Billing Gateway is a mediating device between telephone switching stations and post-processing systems such as billing systems, fraud control systems, etc. The Billing Gateway has also been developed since the early 1990's, and was at the time of our study installed at more than 30 locations worldwide. The system is configured for every customer's needs with regards to, for instance, what switching station languages to support, and each customer builds a set of processing points that the telephony data should go through. Examples of processing points are formatters, filters, splitters, encoders, decoders and routers. These are connected into a dynamically configurable network through which the data is passed.

Also for Ericsson, we have given several examples of how variability is implemented. As with Axis Communications, the favored variability realization technique is that of variant component implementations (Section 3.10), but Ericsson has managed to keep the interfaces and connectors between the software entities intact as the system has evolved, so there is lesser need to augment this realization technique with other techniques. In many ways their architecture is similar to the Infrastructure-Centered Architecture (Section 3.6)

For further reading, see Reference [5][36] and [37].

4.3 Mozilla

The Mozilla web browser was Netscape's Open Source project to create their next generation of web browsers. In 2003, the Mozilla project was transferred to an independent foundation and the remains of the Netscape company were liquidated. One of the design goals of Mozilla is to be a platform for web applications (see e.g. the web site http://www.mozdev.org - a repository for more than a hundred Mozilla based projects). Mozilla is constructed using a highly flexible architecture which makes massive use of components. The entire system is organized around an infrastructure of XUL (a language for defining user interfaces), JavaScript (to bind functionality to the interfaces) and XPCOM (a COM-like model with components written in languages such as C++). The use of C++ for lower level components ensures high performance, whereas XUL and JavaScript ensure high flexibility concerning appearance (i.e. how and what to display), structure (i.e. the elements and relations) and interactions (i.e. how elements work across the relations). This model enables Mozilla to use the same infrastructure for all functionality sets, which ranges from e-mail and news handling to web browsing and text editing. Moreover, any functionality defined in this way is platform independent, and only require the underlying C++ components to be reconstructed and/or recompiled for new platforms. Variability issues here concern the addition of new functionality sets, i.e. applications in their own right, and incorporation of new standards, for example regarding data formats such as HTML, CSS and various XML based languages such as, for example, RSS, XHTML and XBL.

As described above, Mozilla connects its components using XUL and XPCOM. In our taxonomy, this would translate to the use of an infrastructure-centered architecture (Section 3.6). Additionally, it uses an elaborate XML based linking system to relate various resources to each other (XBL). This XBL (XML Binding Language) is used extensively to e.g. associate user interface skins with user interface elements. Both techniques can be seen as variations of the Binary Replacement - Linker Directives mechanism described in Section 3.4.

For further information regarding Mozilla, see e.g. Reference [12][38][39] .

4.4 Symbian - Epoc

EPOC[1] is an operating system, an application framework, and an application suite specially designed for wireless devices such as hand-held, battery powered computers and cellular phones. It is developed by Symbian, a company that is owned by major companies within the domain such as Ericsson, Nokia, Psion, Motorola and Matsushita, in order to be used in these companies' wireless devices. Variability issues here concern how to allow third party applications to seamlessly and transparently integrate with a multitude of different operating environments, which may even affect the amount of functionality that the applications provide. For example, with screen sizes varying from a full VGA screen to a two-line cellular phone, the functionality and how this functionality is presented to the user will differ vastly between the different platforms.

Symbian, by means of EPOC, does not interfere in how applications for the EPOC operating system implement variability. They do, however, provide support for creating applications supporting different operating environments. This was at the time of our studies done by dividing applications into a set of components handling user interface, application control and data storage (i.e. a Model-View-Controller pattern [27]). The EPOC operating system itself is specialized for different hardware environments by using the architecture reorganization (Section 3.1) and variant architecture component (Section 3.2) variability realization techniques.

More information can be obtained from Symbian's website [30] and in Reference [4].

1. The name EPOC has subsequently been changed to SymbianOS. However, at the time we were in contact with Symbian (1999-2000), the software was still referred to as EPOC. Probably, the software has evolved significantly since then.

4.5 Danaher Motion Särö AB

Danaher Motion Särö AB develops general control systems, software and electronic equipment in the field of materials handling control. Specifically, they develop the control software for automated guided vehicles, i.e. automatic vehicles that handle transport of goods on factory floors. Danaher Motion Särö AB's product family consists of a range of software components that together control the assignment of cargo to vehicles, monitor and control the traffic (i.e. intelligent routing of vehicles to avoid e.g. traffic jams) as well as steering and navigating the actual vehicles. The most significant variant features in this product family concern a variety of navigation techniques ranging from inductive wires in the factory floor to laser scanners mounted on the vehicles and specializations to each customer installation, such as different vehicles with different loading facilities, and of course different factory layouts.

The variability realization techniques used in this software product family is mainly by using parameterization, e.g. in the form of a database with the layout of the factory floor, which translates to the realization technique "condition on variable" described in Section 3.12. For the different navigation techniques, the realization technique used is mainly the "variant architecture component" (Section 3.2), which is also aided by the use of an infrastructure-centered architecture (Section 3.6).

For further information about Danaher Motion Särö AB, see Reference [40] and [41]. For a further introduction to the domain of automated guided vehicles, see Reference [42].

4.6 Rohill Technologies BV

Rohill Technologies BV is a Dutch company that specializes in product and system development for professional mobile communication infrastructure, e.g. radio networks for police and fire departments. One of their major product families is TetraNode, a product family of trunked mobile radios. In this product family, the products are tailored to each customer's requirements by modifying the soft- and/or hardware architecture. The market for this type of radio systems is divided into a professional market, a medium market and a low-end market. The products for these three markets all use the same product family architecture, designed to support all three market segments. The architecture is then pruned to suit the different product architectures for each of these markets.

Rohill identifies two types of variability: anticipated (domain engineering) and unanticipated (application engineering). It is mainly through the anticipated variability that the product family is adjusted to the three market segments. This is done using license keys that load a certain set of dynamic linked libraries, as described in the variability realization technique "binary replacement - linker directives" (Section 3.4). The unanticipated variability is mainly adjustments to specific customers' needs, something which is needed in approximately 20% of all products developed and delivered. The unanticipated variability is solved by introducing new source code files, and instructing the linker through makefiles to bind to these product specific variants. This variability is, in fact, using the same realization technique as the anticipated variability, i.e. the binary replacement through linker directives (Section 3.4), with the difference that the binding is external as opposed to the internal binding for anticipated variability.

For further information regarding Rohill Technologies BV and their TetraNode product family, see Reference [14].

4.7 Cummins Inc.

Cummins Inc. is a USA-based company that develops diesel engines and the control software for these engines. Examples of usages of diesel engines involve automotives, power generation, marine, mining, railroad and agriculture. For these different markets the types of diesel engines varies in a number of ways. For example, the amount of horsepower, the number of cylinders, the type of fuel system, air handling systems and sensors varies between the different engines. Since 1994, Cummins Inc. develops the control software for the different engine types in a software product family.

Cummins Inc. use several variability realization techniques, ranging from the variant architecture components (Section 3.2) to select what components to include for a particular hardware configuration, to #IFDEFs, which translates to the realization technique condition on constant (Section 3.11), which is used to specify the exact hardware configuration with how many cylinders, displacement, fuel type, etc. that the particular engine type has. The system also provides a large number of user-configurable parameters, which are implemented using the variability realization technique condition on variable (Section 3.12).

The company Cummins Inc. and its product family is further described in Reference [3].

4.8 Control Channel Toolkit

Control Channel Toolkit, or CCT for short, is a software asset base commissioned by the National Reconnaissance Office (in the USA), and built by the Rayethon Company under contract. The asset base that is CCT consists of generalized requirements, domain specifications, a software architecture, a set of reusable software components, test procedures, a development environment definition and a guide for reusing the architecture and components. With the CCT, products are built that command and control satellites. Typically, only one software system is used per satellite. Development on CCT started in 1997.

The CCT uses an infrastructure-centered architecture (Section 3.6), i.e. CORBA, to connect the components in the architecture. Within the components, CCT provides a set of standard mechanisms: dynamic attributes, parameterization, template, function extension (callbacks), inheritance and scripting. Dynamic attributes and parameterization amounts to the technique condition on variable (Section 3.12). Inheritance is what we refer to as run-time variant component specializations (Section 3.9). Scripting is another example of an infrastructure-centered architecture (Section 3.6). We have not found sufficient information regarding function extension to identify which variability realization technique this is.

Further information on CCT can be found in Reference [3].

4.9 Market Maker

Market Maker is a German company that develops products that presents stock market data, and also provides stock market data to users of its applications. Their product family includes a number of functionality packages to manage different aspects of the customers' needs, such as depot management, trend analysis, option strategies. It also consists of a number of products for different customer segments, such as individuals and different TV networks or TV news magazines. In 1999 a project was started to integrate this product family with another product family with similar functionality but with the ability to update and present stock data continuously, rather than at specified time intervals (six times/day). This new product family, the MERGER product family, is implemented in Java, and also includes salvaged Delphi code from the previous product family.

Market Maker manages variability by having a property file for each customer, that decides which features to enable for the particular customer. This property file translates to the variability realization technique condition on variable (Section 3.12). Properties in the property file are used even to decide what parts of the system to start up, by also making use of Java's reflection mechanism in which classes can be instantiated by providing the name of the class as a text string.

For further information about Market Maker and its MERGER product family, see Reference [3].

5. Related Work

Software Product Families. In the past few years, there have been a number of publications on how to design and implement software product families such as, for instance, Reference [1][2][3]. These and other publications such as Reference [43], our co-author's book [4] and conferences such as SPLC 1 [44] and the upcoming SPLC 2 conference have increased interest in and use of software product families.

Empirical research such as Reference [45], suggests that a software product family approach stimulates reuse in organizations. In addition, a follow up paper [46] provides empirical evidence for the hypothesis that organizations get the greatest reuse benefits during the early phases of development. Because of this we believe it is worthwhile for software product family developing companies to invest time and money in performing methods for determining and implementing variability.

In Reference [43], Bass et al. define a software product family as *a collection of systems sharing a managed set of features from a common set of core software assets*. This is entirely in line with our view that using feature models is an important way of identifying and managing variability [12].

A case study presented in Reference [47] recommends that a focus on simplification, clarification and minimization is essential for the success of software product family architectures. However they also warn not to over simplify since the architecture needs to be adaptable to future needs. In a case where the architecture was over simplified, the time needed to introduce a new feature tripled. Clearly the use of variation techniques is needed to be adaptable and our taxonomy can help selecting the right techniques so that the architecture can be both adaptable and not be too complex. In addition identifying the need for variation using for example feature diagrams (such as in our earlier work [12]). Other methods that may be of use in doing so are the FAST and PASTA [1] and FODA [20].

In Reference [2], a number of variability mechanisms are discussed. However, it fails to put these mechanisms in a taxonomy like we do. In addition, variability is not linked to features. This is an important characteristic of our approach as it is an important means for early identification (i.e. before architecture design) of variability needs in the future system.

A comprehensive work on software product families is Reference [3]. This book presents what a software product family is and is not, the benefits gained by using a product family approach, and a wide range of practice areas, covering aspects in software engineering, technical management and organizational management. This book also presents, in great detail, three cases studies of companies using software product family solutions.

Variability. There appears to be a lot of consensus that domain analysis and feature diagrams in particular are suitable for identifying and documenting variability. FODA [20], for instance, introduces a feature diagram notation that includes things like optional, mandatory and alternative features. In Reference [21], which discusses the FODA derived FORM method, feature diagrams are identified as a means of identifying commonality between products. Related to FODA is FeatureRSEB [22], which extends the use-case modeling of RSEB [10] with the feature model of FODA. Also related is the FAST method described in Reference [1] which also includes analyzing variability. The use of such techniques to organize requirements is also recommended in Reference [3]. This book presents a number of practices and patterns for the development of software product families.

In Reference [11], it is observed that typically changes in a system can be related to individual features or small groups of features. Griss also states that *"Starting from the set of common and variable features needed to support a product-line, we can systematically develop and assemble the reusable elements needed to produce the customized components and frameworks to implement the product"*.

A good overview of domain analysis and engineering methods is provided in Reference [8]. In this book, the authors also include a chapter on feature modeling and the relation of feature models to various generative programming techniques such as inheritance and parameterization. These techniques can be regarded as variability realization techniques as well.

In Reference [48] methodology for using COTS (Commercial Off The Shelf) components is discussed. The discussion also includes what the authors refer to as *alternative refinements*. These alternative refinements can be seen an instance of our variant architecture component technique.

Some recent work has focused on exploiting variation points and providing languages for using these variation points. An example of such a language is Jarzabek's XVCL (XML-based Variant Configuration Language) [28] which is a language that describes systems in terms of variation. A generator then takes care of binding the variation points to the appropriate variants. It is tempting to regard this generation process as a separate variability realization technique. However, we have decided against this because the actual variation is not provided but only described by such languages. These languages merely use the techniques we describe in this paper and do not (at least in their present form) add to them. We are aware of several large companies that are using configuration languages to describe systems. For example, Philips uses the ADL KOALA [24] to describe configurations of embedded software. An overview of ADL's using similar approaches is presented in Reference [29].

Variability realization techniques. In Reference [10], five ways to implement variability are presented, namely: inheritance, extensions, parameterization, configuration and generation. Most of the variability realization techniques we present are based on these implementation techniques. Our contribution is that we explore when it is more suitable to select one technique over another, and what the consequences are of a particular technique. Moreover, we present more than one way in which one can use these implementation techniques.

The two major techniques for variability, as identified in our taxonomy are configuration management and design patterns. Configuration management is dealt with extensively in Reference [49], presenting the common configuration management tools of today, with their benefits and drawbacks. Design patterns are discussed in detail in Reference [15] and [27], where many of the most commonly used design patterns are presented.

Configuration management is also identified as a variability realization mechanism in Reference [50]. This paper primarily focuses on how to model variability in terms of software modules, and is as such a complement to the feature-graphs as discussed above. It does, however, also include a section on how to realize variability in the software product family, which includes techniques such as generators, compilation, adaptation during start-up and during run-time, and also configuration management. Our work complements this work by providing further detail on when to introduce variability, when it is possible to add new variants, and when it is possible to bind to a particular variant. We provide a comprehensive taxonomy that brings these things together into the decision of which realization technique to use, rather than just focusing on one of these aspects.

Another technique for variability, seen more and more often these days, is to use some form of infrastructure-centered architecture. Typically these infrastructures involve some form of component platform, e.g. CORBA, COM/DCOM or JavaBeans [31].

During recent years, code fragment superimposition techniques have received increasing attention. Examples of such techniques are Aspect-, Feature- and Subject-oriented programming. In Aspect-oriented programming, features are weaved into the product code [7]. These features are in the magnitude of a few lines of source code. Feature-oriented programming extends this concept by weaving together entire classes of additional functionality [51]. Subject-oriented programming [52] is concerned with merging classes developed in parallel to achieve a combination of the merged classes.

Historical notes. We received comments on a draft of this article that many of the concepts related to variability and product families that we present here, and that others have presented in recent years, are not new and that ideas like these have been published as early as the 1960's (for example Reference [53] discusses an early notion of software components that can be used and re-used in different environments). One of these pioneers, David Parnas, has written a number of articles in the 1970's discussing program families and extension points. For example in Reference [54], a methodology for reusing a "common ancestor" in different versions of a program is outlined. Ironically, this article makes a number of historical notes too, referring for example to the work of Dijkstra [55] who introduced the notion of stepwise refinement in software engineering. While Parnas deserves credit for his visionary papers in the 1970's, we note that Parnas seems to be have been primarily occupied with variation in time rather than in space as we do (reusing an early version of product A to later develop product B). Additionally, while the examples used were certainly relevant examples in those days, using today's standards they would be considered very small systems. For example, Reference [56] discusses an "address processing subsystem" that consists of 25 function calls (i.e. a moderately sized class in a modern object oriented system), and Reference [54] discusses a space allocation algorithm consisting of (roughly) 25-30 lines of code. The technical detail of these articles are pretty much limited to the types of variation techniques that were common in those days (subroutines, gotos, etc.). Other trends that were emerging during that time are the early development of object orientation (e.g. Simula-67 and Smalltalk), which is a technique that is underlying many of the variability realization techniques we describe in this article, and the already mentioned discussions on software components [53]. More recent contributions to the software engineering field, while still relying on the ground breaking work performed in the 1960's and 1970's, have since then, we argue, evolved and are much more detailed with respect to techniques, methodologies and case studies.

6. Conclusions

We have observed that software systems increasingly rely on variability mechanisms to exploit commonalities between software products. This has resulted in the adoption of so called product families [1][2][3] by numerous companies. However, a problem is that often it is far from obvious what kind of techniques are the most suitable for implementing variability in a product family. In this article we make this decision process more obvious by providing a number of selection criteria and a taxonomy of variability realization techniques based on these criteria.

There are several factors that influence the choice of implementation technique, such as identifying the variant features, when the variant feature is to be bound, by which software entities to implement the variant feature and last but not least how and when to bind the variation points related to a particular variant feature. Moreover, the job is not done just because the variant feature is implemented. The variation points and software entities implementing the variant feature need to be managed during the product's lifecycle, extended during evolution, and used during different stages of the development cycle. This also constrains the choices of how to implement the variability into the software system.

In this paper we present a minimal set of steps by which to introduce variability into a software product family, and what characteristics distinguish the ways in which one can implement variability. We present how these characteristics are used to constrain the number of possible ways to implement the variability, and what needs to be considered for each of these characteristics.

Once the variability has been constrained, the next step is to select a way in which to implement it into the software system. To this end, we provide, in this paper, a taxonomy of available variability realization techniques. This taxonomy presents the intent, motivation, solution, lifecycle, consequences and a brief example for each of the realization techniques. The variability realization techniques in the taxonomy are based around technical solutions we have observed in industry. These are generalised so that the domain-specific and company-specific technical details have been removed so that the underlying principles appear and can be used in different settings by different companies.

We believe that the contribution of this taxonomy is to provide a toolbox for software developers when designing and implementing a software system, to assist them in selecting the most appropriate means by which to implement a particular variant feature and its corresponding variation points.

The contribution of this paper is, we believe, that by taking into account the steps outlined in this paper, and considering the characteristics we have identified, a more informed, and hopefully more accurate, decision can be taken with respect to the variability realization techniques chosen to implement the variant features during the construction of a product or a software product family.

In summary, the criteria and the taxonomy are based on practical experience from a number of companies, generalised into a form that is usable outside of the companies where the criteria and the variability realization techniques were originally observed. The theoretical discussion of criteria to consider when deciding how to implement variability presents a direction towards better and more maintainable practical systems.

To the best of our knowledge, the taxonomy covers most conventional variation techniques. However in future work, we intend to further refine our taxonomy and annotate it with more examples from practice and from other domains. Currently we are working on several case studies (for example in the context of the European IST funded CONIPF project of which the University of Groningen is an active partner). Preliminary experience with these case studies is confirming the results presented in this survey. However, it is too early to report on these cases.

Acknowledgements

We would like to thank the reviewers for their feedback on early drafts of this article. Their constructive criticism has enabled us to make many improvements to this article. Additionally, we would like to thank the companies who have worked with us in the various case studies we cite: Symbian (named UIQ nowadays), Axis Communications AB, Philips, Ericsson Software Technology, Danaher Motion Särö AB, and the Mozilla project.

References

[1] C. T. R. Lai, D. M. Weiss, *"Software Product-Line Engineering: A FamilyBased Software Development Process"*, Addison-Wesley, 1999.

[2] M. Jazayeri, A. Ran, F. Van Der Linden, *"Software Architecture for Product Families: Principles and Practice"*, Addison-Wesley, 2000.

[3] P. Clements, L. Northrop, *"Software Product Lines - Practices and Patterns"*, Addison-Wesley, 2002.

[4] Jan Bosch, *"Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach"*, Addison-Wesley, ISBN 020167494-7, 2000.

[5] M. Svahnberg, J. Bosch, "Evolution in Software Product Lines: Two Cases", in *Journal of Software Maintenance - Research and Practice*, 11(6), pp. 391-422, 1999.

[6] M. Svahnberg, J. Bosch, "Characterizing Evolution in Product Line Architectures", in *Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications*, IASTED/Acta Press, Anaheim, CA, pp. 92-97, 1999.

[7] G. Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, "Aspect Oriented Programming", in *Proceedings of 11th European Conference on Object-Oriented Programming*, pp. 220-242, Springer Verlag, Berlin Germany, 1997.

[8] K. Czarnecki, U. W. Eisenecker, *"Generative Programming - Methods, Tools and Applications"*, Addison-Wesley, 2000.

[9] IEEE, "Recommended Practice for Architectural Description of Software-Intensive Systems", Std. 1471-2000.

[10] I. Jacobson, M. Griss, P. Johnson, *"Software Reuse: Architecture, Process and Organization for Business success"*, Addison-Wesley, 1997.

[11] M.L. Griss, "Implementing Product Line Features with Component Reuse", in *Proceedings of 6th International Conference on Software Reuse*, 2000.

[12] J. van Gurp, J. Bosch, M. Svahnberg, "On the Notion of Variability in Software Product Lines", in *Proceedings of WICSA 2001*, August 2001.

[13] J. Bosch. G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl, "Variability issues in Software Product Lines", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[14] M. Jaring, J. Bosch, "Representing Variability in Software Product Lines: A Case Study", to appear in the Second Product Line Conference (SPLC-2), San Diego CA, August 19-22, 2002.

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley Publishing Co., Reading MA, 1995.

[16] M. Becker, L. Geyer, A. Gilbert, K. Becker, “Comprehensive Variability Modeling to Facilitate Efficient Variability Treatment”, in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[17] R. Capilla, J.C. Dueñas, “Modeling Variability with Features in Distributed Architectures”, in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[18] C.W. Krueger, “Easing the Transition to Software Mass Customization”, in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[19] S. Salicki, N. Farcet, “Expression and usage of the Variability in the Software Product Lines”, in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[20] K. C. Kang, S. G. Cohen, J. A. Hess, W.E. Novak, A.S. Peterson, *“Feature Oriented Domain Analysis (FODA) Feasibility Study“*, Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[21] K. C. Kang, “FORM: a feature-oriented reuse method with domain specific architectures“, in *Annals of Software Engineering* volume 5, pp. 345-355, 1998.

[22] M. L. Griss, J. Favaro, M. d'Alessandro, “Integrating feature modeling with the RSEB”, *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*. IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, xiii+388 pp. p.76-85.

[23] P. Zave, M. Jackson, “Four Dark Corners of Requirements Engineering“, *ACM Transactions on Software Engineering and Methodology*, Vol. 6. No. 1, January 1997, p. 1-30.

[24] R. van Ommering, “The Koala Component Model”, in *Building Reliable Component-Based Software Systems*, I. Crnkovic, M. Larsson (editors), Artech House Publishers, July 2002.

[25] D. E. Perry, A. L. Wolf. “Foundations for the Study of Software Architecture”, in ACM SIGSOFT Software Engineering Notes, vol 17 no 4, 1992.

[26] J. van Gurp, J. Bosch, “Design Erosion: Problems & Causes”, Journal of Systems & Software, 61(2), pp. 105-119, Elsevier, March 2002.

[27] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, *“Pattern-Oriented Software Architecture - A System of Patterns“*, John Wiley & Sons, 1996.

[28] S. Jarzabek, P. Basset, H. Zhang, W. Zhang, “XVCL: XML-based Variant Configuration Language”, in *Proceedings of the International Conference on Software Engineering, ICSE’03*, IEEE Computer Society Press, Los Alamitos CA, pp. 810-811, 2003.

[29] N. Medvidovic, R. N. Taylor, “A classification and comparison framework for software architecture description languages”, in *IEEE Transactions on Software Engineering* **26**(1):70-93, 2000.

[30] Symbian Website (now UIQ website), *http://www.uiq.com/.*

[31] C. Szyperski, *“Component Software - Beyond Object-Oriented Programming”*, Pearson Education Limited, Harlow UK, 1997.

[32] J.K. Ousterhout, “Scripting: Higher Level Programming for the 21st Century”, in *IEEE Computer*, May 1998.

[33] H. Lieberman, “Using Prototypical Objects to Implement Shared Behavior”, in Proceedings on Object-Oriented Programming systems, Languages and Applications, pp. 214-223, 1986.

[34] J. Bosch, “Superimposition: A Component Adaption Technique”, in *Information and Software Technology*, (41)5, pp. 257-273, 1999.

[35] M. Mattsson, *“Evolution and Composition of Object-Oriented Frameworks”*, Ph. D. Thesis, Blekinge Institute of Technology, Sweden, 2000.

[36] M. Mattsson, J. Bosch, “Evolution Observations of an Industry Object-Oriented Framework”, in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press: Los Alamitos CA, pp. 139-145, 1999.

[37] M. Mattsson, J. Bosch, “Characterizing Stability in Evolving Frameworks”, in *Proceedings TOOLS Europe 1999*, IEEE Computer Society Press: Los Alamitos CA, pp. 118-130, 1999.

[38] Mozilla website, *http://www.mozilla.org/.*

[39] I. Oeschger, “XULNotes: A XUL Bestiality”, web page: *http://www.mozilla.org/docs/xul/xulnotes/xulnote_beasts.html*, Last Checked: May 2000.

[40] M. Svahnberg, M. Mattsson, “Conditions and Restrictions for Product Line Generation Migration”, in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[41] Danaher Motion Särö AB website, *http://www.danahermotion.se/.*

[42] T. Feare, “A roller-coaster ride for AGVs”, in *Modern Materials Handling* **56**(1):55-63, january 2001.

[43] L. Bass, P. Clements, R. Kazman. *“Software Architecture in Practice”*, Addison-Wesley, 1997.

[44] P. Donohoe, *“Proceedings of the First Software Product Line Conference”* (SPLC1), Kluwer, 2000.

[45] D. C. Rine, R. M. Sonnemann, “Investments in reusable software. A study of software reuse investment success factors“, in *The journal of systems and software*, nr. 41, pp 17-32, Elsevier, 1998.

[46] D. C. Rine, N. Nada, “An empirical study of a software reuse reference model“, in *Information and Software Technology*, nr 42, pp. 47-65, Elsevier, 2000.

[47] D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, “Applying Software Product-Line Architecture“, IEEE Computer, August 1997.

[48] K. Wallnau, S. A. Hissam, R. C. Seacord, *“Building Systems from Commercial Components“*, Addison-Wesley, 2002.

[49] R. Conradi, B. Westfechtel, “Version Models for Software Configuration Management”, in *ACM Computing Survey*, 30(2):232-282.

[50] F. Bachmann, L. Bass, “Managing variability in software architectures”. *proceedings of the ACM Symposium on Software Reusability: Putting Software Reuse in Context*, pp. 126-132, 2001

[51] C. Prehofer, “Feature-Oriented Programming: A fresh look at objects”, in *Proceedings of ECOOP’97*, Lecture Notes in Computer Science 1241, Springer Verlag, Berlin Germany, 1997.

[52] M. Kaplan, H. Ossher, W. Harrisson, V. Kruskal, “Subject-Oriented Design and the Watson Subject Compiler”, position paper for OOPSLA’96 Subjectivity Workshop, 1996.

[53] M.D. McIlroy, “Mass produced software components”, in *Software Engineering: Concepts and Techniques*, J.M. Buxton P. Naur, B. Randell (eds), Petrocelli/Charter Publishers Inc. (republished 1976), New York NY, pp. 88-98, 1969.

[54] D. L. Parnas, “On the Design and Development of Program Families“, IEEE Transactions on Software Engineering, vol2, no. 1, 1976.

[55] E. W. Dijkstra, “Structured Programming“, in Software Engineering Techniques, J.N. Buxton and B. Randell, (eds), Brussels, Belgium: NATO Scientific Affairs Division, pp. 84-87, 1970.

[56] D. L. Parnas, “Designing Software for Ease of Extension and Contraction“, proceedings of the International Conference on Software Engineering 1978, pp. 264-277, 1978.