AMPLE
Aspect –Oriented, Model-Driven, Product Line Engineering
Specific Targeted Research Project: IST- 33710

Survey of existing implementation techniques with respect to their support for the requirements identified in M3.2

ABSTRACT

This deliverable consists of a list of identified requirements for variability in SPLs and their motivation, the descriptions of the surveyed technologies and evaluation of each technology with respect to the listed requirements. The survey includes the technologies in use at industrial project partners and other promising AOP and MDD technologies with a potential to solve identified shortcomings of currently practised implementation techniques.

Document ID:	AMPLE D3.1
Deliverable/ Milestone No:	D3.1
Work-package No:	WP3
Type:	Deliverable
Dissemination:	PU
Status:	final
Version:	1.2
Date:	2007-07-30
Author(s):	Christoph Pohl, Andreas Rummler (SAP), Vaidas Gasiunas (TUD), Neil Loughran (UNLANC), Hugo Arboleda, Fabricio de Alexandria Fernandes, Jacques Noyé, Angel Núñez, Robin Passama, Jean-Claude Royer, Mario Südholt (EMN)

Project Start Date: 01 October 2006, Duration: 3 years

History of Changes

Version	Date	Changes
0.1	2007-06-01	Initial Version
0.2	2007-06-21	Updated section containing evaluation of SAP techniques
0.3	2007-06-29	structural changes, initial conclusion
0.4	2007-06-29	Sections 2 and 3 by TUD
0.5	2007-07-03	Sections 2 and 4 by EMN
0.6	2007-07-06	Sections 3, 5, and 6 by SAP
0.7	2007-07-13	Update to sections about configuration management & SAP techniques
0.8	2007-07-15	Added sections about component technology and interpreters. Section 2 updated by comments of Uwe.
0.9	2007-07-16	Tool criteria, description and evaluation extended and revised by EMN
0.10	2007-07-23	document consolidation, introduction added
0.11	2007-07-23	Conclusions for section 3 added
0.12	2007-07-23	Updated references in sections 2 and 3
0.13	2007-07-25	Tool criteria & description revised by EMN in section 2. Updated references, comparison table and evaluation in section 4.
1.0	2007-07-27	Final review version
1.1	2007-07-29	Polishing
1.2	2007-07-30	Broken references fixed

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Software Product Lines (SPL) have been discussed for more than a decade as a concept for managing commonalities and variations in features of software product families. Consequentially, numerous approaches and implementation techniques already exist for binding these feature variations in concrete products. The document at hand surveys a range of existing implementation techniques with respect to their support for the requirements identified in Milestones 3.1 and 3.2, representing the different views of requirements posed by variability management for SPL on implementation technology and practices currently in use at industrial partners.

The way of throwing light on implementation techniques in the context of Software Product Lines is not obvious. For this reason this survey starts with the definition of adequate criteria that can be used to analyse and compare those techniques in chapter 2. This chapter is an attempt to identify individual variation mechanisms that are used to express variability instead of analysing entire programming languages as a whole for their support for variability, because such languages may contain multiple features that can be used to implement variation in program code. Due to the diversity of mechanisms, which can be as simple as setting property values and as complex as generating code based on templates, the mechanisms must be viewed through a common scheme, which is described in this chapter.

Based on the criteria in chapter 2 a wide range of implementation mechanisms is enumerated, characterized and classified in chapter 3. The analysed variation techniques range from simple ones like parameterisation to complex ones like feature-oriented and aspect-oriented programming. Chapter 3 also discusses methods that are used to manage variability and create actual products in product line engineering, but which are not directly connected to implementation in the common sense – configuration management may be named here as an example. It should be stated here that it is not realistic to evaluate all possible variation mechanisms, because most mechanisms in common programming languages are related to handling variability in some sense. Although a wide range of techniques is discussed in chapter 3 this range has been limited to the most popular ones which do have importance in an industrial context and which are directly related to the project in the context of SPL. In addition it should be noted that most techniques are not mutually exclusive but can be used in conjunction with others.

Some of the implementation techniques discussed in chapter 3 are complex enough that the support of developers applying these techniques by dedicated tools is desirable. Although many tools used in software development may be applied in implementing variations in program code (i.e. a simple text editor may be used to create source code containing the variation mechanism *parameterisation*), chapter 3 describes only the tools that cover a range of activities in the development of software product lines. Several tools that fulfil this requirement and that provide development environments for SPLs are evaluated in this chapter, because they may serve as a background for tool development in the context of the AMPLE project, where new approaches for tooling for SPLs are to be worked out and integration with mainstream approaches is of importance.

Usually and naturally there is a permanent gap between the state of the art in (academic) research and the practices applied in an industrial context. Of course the selection of implementation techniques and the way they are applied in industry is probably as diverse as the companies that use them. But the analysis of implementation techniques applied by the industrial partners of the AMPLE project in Chapter 5 may serve as a representative to get an insight into variation mechanisms used in real-world developments.

This survey concentrates on existing techniques that are already (at least partially) applied in practice. Potential extensions and combinations of technologies – with special focus on Aspect-Oriented and Model-Driven techniques – will be discussed in the upcoming Deliverable 3.2.

2. Criteria for Evaluation

2.1 Criteria for Variation Mechanisms

2.1.1 Concept of Variation Mechanism

It is difficult to evaluate a programming language (or some other implementation technology) as a whole for its support for implementing variability in product lines. Programming languages contain multiple features and allow different design patterns to express variability, often with different qualitative properties. We will refer to the different techniques to express variability as **variation mechanisms**. Instead of analysing entire programming languages and platforms for their support for variability we should identify and compare individual variation mechanisms.

Variation mechanisms can be very different. They can be as simple as object properties or class inheritance, or as complex as extension with aspects or template based generation. There are plenty of such mechanisms and in order to be able to compare them we must view them through a common scheme. In the following we will describe such a scheme and its constituents.

Figure 1 displays the most general structure of a variation mechanism. The primary goal of every variation mechanism is to improve reusability. It achieves this goal by enabling separation of reusable assets from their variations. Thus, when we talk about a variation mechanism we must at first identify the kind of **reusable assets** it deals with and the kind of variation that it supports. The variation supported by a reusable asset can be characterized by a set of variants that can be used to specialize the reusable asset. We will refer to such sets as **variation types** and their elements as **variants**.

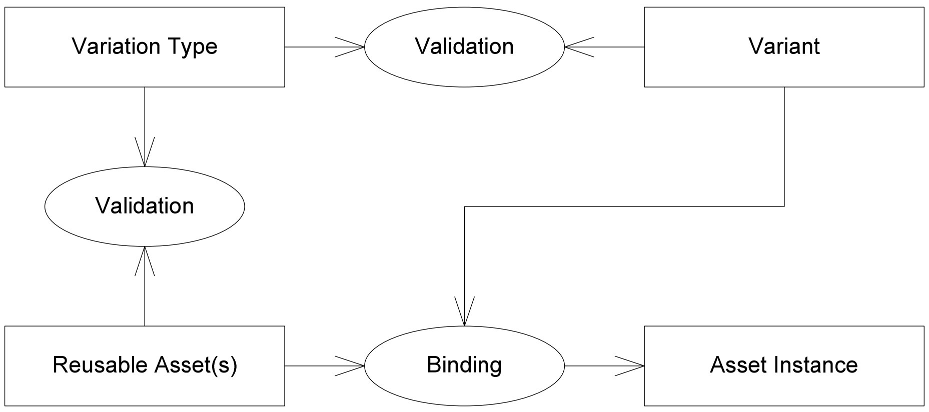


Figure 1. General scheme of variation mechanisms

The minimal scenario is that a variation mechanism provides facilities to instantiate reusable assets by **binding** variants to reusable assets. This process can be as simple as setting a variable value, or as complicated as performing code generation. In a more sophisticated scenario a variation mechanism supports explicit description (or implicit inference) of the type of variation supported by a reusable asset and is able to **validate** the reusable asset and variants against the variation type. On the one hand, it is validated if a variant belongs to the set of variants described by variation type. On the other hand, it is validated, if for all variants that are possible for the specified variation type of a reusable asset, generation will succeed and the result will have certain properties.

For a better understanding of the concepts introduced so far consider the example depicted in Figure 2. Here our variation mechanism is simply a language feature that supports subroutines with one or more parameters. In this case our reusable assets are pieces of code implementing subroutines. The variation type of a subroutine is described by the list of its parameters and their types. Variants are tuples of values that are passed as function parameters, and binding a variant to a subroutine generates a subroutine call with the appropriate parameter values. The two validation processes correspond to type checking on caller side (checks parameters against parameter types) and type checking on callee side (checks subroutine implementation against parameter types).

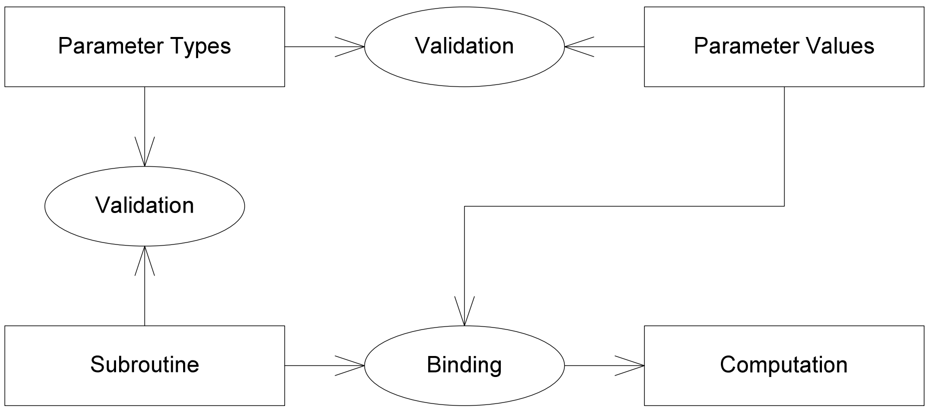


Figure 2. Function as a variation mechanism

Note that the same language mechanisms can be considered from different perspectives depending on what is to be reused. For example, we can consider AOP as technology to extend a reusable code base with unanticipated variations (Figure 3). Then the base code is a reusable asset and an aspect describes the variant. In this case the set of possible variants is not specified, and thus no validation is possible.

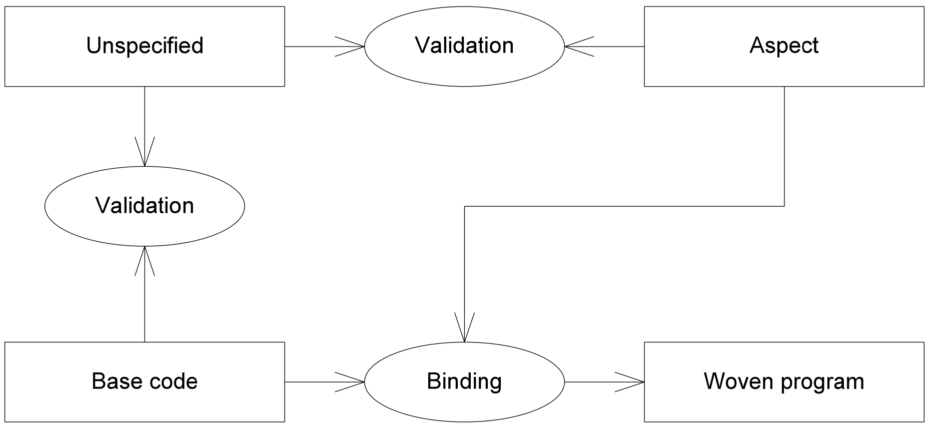


Figure 3. Aspect as extension mechanism

In another scenario (Figure 4) we consider that our product line contains both the base code and a set of aspects that advise this code. In instances of the product line we can select a subset of the aspects. From this perspective our reusable assets are both the base code and the aspects. The variation type is the power set of the set of available aspects. The variation type can additionally impose various constraints on this power set, for example by specifying which aspects are mutually exclusive. We could validate if a given variant (selection of aspects) fulfils the constraints, and if for every allowed selection of aspects the weaver will produce a valid program.

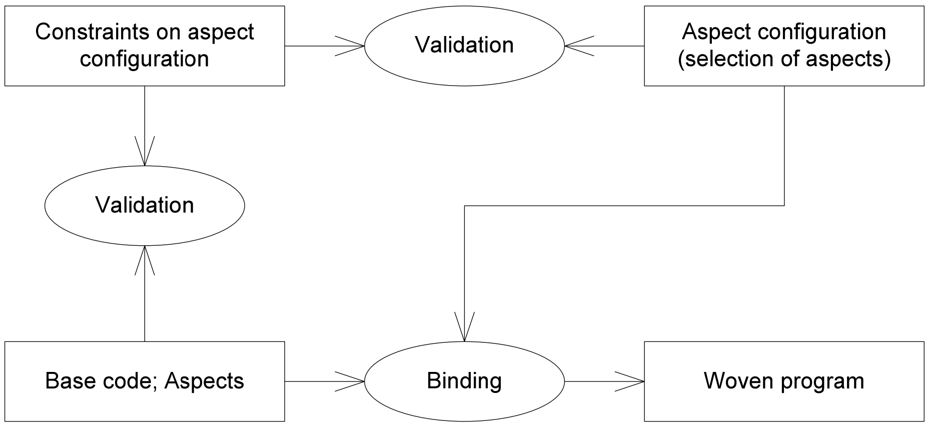

Constraints on aspect configuration
Validation
Aspect configuration (selection of aspects)
Validation
Base code; Aspects
Binding
Woven program

Figure 4. Configuration of aspects as a variation mechanism

2.1.2 Expressive Power

In this section, we will talk about expressive power in a relatively narrow sense: as a characterization of what kind of variation can be expressed by a variation mechanism. In a broader sense expressive power would encompass the issues of further sections, such as modularization possibilities and instantiation model.

First of all we have to identify the object of variation, i.e. the type of the entities produced by the variation mechanism. This characteristic influences all other aspects of the technology. The **type of the produced entities** can be functions, data structures, objects, modules, collaborations, programs or arbitrary artefacts**.**

The essence of a variation mechanism is a transformation that takes reusable assets and a description of the variant as input. Thus the expressive power strongly depends on the **types of the transformations** that we can express:

- **Parameterization.** Reusable assets declare parameters that are used at different places of the assets and the binding process transforms the assets by replacing these slots with given values. There are a large number of mechanisms that use this kind of transformation from simple parameterized routines to conditional compilation. Technically, mechanisms can be implemented in different ways, for example by direct substitution or by using substitution environments.

  Substitution mechanisms can vary by the constraints of **how and where the parameters can be used**, for example generic Eiffel [25] classes cannot inherit from their generic parameter, but this is possible in C++ templates. In C++ conditional compilation the developer can describe only conditionals over the parameters, while in template languages such as Xpand [26] the developer can describe iteration over the parameters.

  The expressivity of parameterization also depends on the supported types of parameter values:

  - **Predefined types.** Usually simple scalar types and strings. This is typical in various configuration languages.
  - **Data structures.** Constructors to build aggregate types, such as arrays or records.
  - **Functions.** Functions can be passed as values.
  - **Objects.** A variant is described by an object, which contains data, operations and references to other objects. Object structures are often used to define models.

  Further criteria for evaluation of parameterization mechanisms are:

  - Support for **default parameters**
  - Support for **partial binding** of parameters

- **Refinement.** Variation can be expressed as a delta to the reusable item. Variation mechanisms can differ by the ways in which they can modify the reusable asset:

  - **Extension.** A transformation can insert new items in the reusable asset. For example AO languages that support only before and after advice can insert new behaviour in the advised code. Another example is Open Classes that allow extending existing classes with new fields and methods.
  - **Overriding.** A transformation can override parts of the reusable asset. An example of overriding is AO languages with support for around advice. Note that in case of class inheritance we can have both parameterization and overriding.

- Refinement mechanisms can further differ by **granularity of refinement**, i.e. the positions where new items can be inserted and granularity of parts that can be overridden. In AOP granularity is usually characterized by a joinpoint model describing the execution points that can be advised.

- An important innovation of AO languages with respect to extension mechanisms is the **possibility to quantify** over the points that have to be extended in an analogous way. The quantification possibilities can, of course, be different and can again be evaluated by multiple criteria.
- **Composition.** Reusable assets describe parts of the entity to be generated and the variation describes how the parts should be composed. There are different types of composition**:**
  - **Merging.** The selected components are merged together. Typical examples are the technologies supporting layered decomposition, such as mixin layers [28], Hyper/J [27], virtual classes with mixin composition [30][29]. In fact, this kind of composition can be derived from almost every refinement mechanism. For example, for class inheritance we can consider multiple inheritance or mixin-based inheritance.

    Merging techniques can differ by the **granularity of composition**, e.g. some of them can compose methods, others not.

    An important issue is how to deal with **ambiguities,** for example when there are alternative implementations for the same method, or when there are multiple aspects advising the same joinpoint. Resolution of ambiguities can be non-deterministic, automatic or manual. The granularity of resolution may vary.
  - **Assembly.** The components are assembled by writing glue code that connects their explicitly exposed interfaces.
  - **Event-based composition.** Components are implicitly composed in an event based system by publishing their events and listening to events published by some other components.
  - **Contribution.** The composed components contribute to some common results of computation. A typical scenario is that components are registered in a list and expose a common interface. Some manager object calls the components in the list and composes their results. For example, components can contribute to building the menu of the main window. This type of component composition is often used to support plug-ins.
- **Arbitrary transformation.** This is a category for very powerful variation mechanisms that do not have clear constraints on the type of transformation that they can do.

2.1.3 Binding Model

The **binding time** of a binding defines the time when a variant is bound to a reusable asset. There are different classifications of binding time. However, for evaluating variation mechanism we are only interested in the technical aspect of binding times, thus, for example, a difference between development time and installation time is not interesting if technically the same kind of binding is done.

So from a technical perspective we can identify the following kinds of binding time:

- **Compile time (static).** The variant is bound before running the software.
- **Run time (dynamic).** The variant is bound in a running system. The runtime binding can further differ by the possibility to **change the binding** during the lifetime of the varying object.

Another characteristic is the **availability time** of the variations, which tells at what point varying artefacts must be available. The artefacts can be available either at compile time or at runtime. In the latter case, it is also possible to change the availability of artefacts during runtime.

We need to differentiate between binding and availability time, because we can have situations where binding takes place at runtime, but the variations must be already available at compile time. For example, in the Strategy pattern [16] we can dynamically decide which strategy we choose, but our choice is still limited by the set of strategies defined at compile time. So we have binding at runtime, but compile time availability. To achieve runtime availability in this scenario we must use some dynamic loading technology.

Another important aspect is the **scope of binding**. Since a reusable asset can be bound to different variants, the variation mechanism can differ by supporting the coexistence of different bindings of the same asset. For example, if we encode variation by a normal class field, we can bind a variant for each object of the class, but if we use a static class field instead, we can only bind one variant for all the objects of that class in the application. There can be different scopes of binding: program, thread, object, module, class, component, collaboration, etc.

2.1.4 Validation

In section 2.1.2 we showed that there are big differences in the expressivity of variation mechanisms. The question is then why not to use the ones that allow expressing arbitrary transformations. The problem with very expressive mechanisms is that it is difficult to check automatically (and often manually) if the transformation will succeed and produce the desired result. As was already mentioned, there are two kinds of validation: the validation of a variant against a variation type and the validation of a reusable asset against a variation type.

In any case, validation needs information about the variation type. First we can categorize the variation techniques by the availability of the variation type:

- **Not available.** In some variation mechanisms the variation type is neither inferred nor explicitly declared.
- **Inferred.** The variation type is automatically inferred from the reusable asset. Note, that the possibility to infer the variation type implies validation of the reusable asset. The classical example here is type inference in various programming languages. A less obvious example is inference of the protected interface of a class, which is sufficient to check validity of subclasses.
- **Explicitly specified.** The variation type supported by the reusable asset is explicitly declared.

Specifications of variation type can differ by their **precision**. If we describe variations as values then the possible precision of describing the intended type of values depends on the power of the type system. For example, in a simple object-oriented language the type of a variable can only specify the class of the object referenced by the variable, while in a more sophisticated system we could also specify the types of the fields of that object. When describing object interfaces we can specify only their signature or also some further semantic properties. When describing meta-models we can limit ourselves to describing their abstract syntax or we can also describe further constraints on the model.

Availability of a variation type, specified or inferred, normally implies that it is checked if variants comply with this type.

However, for **validating reusable assets** it is not always the case. In a lot of cases the validation is not done at all, or is partial in the sense that it does not guarantee successful binding of the reusable asset with all the variants that are possible for the specification of the variation type. The types of partial validations can be very different. For example, in template languages we may validate the generator instructions, but not the code to be generated, or we may validate only if this code is syntactically correct.

In some cases complete validation may be impossible, because variation type is not specified precisely enough. For example, if we have a feature model description language that cannot define dependencies or conflicts between features we are not able to specify all constraints that are necessary to specify which selection of features will result in a valid products.

A lot of variation mechanisms are able not only to validate success of binding of the reusable asset with any of the supported variants, but also to **guarantee further properties of the binding result**. For example, a typical guarantee of class inheritance is that the signature of a class subsumes the signature of its superclass. A lot of variation mechanisms allow one to describe the type of the reusable asset and can check if the result of the binding will always have this type. These possibilities again depend on the power of the type system.

The mechanisms that do not perform validation against variation type must validate variants directly against reusable assets. If such validation is done during binding, there is no sense to distinguish the validation from the binding. However, a variation mechanism with runtime binding can additionally support **static validation of variants against reusable assets**.

2.1.5 Modularity

By modularity we understand separation of concerns. There can be however different **levels of separation**:

- **Structural separation**. Concerns are separated into different modules, but there are no clear relationships between modules.
- **Explicit dependencies**. The dependencies between separated concerns are explicitly declared. Internal consistency of a concern can be modularly checked.
- **Explicit interfaces/encapsulation**. Modules expose explicit interfaces and hide the remaining details from other modules. As a result, the total complexity is reduced, internals can be changed without influencing other modules.
- **Segregated interfaces.** A module can implement multiple interfaces, which are dedicated to different groups of clients. This makes clients of the module more stable, because they depend only upon the interface that they need.

- **Independent structure.** There is no preplanned alignment between the client of a module and the module. They are integrated afterwards.

The higher levels of separation primarily increase the stability and reusability of the modularized concerns, but require more development effort and in some cases may prohibit unanticipated variation.

Another important general criterion for evaluating modularization possibilities is the **granularity (or flexibility) of separation**, e.g. modules can be used to segregate individual classes, methods, or even intra-method constructs

A primary modularization characteristic is the **dependency of reusable code on variation**. Here we can identify the following categories:

- **Unaware**. Reusable code is completely unaware of the variation. This is a typical case when varying code simply uses the reusable code or extends it.
- **Stable abstraction.** The dependency of reusable code on the variation is described by a stable abstraction. The reusable code uses this abstraction without differentiating between variants. This is for example achieved by subtype polymorphism.
- **Inlined variation.** The reusable code differentiates between the variants and this differentiation is done simply in-place using conditional structures. This characteristic applies to variation management with switch/case statements in Java and for conditional compilation in C/C++.
- **Modularized variants.** The reusable code must differentiate between the variants, but the pieces of code that have different dependencies on variation are separated from each other, and the correct piece is selected by some static or dynamic dispatch mechanism.

The separation method depends on the type of transformation. In case of parameterization, the separation is based on **dispatch**. In case of compositional variation, it is important to **separate the components**. In both cases we should evaluate the **level and granularity of separation**.

In case of parameterized variation, we should evaluate the **expressivity of dispatch**: if dispatch is based on subtype relation or on predicate dispatch [63]. In the first case, dispatch expressivity depends on the expressivity of the type system.

We must also evaluate the possibility of **decomposition of reusable assets**, because it can be that a variation mechanism constrains modularization possibilities. A typical example of such a restriction is the expression problem [32][33]. For example, if variation is described by an inheritance hierarchy in Java, all operations, whose implementation depends on these classes, must be defined in the modules containing these classes, while MultiJava [34] allows defining such operations in separate modules.

The possibility of **decomposition of variation type descriptions** may also be an important issue if these descriptions are large enough. For example, one technology may require that a feature model must be defined as a whole, while another technology may allow decomposing the feature model into subtrees even more flexibly.

2.1.6 Other Criteria

The convenience of using variation mechanism is an important criterion in SPL engineering, because configuration of SPL instances sometimes is done by people less experienced in software engineering, for example during deployment. To evaluate the convenience we can evaluate:

- **the complexity** of the language,
- the amount of **infrastructural code** to support the mechanism compared to a fixed binding between the reusable asset and the variant,
- the availability of **graphical tools,**
- the conceptual and optical **distance between the input assets and the binding results**.

The last point is also tightly related to **traceability**. Traceability is understood in the sense of the ability to match the structure of the binding (generation) result against the source assets. Traceability is very important for managing changes and debugging.

The efficiency of a variation mechanism can be evaluated through the following criteria:

- Runtime performance overhead in terms of time and memory
- Amount of generated code
- Performance of transformation and validation steps
- Support for incremental generation

2.2 Criteria for Tool Support

Since SPLs are concerned with software artefacts relevant to all phases of the software lifecycle and at all abstraction level, almost any tool relevant for software development may, in principle, be used in the context of SPLs. In order to define a significant set of criteria, we have delimited the set of considered tools by considering mainly tools that have been explicitly developed for SPL engineering; we have, however, also taken into account features relevant to SPLs but present in general tool suites for software modelling.

SPLs require the management of variability over the whole development process, involving, in particular, software artefacts described at a large range of different abstraction levels. Hence, tools for product development using SPLs should support an integrated development process that allows creating software artefacts, ensuring properties, and generating tests as well as product quality level code.

In this section we present the criteria used to evaluate existing tools for SPLs with respect to their support for integration of the development process. The focus lies on the core concepts supported by the tools, coverage of development activities over the development lifecycle and integration opportunities with other tools that are potentially relevant to the development of SPLs. The overall set of criteria we consider in this section includes, partially with adaptations, the set of criteria introduced in the previous section. We have made some adaptations in order to take into account the fact that some of the previously presented criteria are not applicable to tools or at least have a different range of possible options. In Chapter 4 we evaluate four different tools (three major SPL specific tools and a general-purpose tool for model-driven engineering) with respect to the criteria presented here.

We consider three main groups of criteria: (i) the underlying conceptual and technical concepts, (ii) the extent to which a tool covers the development process and different target implementation infrastructures (such as a component model on top of which the SPL is implemented) and (iii) usage-related parameters, such as the availability of the tool and how it may interoperate with other tools.

2.2.1 Concepts

What are the abstract and technical concepts underlying the tool?

1. **Variation mechanisms**: the techniques used by a tool to express variability.
   - **Variation management principle**: The general way the variability is managed.
   - **Reusable Assets**: the kind of reusable assets that the variation mechanism deal with, e.g. tools may deal with any kind of asset in a generic way, or provide support for more specific assets such as definition of requirements, language-specific source code, etc.
   - **Variation types**: the set of variants that can be used to specialize the reusable assets, e.g. files, requirements, source code structures, etc.
   - **Variants**: the concrete elements of a variation type that can be selected to specialize the reusable assets, e.g. concrete files, specific requirements, etc.
2. **Expressive power**
   1. **Transformation type**: expresses the kind of transformations on assets, e.g. tools supporting general assets may support transformations such as file generation, text substitution, etc.; tools supporting specific assets may support adding or removing requirements, or language-specific transformations such as refinement, composition or arbitrary transformations, etc.
   2. **Granularity**: the smaller unit that can be modified in an asset, e.g. the granularity for tools supporting generic asset may be files, blocks of text, etc.; the granularity for tools supporting specific assets may be requirements, source code structures, components or business logic.
3. **Binding model**
   - **Binding Time**: the time when the binding between assets takes place.
   - **Availability time**: the time when varying artefacts must be available, e.g. some tools may need variants to be completely available before binding, whereas other tools may need variants to be just partially available, and they are completed in the binding.
   - **Scope of binding**: the scope, in term of a software artefact, where the binding applies.
   - **Validation**: how tools ensure the correctness of the variation mechanism
   - **Availability of variation types**: existence of variation types that allows validating variants (variant complies with its type). Variation types may be not available, inferred or explicitly specified, e.g. the variation type can be explicitly defined as a file, and then it has to be checked that all the variants are effectively files.

- **Validation of binding**: the way to ensure that asset instances are correct. It is related to the correct selection of features imposed by possible existing constraints. This validation can be possible or not possible.

4. **Modularity**

- **Structural separation**: the structural mechanisms that the tool provides to separate concerns, e.g. a tool can provide a notion of modules, and/or ways of grouping features.
- **Explicit dependencies**: tells how dependencies between modules are declared, e.g. some tools can provide explicit dependency declarations, whereas for other tools the dependencies can be implicitly inferred.
- **Segregated interfaces**: tells how a module or a complete asset can be instantiated for different groups of clients.
- **Asset-variation dependency**: the dependency of reusable code on variation, e.g. an asset in a tool can be defined without depending on the different variants for the varying part of the asset.
- **Decomposition of assets**: tells how the variation mechanism of a tool allows an asset to be decomposed in different parts that can be treated independently, e.g. the variation mechanism of some tools could restrict the decomposition of an asset, whereas in others such a separation could be always possible.
- **Decomposition of variation descriptions**: tells how the descriptions of the variation can be decomposed.

2.2.2 Functionality

What are the main functionalities provided by the tool in terms of SPL and product lifecycle management?

1. **Process coverage**

- **Definition of SPL:** feature models, DSL
- **Analysis/validation of SPL (domain space):** Are there any specific functionality for domain space analysis or validation, for example analysis of feature models, domain requirement, SPL architecture or reusable assets, etc.?
- **Analysis/validation of products (application space):** Are there any specific functionality for application space analysis or validation, for example validation of product configuration, product requirements, product architecture, product assembly, etc.?
- **Product assembly:** Are there any specific functionality to create a product by assembling assets?
- **Product testing:** Are there any specific functionality to test products?
- **Product execution:** Is there a specific support for product execution?
- **Product maintenance:** Are there any specific functionality for product debugging or evolution?
- **Support for specific application domains:** Are there any specific functionality for one or more specific application domains?

2. **Expressiveness of feature model editors**

- **Hierarchy of features:** boolean
- **Feature selection:** one-of, more-of, optional, mandatory
- **Support for features labelled with values:** boolean
- **Assertions on feature values:** boolean
- **Representation of features:** graphical, textual
- **Multiple feature models:** boolean
- **Feature-model dependencies:** boolean

3. **Product engineering**
   - **Support for managing feature-model instances:** boolean
   - **Support for product instantiation:** boolean
   - **Execution environment:** boolean
   - **Editors to manage dependencies between feature models:** boolean
   - **Code generator:** boolean
   - **Implementation targets:** Are there one or more specific programming languages as implementation target of the tool or is it a technology-agnostic tool?

2.2.3 Usage

All criteria related to the usage of the tool.

- **Availability:** free, licensed, etc.
- **Configurability:** Is the tool configurable for different tasks?
- **Extensibility:** is it possible to extend the tool and by what means?
- **Interoperability:** how does the tool interoperate with other tools? Does it support the use of different file formats?
- **Usability:** Is the tool easily usable and in which context?

3. Existing Variation Mechanisms

This section discusses approaches for implementation of variability in software product lines and evaluates them according to the criteria of Section 2.1. It is not realistic to evaluate all possible variation mechanisms, because most mechanisms in programming languages and other implementation technology deal with some kind of variability. Therefore, we will evaluate the mechanisms that are most popular in industrial applications and the mechanisms that are directly relevant to the project, such as aspect-oriented programming and code generation. It should also be noted that the approaches are not mutually exclusive of one another. For example, some form of configuration management is generally used in any real world approach while many component-based approaches may use object-oriented techniques at their core.

3.1 Object-Oriented Mechanisms

Object-oriented programming languages such as C++ and Java are at the core of the mainstream implementation technology. In this section we evaluate the basic variation mechanisms that are available in such languages. Since Java is selected as base implementation language in the project, we refer to object-oriented mechanisms as they are implemented in Java, unless noted differently.

3.1.1 Parameterization

The most common variation mechanisms are based on parameterization. Parameterization in object-oriented languages is available for different scopes: methods are parameterized by their explicit parameters; objects are parameterized by the values of their fields and constructor parameters; parameterization of classes and packages is possible by setting values of static class fields. In this way we can parameterize reusable assets of different size: methods, classes, and packages.

The types of parameters in object-oriented languages are usually object types, which can describe primitive types as well as complicated models. The variation type is normally described by defining interfaces, and variants are the classes implementing these interfaces. Since functions can be modelled as objects, parameterization by functions is also possible, but usually requires a significant amount of infrastructural code.

By giving initial values to fields and by setting them after object construction we achieve the effects of default parameters and their partial binding. This is, however, not possible for method parameters.

The parameters are bound at runtime. The field values can be freely changed after the binding. The classes implementing the variants must normally be available at compile time. Nevertheless, because of dynamic class loading and reflection in Java it is possible to postpone availability of variants until runtime.

Statically typed object-oriented languages such as Java provide complete static validation of both reusable assets against the declared parameter types as well as validation of the actual parameter values. Usually only the signature of the parameters is specified. Languages such as Eiffel [25] allow definition of semantic properties of the parameters in form of pre- and post-conditions, but they are not validated statically.

Polymorphism and late binding enable a quite good level of separation between reusable and varying functionality: The reusable assets use the varying functionality over stable abstraction defined by the interfaces of their parameters. The implementations of different variants are separated from each other, since they are defined in different classes.

The latter properties characterize the major qualitative advantage of object-oriented parameterization methods over traditional procedural programming, where variations are usually handled by conditional statements. Besides, dispatch over virtual tables makes the object-oriented selection mechanism more efficient than selection using conditionals.

The disadvantages of parameterization as variation mechanism are preplanning and infrastructural code. The variation possibility should be prepared in advance by defining necessary parameters and interfaces. For parameterization using object fields we must additionally define constructors and setter methods.

Object-oriented languages impose limitations on decomposition of the definition of variation type and implementation of variant. All the code that depends on a variant must be defined in the module/class that represents this variant. It is not possible to distribute the variant-dependent code in multiple modules.

As previously mentioned, the object-oriented mechanisms support parameterization with method, class, and package scope. However, there is no explicit support for parameterization at the dynamic scopes larger than a single object. If we want that a collaboration of objects share a parameter value, we must implement infrastructure that passes this value explicitly from object to object.

Design patterns, such as Singleton, Strategy, State, Command, Composite [16], describe special cases of object-oriented parameterization. Singleton describes the case of parameterization on the scope of an entire program. Strategy, State, Command and Composite describes parameterization over object fields in order to deal with variation in different situations: Strategy describes variation of an algorithm, State – variation of object state, Command – variation of an action to be performed, Composite – variation of parts of a data structure.

3.1.2 Inheritance

In class-based languages, such as Java, inheritance is necessary for achieving (inclusion) polymorphism and late binding, the advantages of which were discussed in the previous section. In this section we will evaluate another usage of inheritance, which is called implementation inheritance, and is the only form of refinement directly supported in object-oriented languages.

With implementation inheritance, the reusable functionality is captured by a base class and the varying functionality by its subclasses. Since a class can have multiple subclasses, it is possible to define multiple variations of the same base functionality. The extensions can override the reusable functionality at the granularity of methods.

There is no explicit description of variation type. The extensions are validated statically against their base. The possibility of decomposition of the reusable and varying functionality depends on the support for multiple inheritance. In a single inheritance language, such as Java, reusable and varying functionality can be organized only along a strict linear hierarchy.

The main advantage of inheritance is the possibility of unanticipated variation, which in this case is known as the Open-Closed Principle [35], because the base classes are unaware of their extensions. Besides, inheritance requires a very minimal amount of infrastructural code and can be very efficiently implemented.

The major disadvantage is a low level of separation (explicit dependency) between the varying and the reusable code, which makes evolution of the reusable code problematic. The problem of lack of contract between a class and its subclasses is also known as the Fragile Base Class Problem [15].

Besides, inheritance is limited to static variation. A solution for dynamic refinements in object-oriented languages is described by the Decorator design pattern [16]. The problem is that this solution invalidates some of the advantages of inheritance: it requires a lot of infrastructural code and is much less efficient.

Some of the problems of inheritance as available in mainstream object-oriented languages such as C++ and Java are alleviated by mixin-based inheritance [31]. Mixins are classes, which cannot be instantiated, that are parameterised by their superclass. Unlike standard subclasses, mixins are explicitly abstracted from their superclass. This makes them more stable and reusable.

Mixins can also be used to express a simple form of compositional variability: we can produce different variations of a class by combining different lists of mixins. A similar effect can be achieved with multiple inheritance.

3.1.3 Generics

Generics is a mechanism for parameterization of classes by types. The well-known languages supporting generics are Java [37] and Eiffel [25]. A typical application of generics is the implementation of collection classes that can be used with different element types.

The main difference between generics and simple parameterization is that parameters are not used in expressions, but in types. It is a static variation mechanism. A variation type is expressed by constraints on the parameters types. Java and Eiffel provide an efficient implementation of generics and a complete modular validation of generic classes.

C++ templates are an alternative solution for parameterization of classes by types. The mechanism is semantically more powerful than generics because it additionally enables dispatch of a generic class by its parameters by a mechanism called template specialization: beside generic implementation of a class, the developer can provide its specializations for specific parameter types. The disadvantages of C++ templates are a lack of modular validation of template classes and duplication of generated code.

3.2 Frameworks

Johnson and Foote describe an object-oriented framework as "… a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes." [13] The major characteristic of object-oriented frameworks that distinguishes them from simple class libraries is that the control flow is managed by the framework rather than by the application classes. Application specific behaviour is triggered at predetermined extension points or "hot spots" in the framework.

Extension points in object-oriented frameworks are implemented using inheritance and late binding. The varying code is defined in subclasses of the abstract classes of the framework, and is used in a polymorphic way in the framework. The Observer design pattern [16] describes a solution supporting multiple variations at the same extension point. In procedural languages such as C the extension points are usually implemented using call-back functions.

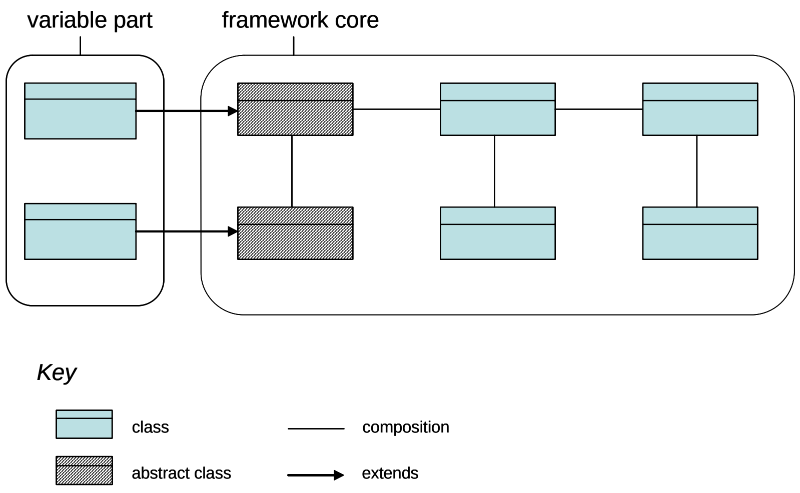


Figure 5. Object-oriented framework

Frameworks are a mainstream solution for large-scale reuse with explicit support for variability. In this case the framework is the reusable asset, and the variants are the application-specific subclasses of the classes of the framework. Frameworks combine both parameterization and refinement, because they are explicitly parameterized by their extension points. On the other hand the framework classes can be extended by inheritance, also in unanticipated ways. Refinements are not possible in so-called black-box frameworks.

The variations are bound to the framework at runtime, and runtime availability of the variants can be achieved using reflection mechanisms for class loading. The framework is separated from its variations by explicit abstractions. Extension of the classes of the framework can lead to tight dependency of varying code on the framework. However, it is not the case when the framework is used only in a preplanned way over explicit abstractions.

Frameworks are well suited only for preplanned variations that must be supported by an infrastructure prepared for this purpose. One of the problems with frameworks is the size and complexity of the infrastructural code to support variations. If a variation is of crosscutting nature, the infrastructural code to support this variation will be scattered in multiple places across the framework.

Another big problem is that frameworks tend to be monolithic. Usually the framework and the application code are separated by strict abstractions, but there are tight dependencies between internal parts of the framework. The consequence is that the framework must always be used as whole even if an application needs only a part of its functionality.

Since the extension points of the frameworks are described by interfaces, each variation must be defined in a single class. Such a mechanism is not well suited to describe more complicated variations that are implemented by multiple classes. This problem is addressed by the Abstract Factory design pattern [16], but this pattern requires additional infrastructural code and has problems with extensibility. Another problem is that the type systems of object-oriented languages are not powerful enough to express covariant dependencies between the classes that implement the variations with the consequence that the covariant classes must use type casts to access the functionality of each other.

Usage of multiple frameworks in an application poses further problems, which are discussed in detail in [38].

3.3 Component Technology

Component-based software engineering (CBSE) [14] refers to the development of software systems from reusable components. There is no general consensus on the definition of the concept of component. In the broadest sense a component is any reusable piece of software with a well-defined interface. From this perspective we can consider modules and classes in programming languages as components. However, other definitions impose further requirements on components: they must be independently deployable, abstracted from their dependencies on other components, abstracted from middleware and so on. In the following we will discuss different aspects of component technology with the focus on their support for variability.

3.3.1 Component-Based Architecture

The basic principle of component-based architecture is decomposition of software into smaller parts that communicate over explicitly defined interfaces. The components can be further hierarchically decomposed into smaller components. In order to avoid tight dependencies, components do not instantiate each other directly but rather declare their dependencies in form of expected interfaces. The interfaces implemented by the component itself are then called provided interfaces. The composition of components is performed on a higher level by linking expected interfaces to compatible provided interfaces. The components can be composed using a conventional programming language or an architecture description language (ADL), which can express the composition more concisely and visualize it. An overview of ADLs is given in [1].

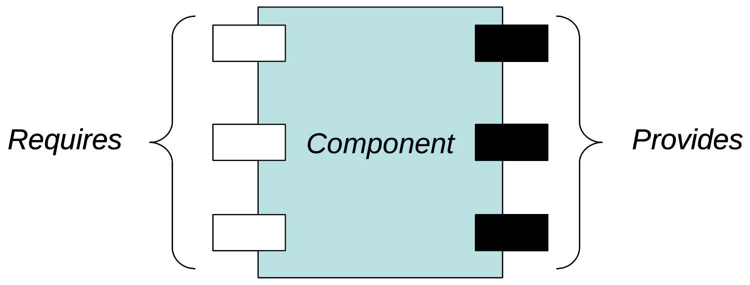


Figure 6. Component interfaces in CBSE

Component-based architectures express compositional variability. The components are reusable assets and variants are different compositions of components. The provided and expected interfaces of components define constraints for composition and thus they can be seen as specification of variation type.

The main advantage of component-based architectures is their strong support for modularity. Components are separated from each other at the level of segregated interfaces. The components as well as their compositions are validated in a modular way, which also enables good support for separate compilation.

The problem with component-based architectures is their assumption that software can be easily decomposed in a strictly hierarchical way. Components are usually implemented as classes or as groups of classes, which means that they cannot cross class boundaries. A hierarchical architecture also implies that subsystems that are defined as compositions of components are responsible for completely defining interactions between these components. The problem is that variations in product line are often defined in terms of features that cross boundaries of classes and their compositions.

Component-based composition is not suitable for expressing unanticipated variability. It is difficult to extend components and their compositions with new functionality or to integrate components in unanticipated ways.

3.3.2 Abstraction from Middleware

One of the primary goals of the popular component frameworks such as J2EE [40] and .NET [41] is separation of business logic from various non-functional concerns, such as distribution, persistence, transaction management, and security. These concerns are then specified concisely using domain-specific abstractions, which leads to significant reduction of code size and complexity.

Separation of non-functional concerns also enables their independent variation. For example, separation of the distribution concern enables independent variation of physical architecture of the system. Complete separation of some other concerns such as persistence or transactions appeared to be less useful in practice. Therefore, these concerns are often specified in class annotations. This means that they are not syntactically separated from the business logic. Nevertheless, such a solution still supports variation of implementation of non-functional concerns. For example, if persistence and transactional logic of a class is specified by standardized J2EE annotations, this class can be used in different application frameworks. This means that declarative specifications of non-functional concerns enable variation of the platform.

Technically, declarative specifications of non-functional concerns are implemented using the approaches for implementing DSLs that are discussed in Section 3.7 and Section 3.8.

3.3.3 Abstraction from Implementation Language

An interesting feature of the CORBA [42], .NET [41] and COM [43] frameworks is support for multiple programming languages. A software system can be built of components implemented in multiple programming languages. In CORBA and COM this is enabled by language independent interface specifications, while .NET achieves this by translating all specific languages to a common intermediate language. From the perspective of variability management, these technologies enable composition of reusable assets that are implemented in different programming languages.

3.3.4 Independent Deployment

Using programming languages, such as C++, a change in one module of an application usually requires recompilation of the modules that use it. Besides, there is no possibility to install an application in parts or to replace parts of an already installed application.

Such a situation is not acceptable from the perspective of component-based development, because components can be developed independently from each other by different vendors or teams. It may be necessary to upgrade components of a software system independently of each other or to install new components providing new functionality. From the perspective of variation management independent deployment is interesting, because the components that implement different variations can be added to the system independently from each other. Since the set of available components is known only at runtime, independent component deployment enables runtime availability of variations.

Independent deployment of components is enabled by technologies supporting some form of dynamic linking and naming service. For example, dynamic deployment of COM components is based on dynamically linked libraries (DLL) and entries in Windows Registry that relate component names to corresponding DLLs.

Since Java links classes at load time, it is possible to replace individual Java classes and libraries in an installed system. OSGi [44] is a Java-based framework that supports a more controlled form of dynamic loading. The components in OSGi are bundles of Java classes and other artefacts that declare dependencies on other bundles. OSGi not only loads and links the bundles at load time, but also provide facilities to install, start, stop, update and uninstall them in a running system.

3.3.5 Service-Based Composition

So far we discussed explicit composition of components, in which the developer must explicitly link provided and expected interfaces of individual components. In a lot of cases this matching can be done automatically. The interfaces correspond to the services that are provided by or needed for the components, and a component that needs some service can be automatically linked to a component that provides that service. Technically, this can be achieved either by service registries or by dependency injection [45].

An example of service registry mechanism can be found in the OSGi framework. The framework maintains a registry, which maps services (identified by corresponding interfaces) to objects that provide them. An object that needs some service can use the registry to retrieve the list of its providers. Service providers can be registered and unregistered at runtime. The cases when there is one provider or there are multiple providers for certain service must be handled by the users of the service.

In the dependency injection approach each component belongs to some container, which is responsible for finding and passing the necessary services to the component. There are at least 3 different dependency injection techniques: interface injection, constructor injection, and setter injection. These techniques differ in the way the container passes services to the component. For example, in setter injection a class declares a setter method for each service that it needs. Each setter has a parameter of a type that identifies the service. Dependency injection is implemented in Pico Container [46] and in the Spring framework [47].

Dependency injection requires much less infrastructural code than the service registry approach, but service registries can be useful when more flexibility is needed, because it supports multiple providers for the same service and possibility to register and unregister them at any time.

Service-based composition in general implements a somewhat different type of variation than explicit component composition. The variant in this case is described as a set of selected components, while their wiring is determined automatically. Thus it is less expressive than explicit definition of the wiring, but on the other hand it is much more concise in the cases when the expressivity is sufficient.

The automated composition is also less reliable, because there is no static validation that guarantees that all components will be connected to all services that they need. However, by sacrificing some static safety service-based composition provides more flexibility for runtime variation of component composition.

3.3.6 Event-Based Architectures

The behaviour of object-oriented architectures is often seen as interactions in which objects pass messages (or events) to each other. However, there are certain constraints related to this form of interaction: the message is sent from one sender to one receiver, the sender must have a reference to the receiver, the message is sent synchronously. The primary goal of event-based architectures is to enable more flexible patterns of communication between components: multiple senders and receivers, event filtering by various conditions, asynchronous communication, etc.

As in service-based composition, the components are not explicitly linked to each other. Instead they are implicitly linked over the events that they produce and consume. Also, analogously to service-based composition, the variant is described by a set of selected components, while component connections are determined automatically. The major difference is that service-based composition is based on synchronous calls between two components, while in event architectures the calls can be asynchronous and an event can be consumed by multiple components. Thus, event-based architectures provide more expressivity for defining connections between components.

3.4 Aspect-Oriented Programming

Aspect-oriented programming addresses the problem of separation of scattered and tangled concerns. Aspect-oriented languages enable modularization of functionality that crosscuts boundaries of classes, and provide quantification mechanisms for concise expression of the relationship between the crosscutting and the base functionality. In this evaluation we will use AspectJ [48] and CaesarJ [29] as reference aspect-oriented languages, because AspectJ is the de facto standard in industry, and CaesarJ is a language developed by one of the project partners.

From the perspective of variability management, aspect-oriented languages are interesting, because they enable separation of crosscutting varying functionality from the rest of the program. The transformation supported by aspects is refinement with possibility of overriding. The overriding is possible only in the aspect-oriented languages that support so called *around* advices.

The granularity of extension supported by an aspect-oriented language depends on its *jointpoint* model, because joinpoints are the places where aspects can attach new functionality. AspectJ and CaesarJ support joinpoints that cross boundaries of methods, e.g. method calls and field accesses. In this way extensions at very fine level of granularity are supported. AspectJ also supports *introductions* that enable extension of the static structure of existing classes. They can introduce new fields, new methods and inheritance relations.

The conciseness of expression of the extensions largely depends on the pointcut language, which determines the possibilities of quantification over joinpoints. The quantification in AspectJ and CaesarJ is mostly based on the static structure of programs. Quantification over various dynamic conditions is more powerful, but its efficient implementation is still a topic of ongoing research. [49]

Since in AspectJ all active aspects must be known at compile time, it is not possible to bind a variant at runtime. CaesarJ enables runtime binding by supporting dynamic aspect instantiation and activation. The aspects that can be selected for activation must still be available at compile time. Runtime availability is possible only in implementations that support load-time or runtime weaving [50][51].

The main advantages of aspects are support of variation of crosscutting functionality, possibility of unanticipated variation and small amount of infrastructural code.

The biggest problem of aspects is their impact on modularity. The separation of the aspects from the modules that they advise is only at the structural level. It is difficult to understand the dependency between the aspects and the base code and to support safe evolution of the base code.

Another problem is the interaction between aspects. It can be difficult to predict the effect of multiple aspects advising the same joinpoint. Especially problematic are conflicts and implicit dependencies caused by introductions. For this reason, introductions are not supported in CaesarJ. For extension of classes with new state CaesarJ provides a mechanism of wrappers, which are defined locally inside aspects and therefore cannot lead to conflicts and implicit dependencies.

Modularity of aspects can be improved using a pattern called Crosscutting Interfaces [54], which uses a set of static pointcuts to define a boundary between the base functionality and the aspects. Open Modules [53] propose dedicated language features to control aspect visibility.

The runtime overhead of aspects without the dynamic features, dynamic activation or usage of dynamic conditions in pointcuts, is negligible. An efficient implementation of some of the dynamic features is possible in dedicated virtual machines, such as Steamloom [50]. Since aspects have global effect, support for incremental weaving is problematic. Compilation time is also a difficult issue in aspect-oriented languages, based on static weaving [52].

3.5 Feature-Oriented Programming

A feature can be seen as a logically cohesive piece of functionality, which normally corresponds to a set of related requirements in a functional specification of a program. Thus, varying (optional, alternative, etc.) requirements can be represented by varying (optional, alternative, etc.) features.

The goal of feature-oriented programming (FOP) is to modularize programs into features. Since features are present in most stages of software development, such modularization has various advantages, but in this document we will consider only the advantages related to variability management.

Variability in FOP is achieved by decomposing a program into pieces of code that implement different features and then by generating different variations of the program for different selections of features. The reusable assets in this case are the modules implementing individual features, the variants describe selection of features, and the binding are compositions of the modules implementing the selected features.

A class can implement multiple requirements that belong to multiple features. On the other hand, the implementation of a feature can require multiple classes. Thus the modules in FOP are collections of partial class implementations. In the following, we will evaluate different approaches that support such modularization.

FOP is implemented in GenVoca [12] and AHEAD [11], which provide tools for composition of so-called *mixin layers* consisting of partial definitions of classes. The transformation type of the mechanism is composition based on merging at the granularity level of methods. However, slices of the implementation of a method can also be distributed into multiple layers and combined using the semantics of super calls. The name clashes that occur during merging are resolved by the order in which the layers are composed, i.e. the methods of a layer can override the corresponding methods of the layers that are further according to the order of composition. In this way more specific features can override the functionality of more general features.

The composition takes place at compile time, and the result of each composition is a separate program. The approach requires very little infrastructural code and does not introduce any overhead on runtime performance.

The main problems with AHEAD are related with its weak support for validation and modularity. The separation of feature modules is at structural level, which means that the dependencies between feature modules are not declared. This makes it impossible to validate them in a modular way and to support their incremental compilation.

These problems are alleviated in other approaches that support similar layered decomposition.

Definitions of classes can also be distributed in multiple modules in *Open Classes*, implemented in MultiJava [34]. This achieves separation of features at the level of explicit dependencies, and provides their modular checking and incremental compilation. However, modular checking in Open Classes is achieved at the cost of flexibility of separation: an implementation of a method must be located in the module of its class or in the module where the method was introduced. Besides, Open Classes do not support the composition and overriding semantics of method implementations available in mixin layers.

Layered decomposition is also possible in languages supporting *virtual classes* and *deep mixin composition*, e.g. gbeta [30], CaesarJ [29] and J& [55]. The composition semantics of these approaches also support overriding and composition of method implementations. The abstract virtual classes of CaesarJ enable definition of interfaces for feature modules. In this way it is possible to separate the implementations of different features at the level of segregated interfaces. Besides, virtual classes support coexistence of multiple different combinations of feature modules within the same program and their dynamic instantiation.

3.6 Conditional Compilation

Conditional compilation is a simple and widely used variability mechanism in languages such as Ada, C and C++. Variant, optional, and alternative code segments are marked using pre-processor directives. In the example shown in Figure 7, platform specific code is marked using a series of `#ifdef` statements. Defining a token `WINDOWS`, by using `#define WINDOWS`, will then allow the pre-processor to include code segments that delineate the windows specific code.

```
//Non platform specific code…
//…
#ifdef WINDOWS
Windows specific code…
#endif


#ifdef MACINTOSH
Macintosh specific code…
#endif


#ifdef UNIX
Unix specific code
#endif
```

Figure 7. Conditional compilation

Conditional compilation is an easy way to configure software by allowing fragments of program code to be included or omitted from the final code, depending on whether a symbol has been defined or not. In the C and C++ languages, in addition to the `#ifdef` command there are also `#ifndef` (if token not defined) `#undef` (undefine the token) and `#else` (alternative code) directives among others.

Pre-processor directives were not seen as important for the Java programming language, mainly due to its platform independence and ability to do a simple form of conditional compilation by surrounding variant code segments in a test (Figure 8). Therefore, if the compiler can prove the code will never get executed it removes it, although this is rather simplistic and inflexible for anything but small-scale problems.

```
private static final boolean DEBUG = false;

…code

if(DEBUG){
…debugging code (included if DEBUG set to true)
}
```

Figure 8. Conditional compilation in Java

Conditional compilation is usually used in combination with build scripts, which define the process to build various configurations of a program. The variation is achieved by controlling the set of the files to be compiled and by specifying the values of the variables used by the pre-processor. There can be multiple build scripts for building different configurations of a program or a single script parameterized by corresponding configuration variables.

Conditional compilation implements parametric variation. The reusable assets are the source files and variants are expressed by values of the configuration variables. These variables are not typed and can take primitive values, usually Boolean flags and numbers. The variation is bound at compile time.

The biggest problem with such a variation approach is modularity, because the parts of code depending on different variants are not separated from each other, which also often leads to complicated, obfuscated code.

The supported variation is usually described informally in form of comments of configuration variables. Validation of the variables is usually implemented by explicit checks in the build scripts. However, the biggest problem is validation of the reusable assets. Validation takes place only during the build, which means that validation is performed only for specific variants; therefore, a change for one configuration can lead to inconsistencies in other configurations.

Incremental compilation is supported in the context of fixed configuration. Changing a value of a configuration variable usually requires complete rebuilding.

3.7 Code Generation

Code generation [10] is an increasingly popular generative technique that produces code from a higher abstraction. In the past, the term code generation was used to describe the process of turning source code into assembly code, although modern usage typically means the production of the programming code itself. Examples of code generation in the modern sense include GUI builders, domain-specific languages, macros, template languages and so forth. Code generators encapsulate the complexity and finer details of program code, allowing developers and system configurators to concentrate on domain-specific configuration details.

A prime example of the strength of code generation is in the elimination of the redundant complexity that pervades the J2EE platform. A large database application that utilizes Enterprise Java Beans (EJB) typically requires two interfaces per table. Additionally, each table may also require multiple classes to handle the mappings between the different tables. In a database system that has hundreds of tables this can result in literally thousands of files, with many of these files sharing a great deal of 'boilerplate' code commonality with one another. By using a code generator and suitable templates, it is possible to automate the creation of these files. Moreover, changes to a schema are propagated throughout the code, and the templates are reusable for other database applications. In contrast, if the application were entirely hand coded from scratch the chances of anything being reused would be opportunistic at best.

The input for code generation is usually a model that describes some higher-order abstraction of the system. The models are expressed in domain-specific languages (DSL) that can have a textual as well as a graphical concrete syntax. This approach of building software systems is known as model-driven development (MDD) [7]. MDD technology consists of a variety of tools for different purposes: defining and implementing DSLs, building model editors, validation of models, defining model to model transformations, and code generation, which is also known as model to code transformation. An extensive overview of the MDD techniques is given in our survey of the state of the art in product line architecture design [1]. In the following we will only evaluate code generation techniques by the criteria of Section 2.1.

The reusable asset in the code generation is the generator itself, i.e. its implementation and all artefacts that it uses for generation. The input models are the variants, and their meta-model describes the variation type.

The expressivity of the transformation depends on the expressivity of the language used to implement the generator. Using *brute force* generation, generators are implemented in a general purpose programming language (GPL). Often the generator language is the same as the language of the generated code. Another approach is to use specific DSLs for generation. The advantages of DSLs for code generation are in fact the same as the advantages of DSLs over GPLs in general:

- By providing specifically designed constructs, DSLs can express generators more concisely, reduce the implementation complexity and the amount of infrastructural code
- By constraining expressivity, DSLs protect developers from wrong design decisions. In GPL developers must work out suitable idioms and rules and follow them in a disciplined way. Besides, there is a danger that someone who does not know these rules will break the design assumptions.

Typical DSLs for building generators are various template languages, such as XSLT [58], XPand of openArchitectureWare [26], XVCL [56] and ANGIE [57]. The generators are implemented by a set of templates. Templates mix pieces of code to be generated with template instructions that control generation. The languages provide constructs for parameterization of the source code, conditional generation, code repetitions, navigation over the input model, and search in the model.

Generator code is modularized by decomposing it into multiple templates. The templates can have parameters and use other templates in their implementation. XPand provides a dispatch of templates on their model parameter. In this way it is possible to separate code depending on different variations in the model. XPand also has aspect-oriented constructs that enable modularization of crosscutting concerns at the level of templates.

An advantage of template-based generation is that it minimizes the optical difference between the generator and the generated code, and thus it makes it easier to understand the output of generation. The distance between the generator and its output is especially big in API-based generation, which makes such generators very difficult to understand.

Template languages, like brute force generation, are not bound to a specific target language. An obvious advantage is that they can be used for generation of code in different languages and even for non-code artefacts. The disadvantage is that such generalization makes it impossible to provide any validation of the pieces of target code contained by the templates. Therefore, only the instructions of the template can be validated, which, of course, is not sufficient to guarantee any properties concerning consistency of the generated code. Lack of modular validation is not so critical in template-based generation, because generation takes place at compile time, thus, the generated code can be validated at compile time by some compiler of the target language.

A better support for validation is available in language-specific template-based generation approaches, such as template meta-programming in C++ [5][59] or in Haskell [60]. These approaches check the syntactic correctness of the templates. They are however not specifically designed for code generation from models and, thus have similar problems as brute force generation with GPLs.

Syntactic checking is also an advantage of API-based generation, because the generators work at the level of the abstract syntax trees. They are suitable for complicated transformations, especially when no larger solid pieces of generated code can be identified. In other cases the generators implemented in this way are much larger and complicated than their template-based counterparts.

Complete type checking of code generators is difficult and possible only by constraining their expressivity. Type-safe solutions are proposed for conditional variation of method declarations in a class [61] and iteration over methods of a class to generate methods of another class [62]. These solutions are very new and cover only very special cases of code generation.

3.8 Interpreters

In a previous chapter, we mentioned that variation can be expressed by models of various DSLs, which are then transformed to code of a conventional programming language using code generation. Another approach to implement a DSL is to write an interpreter for it in some programming language. In such a scenario, the interpreter is the reusable asset and variation is bound at runtime by calling the interpreter with some program.

The advantages and disadvantages of using interpreters instead of code generation are discussed in [7]. It is argued that it is common to use code generation to express static aspects of a system, while interpreters are suited when the DSL expresses behaviour. In fact, if there no dynamic behaviour related with a DSL, there is nothing to interpret. Thus, it makes sense only to compare interpreters or code generation with respect to implementation of DSLs with dynamic behaviour.

The major advantage of interpreters is that they provide variation at runtime. However, in the languages supporting dynamic loading, such as Java, it is possible to use code generated at runtime. Explicit execution of DSLs in interpreters can be advantageous for traceability, because there is a direct link between the execution state and the executed program.

The major argument against using interpreters is loss of performance. The generated code can be compiled, optimized, and executed directly by the processor, while an interpreter executes the program as a virtual machine, which itself is executed by the physical machine. This additional level of indirection usually causes significant drop of performance.

The static nature of code generation can also be an advantage, because the generated code can be statically checked by its compiler. The interpreter can only perform static checking of DSL code before executing it, but this kind of validation is also possible in case of code generation.

3.9 Configuration Management

Configuration management (CM) is the process of identifying, defining and recording changes to configuration items in a system. CM also reports on the status, completeness and correctness of the configuration items and is used for storing different versions of a component or software artefact for use in different products. Pressman defined CM in [24] as a set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made. It is used in conjunction with a wide variety of the following approaches such as object-oriented frameworks and component-based models for example. However, while the usage of this approach to variability management is widespread in industry, it does not represent the state of art for software product line development. Krueger in [6] discusses the problem that traditional configuration management tools have in managing “variations in space” (the differences between individual products in the domain space at any fixed point in time) as opposed to “variations in time” (tracking changes and evolution). That being said, CM should still be used in software product line development for versioning of the product line architecture.

Although configuration management consists of a wide range of activities concerning variation, it is often considered to be equal to revision/version control. Instead the latter one is a part of CM. There are plenty of tools supporting the management of different revisions of source files. Systems like CVS or Subversion are mostly used in the open source community; examples for commercial tools are IBM Rational ClearCase, Perforce, PVCS or Microsoft SourceSafe. These tools are much more focused on management of files than of more abstract concepts like features. These tools could be used for feature and variation handling, but such an approach would be far away from being efficient. Revision control systems offer the ability of managing different versions of a software product on different development streams. These streams are called branches. Changes on one branch may be propagated to other branches. While there is tool support for merging functionality from one branch into another and for dealing with conflicts that may arise, using this functionality for product line management may be at hand. But the management of products of a product line put on different branches becomes tedious with an increasing number of products. While such tools may be used to develop three or four products in parallel, the management of a product line consisting of more than 15 or 20 products is almost impossible. For being able to handle concepts like features and their relationships more tailored tools are necessary. Some are discussed in chapter 4 of this document.

As already indicated configuration management does not only contain activities regarding the management of changes to source code, it is also related to activities like actually building a product. This includes the creation of components, handling conflicts and assembling a product out of these components. It is easy to draw a connecting line to feature modelling in SPLs, which may be used as a fundament, when applying configuration management systems in SPL engineering.

3.10 Conclusion: Elements of Variation Mechanisms

In spite of the diversity of variation mechanisms that were evaluated in this chapter we can identify several reoccurring techniques that form the basis of these mechanisms:

- **Parameterization.** A large part of variation mechanisms implement some form of parameterization. In parametric variation reusable assets expose a set of explicit parameters that can be bound to different values. Parametric mechanisms differ by the object of parameterization, the type of parameters, the way the parameters can be used and the binding time. Conventional object-oriented languages provide parameterization by objects at runtime at the scope of methods, objects and programs. Frameworks and various design patterns, e.g. Strategy, State, Decorator, specialize primitive parametric mechanisms of object-oriented languages for specific problems and scenarios. The conventional mechanisms of programming languages do not support static parameterization. This hole is filled by more specific language features, such as generics or C++ templates that enable static parameterization of classes, and pre-compilation techniques, such as conditional compilation and template-based generation, that enable static parameterization at the scope of modules and other artefacts.

  The general characteristic of parametric variation is its high expressive power, because variation is expressed as computation with the variant as input. Another advantage of parametric variation is that variants are independent from reusable assets. However, the cost for this is usually a strong dependency of reusable assets on variants, which makes it difficult to extend the system with new variants. Because of weak support for extensibility, the reusable assets must be specially prepared to support required variations, which means that parametric variations are best suited for anticipated variation. Another observation is that there is a trade-off between the high expressivity of a parametric mechanism and the static validation of its possible results. For example, in the template-based generation it is even not possible to determine if a template will always produce syntactically correct code. An opposite example is generics, which supports complete type checking of parameterized classes, but the generic parameters can be used only in very strict way.

- **Dispatch.** Parametric mechanisms are often supplied with some form of dispatch on the parameter values. The most well known example is dispatch of method implementation by the type of receiver in object-oriented languages, also known as late binding. Other interesting examples of dispatch are partial template instantiation in C++ and dispatch of templates in XPand.

  Dispatch improves modularity of parametric mechanisms, because it reduces dependency of reusable assets on specific variants. The pieces of code that depend on different variants can be isolated from each other. In this way stability and extensibility of the system is increased, because new variants can be introduced without changing existing code.

- **Refinement.** Refinement is an alternative to parameterization: instead of specifying variation by explicit parameters, we open the structure of reusable assets for extension and overriding. The refinement mechanisms that were discussed in this chapter differ by their scope: class inheritance enables refinement with class scope, while AOP and FOP define refinements with program scope. Refinement with collaboration scope is possible with virtual classes.

  The major advantage of refinement lies in its support for unanticipated variation, because variation is enabled implicitly by leaving the structure of reusable assets open for extension. Another advantage is that a reusable asset is completely independent from its variations. However, this independency is usually achieved at the cost of a strong dependency in the opposite direction. The strong dependency of extensions on the code they extend is the major problem of refinement mechanisms. Support for independent extensions is also problematic, because the extensions are not aware of each other and can interact in unpredictable ways. Examples of this problem are: the aspect interaction problem and the problem with name clashes in multiple inheritance.

- **Quantification.** The major innovation of aspect-oriented languages is possibility of quantification over the points of extension. Quantification improves expressivity of refinement mechanisms, because crosscutting extensions can be expressed in a concise way. Quantification can also improve stability of extensions, because instead of explicitly enumerating extension points we refer to them by their properties.

- **Composition.** Variations are often expressed as different combinations of a predefined set of reusable assets. Technically, compositional variability can be achieved using the same parameterization and refinement mechanisms: in the first case we compose a reusable asset with a parameter value, while in the second case we compose a reusable asset with its extension. The main difference is that in case of compositional variability we treat the parameter values and extensions as reusable assets.

  Compositional variation is often an alternative to implementation of configuration using simple parametric variation, e.g. conditional compilation or conditions on global variables. The advantage of compositional variation is better modularization of variant dependent code: each component contains the code that depends on the selection of that component in the variant. Besides, we can extend the system with new components. Other properties of compositional mechanisms are very different and rely on the properties of underlying parametric and refinement mechanisms.

We can see that different variation techniques have different advantages and disadvantages, and what variation mechanism we choose depends on our requirements to variability support. In order to cover a broader spectrum of variability requirements, we must either alleviate the disadvantages of individual variation mechanisms or to develop a coherent implementation technology that integrates multiple variation mechanisms.

Programming languages are such coherent technologies that integrate multiple variation mechanisms, but they are still far from covering the complete spectrum of variability requirements. Mainstream programming languages such as C++ and Java lack variation mechanisms with a scope larger than classes. With the exception of C++ templates there is almost no dedicated support for static variation.

The need for large-scale refinement mechanisms is addressed by feature-oriented and aspect-oriented programming. Large-scale parametric mechanisms are provided by template-based languages, but differently from aspect-oriented mechanisms, templates are not integrated in the programming languages: what makes it impossible to validate them in a modular way.

A lot of variation mechanisms are available only at compile time or only at runtime. This means that if we decided to switch to change binding time, we have to reimplement our assets with different mechanisms. An interesting research direction would be to make certain static variation mechanisms available at runtime, or the other way around.

It was mentioned that dispatch improves modularity and extensibility of parametric variation mechanisms. We can observe, however, that the mainstream parametric mechanisms have weak or no support for dispatch. More sophisticated forms of dispatch such as multi-dispatch or predicate dispatch [63] are available only in research languages and are not used in practice.

4. Evaluation of Existing Tools

In this chapter we evaluate four different tools with respect to the criteria introduced in Section 2.2. We consider here only the tools that cover a range of activities of the development of SPLs. The following three tools fit this requirement, provide typical development environments for SPLs, and are among the most feature-rich tools:

1. pure::variants from pure-systems™, a German software development company;
2. Gears from BigLever Software, Inc.™, a US software provider;
3. fmp2rsm, a SPL tool developed by Prof. Czarnecki and his group.

Furthermore, we evaluate a representative tool suite for general-purpose modelling and DSL engineering:

4. openArchitectureWare, an OSS effort.

This set of tools seems particularly useful for evaluation in the context of the AMPLE project, in which new approaches to tooling for SPLs are to be developed but for which integration with mainstream approaches is also of importance.

In the section, the main characteristics of the three tools listed above are presented. The tool evaluations are structured according to the top-level categorization of the set of criteria defined in section 2.2.

4.1 pure::variants

Pure::variants [3] is a commercial tool with a free entry-level edition that provides the currently most complete support for SPL development over the software lifecycle.

Concepts. pure::variants provides explicit representations for sets of variants of components (as part of its "family models") and features. Family models are built from concrete assets (typically files), declare relationships between them (e.g. arity relationships), and make it possible to associate existing implementations to components. Feature models may be hierarchical. Relationships between components and features can be defined using logic-based constraints or in terms of a table-based representation. Variant description models are defined to represent a set of models for configuration and transformation. Each different variant product has an associated variation description model. Transformations occur by replacing fragments of file that represent assets. A validation is done by automatically checking the selection of desired features for the variant product.

pure::variants is agnostic to the concrete implementation *technology* used for assets (a property that is shared by, e.g., the Gears tool that is described below). Nevertheless, it comes equipped with several extensions that integrate with different industrially relevant implementation frameworks, e.g., SAP's ERP systems as well as the DOORS requirement support system.

Functionality. pure::variants directly supports three different phases of the *software lifecycle*: (i) feature definition and asset-feature mapping, (ii) configuration of an SPL and (iii) assembly of a product by selecting features and corresponding components from an SPL. It provides limited support for requirement and configuration management. However, pure::variants does not provide direct support for other tasks, such as requirements and architecture analysis, asset development and the execution of products. Some of these tasks are supported by interfaces to third-party tools (e.g., requirements analysis with DOORS, simulation with Simulink). SPL models based on other information (e.g., stemming from an architecture analysis), variant models as well as feature models may be transformed using a specially-tailored XLST framework.

pure::variants is agnostic to concrete implementation languages and underlying implementation technologies in the sense that these are either provided through suitable import and export plugins or can be supported by specially-tailored plugins.

pure::variants in itself is not specialized to any *application domain*. However, specific integration modules exist for some industrially relevant domains, such as SAP's ERP software.

The tool provides user interfaces with graphical representation of the models and ways to define and modify these models. Specific interfaces are also used for the definition of all provided models, definition of features characteristics, model dependencies, and transformation management.

Usage. pure::variants is available in different commercial versions and a free community edition, the latter lacking some extensibility features and being restricted to small feature models.

Extension of this tool is explicitly supported through the Eclipse [8] extension framework. Furthermore, XSLT transformations can be used to extend the provided family and feature models and new features can also be implemented through SOAP and COM/OLE interfaces.

Pure::variants supports *interoperability* with other tools by several predefined or user-defined export mechanisms for feature and family models. Source code of assets can be used to import code artefacts natively for a small set of languages and can be handled using plugins otherwise.

Usability of the development environment is enhanced by supporting (the manipulation of) textual as well as graphical representations of key abstractions.

4.2 Gears

Gears [2] is a commercial tool based on a graphical user interface or command-line user interface. It allows the definition of variation points and features, and the definition of concrete products through selection of variants and assembly of the corresponding products.

Concepts. An *asset* in Gears is a group of various forms of files related to source code, test- and maintenance-related data. Gears provides a notion of *variation points* that represent parts of software assets that can be configured according to *features*.

Gears also provides a special purpose language for defining how different features may modify an asset: this language basically makes it possible to test for feature values and select concrete assets accordingly. First, an abstract file of an asset can be created by selecting a concrete file. Second, the text of the concrete file can be customized by text substitution using text patterns defined in a pattern file. Features are defined using a textual special purpose language that makes it possible to hierarchically define sets of simultaneously applicable or mutually exclusive options.

Composition within products is supported by three different mechanisms: a module abstraction for grouping software assets, a mixin-like abstraction to support crosscutting definitions over modules and a composition matrix. Nesting of product lines is also supported.

Functionality. Like pure::variants, Gears supports three different phases of the *software lifecycle*: (i) feature definition and asset-feature mapping, (ii) configuration of an SPL and (iii) assembly of a SPL. However, it does not provide direct support for other tasks, such as requirement and architecture analysis, asset development and the execution of SPLs.

Gears is agnostic to *other methodologies and technologies* used to support SPL that are not directly related to feature management. Software artefacts generated by, e.g., tools for software design, can be used as assets within Gears and assets may be implemented according to different, e.g., industrial, component standards. In contrast to pure::variants, Gears does not provide specific interfaces for existing infrastructures but relies on direct manipulation of compatible software assets.

Finally, Gears is not targeted at particular application domains.

Usage. Gears is licensed on a per-user basis and only different evaluation versions are freely available on request. The underlying development environment does not provide an explicit support for extension or configuration. All main concepts of the tool - such as variation points, features and composition abstractions – are easily manipulable in a textual and graphical way.

4.3 fmp2rsm

fmp2rsm is an implementation of Feature-Based Model Templates for IBM Rational Software Modeler (RSM) [67] and IBM Rational Software Architect (RSA) [68], which are UML modelling tools. fmp2rsm integrates the Feature Modeling Plug-in (fmp) [69][70] with RSM and enables product line modelling in UML and automatic product derivation. The fmp is an Eclipse plug-in for editing and configuring feature models.

Concepts. As presented in [71], fmp2rsm provides explicit representations for sets of family models. A family model is represented by a feature model and a model template. The feature model defines features with constraints on the possible configurations. The model template contains the union of the elements in all valid template instances. The set of the template instances corresponds to the scope of the model family. The elements of a model template may be annotated using presence conditions (PCs) and meta-expressions (MEs). These annotations are defined in terms of features and feature attributes from the feature model, and can be evaluated with respect to a feature configuration. A PC attached to a model element indicates whether the element should be included in or removed from a template instance. MEs are used to compute attributes of model elements, such as the name of an element or the return type of an operation. An instance of a model family can be specified by creating a feature configuration based on the feature model. Based on the feature configuration, the model template is instantiated automatically. The instantiation process is a model-to-model transformation with both the input and output expressed in the target notation. It involves evaluating the PCs and MEs with respect to the feature configuration, removing model elements whose PCs evaluate to false and, possibly, additional processing such as simplification.

Functionality. fmp2rsm supports three different phases of the *software lifecycle*: (i) feature definition and RSM/RSA-feature mapping, (ii) configuration of an SPL and (iii) generation of template-instance (models) of a product line member by selecting features. fmp2rsm follows a model driven development (MDD) strategy relying on model-to-model transformations and does not provide model-to-text facilities. For this, the use of specialized tools such as MofScript [72] or Acceleo [73] is suggested. fmp2rsm provides limited support for requirement and configuration management. Based on fmp, fmp2rsm provides an automated verification procedure for ensuring that no ill-structured template instances are generated from a correct configuration. It also provides functionality for creating staged configuration of product line members, useful, for example, for sharing the responsibility of configuring products between different users/roles, and extends the traditional semantics of feature models allowing the creation of clonable features, or feature nodes with typed attributes. As pure::variants, fmp2rsm does not provide direct support for other tasks, such as requirements and architecture analysis, asset development and the execution of products. Some of these tasks are supported by RSM/RSA. fmp2rsm is not specialized to any *application domain*.

Usage. fmp2rsm is freely available through its homepage (see [74]). However, it works on the platform of IBM Rational Software Modeler or IBM Rational Software Architect, which are commercial tools. RSM/RSA 30-day trial versions can be accessed from [75]. Extension of fmp2rsm is supported through the Eclipse-plugin mechanism.

4.4 Modelling tools for SPL: the example openArchitectureWare

openArchitectureWare (oAW) is a free tool for model-driven development, that provides complete support for the design of Domain Specific Languages (DSLs) [26]. It does not explicitly support the design and implementation of SPLs by means of dedicated representations.

Concepts. oAW supports the implicit management of variability using model transformation techniques. Domain meta-models define domain concepts and possible variations are represented using a tool for defining DSLs (using a formalism based on

BNF-style grammars) together with additional semantic constraints. The description of a variant product is made by writing DSL code. The validity of variant products can be checked using language parsing and model validation techniques. A model transformation chain is used for the generation of source code for products; assets are also defined and manipulated using meta-models, transformation rules, and generation templates. Model transformation supports the use of Aspect-Oriented Software Development techniques. Transformations can take as input many models (instances of one or more meta-models that themselves can be instances of different meta-meta-models) and can define the way models are woven in order to produce one or more models. From a code generation perspective, some templates can be used to customize other existing templates without having to modify them, thus making it possible to manage separation of concerns and unanticipated variants. These facilities can be harnessed as part of a configurable workflow: the workflow itself can be split into parameterized components, enabling variability of the product design and generation process itself.

Functionality. oAW provides a set of specific languages to support the entire design process of a DSL [26]: a first phase consists in domain meta-modelling, a second phase consists in defining the grammar of the DSL language. To this end, a dedicated language with a simple BNF-like syntax named *Xtext,* is provided. *Xtext* descriptions are used by oAW to produce syntactic analysers and a corresponding meta-model. A third step consists in the definition of transformation rules between meta-models (and particularly between the domain and the syntax meta-model). The *Xtend* language is provided to define transformation rules: it allows weaving different models, instances of different meta-models, and supports aspect oriented model transformations. As part of a fourth step, constraints can be defined to validate models. A language, named *Check* is used to describe model validation rules and error messages. Finally, the fifth and last phase consists in defining templates for source code generation. A template language named *Xpand*, is provided to this end (cf. Section 3.8). All of this process (parsing, checking, transforming and code generating) is supported by the oAW workflow engine and configured with specific workflow configuration files, thus providing large flexibility.

This process does not necessary include the use of all languages and other tools. Complementary facilities of the ECLIPSE modelling world can be used instead or in addition. oAW is agnostic to concrete implementation languages and underlying implementation technologies in the sense that any type of source code can be generated by the transformation chain. It is also not specialized to any application domain.

Usage. oAW is a freely available ECLIPSE plug-in [26], deeply integrated with the well known plug-in EMF [9]. oAW supports, with import and exports facilities, the use of many ECLIPSE standard formats (and particularly *ECore*) during the DSL design process, making its usage rather well interoperable with the rest of the Eclipse modelling world (GMF, UML2, OCL, etc.).

Furthermore, the oAW workflow is highly configurable: many kinds of existing technologies can be used instead of oAW languages for defining meta-models (even some non-Eclipse ones), transformations rules (e.g. ATL), constraints (e.g. OCL) and templates (e.g. JET). *Extension* of this tool is explicitly supported through the Eclipse extension framework.

4.5 Evaluation

The evaluation of the four tools given in the previous section gives rise to different major issues that are relevant for the endeavour of integrating new support for SPL with existing tooling as well as underlying software development methodologies and implementation technologies:

- All evaluated tools only cover directly the phases of variant and feature definition as well as product composition by feature selection. There is, however, almost no explicit support for the testing, execution and maintenance of SPLs.
- Existing tools offer almost no means for some key features relevant to SPL development. Traceability over the software lifecycle, for instance, is covered only rudimentarily by the existing tools. The tools do not allow, for instance, tracing information during execution within parts of a module that constitutes one asset on the level of the variant and feature models. However, this information can very well be relevant to improve the feature model of the underlying product, especially during evolution of the product. In addition, the evaluated tools lack of orthogonal mechanisms for managing the whole SPL development activities such as requirements management or configuration management and the respective versions control.
- Even when some tools use different kinds of models as core assets for composing entire SPL members, there is limited integration of the model driven development principles, related with the development of domain-specific model languages or the definition of reusable model transformation rules, that would allow composing or refining models until the final SPL members are obtained.
- Integration with existing tool development environments, such as Eclipse, is advantageous compared to a standalone tool, in particular, for achieving extensible and interoperable tool support for SPLs. Tool development environments support these criteria in two ways; first, directly through extension mechanisms (such as the Eclipse extension mechanism or plugin mechanisms) that can be leveraged for SPL development; second, indirectly through better interoperability with already existing tools that are relevant to SPL development.
- DSLs and feature models are complementary approaches for the engineering of large scale SPLs. Feature modelling is well suited for configuring the main features of a line of products. When considering in detail some complex and/or highly configurable parts of a product, the use of a DSL can be very useful. For example, if some parts of a product require specific behaviour it is generally more intuitive, easy and powerful to express this behaviour with a DSL. The DSL approach is generally well-suited for engineering SPLs "in the small". No tool currently exploits the respective qualities of both approaches.

Tabular comparison

Table 1. Tabular comparison of the tools at the conceptual level

	pure::variants	**Gears**		**fmp2rsm**	**oAW**
Variation mechanism		File realization	Text Substitution		
Variation management principle	a piece of text is replaced by a substitution text	an abstract file (directory) is realized by a concrete file (directory)	a piece of text is replaced by a substitution text	A model-template instance is created from a model-template and a feature configuration	source code is generated from DSL code via a model transformation chain
Reusable Assets	files (code, documentation)	any kind of asset implemented using files (directories)	any kind of asset implemented using text files	models	meta-models, transformations rules, generation templates
Variation Type	parts of files	an abstract file (directory)	piece of text in an asset that matches one of the patterns defined in a pattern file	Multiple stereotypes in models, cardinality, and model elements	defined by DSL syntactic and semantic rules, instantiation of meta-models and templates parameters
Variants	fragment of text (code, documentation)	concrete files (directory)	the different substitution texts defined for the matched text in a pattern file	Concrete stereotypes and cardinality, and concrete model elements	DSL code, models, source code
Expressive Power					
Transformation Type	replacement	file (directory) realization	text substitution using pattern matching	Composition/ Merging	model transformation, (AO) code generation
Granularity	fragment of files	files	characters	Fragments of models	fragment of meta-models, fragment of

					templates
Binding Model					
Binding Time	assembly	assembly		assembly	assembly
Availability Time	before binding	before binding		before binding	before binding
Scope of Binding	file	file		models	Meta-model, transformation rules, templates
Validation					
Availability of variation type	explicitly specified	explicitly defined as a file (directory)	explicitly defined as files	explicitly specified	explicitly specified as a DSL BNF and meta-models
Validation of binding	possible	possible, there are constraints in the feature selection that assure that a selection can result in a valid product	possible, there are constraints in the feature selection that assure that a selection can result in a valid product	possible	Possible via DSL syntactic analyzers and model constraints checking
Modularity					
Structural Separation	not clear	There is the notion of module, which define the scope of feature declarations		not clear. Features can be grouped and after referred by other features. Model templates are managed separately.	No explicit description of features. Meta-models and templates can express different concerns
Explicit dependencies	it is possible to declare restrictions and constraints that refer to features	modules can depend on mixin modules. The dependencies are implicit in the context of a module, but they become explicit in the definition of a concrete product		It is possible to define dependencies between features, between model templates, and between feature and models templates	dependencies between meta-models are explicit in transformation rules. Explicit dependencies between templates and meta-models.
Segregated	variation description	using the concept of a matrix, different products		It is possible using the	not relevant

interfaces	models	can be instantiated for different clients	concept of staged feature configurations and specializations	
Asset/Variation Dependency	unaware	unaware	unaware	unaware
Decomposition of Assets	assets in different files	assets in different files	Assets in different files that represent different models	no
Decomposition of Variations	features model hierarchy	features model hierarchy	features and model-templates hierarchy	not explicit

Table 2. Tabular comparison of the tools at the functional level

Process coverage				
definition of SPL	feature model	feature model	feature model	DSL
analysis/validation of SPL (domain space)	yes, checks compatibility between feature model and family model	yes, the tool includes a statistics report, which, for instance, computes the number of potential products based on the number of feature declarations and definitions	yes(computing number of configurations represented by a feature model, propagating configuration choices, and so on [65])	no
analysis/validation of products (application space)	yes, check compatibility between feature model and variation description model	yes, checks that the feature selections of a product instance are correct with regard to the corresponding feature model and check the correct selection of product instances for each module and mixin module	yes (verifying feature-based model templates against well-formedness OCL constraints [66])	yes (checking model validity against OCL constraints)
product assembly	yes	yes	no	yes
product testing	no	no	no	no
product execution	no	no	no	no

product maintenance	no	no	no	no
Support for specific application domains	no	no	no	no
Expressiveness of feature model editors				
Hierarchy of features	yes	yes	yes, including the possibility of creating cloned features	not relevant
Feature selection	one-of, more-of, optional, mandatory	one-of, more-of, optional, mandatory	one-of, more-of, optional, mandatory	not relevant (not explicit in DSL code)
Feature with values	yes	yes	yes, including typed attributes.	not relevant
Assertions on feature values	yes	yes	yes	not relevant
Feature Representation	graphical	textual	graphical and textual.	not relevant
Multiple feature models	yes	yes	yes. Including constraints between them that allow staged specialization.	not relevant (but multiple DSL can be used)
Feature-model dependencies	yes	yes	yes. Including OCL constraints.	not relevant
Product creation				
Support for managing feature-model instances	yes	yes	yes	no

Support for product instantiation	yes	yes	no, just model instantiation	yes (generation of source code from DSL code)
Execution environment	no	no	no	no
Editors to manage dependencies between feature models	yes	yes, it is textually possible to manage the dependencies on the feature models of mixin modules	yes	no
Code generator	yes	no	no	yes
Implementation targets	technology agnostic, specific support for C/C++ and Java	technology agnostic	no	technology agnostic

5. Existing Implementation Practices Applied at Industrial Partners

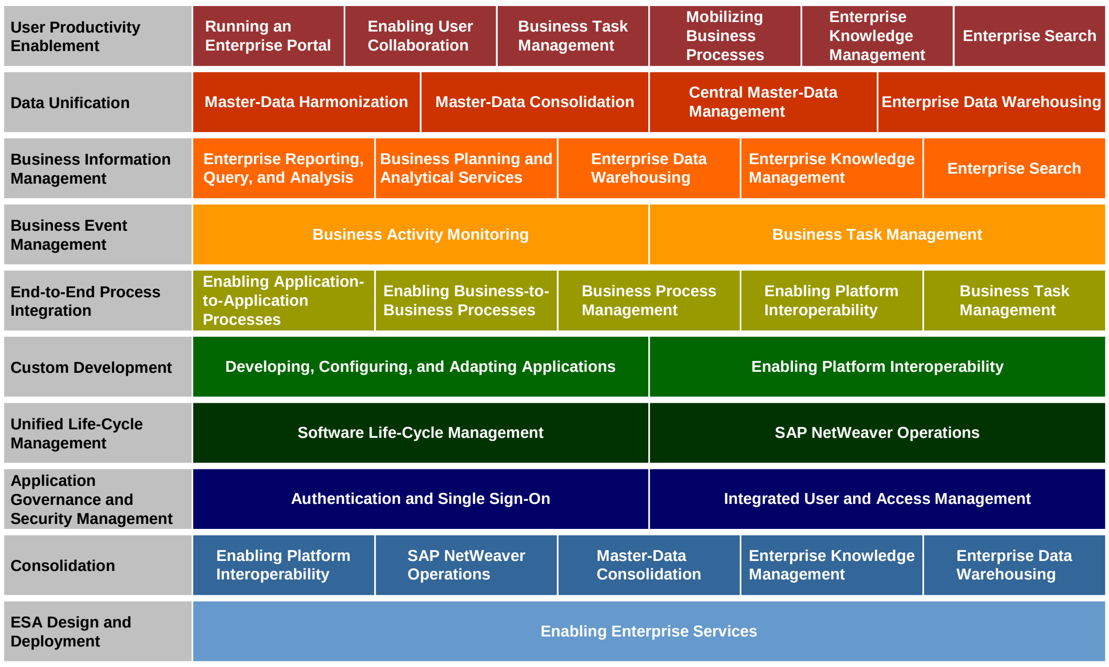
This chapter gives an overview about the various implementation techniques used at industrial partner sites.

5.1 SAP

At SAP various techniques are already applied to implement variability in software products. This section gives an overview of technologies currently used in SAP.

5.1.1 SAP NetWeaver Platform

SAP NetWeaver is the underlying technology platform of all SAP applications. The following figure gives an overview of the SAP NetWeaver solution map:



User Productivity Enablement	Running an Enterprise Portal	Enabling User Collaboration	Business Task Management	Mobilizing Business Processes	Enterprise Knowledge Management	Enterprise Search
Data Unification	Master-Data Harmonization	Master-Data Consolidation	Central Master-Data Management	Enterprise Data Warehousing		
Business Information Management	Enterprise Reporting, Query, and Analysis	Business Planning and Analytical Services	Enterprise Data Warehousing	Enterprise Knowledge Management	Enterprise Search	
Business Event Management	Business Activity Monitoring	Business Task Management				
End-to-End Process Integration	Enabling Application-to-Application Processes	Enabling Business-to-Business Processes	Business Process Management	Enabling Platform Interoperability	Business Task Management	
Custom Development	Developing, Configuring, and Adapting Applications	Enabling Platform Interoperability				
Unified Life-Cycle Management	Software Life-Cycle Management	SAP NetWeaver Operations				
Application Governance and Security Management	Authentication and Single Sign-On	Integrated User and Access Management				
Consolidation	Enabling Platform Interoperability	SAP NetWeaver Operations	Master-Data Consolidation	Enterprise Knowledge Management	Enterprise Data Warehousing	
ESA Design and Deployment	Enabling Enterprise Services					

Figure 9. SAP NetWeaver Solution Map

Obviously, a whole range of different technologies, frameworks and libraries are integrated in this platform. At the bottom there are two different language stacks

At the bottom there are two different language stacks that are coexisting. SAP software may be implemented on top of both of these stacks:

1. ABAP (Advanced Business Application Programming) [22] was developed and extended by SAP as the primary language for writing business applications. The ABAP stack will remain the strategic platform for business logic running on backend servers, also in the advent of the upcoming Enterprise SOA based, component-oriented Business Process Platform.
   Although legacy plain ABAP programs are still supported, new applications are almost exclusively written in ABAP Objects, the downward compatible Object-Oriented extension of ABAP. ABAP Objects has all major features of modern OO languages, except for method overloading. However, the lack of this feature can be circumvented by a number of alternative best practices.
   ABAP furthermore has a number of built-in language features like direct

access to database tables, which predestine it for implementing data-intensive business software.
In addition ABAP features aspect-oriented characteristics, which are explained in section 0 in greater detail.

2. Java on the other hand is primarily used for most web-based UI/portal technologies (on a JEE basis). Java also plays an increasing role for implementing service consumption and service composition on top of the Business Process Platform. This strategic decision for a wide-spread industry standard language enables SAP partners and ISVs to recruit developers from a far larger community than in a pure ABAP-based environment.

While ABAP development (programming, debugging, deployment etc.) is supported by a set of dedicated development transactions, which are executed on the host server, all Java development at SAP is performed using the client-side IDE of NetWeaver Developer Studio (NWDS). NWDS is an extension of the popular Eclipse tool platform [20], [21], which consists of a large number of SAP-specific plug-ins. Plug-ins are the primary extension mechanism for addition new features to the Eclipse platform.

Connectivity between distributed components is established via three technologies: Remote Function Calls (RFC), the J2EE Connector Architecture (JCA) and the Java Message Service (JMS).

RFC is the standard SAP interface to communicate with SAP backend systems and non-SAP systems, where functions can called to the executed on remote systems.

The JCA is a specification that defines the standard architecture for connecting the Enterprise Edition of the Java Platform (J2EE) to heterogeneous Enterprise Information Systems (EIS), which may include ERP and database systems. The mechanisms that the connector architecture defines are scalable and secure and enable integration of the EIS with application servers and enterprise applications. An EIS may supply so-called resource adapters, which are used to connect to the EIS. The connectors can be plugged into an application server and provide connectivity between the EIS, the application server and the enterprise application. When an application server supports this connector architecture, it provides seamless connectivity to multiple EISs.

JMS is a set of interfaces and associated semantics that define how a JMS client accesses the facilities of an enterprise messaging product. A JMS application is made up of a set of application defined messages and a set of clients that exchange them. Products that implement JMS do this by supplying a provider that implements the JMS interfaces. Messages are asynchronous requests, reports or events that are consumed by enterprise applications.

Enterprise systems need to persist large amounts of data. To achieve this task the NetWeaver Platform enables the use of several technologies for establishing persistence.

OpenSQL is the SAP database abstraction layer implemented in ABAP that translates abstract SQL statements to native database SQL statements. OpenSQL covers the Data Manipulation Language (DML) part of the SQL standard and extends the SQL standard by offering options to simplify and accelerate database access.

Java Database Connectivity (JDBC) technology provides cross-DBMS connectivity to a wide range of SQL databases and access to other tabular data sources, such as spreadsheets or flat files. It is supported by the NetWeaver Platform for J2EE development. With a JDBC technology-enabled driver it is possible to connect all corporate data independent from homogeneous or heterogeneous environments.

The Java Data Objects (JDO) API is a standard interface-based Java model abstraction of persistence. It is supported by the NetWeaver Platform as an alternative to JDBC. JDO technology has the advantage to be able to store Java domain model instances directly in a database. The process of mapping data to relational databases is transparent for a developer.

For implementing business logic both of the language stacks mentioned above can be used. ABAP is tailored to implementing business applications. It allows quick development of business applications providing powerful macros to create the actual business logic based on SAP backend systems. There is a huge amount of existing business objects on which a developer may rely on.

The Composite Application Framework (CAF) offers a methodology and toolset to create and manage composite applications. It leverages information and data from existing applications to solutions by composing existing or new services, user interface components, and business processes. CAF is based on the Enterprise Services Architecture (ESA) and comprises an abstraction layer for services and processes as well as design tools and integrates many key capabilities of the NetWeaver Platform.

In the area of user interaction Web Dynpro is the recommended NetWeaver programming model. The Web Dynpro model is based on the Model-View-Controller (MVC) programming model and allows a clear separation of business logic and display logic. The development environment provides powerful graphical tools to layout the user interface.

However, there are other technologies that are supported alongside. Business Server Pages (BSP) are a page-based Web programming model with server-side scripting in ABAP. BSPs gives complete freedom when designing UIs since any HTML and/or JavaScript can be sent to the client. With the HTMLB BSP extension SAP also offers a library of predefined UI elements that simplify the creation of BSP pages. The pendant are Java Server Pages which enable page-based web programming with server-side scripting in Java. In addition there are frameworks on a higher abstraction level like for instance Guided Procedures (GP). GP provides tools and a framework for modelling and executing user-oriented workflows. It supports business specialists in implementing processes and guides casual users through the execution of these processes.

5.1.2 SAP Exchange Infrastructure (XI)

An important cornerstone of integration technology built into the NW platform is the SAP Exchange Infrastructure (XI) [19], an Enterprise Application Integration (EAI) solution supporting also message-oriented / event-driven “hub and spoke” [18] style business-to-business (B2B) interactions, which loosely couple heterogeneous applications. This corresponds to the event-based component interaction introduced in section 3.3.6 as a highly modular architecture style with independent structures whose variability can be bound very late in software lifecycle.

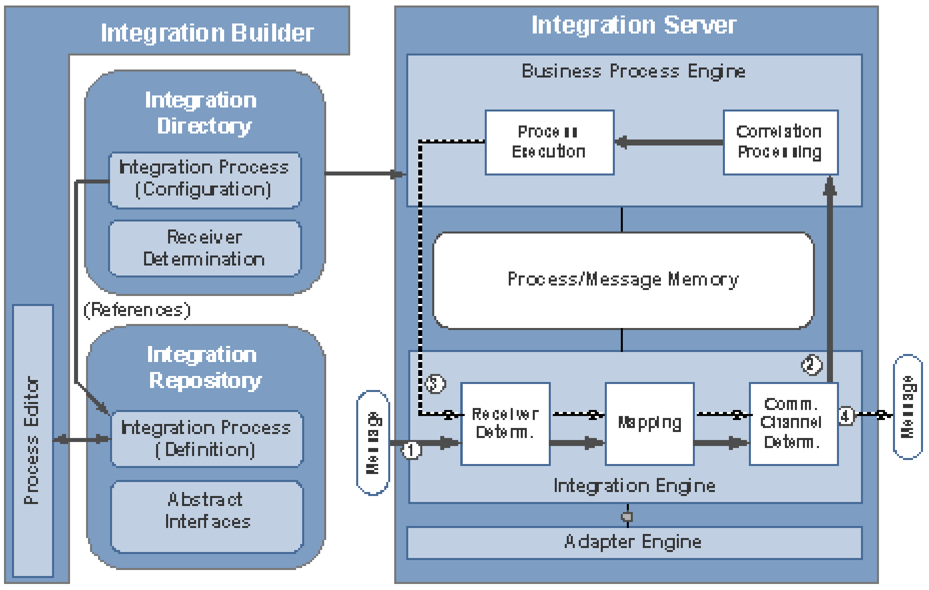


Figure 10. SAP XI Architecture

SAP XI – now being renamed to SAP Process Integration (PI) – runs on the SAP Web Application Server (SAP Web AS) component. SAP XI reduces integration and maintenance costs of IT systems by providing a common, central repository for interfaces. It supports cross-component business process management (BPM) within the same solution. And, it offers an integrated tool set to help organizations build their own integration scenarios by defining the appropriate messaging interfaces, mappings, and routing rules.

5.1.3 Configuration and extension mechanisms in the ABAP stack

Implementation Guide for R/3 Customizing (IMG)

In older existing R/3 applications, the Implementation Guide (IMG) allows the customization of selected business processes. It lists all necessary and optional actions required for implementing a SAP system. Its primary purpose is to allow a user to control and document the whole implementation process. It is also used for making customer-specific settings in an SAP system.

The base is the *Reference IMG*, which contains all IMG activities and relevant documentation. It covers all topics of an SAP system, for example, enterprise structure, financial accounting, controlling, materials management or production planning. The IMG guides the attention of a user on which configuration options exist and which need to be used for certain application fields.

The Implementation Guide is structured hierarchically, its structure follows the hierarchy of the application components (i.e. *Recruitment* is located under *Personnel Management*). The central parts are so-called IMG activities that enable ways to customization and perform important system configuration tasks. The implementation team accesses the documentation part of the IMG to perform settings in an actual project via the IMG.

The Enhancement Framework (EF) was designed to overcome older techniques to enable users to modify the standard behaviour of an SAP system. The EF tries to combine the easy maintainability of standard software with the high flexibility of proprietary solutions while avoiding the drawbacks of both (lack of flexibility in standard and upgrade issues in customized software). The EF is not a single mechanism; instead it is the integration of various techniques for modifying development objects.

In previous releases of the SAP system, there were predefined points at which users were able to insert so-called *modifications*. This procedure was supported by a *Modification Assistant*, which was able to observe user add-ons (up to a certain degree). There are several shortcomings that are connected to these modifications:

1. There is no support for system upgrades; an upgrade may render modifications unusable.
2. It is quite difficult to trace developments made in different parallel system back to one central system.
3. There is a high cost for testing systems with a lot of user modifications.

The Enhancement Framework has been introduced in SAP NetWeaver 2004s, Release 7.0, and aims to unify possible types of modifications/enhancements as well as organize enhancements as effectively as possible. At the core of the framework there is a simple structure consisting of a hook and an element that can be attached to this hook. The EF is supported by a dedicated tool, the *Enhancement Builder*.

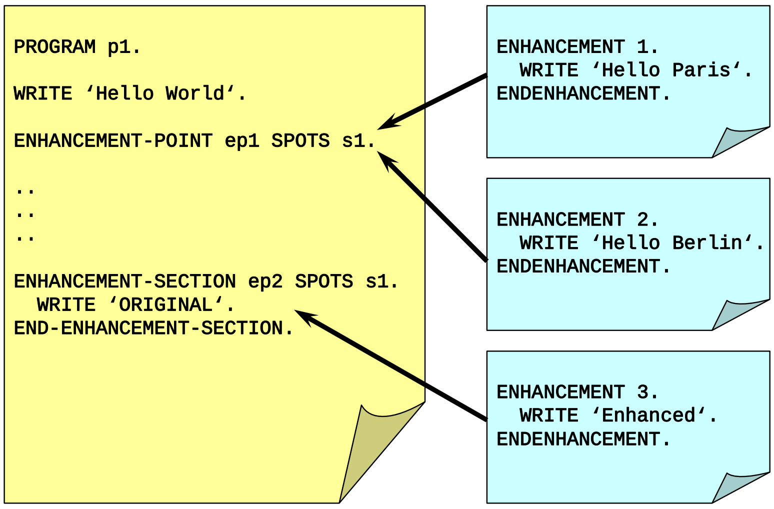
The main function of the EF is the modification, replacement and enhancement of repository objects and foreign objects – objects that form the technical basis of an SAP system. Control over these objects is provided via the *Switch Framework*, which is explained in more detail in another section below.

There are three elementary concepts in the Enhancement Framework for modifying/enhancing development objects:

1. *Enhancement Options* (EO) defined as positions in repository objects, where enhancements can be made. Two types of EO exist: explicit options and implicit. An explicit option is created when points or sections in source code of ABAP programs are explicitly flagged as extensible. These options are *managed* by Enhancement Spots and *filled* by Enhancement Implementations. In contrast to explicit options, implicit options are special points in ABAP programs, which can be enhanced. Examples for such special points are the end of a program or the beginning of a method. Implicit options can be enhanced by source code, additional parameters for the interface of function modules or global classes.
2. *Enhancement Spots* (ES) are used to manage explicit Enhancement Options and carry information about the actual position of possible options. A spot can manage more than one option. ES are directly supported by the Enhancement Builder which is integrated in the ABAP Workbench.
3. *Enhancement Implementations* (EI) are the counterpart for ES. At runtime one or more EI can be assigned to a single ES. There are several types EI: Source Code Enhancements, Function Module Enhancements and Global Class Enhancements. Source Code Enhancements represent the direct insertion of source code at predefined locations in ABAP programs. These locations can

be defined by implicit and explicit Enhancement Options. Function Module Enhancements represent the enhancement of parameter interfaces. For example a new optional parameter can be added to the interface of a function module. In addition via Global Class Enhancements new attributes can be added to repository objects or special pre-/post-methods can be realized, which are called directly before/after ABAP methods.

Obviously, these concepts can be roughly compared to concepts of **Aspect-Oriented Programming**: Enhancement Options resemble *Pointcuts*, Enhancement Spots map to *Join Points*, and Enhancement Implementations to *Advices*. An example is shown in Figure 11. In this example a simple program is extended by several enhancement implementations. Enhancement 1 is inserted at the position marked with ENHANCEMENT-POINT and can optionally be overwritten by Enhancement 2. In contrast Enhancement 3 is not inserted at some particular point, but replaces a section marked with ENHANCEMENT-SECTION.



```
PROGRAM p1.

WRITE ‘Hello World‘.

ENHANCEMENT-POINT ep1 SPOTS s1.

..
..
..

ENHANCEMENT-SECTION ep2 SPOTS s1.
  WRITE ‘ORIGINAL‘.
END-ENHANCEMENT-SECTION.
```

```
ENHANCEMENT 1.
  WRITE ‘Hello Paris‘.
ENDENHANCEMENT.
```

```
ENHANCEMENT 2.
  WRITE ‘Hello Berlin‘.
ENDENHANCEMENT.
```

```
ENHANCEMENT 3.
  WRITE ‘Enhanced‘.
ENDENHANCEMENT.
```

Figure 11. Example for an ABAP code enhancement

Business Add-Ins (BAdI)

SAP Business Add-Ins (BAdIs) are one of the most important technologies to adapt SAP software to specific requirements. BAdIs were introduced in Release 4.6 in order to replace function exits. As of Release 7.0 they are part of the enhancement framework. They are realized as explicit *Enhancement Options* (so-called classic BAdIs). New BAdIs are directly supported by the ABAP runtime environment through dedicated ABAP statements.

BAdIs are the basis for *object plugins* that modularize function enhancements in ABAP programs. There is an explicit distinction between the definition and the actual implementation of BAdIs. The definition of a BAdI contains an interface, a set of selection filters and settings for runtime behaviour. The implementation contains a class implementing the interface and a condition imposed by the filters. An example of a BAdI structure can be seen in Figure 12. In this example a BAdI A may be used for tax calculation. The definition of this procedure is made in the Enhancement Spot for the BAdI, while the actual calculation logic can be found in Implementation 1 for BAdI A. There may be more than one (two in this example) implementations for the definition, which can be used mutually exclusive.

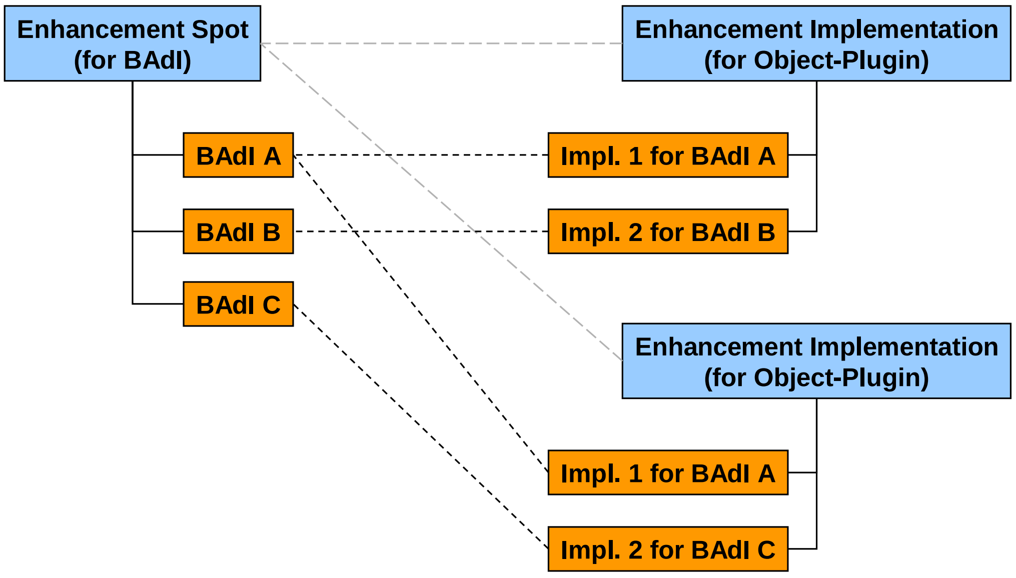

Enhancement Spot (for BAdI)
BAdI A
BAdI B
BAdI C
Enhancement Implementation (for Object-Plugin)
Impl. 1 for BAdI A
Impl. 2 for BAdI B
Enhancement Implementation (for Object-Plugin)
Impl. 1 for BAdI A
Impl. 2 for BAdI C

Figure 12. Structure for Business Add-ins

Clearly this mechanism is not related to Aspect-Oriented Programming, rather it resembles patterns from Object-Oriented Programming, where certain behaviours are defined via interfaces and implemented by a combination of abstract and concrete classes.

Switch Framework

The Switch Framework (SF) allows the control of the visibility of repository objects or their components by means of switches. The SF is integrated in the ABAP workbench and works closely together with the Enhancement Framework. While the Enhancement Framework enables and supports the actual implementation of solutions, the SF controls which of those implementations are finally utilized.

The main purpose of the SF is the simplification of an ABAP-based system landscape by adopting one or more industry solutions in a standard system. Solutions are delivered with all objects/functions deactivated, only appropriate objects are activated on demand. For this reason, the Switch Framework is a modification-free enhancement concept.

The basis of the SF are three main components:

1. A *Business Function Set* (BFS) is a set of Business Functions and corresponds to an industry solution. Inside a SAP system several BFS may exist, but only one may be active at a time.
2. A *Business Function* (BF) is a self-contained function from a business perspective and consists of a set of switches. A BF is some kind of building block for BFS, activating a BF means activating all its switches.
3. A *Switch* is the elementary component in this context; it is a repository object that is able to control the visibility of other repository objects. This applies to single objects like screens or collection of objects like a package. A switch can be assigned to several Business Functions and vice versa several switches can be assigned to one Business Function. A conflict arises if two switches turn on

objects that may not be used together. This situation is resolved by special conflict switches and appropriate conflict-resolving enhancement implementations.

The relations between those elements are shown in Figure 13. In this example the BFS contains five BF, where the first and the fourth are activated. Both trigger appropriate switches, which leads to the application of a certain package and some arbitrary component. The whole structure is similar to feature trees, although there is only a limited depth of two or three levels, depending on how fine- or coarse-grained a feature is defined.

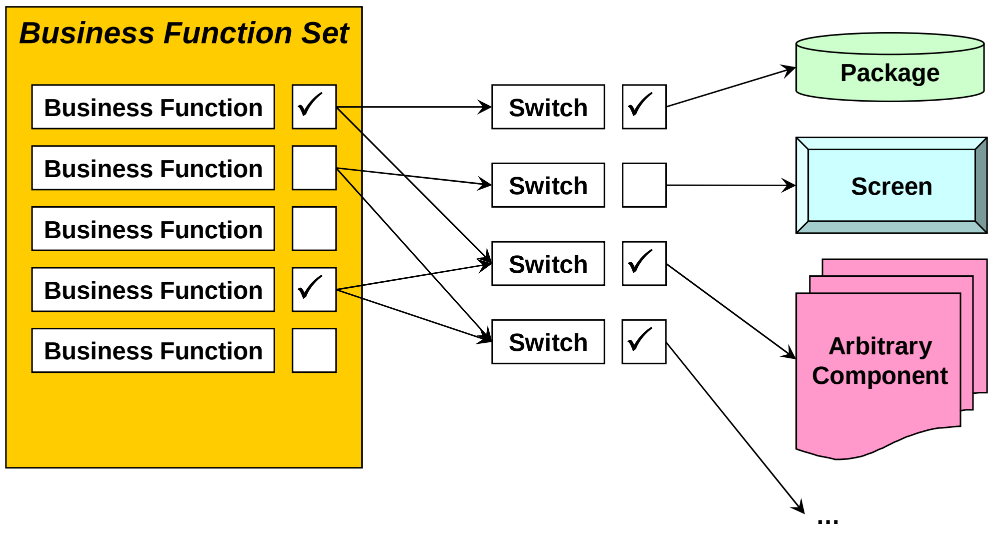


Figure 13. Structure of a Business Function Set

The whole configuration of a Business Function Set is stored in so-called *Switch Business Configurations* (SBC). These are data containers with database table entries for industry solutions. Such solutions may contain several SBC, which can be activated in subsequent systems of the solution.

There is a differentiation between *industry* BFS (*industry extensions*) and *generic* BFS (*enterprise extensions*). The Switch Framework can activate exactly one industry BFS, but several generic BFS. Examples for industry extensions are media, telecommunications or oil & gas, as examples for enterprise extensions financial services, global trade or human resources may be mentioned.

Table 3. Comparison of mechanisms in the SAP ABAP stack

	SAP Enhancement Framework (EF)	**SAP Business AddIns (BA)**	**SAP Switch Framework (SF)**
Concept			
Reusable Assets	ABAP Code	ABAP Code	Business Function
Variation Type	ABAP Code Fragments	Business AddIn	Set of Switches
Transformation Type	Refinement	Refinement	Composition

Granularity	Fine, on code level	Fine to Coarse, similar to component level	Coarse, on business logic level
Modularity			
Level of Separation	Structural	Structural	Structural
Dependency on Variation	Unaware	Unaware	Unaware
Binding Model			
Binding Time	Runtime	Runtime	Startup Time
Availability Time	Runtime	Runtime	Startup Time
Scope of Binding	Program	Program	Program
Modularity			
Asset/Variation Dependency	Stable Abstraction	Stable Abstraction	Stable Abstraction
Decomposition of Assets	Possible	Possible	Impossible
Decomposition of Variations	Possible	Impossible	Impossible
Efficiency			
Runtime Overhead	Highly dependent on discovery of enhancements, medium/high	Highly dependent on discovery of enhancements, medium/high	Implementation dependent, low
Memory Overhead	low	low	unknown
Compilation Effort	low	low	low
Other Criteria			
Complexity	low	low	high
Infrastructural Code	low, marking of enhancement points	low, definition of enhancement spots	unknown
Tool Support	yes, via Enhancement Builder	yes, integrated in ABAP Workbench	yes, integrated in ABAP Workbench
Tracing Support	no	no	no

The table above compares three SAP techniques by several important criteria defined in chapter 2.

The concepts of the techniques are different, depending on the level of abstraction they are used to vary existing functionality. While EF and BA allow variations on a code level, SF has got a notion of variation on a higher abstraction level, although this technique is also an implementation technique. While the first two allow refinements, the SF can be used for compositional variations. For this reason the granularity is coarser.

In terms of modularity all three approaches are looking alike. The concerns are structured into separate modules without clear relations between each other. In addition the reusable code is unaware of possible variations and will work without taking the functionality of potential extensions into account.

All approaches support the concept of late binding, that is, resolution takes place at startup- resp. runtime. For EF and BA variability is resolved at runtime, while SF relies on a database containing the values for switches which are evaluated at startup time. In addition the actual variations must be initially present at the same point of time, which allows a decoupled development of assets and variations.

All three approaches feature stable abstractions; unlike in AOP code injections at arbitrary positions are not possible. The decomposability is different in each technique; both assets and variations may or may not be decomposed.

Statements about the efficiency of the approaches are relative. Usually the runtime overhead in dynamic techniques like EF and BA are higher than in static approaches. The runtime overhead for SW is dependent from the actual implementation and also depends on the underlying database containing the value of the switches. The same is valid for statements about the memory overhead. Compilation effort is in every case low.

All approaches are supported by dedicated tools, but lack tracing support. The complexity is connected directly with the abstraction layer of the variations.

5.1.4 Business Rule Engines

A key property of SAP customers is that every business is different. Although there are many common parts (predefined business content and built in business best practice are actually major reasons why customers buy SAP software), most companies draw their competitive advantages out of subtle deviations from standard business processes. These variations often go beyond simply enabling/disabling switches or changing parameter values. Business experts need means for “programming in the large”, i.e., wiring state transitions and message-based process interactions, and “programming in the small”, i.e., being able to model conditional and/or parallel execution of business process steps, ideally supported by graphical tools.

The Business Process Execution Language (BPEL) [23] was standardized by the OASIS group for exactly that purpose. It interacts with external Web Services to orchestrate higher-level business processes out of these building blocks. Graphical tool support for constructing orchestrations is available, for instance, using the Business Process Modeling Notation (BPMN), as a graphical front-end to capture BPEL process descriptions. Numerous BPEL engines from different vendors already exist today for executing BPEL-based process descriptions.

An example for such business rule engines is the Business Process Engine (BPE) as part of SAP XI (see above): The business process engine (BPE) is tightly connected with the integration engine and fully integrated into the integration server. During message flow between heterogeneous systems, the engine makes use of all the shared collaboration knowledge needed to execute a business process. An easy-to-use graphical modeller gives you access to the message types and interfaces involved in a process. It lets you define the series and sequence of steps and actions required to run the process. During execution, the BPE also correlates and links related messages based on a unique, user-defined identifier.

In summary, there is a clear need for flexible configuration/variation of runtime behaviour by business domain experts (i.e., end users without sophisticated programming skills). Hence, DSLs or other formats for representing executable models are required, which can be dynamically loaded, interpreted and/or compiled at runtime.

5.2 Siemens

Siemens AG is a collection of business units that operate in different domains with different product innovation cycles and different business models. Therefore there is neither one common development process for all Siemens business units, nor one consistent set of development practices for product line engineering. Main differences are

- Solution versus product driven businesses, e.g. postal automation system solutions build on a common set of base assets, but have to be customized heavily for each customer, while telephone switches are standardized products.
- Product/innovation cycles range from a couple of months for e.g. mobile phones up to decades for rail traffic control technology.
- Security and reliability requirements, e.g. medical devices or traffic control systems have to fulfil high reliability and security requirements, while those requirements are a lot less critical for car entertainment systems.

Here, a rough overview over the implementation practices employed at Siemens.

5.2.1 Implementation Techniques for Variability

For efficiently handling a family software systems in a domain it is essential to know the domain abstractions and to generalize and separate them with stable interfaces. Stable interfaces are the most profound mechanism for reuse and exchangeability of implementations, which is a way to support variability.

Beyond that the following main technical options exist to cope with variations of base assets during software architecture, design and development :

- Another level of indirection—In this category fall the typical design patterns used for decoupling and configuration, such as Factory, Strategy, Extension Interface, Bridge and Adapter, but also general framework principles such as inversion of control and dependency injection, as intensively used by the Spring framework [47]. To avoid the mingling of variations and allow for easy re-configuration, configuration options are externalized into configuration files, where variations can be expressed declaratively. Certain architectural patterns, sometimes also referred to as architectural styles, such as event-based communication and Pipes and Filters architectures allow for more easy variation, as they inherently decouple a system into exchangeable parts.
- Language support—This includes approaches, such as aspect-oriented programming, where variations are encapsulated as aspects, template meta programming, where commonalities are expressed in templates, or domain-specific languages (DSL) combined with code generation. Further, macro languages, such as the C++ #ifdef construct, allow to for compile-time binding in source code.

All of those options are used in Siemens product lines, though generative approaches including AO are still rare.

The typical means are OO in combination with stable interfaces for important varying domain abstractions. A simple example for the latter is hardware abstractions in automation and control systems. Devices like motors, sensors or higher level entities like cameras or conveyor belts, which themselves group sensors and actuators, are represented as abstract interfaces to the machine control software. The gap between the interface provided by the hardware element and the interface required by the software has to be implemented for each device, but typically there are no adaptations to the control software required if new devices of a known type are integrated.

Nevertheless, developers of component-oriented business applications make increasing use of aspect-oriented programming, also within Siemens. Frameworks such as Spring or J2EE compliant containers like JBoss already offer aspect-oriented extensions. The very existence of frameworks, like EJB, and specific design patterns to decouple responsibilities confirms the need for AOP. They were developed to untangle concerns to be able to evolve and reuse infrastructure and business code separately. The advantage of AOP is that it is not limited to a single domain in the way that EJB is limited to server-side component computing [40]. Examples for AO in product lines in Siemens are Spring aspects for security and life cycle management in a platform for telecommunication applications and JBoss AOP in an IP based communication service for voice, video, unified messaging and instant messaging for service aspects.

The typical usage scenario for generative approaches including MDD are currently either generating glue code for embedding business components in a given platform or for easily formalize-able code like communication code in embedded systems. An example for the former is a DSL and a generator for generating interception proxies for a telecommunication application platform. The DSL allows attaching interceptors to business components, simulating a simple AO infrastructure. Another example is the generation of MOST-bus specific communication code for small controllers in a medical imaging system. Communication partners (device controllers) and the payload are specified in tables, supported by dedicated editors.

5.2.2 Binding Variability

Depending on requirements like footprint, security, and runtime flexibility different measures are taken for implementing and binding variability. E.g. telecommunication enterprise applications need runtime configurability and therefore implement component containers and composition filters for flexibly changing the runtime configuration of a system. Automation and drives software requires often small runtime resource footprint, therefore the variability will be bound at load time through configuration files. For high security domains, e.g. train traffic control and supervision systems the code has to be certified by national certification bodies. Variation is only allowed before compile time, so variability usually gets incorporated via #ifdefs. The actual code for a variant is specified by preprocessor defines and must not change after certification. Code for new variants is introduced in new conditional compilation blocks only.

5.2.3 Platforms

Business units that do not have a dedicated product line engineering approach nevertheless usually have at least a common base asset for domain specific infrastructure services called a platform. Such platforms typically care for communication, persistence, user interface support, some introspection support like tracing and debugging features and usually typical domain specific extensions like image processing for optical systems.

A common platform is often the first step towards product line engineering, since practices like commonality/variability analysis have to be introduced once the capabilities of a platform reach beyond general purpose middleware responsibilities.

Siemens has several examples of platforms that are the basis for further platforms, e.g. in medical engineering one platform for all imaging systems is the basis for further platforms in product lines for e.g. magnetic resonance systems or computer homographs.

5.2.4 Application Engineering and Product Derivation

For product driven business application engineering and product derivation can be as simple as assembling the product from the pre-built base assets. Often however, and definitely for solution driven business the product/solution hast to be customized or even product/solution specific extensions have to be implemented.

The goal however is to avoid implementation and derive new products mostly through customization. For example in automation systems it is common to have a staged approach for customization. On the top level the layout of an automation system is configured according to the hardware and mechanical capability of a machine. On the next level of configuration machines offer specific functions for calibration, where the machine either automatically or guided by an operator determines reference positions or settings and keeps acquired data for production runs. On the last level, a customer specific customizations can be set by “programming” the machine through teach-in or with dedicated domain specific programming languages.

Next to configuration files configuration and build management tools are the state-of-the-art tooling for product derivation. Configuration management tools are used for keeping and managing variations of base assets and allow to assign a label to a set of base assets, and for each of them exactly one version, that then form a base line or a product. Build systems can either use this information or get the information in their own scripting language on where base assets can be found and set pre-compiler variables and compiler switches for generating products.

While those mechanisms are proven technologies, the mapping between the information kept in build scripts and configuration management labels and the information on the feature set selected by those mechanisms is not well supported by tools. This information has to be kept separately.

5.2.5 Summary

According to the preceding sections the following techniques for implementing variability are in use at Siemens.

Development is (mostly) carried out in object-oriented languages. These are used in combination with stable interfaces for important varying domain abstractions. The actual implementation relies frequently on design patterns such as Factory , Strategy, Extension Interface, Bridge and Adapter, which are used for decoupling and configuration. In addition framework principles such as inversion of control and dependency injection are applied.

Apart from the actual implementation configuration options are externalized into configuration files, where variations can be expressed declaratively. It is at hand that this mechanism supports load time binding of variability. On the side of the system itself this is supported by component containers and composition filters for flexibly changing the runtime configuration of the system. Here, appropriate platforms are used, which form a common base asset for domain specific infrastructure services. In addition it should be mentioned that aspect-oriented programming is also used, where variations are encapsulated as aspects. As supporting technologies aspect-oriented frameworks like JBoss can be mentioned.

Design time variability binding can also be identified, applied examples are Template Metaprogramming, where commonalities are expressed in templates and generative approaches including MDD, where domain-specific languages (DSL) are combined with code generation. Macro languages, such as the C++ #ifdef construct, which allow compile-time binding in source code are popular in areas like Embedded Systems.

On architectural level architectural patterns are used (which are sometimes also referred to as architectural styles), such as event-based communication and Pipes or Filters.

5.3 HOLOS

Besides specific customer driven software development, HOLOS has been developing software for the European Space Agency using the Agile Modelling Methodology (http://www.agilemodeling.com).

This methodology is strongly used in projects where variability is not only a requirement at the end of the project (reusing project modules and lessons learnt from one project to another are current practice within ESA projects), but also during the development of the projects themselves.

Motivation for the use of this methodology has its roots in the need to strengthen the end-user involvement in the project and ensure the compliance with requirements throughout the development cycles. The active involvement and cooperation of the end-users is expected and necessary to take advantage of the proposed development approach.

The Agile methodology envisages three major iterations for the implementation process, which are:

- Functional Model iteration;
- Design & Build iteration;
- Implementation.

from which only implementation should be examined in this context.

The Implementation phase is the scenario where the latest increments in the iterative development methodology drive to a prototype that is fully released to the end user. This represents the transition from the development to the operational scenario – including final tuning – as well as the effective handing over to the end user, who – conveniently assisted - will perform the operational validation.

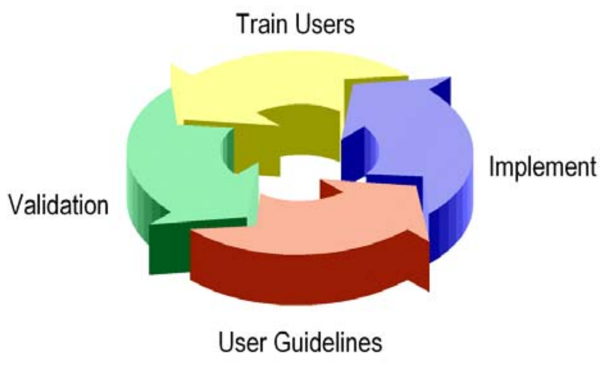


Figure 14. Implementation

The Implementation is subdivided into four major tasks:

- the "user guidelines" tasks provides the straightforward connection with the end user and helps to plan the last cycle of iterations;
- the effective implementation of the operational ready prototype;
- the assistance to the users in their operation in the prototype. The Handing Over Process is expected to have taken place in the meantime.

Prototype presentation, demonstration and effective validation – at the client's premises – closes the nominal iteration cycle of this phase.

HOLOS view on the application of this methodology is as follows. The results of the application of this methodology ensure that the components of software developed are fully compliant with the end-users' requirements, since they are involved throughout the whole process.

Partial test of the prototypes being developed also presents the end-user with possible limitations of the technology at an early stage, which, in turn, gives rise to revision of requirements, but also ensures that at the end, the user is presented with a system whose "usability" is directly what he/she expects. At an early stage the prototypes are released to the end user where tests are conducted, most of the times with real data and on real operational conditions. Early test of prototypes helps identifying possible bottlenecks (e.g. performance) and provides a forum for the discussion and selection of alternatives that effectively meet the requirements or the revision of requirements (even those introduced during the process).

The development of the prototypes is always done in a modular fashion where module interfaces are agreed upfront, thus allowing for reusability. Based on the design of the early prototype the actual implementation stage starts. The development team assembles in a meeting where the architectural design is reviewed and the new modules are defined. At the end of the meeting the team will start producing the necessary refactoring of the code and produce the new code. Again, all these steps are closely followed by the client's team that always provides some feedback on the produced items.

The end of the implementation produces the review of the test documents. A requirement vs. test case matrix is produced to confirm that all the final requirements are covered by a test.

6. Conclusion

As shown in chapter 3, there is a big variety of techniques for handling variability on implementation level. Different techniques are tailored for application on different scopes and have different qualitative properties. This level may be as low as the parameterization of a method and as high as the composition of software systems out of reusable components.

In chapter 2 we defined a concept of a variation mechanism that can capture very different approaches of variation management. We introduced a coherent terminology that establishes a unified view on very different variation mechanisms and in this way enables their comparison. Besides, we described a broad range of criteria that provide an insight into the design space of the variation mechanisms and form a basis for the analysis of their differences, advantages, disadvantages and combination possibilities.

The concepts and criteria formulated in chapter 2 were applied in chapter 3, were we analysed various technologies for their support for variability. Our analysis includes mainstream technology, such as object-oriented programming languages, design patterns, frameworks, component technology and conditional compilation. These approaches are in use for a long time in industry, and therefore are well understood. In addition, we evaluated a set of more advanced technologies that are of special interest to the project. Feature-oriented programming, aspect-oriented programming and model-driven development are falling to this category. The combination of aspect-oriented and model-driven techniques has not been elaborated intentionally, because this is the focus of Task 3.3, which will identify respective strengths and weaknesses of AOP and MDD. The results of this task will constitute a part of the upcoming deliverable D3.2.

Some of the evaluated approaches, e.g. component technology, are not traditionally seen as technology for variation management, but the conceptual basis formulated in chapter 2 allowed us to view these mechanisms from a new perspective and to identify their support for variability. The criteria formulated in chapter 2 will also be useful for evaluation of the future contributions of the project.

Variations mechanisms are techniques, but in order to be applied techniques require support by appropriate tools. Tool support is an important aspect in the evaluation of applicability of techniques. For this reason additional criteria for the comparison of tools supporting variation mechanisms have been defined in section 2.2. Here, aspects like the underlying conceptual and technical concepts, the extent to which a development process is covered and availability and interoperability have been analysed.

An overview about tools itself is given in chapter 4. In principle, almost any tool related to developing software may also be used in the context of software product lines. To narrow the choice of potential candidates, the set of considered tools has been limited to tools that have been developed for SPL engineering and management explicitly.

Another aspect of this deliverable is the analysis of existing practices applied at the sites of the industrial partners of the AMPLE project. One might expect that the applied techniques are quite the same. But in fact this is not entirely the case. SAP relies heavily on in-house technologies for variation handling like the Switch

Framework or the Enhancement Framework, while at Siemens a much broader diversity of external and internal techniques and tools is deployed.

From an abstract point of view however, similarities can be identified. Well understood “mainstream” technologies like OOP, design and architectural patterns, conditional compilation, frameworks techniques, Component-Based Software Engineering, and even Model-Driven Software Development are widely used. On the other hand, there is a lacking adoption of advanced research approaches like AOP, Feature-Oriented Programming, Feature Modelling, etc. The reasons are twofold: a) Maturity issues like scalability for large systems, tool integration, or debugging support, etc., which are rather out of scope for AMPLE in work package 3; and b) conceptual problems like resulting in maintainability concerns, where contributions are possible in the context of work package 3.

For instance, one road block for the adoption of AOSD techniques is the strong coupling between aspect and base code, which complicates upgrade releases and allows customers to create unsolicited extensions to core components. Research in explicit aspect interfaces could address this problem.

Another key challenge is the flexibility to arbitrarily change the binding time of some variability. Currently, the choice for a certain variation mechanism ultimately determines the binding time. Further research on Model-Driven techniques seems promising since MDD allows binding variability to code at various points; it could even produce runtime-interpretable DSLs out of some models to defer binding of variability to runtime.

The identified challenges and potentials for improvement will be taken up in the ongoing Task 3.3. Conclusions for concrete improvements of SPL implementation techniques will be given in the upcoming Deliverable 3.2.

References

[1] N. Loughran, P. Sánchez, N. Gámez, A. Garcia, L. Fuentes, C. Schwanninger, and J. Kovacevic: *"Survey on State-of-the-Art in Product Line Architecture"*, AMPLE deliverable D2.1, March 2007.

[2] Gears home page: http://www.biglever.com/solution/product.html.

[3] Pure::variants home page: http://www.pure-systems.com/Variant_Management.49.0.html.

[4] VarMod home page: http://www.sse.uni-due.de/wms/en/?go=256.

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

[6] C. Krueger, *"Variation Management for Software Production Lines*", Proceedings of SPLC 2002 , pg 37-48.

[7] T. Stahl, and M. Voelter, *"Model-Driven Software Development"*, Wiley, 2006.

[8] Eclipse website, http://www.eclipse.org/.

[9] EMF website, http://www.eclipse.org/modeling/emf/?project=emf.

[10] J. Herrington, *"Code Generation in Action"*. Manning, 2003.

[11] D. Batory. 2004. *"Feature-Oriented Programming and the AHEAD Tool Suite"*. In Proceedings of ICSE'2004. IEEE Computer Society, Washington, DC, 702-703.

[12] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, *"The GenVoca Model of Software-System Generators"*. IEEE Software 11, 5 (Sep. 1994), 89-94.

[13] E. Johnson, and B. Foote, *"Designing reusable classes"*, Journal of Object-Oriented Programming 1, 2 (June/July 1988), 22-35.

[14] G. T. Heineman, and W. T. Councill, *"Component-Based Software Engineering: Putting the Pieces Together"*. Addison-Wesley Professional, 2001.

[15] L. Mikhajlov, and E. Sekerinski, *"A Study of the Fragile Base Class"*. Proceedings of ECOOP'1998. LNCS, Vol. 1445, pages 355 - 382.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*. Addison-Wesley Inc., 1995.

[17] B. Meyer, and K. Arnout, *"Componentization: The Visitor Example"*, IEEE Computer, vol. 39, issue 7, pages 23-30, July 2006.

[18] C. Bussler. *"B2B Integration - Concepts and Architecture"*. Springer, 2003.

[19] J. Stumpe, and J. Orb. *"SAP Exchange Infrastructure"*. SAP Press., 2005.

[20] J. McAffer, J.-M. Lemieux. *"Eclipse Rich Client Platform. Designing, Coding, and Packaging Java Applications"*. Addison Wesley, 2005.

[21] E. Clayberg, and D. Rubel. *“Eclipse. Building Commercial-Quality Plug-Ins”*. Addison Wesley. 2006.

[22] Horst Keller, and Sascha Krüger. *„ABAP Objects. ABAP-Programming in SAP NetWeaver”*. Galileo Press. 2007.

[23] *“WS-BPEL 2.0 Specification.”* OASIS Standard. 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[24] R.S. Pressman, *“Software Engineering: A Practitioner's Approach”*, McGraw-Hill, 2001.

[25] B. Meyer. *"Eiffel: the Language"*. Prentice-Hall, Inc., 1992.

[26] openArchitectureWare (oAW) website, http://www.eclipse.org/gmt/oaw/.

[27] P. Tarr, H. Ossher, W. Harrison, M. Sutton. *"N degrees of separation: Multi-dimensional separation of concerns"*. Proceedings of ICSE'1999.

[28] Y. Smaragdakis, and D. Batory. *"Implementing layered designs with mixin-layers"*. Proceedings of ECOOP’1998.

[29] I. Aracic, V. Gasiunas, M. Mezini and K.Ostermann. *"Overview of CaesarJ"*. Transactions on Aspect-Oriented Software Development I. LNCS, Vol. 3880, pages 135 - 173, Feb 2006.

[30] E. Ernst. *"gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance"*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[31] G. Bracha, and W. Cook. *"Mixin-based inheritance"*. Proceedings of OOPSLA’1990.

[32] P. Wadler. *"The expression problem"*. Posted on the Java Genericity mailing list, 1998.

[33] E. Ernst, *"The expression problem, Scandinavian style"*. MASPEGHI 2004.

[34] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. *"Multijava: modular open classes and symmetric multiple dispatch for Java"*. SIGPLAN Not.35(10), pages 130 - 145, 2000.

[35] B. Meyer. *"Object-Oriented Software Construction (2nd ed.)"*. Prentice-Hall Inc., 1997.

[36] L. Mikhajlov, and E. Sekerinski, *"A Study of The Fragile Base Class Problem"*. Proceedings ECOOP'1998, LNCS, Vol. 1445. pages 355 - 382, 1998.

[37] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler, *"Making the future safe for the past: adding genericity to the Java programming language"*. Proceedings OOPSLA'1998.

[38] M. Mattsson, J. Bosch, and M. Fayad, *"Framework integration problems, causes, solutions"*. Communications of the ACM 42,10, pages 80 - 87., Oct 1999.

[39] G. Heineman, and W. Councill, *"Component-Based Software Engineering: Putting the Pieces Together"*. Addison-Wesley Professional, 2001.

[40] V. Matena, B. Stearns, and L. Demichiel. *"Applying Enterprise Javabeans: Component-Based Development for the J2EE Platform (2nd ed.)"*. Pearson Education, 2003.

[41] J. Lowy, *"Programming .NET Components (2nd ed.)"*, O'Reilly Media Inc., 2005.

[42] S. Vinoski, *"CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments,"* IEEE Communications Magazine, Vol. 14, No. 2, Feb 1997.

[43] D. Rogerson, *"Inside COM"*. Microsoft Press, 1997.

[44] OSGi Alliance website, http://osgi.org.

[45] M. Fowler. *"Inversion of control containers and the dependency injection pattern"*, http://www.martinfowler.com/articles/injection.html.

[46] Pico Container website, http://www.picocontainer.org/.

[47] Spring Framework website, http://www.springframework.org/

[48] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. *"An Overview of AspectJ"*. Proceedings of ECOOP'01, LNCS, Vol. 2072. pages 327-353., 2001.

[49] K. Klose, K. Ostermann, M. Leuschel *"Partial Evaluation of Pointcuts"*. Proceedings of the Ninth International Symposium on Practical Aspects of Declarative Languages (PADL), Jan 2007.

[50] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann *"Virtual Machine Support for Dynamic Join Points"*. Proceedings of AOSD'2004.

[51] R.Hirschfeld. *"AspectS - Aspect-Oriented Programming with Squeak"*. Objects, Components, Architectures, Services, and Applications for a Networked World, LNCS, Vol. 2591, pages 216-232, 2003.

[52] C. Bockisch, M. Haupt, M. Mezini, R. Mitschke *"Envelope-based Weaving for Faster Aspect Compilers"*. Proceedings of Net.ObjectDays, 2005.

[53] J. Aldrich. *"Open Modules: Modular Reasoning about Advice"*. Proceedings of ECOOP'2005, LNCS, Vol. 3586, pages 144 – 168.

[54] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, H. Rajan, *"Modular Software Design with Crosscutting Interfaces"*. IEEE Software. 23, 1, Jan 2006.

[55] N. Nystrom, X. Qi, A. C. Myers, *"J&: Software Composition with Nested Intersection"*. Proceedings of OOPSLA'2006.

[56] S. M. Swe, H. Zhang, S. Jarzabek, *"XVCL: a tutorial"*. Proceedings of the 14th international Conference on Software Engineering and Knowledge Engineering, Jul 2002.

[57] Delta Software Technology, ANGIE website, http://www.d-s-t-g.com/neu/pages/pageseng/et/common/techn_angie_frmset.htm

[58] W3C Consortium. *"XSL Transformations (XSLT), Version 1.0"*, J. Clark, Editor, W3C Recommendation, November, 1999, http://www.w3.org/TR/xslt.

[59] A. Alexandrescu: *"Modern C++ Design: Generic Programming and Design Patterns Applied"*. Addison-Wesley, 2001.

[60] T. Sheard, S. P. Jones, *"Template meta-programming for Haskell"*. In Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell.

[61] S. S. Huang, D. Zook, Y. Smaragdakis, *"cJ: enhancing java with safe type conditions"*. Proceedings of AOSD'2007.

[62] S. S. Huang, D. Zook, Y. Smaragdakis, *"Morphing: Safely Shaping a Class in the Image of Others"*. Proceedings of ECOOP'2007.

[63] M. D. Ernst, C. Kaplan, C. Chambers. *"Predicate dispatching: A unified theory of dispatch"*. Proceedings of ECOOP '1998, LNCS, Vol. 1445, pages 186 - 211, 1998.

[64] C. Chambers. *"Object-oriented multi-methods in Cecil"*. Proceedings ECOOP '1992, LNCS 615, pages 33 – 56.

[65] K. Czarnecki and C. H. P. Kim. *"Cardinality-Based Feature Modeling and Constraints: A Progress Report"*. In OOPSLA'05 International Workshop on Software Factories (online proceedings), 2005.

[66] K. Czarnecki and K. Pietroszek. *"Verifying Feature-Based Model Templates against Well-Formedness OCL Constraints"*. Proceedings of GPCE'2006 - Generative Programming and Component Engineering.

[67] IBM Rational Software Modeler (RSM) home page: www.ibm.com/software/awdtools/modeler/swmodeler/index.html.

[68] IBM Rational Software Architect (RSA) Home Page: www.ibm.com/software/awdtools/architect/swarchitect/.

[69] M. Antkiewicz, K. Czarnecki. *"FeaturePlugin: Feature Modeling Plug-in for Eclipse"*. In Eclipse '04: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange, Vancouver, British Columbia, Canada, 2004.

[70] Feature Modeling Plug-in (fmp) Home Page: http://gp.uwaterloo.ca/fmp.

[71] K. Czarnecki, M. Antkiewicz. *"Mapping features to models: A template approach based on superimposed variants"*. Proceedings of GPCE'2005 - Generative Programming and Component Engineering, LNCS, Vol. 3676, pages 422 - 437.

[72] MofScript home page: http://www.eclipse.org/gmt/mofscript/.

[73] Acceleo home page: http://www.acceleo.org.

[74] Mapping Features to UML 2.0 Models Plug-in Home Page: http://gp.uwaterloo.ca/fmp2rsm.

[75] IBM Rational Downloads Home Page: http://www.ibm.com/developerworks/rational/downloads.