Fraunhofer Institut
Experimentelles
Software Engineering

Product Line Implementation Technologies

Programming Language View

Authors:
Thomas Patzke and
Dirk Muthig

Funded by Stiftung Rheinland-Pfalz für Innovation, Project #15202-386261/51: “Entwicklung und Erprobung eines Methodenleitfadens für Software-Produkt-linien-Implementierungstechnologien”

IESE-Report No. 057.02/E
Version 1.0
October 31, 2002

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Executive Summary

Nearly all software organizations today develop and maintain more than a single product. This holds for organizations that develop tailored systems individually for single customers, as well as for organizations that develop products for a mass market. The products developed by an organization typically are similar applications in the same application domain. Hence, these products share some common characteristics and thus can be viewed as a software product line.

To implement a product line approach in practice, special technologies are required that effectively support the identification of reusable artifacts, as well as explicit means for capturing and controlling commonalities and variabilities. The focus of the PoLITe project are product line technologies at the implementation level. PoLITe defines three categories of implementation technologies [MAL+02], namely configuration management, component technologies, and generative features of programming languages (including generators). This report summarizes the product line implementation aspects in the programming-language dimension.

The report first defines the term “programming language” and its role in the PoLITe context, surveys the actual distribution of programming languages in practice. It further categorizes variability techniques in the programming-language dimension and describes numerous of them in detail. This categorization is complemented by a high-level process that integrates the systematic usage of variability techniques throughout the implementation activities. Finally, the application of variability techniques is illustrated by two case studies: one uses C++, the other Java analogously.

Keywords: software product lines, product line implementation technologies, variability mechanisms

Table of Contents

1 Introduction

Nearly all software organizations today develop and maintain more than a single product. This holds for organizations that develop tailored systems individually for single customers, as well as for organizations that develop products for a mass market. Even for organizations that believe to develop a single product only, surveys have uncovered that also these organizations spend most of their resources on tailoring their systems to the needs of individual customers or enhancing systems by features that are newly required by customers [KHS98], and thus also these organizations must maintain and evolve a set of customer-specific variants.

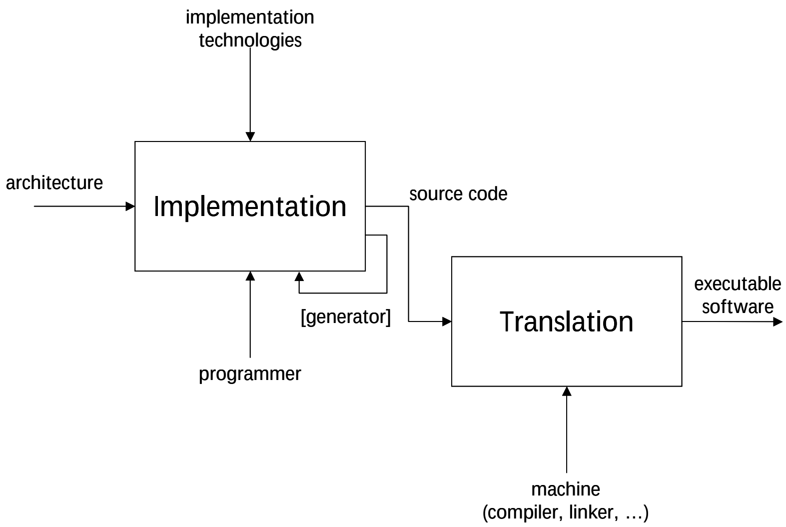
The products developed by an organization typically are similar applications in the same application domain. Hence, these products share some common characteristics and thus can be viewed as a software product line. Typically, the complexity and size of these software products, today, rapidly increases and customers are requesting more and more quality products tailored to their individual needs. Due to these increasing challenges, also the requirements on the development skills of an organizations increase. Hence, there is a need for organizations to learn how to manage a product line.

Software product lines are a reuse approach; the reuse approach that promises to master all of the described challenges [ABB+01]. To implement a product line approach in practice, special technologies are required that effectively support the identification of reusable artifacts, as well as explicit means for capturing and controlling commonalities and variabilities. The focus of the PoLITe project are product line technologies at the implementation level.

1.1 Product Line Implementation

In traditional software development approaches implementation typically means producing source code [Cop99]. The implementation activities refine a system's architecture down to a level that can be interpreted by a machine. Hence, the information captured by implementation artifacts is at the lowest level of abstraction compared to the content of all other development artifacts. The implementation artifacts (i.e., the source code) can be transformed automatically into an executable form. The implementation process is, at a general level, depicted in Figure 1 (using the IDEF0 notation). It covers all implementation-related activities in the general product line life-cycle and integrates them into a single implementation process. Hence, this process only implicitly distinguishes between implementation-for-reuse and implementation-with-reuse. In the

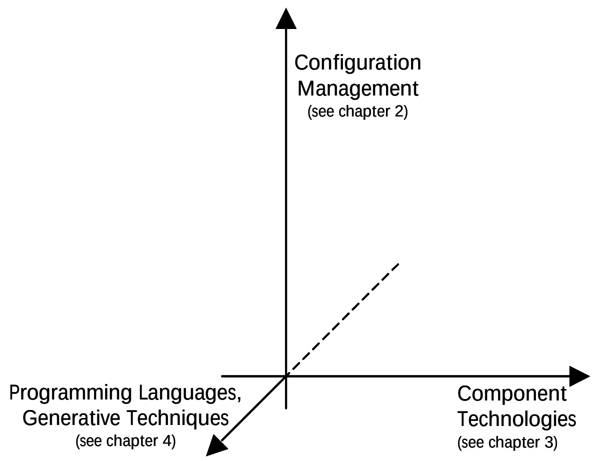
Figure 1:
Software implementation



PoLITe project, we discuss implementation technologies in the context of this process because it allows both aspects of an implementation technology to be discussed together independent of life-cycle issues.

The PoLITe project defines three categories of implementation technologies [MAL+02], namely configuration management, component technologies, and generative features of programming languages (including generators). The three dimensions are depicted in Figure 2. Initially, each dimension is investigated separately.

Figure 2:
Technology dimensions enabling the automation of implementation approaches



This report summarizes the product line implementation aspects in the programming-language dimension.

1.2 Programming Language Dimension

This report purely presents an approach to manage currently available implementation technologies from a programming language point-of-view. Note that the other two dimensions, configuration management and component technologies, are investigated and described in reports of their own.

The underlying theme is to find an efficient bridge between product line architectures and its generic implementation. To understand the situation we first look at the way single systems are implemented. In single-system cases, software is created on a one-by-one basis without any pro-active reuse activities. That is, the models describing a required system are realized straight-forward as illustrated in Figure 3.

In a product line context, the series of products delivered by an organization is seen as a series of products that are variations of the same infrastructure. From this viewpoint, models describe not only a single system but a set of similar systems in the same application domain. Hence, the models capture generic information.

Product line approaches have primarily been concentrated on the activities producing models covering the right scope of genericity, that is, on activities early in the software engineering lifecycle. The scope of this report, however, is to start with such generic models and systematically explore ways for supporting and guiding their generic implementation.

So we assume that models may contain besides common also variable elements. The output of implementation activities, that typically is, source code, must then reflect all these elements explicitly. In other words, implementation activities become more complex because they have to realize also variation. The challenge is to understand the techniques available for realizing variation and to know which of these techniques fits best for a given variability at hand. Figure 4 illustrates the generic implementation that realizes variant model elements by using variability techniques provided at the programming language level.

Figure 3: Implementing one-at-a-time systems

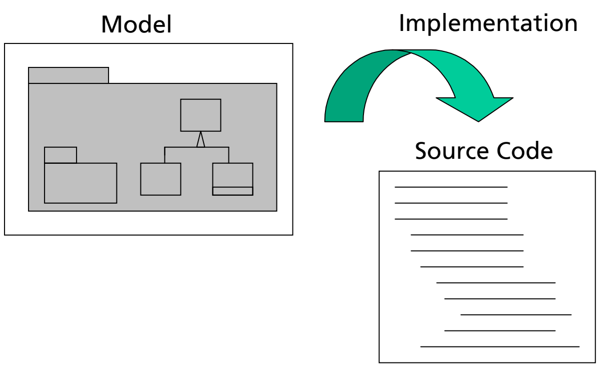
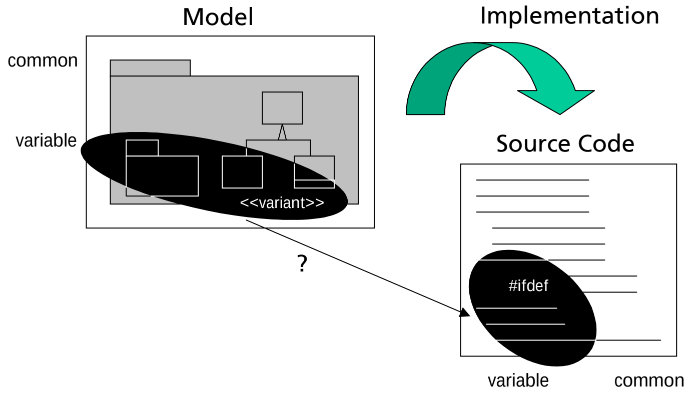


Figure 4: Implementing Software Product Lines



1.3 Outline

This report summarizes product line implementation technologies from a programming language point-of-view. The remainder of the report is organized in the following way:

Chapter 2 defines the term “programming language” and its role in the PoLITe context, surveys the actual distribution of programming languages in practice, and selects - based on the survey results - the programming languages used in the case study: C++ and Java.

Chapter 3 describes the implementation activities as illustrated by Figure 4 in more detail. The main enhancement is that already the models are organized in a component-based way and then only the thus smaller models of single components are realized separately and in a generic way.

Chapter 4 represents the core of the report: it categorizes variability techniques in the programming-language dimension and describes numerous of them in detail. Chapter 4 is complemented by Chapter 5, which introduces a high-level process that integrates the systematic usage of variability techniques throughout the implementation activities.

Chapter 6 demonstrates some of the variability techniques in a case study and shows the benefits of these techniques compared to single-system practices concretely. Chapter 7 summarizes the report and gives an outlook on future research activities to further improve the current state-of-the-practice.

2 Programming Languages

Traditional product line approaches have concentrated on the early phases of development like domain analysis and scoping. There are at least two reasons for this: First, product line engineering has its roots in the software reuse community who favour a top-down approach. Second, there has been lack of guidance in implementation technologies such as programming languages.

In order to explicitly support product line concepts in later phases of development, we have to address two conflicting forces:

On the one hand, when focussing on how general product line concepts can be implemented, we have to start with a general, abstract view on programming languages. Here we concentrate on the commonalities of the different programming languages and explicitly avoid details specific to a certain language. On the other hand, a specific programming language is used in practice and so just these details become highly important.

The following subsections are organized as follows: First, a general characterization of programming languages is presented, followed by an automation model for programming languages. After that, a survey of the current distribution of programming languages is presented. Finally, the most important ones from the perspective of our target group, small- to medium-sized companies, are selected for our case studies of product line implementation technologies for programming languages (see section 6).

2.1 Definition

According to [Bro], a programming language is

An artificial, formal language for the formulation of work instructions to a computer system. The programming language explicitly defines the vocabulary and the grammar a correct program text has to be written in.

The choice of programming languages is dependent on the needs of the customers of the products of the programming languages ([Mar02], pp.1010-1012). Three kinds of concerns for programming language adoption can be identified: contextual concerns, problem domain issues and computing models (Figure 5).

Figure 5: Programming Language Concerns

- Contextual Concerns
  - Social
  - Physical
- Problem Domain Issues
  - General-Purpose Language
  - Domain-Specific Language
- Computing Models
  - Imperative
  - Declarative
  - Object-Oriented

Contextual aspects comprise social or physical contexts of use (e.g. that the program produced by the language runs sufficiently fast on a specific platform, that it can co-exist with legacy code, or what level of support or training is available for the language).

Problem domains are those areas in which the languages are used to solve problems, for examples in business, embedded, scientific or web systems. General-purpose programming languages are those which are useful in most problem domains, whereas a domain-specific language (DSL) is a specialized, problem-oriented language ([CE00], p.140) like (Embedded) SQL or $\TeX$.

The ease of use of a general-purpose programming language in a particular problem domain is affected by its computational model. The main models are the imperative, the constraint-based and the object-oriented model.

In imperative (procedural) languages a program consists of a sequence of operations which describe how to change data (state). Ada, C, Fortran and Pascal belong to this category.

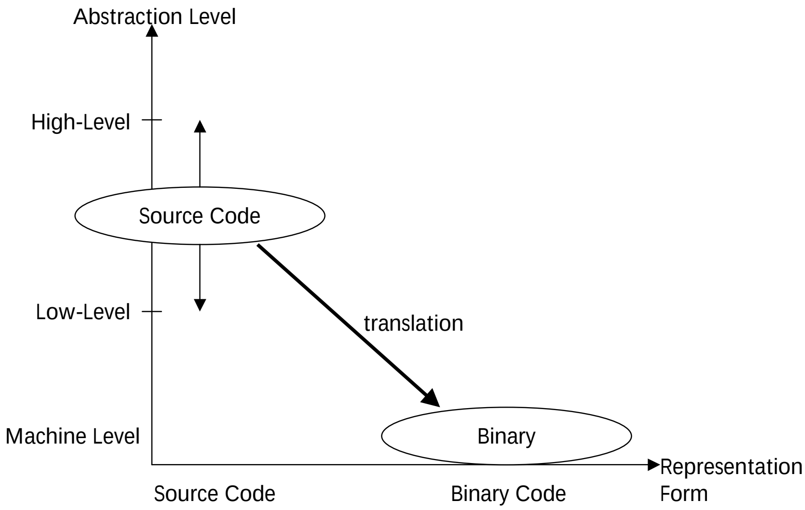
Whereas imperative languages focus on *how* to solve problems, declarative languages are concerned with *what* to solve, i.e. the problem. Two categories of these languages are the functional programming languages like Lisp and Haskell and the logical programming languages like Prolog.

In object-oriented programming languages all information for solving a problem (data and operations) is represented as objects. Pure object-oriented languages like Java and Smalltalk exclusively support this model, whereas hybrid languages like C++ also support other models.

2.2 Automation

Like other implementation technologies, programming languages support automation. They take as input a description in a human-readable form, usually text (source code), which symbolically represents programs, and transform it into a binary, machine-readable form (figure 6). The input is at a higher level of abstraction than the output, and depending on the input's level of abstraction programming languages are categorized as high-level (favouring humans) or low-level (machine-oriented).

Figure 6: Automation in programming languages



Programs written in a high-level language are usually transformed by a compiler or interpreter. In both cases the source code is first translated into an intermediate form. Whereas the interpreter translates each line of code in succession and immediately executes it, the compiler translates entire pieces of source code into modules of object code, which can then be transformed into executable binaries via a linker.

2.3 Programming Languages in Practice

In order to be useful for technology transfer, the specific technologies have to be focused on which are in common use. Thus, the most widely used programming languages in industry should be considered, which support a minimal set of mechanisms for product line implementation. In this section, we present a survey of important programming languages at the time of this writing, and select those two with the most practical importance.

2.3.1 Distribution

The total number of programming languages is estimated to be far more than 1000 and about 20 are wide-spread [Bro]. A survey of popular programming languages mentioned on the internet between mid-2000 and mid-2002 led to the following results:

For a first overview, the number of links to specific programming languages at major programming language-related websites was examined. The results are captured in table 1.

Table 1:
Links to programming languages (October 2002)

	Dmoz	Google	Cetus	Amazon	Sourceforge
Java	**2338**	**2570**	**2691**	**1922**	**6661**
C++	**703**	**793**	**1094**	**1343**	**8425**
Perl	**895**	**943**	118	497	3897
Visual Basic	412	453	**753**	**1203**	1018
Delphi	466	523	374	274	796
C#	127	131	230	154	394
JavaScript	454	464	603	368	862
Python	321	343	219	147	1912
PHP	517	449	115	753	5041
Ruby	116	99	20		168
Cobol	146	157	121	545	
C	188	224			**9072**
Fortran	111	126		666	73
Smalltalk	202	231	197	61	39
Ada	404	408	166		49
Lisp/CLOS	325	351	90	165	207

The Open Directory Project maintains a list of programming languages where Java, Perl and C++ have the highest numbers of links [Dmoz].

These languages also appear most in the Google directory [Google].

The Cetus Links are an index to over 18,000 internet addresses about object-orientation and component-orientation. The figures mentioned are the number of sections on programming languages; Java, C++ and Visual Basic are the dominant languages [Cetus]. Note that these figures have not changed within 2002, so they may be outdated.

The same results were gained in a search for books about a specific programming language at Amazon.com.

The world's largest Open Source development website also maintains figures of the numbers of registered projects for a specific programming language, with C, C++ and Java heading [SourceForge].

These two recent German surveys determined distributions of major programming languages:

Table 2:
Recent distributions of programming languages in Germany

	iX, 11/00	iX, 11/02	Gulp, 4/01
Java	38%	38%	18.9%
C++	36%	32%	13.8%
Perl	39%	40%	2.8%
Visual Basic	16%	8%	2.4%
Delphi			0.2%
Cobol			10.3%
C	38%	35%	
Python	5%	8%	

In November 2000 and 2002, the use of programming languages was published as part of a readers' survey in German IT magazine iX ([iX1], [iX2]). In both surveys, Perl, Java and C headed the ten languages mentioned most, Delphi was not among the top ten languages, and Visual Basic was significantly losing importance.

Gulp, a German portal for IT projects compared the market situation of Java vs. C++ in April 2001. They store qualification profiles for over 29,000 free-lancers and offer queries for qualification profiles of these. An analysis of the number of such queries revealed that Java, C++ and Cobol were the most requested programming languages [Gulp].

Two 2001 surveys from Evans Data [Byte] and Meta Group [Infoworld] have also shown Java and C++ as the most popular programming languages.

2.3.2 Selection

For our case studies we selected two programming languages: C++ and Java. On the one hand, they belong to the most popular programming languages in industry at the time of this writing[1], on the other hand they support component-based design and are object-oriented, which is beneficial in a product line context.

1 [Mar02] report a C++ user base of at least 1.5 million.

The dynamically typed languages (Perl, JavaScript, Python, PHP and Ruby) also offer these benefits, but are still not that widely used as general-purpose programming languages in industry. The same applies to C#.

Smalltalk could also be an implementation language for product lines, but has been losing popularity, which also applies to Visual Basic and Delphi.

Although the older languages like C, Fortran or especially Cobol have still importance in parts of industry, they are not considered here because they lack important mechanisms for implementing product lines. Reengineering their code, and then using these wrappers by more suitable languages can be a way to introduce product lines in these cases.

3 Generic Component Implementation

An approach to handle system complexity in practice is to decompose functionality common to several systems into components which are meant to be used as-is. However, during integration these fixed components often need adaptation for each particular system in order to be reused. Thus, there is a need to handle variability as well when implementing components.

In this chapter, generic component implementations are introduced at an abstract level. In that context, chapter 4 will then deal with major implementation mechanisms in detail.

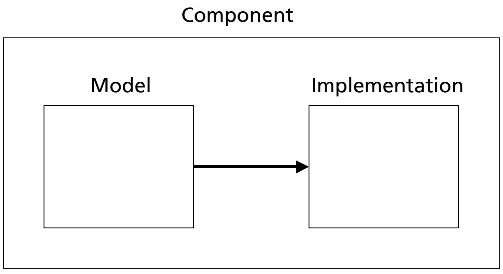
3.1 Component

Ideally, large-scale reuse as employed in software product lines leads to a software system which corresponds to exactly one huge component, often referred to as a platform. In contrast to unstructured monolithic systems as often found in practice [FY00], however, this kind of system can be decomposed recursively into a hierarchy of subsystems.

The other characteristic of these systems is that huge parts of several systems share the same components. Thus the components are meant to be used several times in the same system or in different systems, so that functionality often becomes a predominant partitioning criterion.

We describe each component by both a logical[1] model and a component implementation, as depicted in the following figure.

Figure 7:
Structure of a single component



1 The physical representation of files and directories is evaluated as another dimension of product line implementation technologies in a separate report on configuration management.

A model serves to describe the component at a coarse-grained level. Note that in this case we mean an analytic model as opposed to the analogic model, which is also used in software, but does not represent a description but another reality with similar properties ([Jac01], p.12).

After a component has been defined in a model, it is represented at the fine-grained level in the component implementation.[1]

3.2 Implementation

In the traditional view, implementation is regarded as a synonym for source code, as Coplien observes:

Implementation means actual code - not just diagrams or formalisms ([Cop99], p.9).

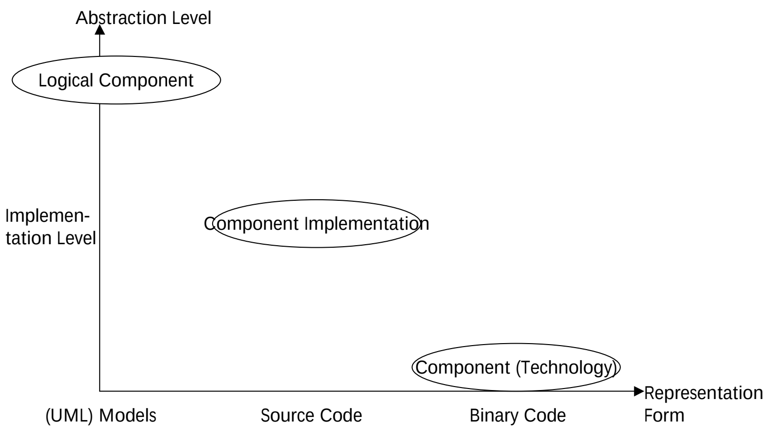
This view is also shared in present component technology[2], where a component implementation is seen as a block of code with a certain functionality which can be used without modification in several places as long as these are plug-compatible.

More generally, implementation is concerned with realizing a software work product which can be automatically transformed to an executable form. Figure 8 illustrates that the component implementation is in-between the logical and technical component; and this in two dimensions: at the abstraction level and in representation form. If the source code has a form which can be understood by a translation tool, like a compiler or interpreter, it can be automatically transformed to the binary, changing both its abstraction level and representation form at the same time.

In our view, the workproducts of product line implementation technologies from a programming language perspective differ in one important respect from those seen in component technology: they are generic.

1 According to ([CE00], p.23), components are one of the work products of Domain Implementation. Others include patterns, frameworks, DSLs and generators.
2 The component technology dimension of product line implementation technologies is dealt with in [AM02] and in a separate report.

Figure 8:
Source code within an implementation level



3.3 Generic

One characteristic of traditional components is that they can be used without modification in different (parts of) systems. They contain a fixed functionality which always stays the same. Consider the Open-Closed-Principle, whose intent was to improve quality attributes like reusability and maintainability in object-oriented technology: "Software entities (classes, modules, functions etc.) should be open for extension, but closed for modification." [Mar96a]

However, in many situations only a similar, but not exactly the identical functionality offered by a component is required. Thus, in order to reuse the component, its implementation has to be altered. Otherwise the fixed component would have to be wrapped or it would have to be reimplemented in its entirety just to reuse it in a slightly different context, resulting in code overhead or code duplication.

As Basset explains, software reuse is the process of adapting a component in order to make it usable ([Bas96], p.350). To summarize:

reuse = use + adaptation

In this sense, product line engineering distinguishes between development for reuse, i.e. creating generic artifacts for a family of systems[1], and development for use, i.e. creating specific artifacts for a single system [AM02].

1 Generative Programming [CE00] additionally includes the process of creating concrete product instances, where configurability [Som99] becomes a major issue.

Adaptation leads to variabilities, and thus generic implementation results in variant code. Figure 2 summarizes that variation points in the logical model are mapped to variant parts of code. In the next section we show how to manage this.

4 Variability Mechanisms

As shown in table 3, logical models contain both common and variable parts with different characteristics.

Table 3: Commonalities and variabilities within the logical model

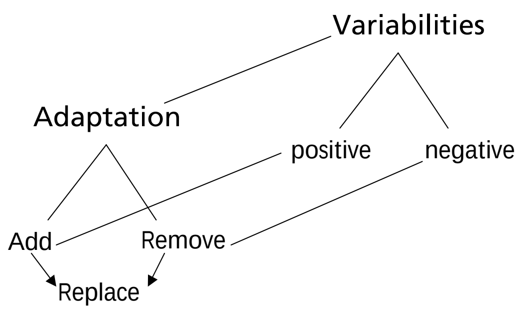
	Commonality	Variability
what?	structure, identity	behavior
focus is on	the same (as-is)	similar (same as-except)
parts are	complete	incomplete (gaps for adaptation)

The common parts form the long-term structure of the system, emphasizing its identity, sameness or completeness. As we have seen in the previous section, reuse often entails slight deviations from this sameness, and these similarities are often closely related to behavioral issues. The reusable models are incomplete, containing gaps which are deliberately left open to enable adaptation.

Two basic kinds of adaptation can be distinguished: Add and Remove. A third kind, Replace, is obtained by combining these. As illustrated in the next figure, the basic kinds of adaptation are reflected in the two kinds of variability, positive and negative variability: Whereas the former often serves to add behavior, the latter mainly helps to remove common behavior.

Negative variabilities should be avoided and turned into positive ones, because they break the common structure, which is the skeleton of the component.

Figure 9: Kinds of adaptation and corresponding variabilities



4.1 Classification

Variabilities are represented by variation points, and these generally describe alternatives. Three simple kinds of variability can be distinguished ([SVGB00], [VGBS01]), as depicted in figure 10.

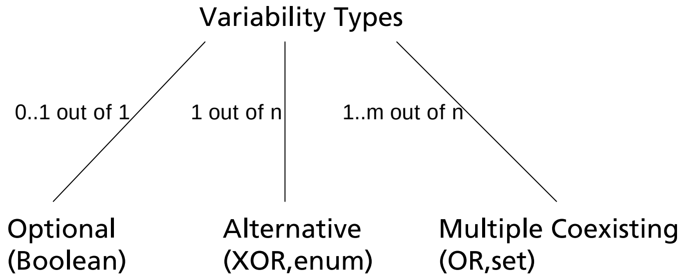


Figure 10:
Simple kinds of variability

In the simplest case, only one possibility exists, which can be included or excluded. This optional variability corresponds to a boolean type of variability point; 0 or 1 out of 1 variability has to be resolved.

In the other cases more than one possibility exists:
All these possibilities can be mutually exclusive so that just exactly one is selectable.This alternative XOR-kind of variability corresponds to an enum-type of variability point; 1 out of n variabilities has to be resolved.
Otherwise, the possibilities are not mutually exclusive so that one or more can be selected. The OR- kind of variability is reflected by a set-type of variability point[1]; m out of n variabilities, can be resolved, that means, multiple coexisting entities are available.

By combining these three basic kinds of variation point more complex types can be obtained, for example:

- Merging an optional with a mutually exclusive one results in a variation point where 0 or 1 out of n variabilities have to be resolved;
- Combining a non-exclusive and an optional one yields a variation point where 0 or more out of n variabilities can be resolved;
- Merging an optional one with itself results in a variation point where the same kind of variability can optionally be selected up to two times.

Another kind of variation point is the range variation point [Mut02].

1 The boolean, enum and set-types of variability points appear in the product line tool BigLever Gears.

The three kinds of adaptation often result in the three basic kinds of variation point, as shown in the following table:

Table 4:
Typical relationship between kinds of adaptation and variation point

	Add	Replace	Remove
optional	X		X
XOR		X	
OR		X	

A number of traditional and emerging language mechanisms exist for expressing behavioral variabilities. Table 5 lists which kinds of basic variabilities these mechanisms are primarily concerned with. Bold marks indicate that the mechanism can easily be applied for expressing the variability, small marks mean that the mechanism is sometimes applied for implementing the variability, and no mark indicates that expressing the variability through the given mechanism requires some effort and thus is not applied that often.

Table 5:
Variability types expressed by language mechanisms

	Optional	XOR	OR
Conditional Compilation	X	x	
Subtype Polymorphism		X	
Parametric Polymorphism	x	X	X
Overloading		X	
Coercion	x	x	x
Defaults	X		
Collaborations	x	x	X
AOP	x	x	X

In the next sections, each of these language mechanisms is explored in greater detail.

4.2 Conditional Compilation

Variabilities on the conceptual level are physically implemented as optional or alternative code parts (see figure 10). The problem is to keep all these variable as well as the common code parts in one place (e.g. file) while at the same time being able to choose one specific configuration. Moreover, one kind of variability often affects several files, whereas the configuration should concentrate in exactly one place.

As an example, consider debugging support. Spread throughout the code, at certain places additional checks should be applied, which should be disabled in the final executable for efficiency reasons.

Macros are one commonly used (often negative) variability mechanism in such cases. They represent an early abstraction mechanism introduced in assembler languages. A macro is
a module containing a body of code that is copied during compilation or assembly when invoked by name. It is not readily modified during invocation ([Bas96], p.347f.).

Macros provide for statically expressing optional and, to a lesser degree, alternative code parts at a maximum level of granularity, i.e. they can even affect single statements or parts thereof.

However, the fine-graininess is also one disadvantage of macros: for large projects, they tend to clutter the code, impeding comprehensibility as a whole. Moreover, conventional macros only have a global scope and, being preprocessing constructs, do not have any language support. They or their parameters might interfere with each other or with constructs of the programming language; they may even be used to change the semantics of the programming language (for example, by redeclaring the keyword private as public). Experience has shown that the use of conventional macros as a product line implementation technology tends to be problematic [AG01].

Frame technology [Bas96] uses an advanced macro-processor which avoids some of the drawbacks of conventional macros. Components are expressed in hierarchies of frames, which are adaptable components expressed in an arbitrary programming language. A frame contains both program code and frame commands, which serve to add from other frames' capabilities as the application requires.

4.3 Polymorphism

As illustrated in figure 11, another major common mechanism for handling variability is polymorphism, which is the ability of code to work with different types [CE00]. As opposed to macros, polymorphism has direct programming language support, which simplifies the detection of programming errors. Four kinds of polymorphism can be used for expressing variability: subtype (inclusion) polymorphism, parametric polymorphism, overloading and casting (coercion).

Figure 11:
Well-known variability mechanisms

- Conditional Compilation
- Polymorphism
  - Subtype Polymorphism
  - Parametric Polymorphism
  - Overloading
  - Coercion

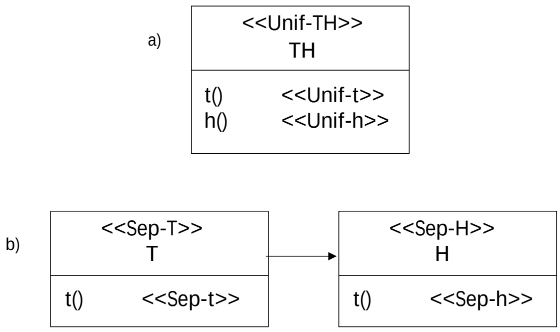
4.3.1 Subtype Polymorphism

Subtype polymorphism is implemented in object-oriented languages by defining one abstract class (interface) as a common point of access and implementing the alternative behaviour in concrete subclasses; thus it is mainly concerned with the XOR-kind of variability.

This mechanism represents one key element of object-oriented design patterns [GHJV95] and object-oriented frameworks [JF88]. [FPR01] identify two essential construction principles of object-oriented frameworks which are based on the arrangement of template and hook methods (see figure 12): Either both kinds of methods share the same class hierarchy, which they call the unification principle, or the two are in different class hierarchies (separation principle).
In the case of the unification principle, their notion of template methods corresponds to the classical Template Method pattern [GHJV95], which plays a prominent role among the targets of object-oriented refactorings [FBB+99]. Major design patterns based on the separation principle are State, Strategy[1] and Bridge. The inheritance-based unification principle underlies white-box frameworks, while the composition-based separation principle is used in the more mature black-box frameworks.

1 Note that [GHJV95] mention the concept of families in two of their pattern rationales: Strategy defines a family of algorithms, and Abstract Factory creates families of related objects.

Figure 12:
Unification (a) and separation principle (b)



Subtype polymorphism as a variability mechanism imposes two kinds of drawbacks concerning quality attributes:

First, as subtype polymorphism is applied dynamically (its binding time is run-time), its extra level of indirection appears in the executable code, which has a negative performance impact. In some cases, this run-time flexibility is actually needed as a functional requirement at the same time, for example when one out of several classes is chosen to be instantiated as the result of a run-time event. But the design should explicitly reveal if the binding time has to be run-time, because otherwise a non-run-time mechanism could be more effective.

Second, and worse, subtype polymorphism restricts reusability ([Bas96], p.13). In contrast to generic implementations, it only allows you to effectively add one kind of variability at the same time, for example by subclassing. In other words, variabilities of the OR-kind cannot be cleanly expressed, so that each class represents a feature combination, not a single feature, in its entirety. This results in a huge number of classes and a large amount of code duplication when adding new features. [OT01] refer to this phenomenon as the “tyranny of the dominant decomposition” because one way of composing the system (e.g. into a certain class hierarchy) forces all other features to conform to this composition, even if this restricts their quality attributes (e.g. by duplicating common features). This problem can only be resolved by applying another set of techniques like multiple inheritance, multiple class hierarchies [Mar96] or by applying design patterns like Visitor, with negative impacts on complexity and performance.

4.3.2 Parametric Polymorphism

Subtype polymorphism as embodied in the unification (with class inheritance) or separation principles (with object composition) is one way of composing behavior in object-oriented systems, and a third way is to employ parametric polymorphism ([GHJV95], p.22). It represents an alternative non-OO implementation technique for product lines. As with subtype polymorphism, it allows for implementing common and variable, optional and XOR-alternative parts of code, as well as template and hook methods. However, this is not done at run-time, but at construction time[1], which avoids the aforementioned performance and combinability drawbacks. One the one hand, it allows the compiler to inline the code, so that the extra level of indirection only appears at construction time and so no run-time performance penalties arise. On the other hand, it enables us to implement OR-variabilities in a straightforward way, resulting in a far larger amount of feature combinations with less code because whole features (i.e., combinations of incomplete features) can be combined in arbitrary ways, which provides a means for the separation of concerns [TOH+99].

Paul Basset emphasizes the duality between run-time and construction time as well when he defines the latter as
those times when we vary the data structures and methods that we choose to hold fixed at run-time ([Bas96], p.13).

Like macros, parametric polymorphism is a static variability mechanism[2], but it has programming language support and so avoids many of the disadvantages of macros.
The disadvantages of parametric polymorphism are that compile-times might increase (which can be a problem if compilations occur frequently), that it is supported by fewer programming languages, and that less experience has been gathered with it.

Whereas in unbounded polymorphism (unconstrained genericity) no constraints on the parameter of variation are expressible, this can be done in bounded parametric polymorphism (constrained genericity). However, this means for explicitly expressing required interfaces is supported by only a few programming languages like ML, Eiffel, or Ada95 [DGD01].

4.3.3 Other Kinds of Polymorphism

In addition to this group of universal polymorphism, the older[3] mechanisms of ad-hoc polymorphism, overloading and casting, can be used as variability mechanisms as well. In combination with object composition, they offer a way to compose behavior in object-oriented systems.

1 In this report we do not distinguish between the different kinds of construction time like source-, compile- or link- (load-) time as presented in ([Cop99], p.73).

2 The parametric polymorphism of C++ was initially implemented, like its macro support, with a preprocessor [Str94].

3 Overloading and coercion were introduced in the late 60s in the programming languages APL and Algol68 ([Mar02], p.1015).

Overloading, emphasizing the procedural paradigm, provides different function implementations sharing the same name, but with different operand types.

Coercion (casting) serves to automatically transfer one type of object into another, enabling black-box adaptation by wrapping.

4.3.4 Defaults

In all these four polymorphism techniques defaults are a mechanism for explicitly providing one variation point for optional code parts. Defaults are a means to express commonality, for example by Null Objects [Woo98], default function arguments or default parameterized types.

Null objects offer a default behaviour for hook methods, often as an empty operation (In the casting mechanism, the wrapped object can be implemented as a Null Object.). They can either share the same base class as the template method, or they are in a separate inherited class. In the former case, like for default function arguments or default parameterized types, a means of cancelling the commonality by providing a different entity than the default parameter or Default Object serves as a mechanism of negative variability.

It is interesting to note that Coplien recently established a connection between these polymorphism variability mechanisms and symmetry:

Programming languages are good at expressing symmetries such as subtyping, overloading and parameterization. They fail to capture the full system complexity required by situations that demand inheritance cancellation, a dynamic form of overloading called dynamic multiple dispatch, or specialized instances of parameterized types. These latter problems are most often handled with patterns; for example, Bridge to handle inheritance with cancellation and Visitor to handle dynamic multiple dispatch. But some language constructs exist to handle this broken symmetry, like template specialization in C++ [Cop00].

4.4 Refinements

Recently emerging approaches combine several of the aforementioned polymorphism techniques with nesting techniques to implement more scalable generic components. In these post-object-oriented approaches, just fragments of classes are encapsulated as components, instead of entire classes. They explicitly leave the variation points open which e.g. the aforementioned defaults close. [SB02] call these emerging mechanisms refinements[1]. They all represent a functionality addition to a program which introduces a new feature, affecting multiple implementation entities simultaneously. As presented in figure 13, two broad categories of refinement mechanisms exist: collaboration-based and aspect-oriented mechanisms.

Figure 13:
Less-known variability mechanisms

- Frame Technology
- Refinements
  - Collaboration-based
    - GenVoca
    - Role-based
  - Aspect-Oriented
    - Aspect-Oriented Programming
    - Multi-Dimensional Separation of Concerns
    - Composition Filters

4.4.1 Collaboration-Based Mechanisms

Collaboration- and role-based design and implementation approaches were first developed in the 90s ([CB89], [Ree96], [VH97]) and have recently gained increased attention in the area of product lines and object-oriented frameworks ([BCS99], [Sma99], [SB02], [Rie00]). Whereas conventional object-oriented methods primarily view objects as instantiations of classes, role modeling emphasizes collaborations of objects in which the objects play certain roles. Classes are synthesized by composing the roles their instances play in different collaborations.

1 Note that Coplien also uses the term “refinement” in a similar context: “Perfect reuse is perfectly symmetric, so you need parameterization and refinement instead of simple cloning” [Cop01]. In connection with generators, [CE00] define refinements as compiler transformations with a vertical component, i.e. a transformation of a higher-level representation into a lower-level one with preserving the modularity of the higher-level one.

A collaboration is a view of an object-oriented design from the perspective of a single concern, service, or feature. Collaborations define the building blocks for application families ([SB02], p.4).

In an early implementation approach, each class was mapped to one role [VH97]. Roles were extended by parameterized inheritance, a mechanism combining subtype and parametric polymorphism. Inter-role relationships were also represented by parametric polymorphism. The class representing the role was thus incomplete and the collaboration could be instantiated at compile-time by configuring all the role classes. The drawback of this approach is that manually (re)configuring a larger collaboration can become difficult.

A more scalable mechanism for implementing collaboration-based designs is based on the GenVoca model [BG97]. As in the aforementioned approach, classes are refined by parameterized inheritance, but their interrelationships are more fixed. On the other hand, several interrelating classes share a common layer, which represents their collaboration. These so-called mixin layers are stacked via parameterized inheritance as well.

Figure 14:
Typical Collaboration-Based Design

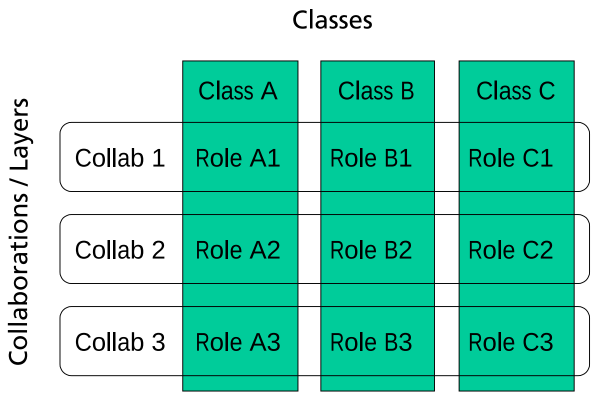
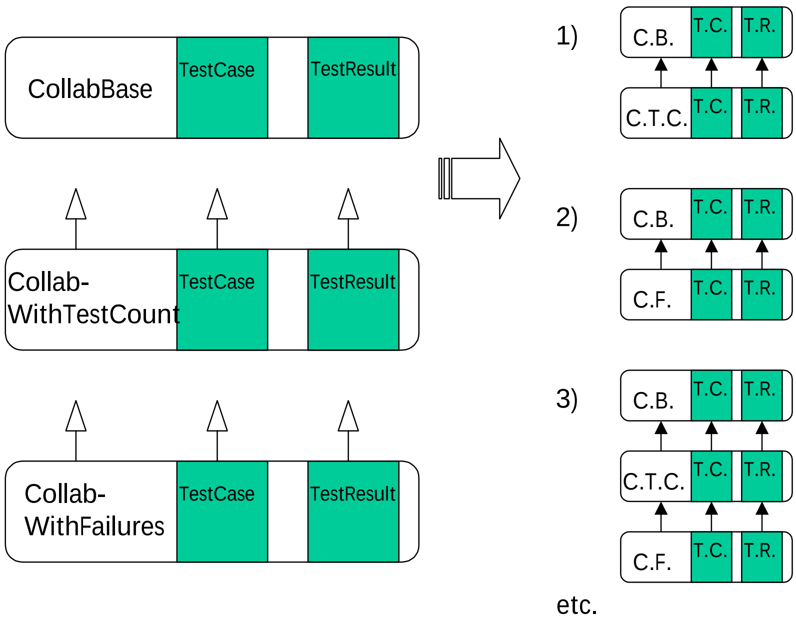


Figure 14 shows a refinement consisting of 3 classes and 3 collaborations. Within the collaboration layers, each class represents a role, which is only a partial implementation of the entire class that is created by stacking the collaborations. Note that not all of the roles have to be present; for example if class C does not participate in collaboration 2, role C2 can be omitted.
The layers can be stacked in quasi-arbitrary ways, and even the same collaboration might appear in different layers, so that a large variety of generic component implementations can be produced. Figure 15 presents an example of a collaboration-based design consisting of 3 collaborations, and 3 simple products which can be obtained by stacking the collaborations in different combinations.

The mixin-layer approach is more scalable than the simple role-based one because fewer parameters have to be passed in order to configure the collabo-

Figure 15:
Example of Collaboration-Based Design



rations; the interrelationships do not have to be configured explicitly. Moreover, roles can themselves be implemented as stacks of mixin layers, so that components in different scales of granularity can be implemented using the same technique.

As an alternative to this inheritance-based implementation, a forwarding based one is presented in ([CE00], pp.377-379).

An interesting point is that many characteristics of the collaboration-based techniques with their local symmetries (e.g. inheritance is possible in both directions, generic up- and down-calls) and self-similar, fractal-like properties (different levels of scale, repetition) play a major role in the recent work of Christopher Alexander ([Ale02],[Gab96b], [Gab98], [Cop01]).

4.4.2 Aspect-Orientation

A second category of refinements comprises aspect-orientation, with several technologies like Aspect-Oriented Programming (AOP), Multi-Dimensional Separation of Concerns (MDSOC), Demeter/AP ([Lie96], [LOO01]) or Composition Filters [BA01].

Aspect-orientation is a separation of concerns technology in which the various crosscutting concerns (properties or areas of interest) of a system and partially their relationships are implemented and then composed automatically, often by an AOP tool.

Concerns can be classified into high-level vs. low-level, functional vs. non-functional or development vs. production. Examples for functional concerns are data concerns, feature concerns, business rule concerns or variant concerns. Non-functional concerns comprise synchronization, distribution, error handling or transaction management. Among development concerns we find debugging, tracing, testing or profiling; product concerns include user-visible functionality.

In the original AOP technique [KLM+97], these properties crosscutting several functional components are captured in modular units called aspects. In order to instantiate the complete system these are combined by the Aspect Weaver, which is a programming-language-dependent tool. [KHH+01] give a more detailed introduction into AOP.

In Multi-Dimensional Separation of Concerns, aspects do not only augment classes but other aspects as well. In addition, it provides for adapting existing systems. Its tools allow a developer to compose a collection of separate models, called hyperslices, each encapsulating a concern by defining and implementing a (partial) class hierarchy appropriate for that concern. [OT01] introduce MDSOC in detail.

Aspect-oriented programming, however, is not restricted to a specific tool; it can just as well be applied using common programming languages [CME01a-c].

4.5 Other Mechanisms

Other variability mechanisms are extension as expressed in the Extension Object [Gam98] or Decorator [GHJV95] design patterns, Active Libraries ([Sim95], [IP], [VG98]), reflection and domain-specific languages.

5 Implementation Process

This section covers a process for introducing programming language mechanisms for implementing product lines, as those mentioned in the previous chapter. Bosch identifies two relevant dimensions with respect to initiating a product-line approach ([Bos00], p.166ff): First, either an evolutionary or revolutionary approach could be applied, and second, a set of existing systems could be used to introduce a product line or a new product family can be created from scratch. Both dimensions should have also been dealt with in earlier phases, e.g. in scoping.

Experience in introducing new software implementation technology has often favoured an evolutionary, iterative and incremental approach (Piecemeal Growth [Gab96], [FY00], [Cop99a]). An example of this process can be found in the Evolving Frameworks pattern language [RJ98], which describes an incremental process for creating frameworks, starting with choosing examples from existing applications, then creating a white-box framework, evolving it into a black-box framework and finally providing for (visual) configurability.

With respect to programming languages, Gabriel notes:

Languages are accepted and evolve by a social process, not a technical and technological one. [...]
[For a language to flourish], it must be similar to existing popular languages. The acceptance process is social because it relies in the actions of people in response to a language. [...] The acceptance process is not unique to languages-it works well and sometimes better with other things like application programs and operating systems ([Gab96], pp.112f.).

Later he observes that

For products in the real world, you have already had the decades, centuries, and millennia to go through this iterative process, and the software guys have had 50 years total [Gab01].

He adds that the most successful software applications combined the familiar with the new.

Before deciding to apply a specific mechanism, the currently used programming language(s) should be evaluated. The reason for this is that there has already been gained experience in using it, so that its fitness for use can be assessed. Additionally, in an incremental process only a minimal set of technologies should be changed at the same time. Thus changing the programming language beforehand should only be considered if the existing one would seriously hamper the introduction of product line mechanisms. Furthermore, the focus should be on just one single programming language, as a mixed programming environment would represent another obstacle.

Domain Design should have revealed the important commonalities and especially the variation points of the product line. Variations should as much as possible have been expressed in terms of positive variability; negative variabilities should have been refactored into positive ones. Examples of this are presented in ([Cop99], pp.146-148). If possible, the kind of variability (optional, XOR, OR etc.) should be determined. Additionally, the product line features should be ordered according to their priority and, as mentioned before, if their binding time has to be run-time this should be recorded explicitly.

As a second step, it should be evaluated which traditional mechanisms of positive variability the language offers, especially polymorphism, and if these impose an unacceptable penalty on the quality attributes at hand like performance, understandability or maintainability. If this is the case, an alternative traditional positive variability mechanism should be considered, or, if no other exists, a traditional small negative variability mechanism like a macro could be used. The mechanisms presented in section 4 can serve as a starting point.

The summary in Table 6 indicates that all of the mechanisms provide some support for the implementation of product lines, but support of different quality. Each mechanism must be seen in the tension of two dimensions: coverage of variability types and complexity. The first dimension characterizes the quality of a mechanism in implementing variabilities, the second one the suitability for being accepted in practice. Both dimensions are important while planning the systematic introduction of variability mechanisms into a running organization.

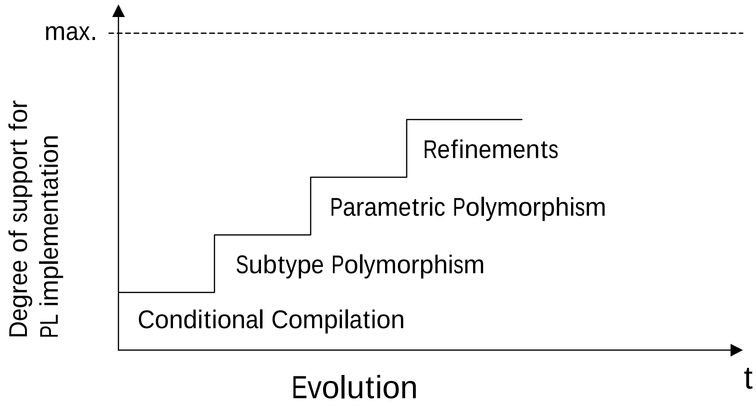
Note that the list of mechanisms mentioned is language-independent and incomplete; other mechanisms (e.g. those offered by language-specific idioms) may have to be added or inappropriate ones be removed in the specific context.

Table 6:
Advantages and disadvantages of language mechanisms

	Advantages	Disadvantages
Conditional Compilation	well-known, no space/ performance loss fine-grained	no direct language support, unscalable
Subtype Polymorphism	often well-known, dynamic	performance loss, OR-variability difficult to express
Parametric Polymorphism	no performance loss, all 3 kinds of variability expressible, built-in	less known, less supported
Ad-Hoc Polymorphism	often well-known	less important than universal polymorphism
Collaborations	good support for PL Implementation	generally not well-known, might require tools
AOP	good support for PL Implementation	often requires tools, market caution
Frame Processor	good support for PL Implementation	not well-known, additional tool

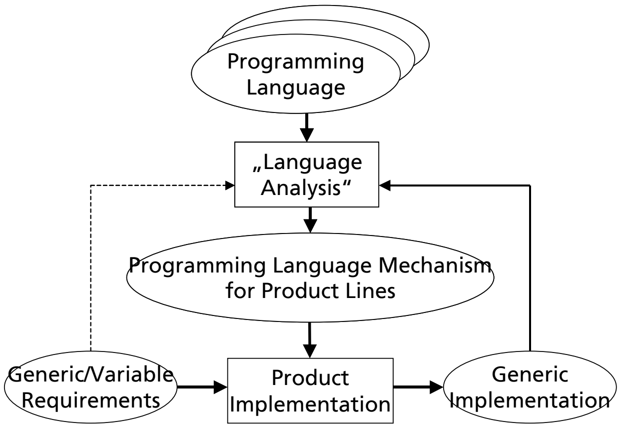
Figure 16 ranks the variability mechanisms from the previous section with respect to these two dimensions. During a technology transfer project both coverage and complexity should be gradually increased, so that the mechanisms would be introduced from left to right, or respectively from bottom to top.

Figure 16:
Incremental support for product line implementation



Taking a suitable language mechanism and the generic requirements, the product implementation finally leads to the generic implementation. Figure 17 depicts this process.

Figure 17:
Process for introducing programming language mechanisms for product lines



In the next section, a concrete example component is implemented using the variability mechanisms presented, as offered by two of the most commonly used general-purpose programming languages at the time of this writing: C++ and Java.

6 Case Study: TestComponent

6.1 Abstract Component

An abstract component is a technology-independent description of a logic unit of a software system. It consists of a specification and realization both encompassing several (UML) models. The specification hides the internals of a component but describes the offered services and the externally visible behavior, as well as the relevant concepts and entities. Figure 18 captures the specification class diagram of a test component TestSuite, which acquires a component to be tested and offers a single service that runs the test cases the test suite consists of. The service is specified in table 7.

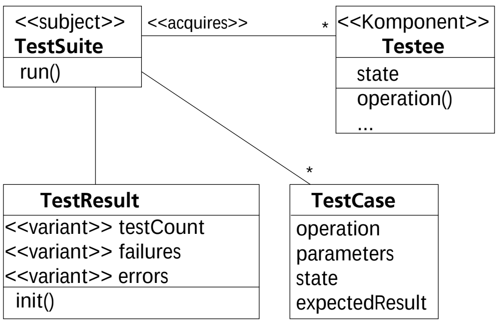


Figure 18: Specification class diagram of a generic test component

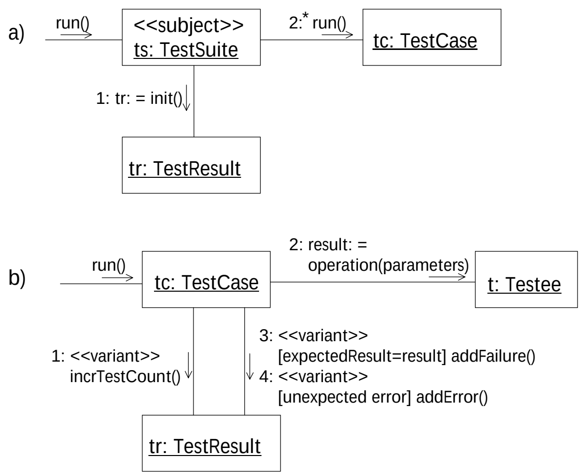
Table 7: Specification of the operation TestSuite::run()

Name	TestSuite.run()
Description	Runs all its test cases and stores their results
Sends	testee.<tc.operation>(<tc.parameters>) for all test cases tc
Changes	testresult:TestResult
Result	All test cases have been run (i.e., Testees' operations with defined parameters as defined by the test cases have been invoked) and their results have been stored in testresult

The externally visible behavior of a TestSuite component is omitted because it consists only of two trivial states: either the TestResult is valid or it is not.
As indicated by the usage of the stereotype <<variant>>, concrete TestSuite instances may produce differently detailed results. That is, TestSuite is a generic component and on the concrete characteristics of a particular instance is only decided when it is (re-)used according to the required test result information.

Running a test suite means first initializing the TestResult and then sequentially running all of the test cases and updating the TestResult according to their outputs. This algorithm is captured in the collaboration diagrams of TestSuite's realization (see figure 19).

Figure 19:
Collaboration diagrams for TestSuite.run() (a) and TestCase.run() (b)



According to the collaboration diagrams, the specification class diagram is refined by adding operations to TestResult and TestCase. The realization class diagram is depicted in figure 20. The run() operation of TestCase is specified in table 8.

Figure 20:
Generic realization class diagram of a test suite

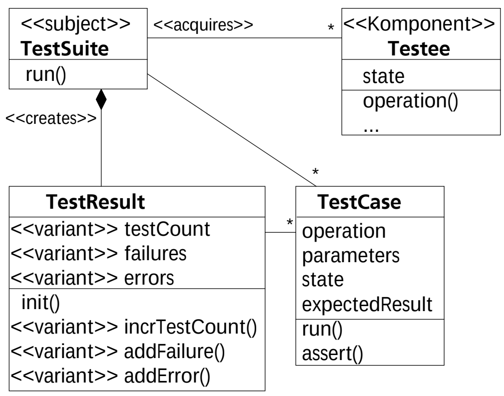


Table 8:
Specification of the TestCase.run() operation

Name	TestCase.run()
Description	Invokes a testee's operation with the defined parameters, analyzes operation output wrt. expected result, and stores test information in testresult
Sends	testee.operation(parameters)
Changes	testresult:TestResult
Assumes	testee is set up correctly and testresult exists
Result	<variant> testresult.TestCount has been incremented, </variant> testee.operation(parameters) has been invoked the operation's result or the Testee's state have been retrieved and compared with expectedResult <variant> The anticipated (failure) or unanticipated problems in testresult have been updated </variant>

6.2 Component Implementation in C++

Consider the variant attributes of the TestResult produced by a TestSuite component instance. The simplest way to implement these instances with different capabilities is to create separate isolated implementations as shown in the following C++ code:

Listing 1:
Naive test component Implementation

```
class TestResultWithTestCount
{ // stores the number of tests run
public:
  void incrementTestCount() { ... }
};

class TestCaseWithTestCount
{ // counts the number of tests run
public:
  void run(TestResultWithTestCount& tr) {
    tr.incrementTestCount();
    // perform test
  }
};

class TestResultWithFailures
{ // stores the detected failures
public:
  void addFailure() { ... }
};

class TestCaseWithFailures
{ // considers failures
public:
  void run(TestResultWithFailures& tr) {
    // perform test
    if(failureDetected)
    tr.addFailure();
  }
};

void main(int,char**) {
  TestCaseWithTestCount tc;
  TestResultWithTestCount trc;
  tc.run(trc);

  TestCaseWithFailures tf;
  TestResultWithFailures trf;
  tf.run(trf);
}
```

This way of implementing the generic TestSuite component is simple but not scalable because it results in a combinatorial number of classes with no reuse at all. In order to prevent this, the variabilities of a component must be controlled more systematically also at the implementation level.

C++ is a hybrid language and supports the procedural paradigm of its ancestor C as well as the object-oriented or parametric one. Recent libraries like FC++ ([FC++],[MNS00a]) even enable functional programming in C++. As a multi-paradigm programming language [Cop99], C++ offers multiple ways of organizing system abstractions around properties of commonality and variation. The following subsections illustrate how the test component can be implemented in C++ using the some of the variability mechanisms introduced in section 4.

6.2.1 Conditional Compilation

Conditional compilation is directly supported by the C and C++ preprocessor in the form of #ifdef or #if directives; this has been one of the most widely used techniques for implementing product lines. As table 5 showed, it is mainly used to express optional variabilities (using #ifdef), and less often XOR-alternative ones (with #if defined() #elif if defined()).

The C++ implementation of the test component with conditional compilation is shown in listing 2.

Listing 2:
Test component implementation using conditional compilation

```
class TestResultCC
{ // TestResult implemented with conditional compilation
public:
#ifdef WITH_TEST_COUNT
  void incrementTestCount() { ... }
#endif
#ifdef WITH_FAILURES
  void addFailure()         { ... }
#endif
};

class TestCaseCC
{ // TestCase implemented with conditional compilation
public:
  void run(TestResultCC& tr) {
#ifdef WITH_TEST_COUNT
    tr.incrementTestCount();
#endif
    // perform test
#ifdef WITH_FAILURES
    if(failureDatected)
    tr.addFailure();
#endif
  }
};
```

Listing 2:
(continued)

```
void main(int,char**) {
#define WITH_TEST_COUNT
  TestCaseCC tc;
  TestResultCC trc;
  tc.run(trc);
#undef WITH_TEST_COUNT
#define WITH_FAILURES
  TestCaseCC tf;
  TestResultCC trf;
  tf.run(trf);
#undef WITH_FAILURES
}
```

Compared to the naive implementation (listing 1), the number of classes has decreased. However, the class internals (both the interface, as in TestResultCC, and the implementation, as in TestCaseCC::run()) are cluttered with macros. Moreover, for each new variant (e.g. storing errors) the internals of potentially all classes have to be changed manually. This clearly violates the Open-Closed-Principle [Mar96a], leading to maintenance problems.

6.2.2 Subtype Polymorphism

C++ can express all the four aforementioned kinds of polymorphism directly. The object-oriented part of C++ allows us to express subtype polymorphism. An interface in C++ is implemented as an abstract class, which is a class that neither contains or inherits from classes that contain attributes, non-virtual functions, or any private or protected members. All of its member functions other than the destructor including inherited functions are declared pure virtual and left undefined [Lak96].
A test component implementation using subtype polymorphism (unification principle) is sketched in listing 3.

Listing 3:
Test component implementation using subtype polymorphism

```
class TestResultSP
{ // interface for TestResults implemented with subtype polymorphism
public:
  virtual void incrementTestCount() {}
  virtual void addFailure()         {}
};

class TestResultSPWithTestCount:public TestResultSP
{ // stores the number of tests run
public:
  void incrementTestCount() { ... }
};

class TestCaseSP
{ // TestCase using subtype polymorphism
public:
  virtual void run(TestResultSP& tr) {
    tr.incrementTestCount();
    // perform test
    if(failureDetected)
    tr.addFailure();
  }
};

void main() {
  TestCaseSP tc;
  TestResultSPWithTestCount trc;
  tc.run(trc);

  TestResultSPWithFailures trf;
  tc.run(trf);
}
```

Compared to the previous implementation (listing 2), the number of classes has slightly increased, but is still lower than for the naive one (listing 1). In case of introducing a new variant for storing errors, not all classes have to be changed, and where change is necessary, it is more localized (e.g. for handling errors, the interface of TestResultSP has to be extended, as well as TestCaseCC's implementation). However, implementing OR-variabilities, e.g. handling both test counts and failures, might lead to code duplication (unless multiple inheritance is used). Moreover, there will be a proliferation of classes as the number of alternatives grows, as mentioned in section 4.3.1.

6.2.3 Parametric Polymorphism

In addition to the run-time variability mechanism of subtype polymorphism, the compile-time mechanism of parametric polymorphism is supported by C++ via templates. An early template implementation which popularized generic programming in C++ was the Standard Template Library (STL) [MS96], which has meanwhile become part of the Standard C++ Library. Other well-known recent generic libraries include Blitz++ [Blitz], Loki [Ale01], Boost [Boost], FC++ [FC++], the Lambda Library [Lambda], MTL [MTL] and Spirit [Spirit].

This is how the test component can be implemented in C++ using parametric polymorphism:

Listing 4: Test component implementation using parametric polymorphism

```
class TestResultPP{ // static interface for TestResults
public:
  void incrementTestCount() {}
  void addFailure()         {}
};

class TestResultPPWithTestCount:public TestResultPP
{ // stores the number of tests run
public:
  void incrementTestCount() { ... }
};

class TestResultPPWithFailures:public TestResultPP
{ // stores the detected failures
public:
  void addFailure() { ... }
};

template<class TESTRESULT>
class TestCasePP
{ // TestCase using parametric polymorphism
public:
  void run(TESTRESULT& tr) {
    tr.incrementTestCount();
    // perform test
    if(failureDetected)
    tr.addFailure();
  }
};

void main()
{
  TestCasePP<TestResultPPWithTestCount> tc;
  TestResultPPWithTestCount trc;
  tc.run(trc);

  TestCasePP<TestResultPPWithFailures> tf;
  TestResultPPWithFailures trf;
  tc.run(trf);
}
```

In this approach, the number of classes can be kept to a minimum, and, as the TestResult classes are parameterized as well (not shown here; idioms for configuring this are e.g. traits [Mye95] or type lists [Ale01],[CE00]), the class proliferation can be avoided.

6.2.4 Collaboration-Based Mechanisms

The mixin technique, in which a class is parameterized by its subclass, has been applied to implement role-based [VH97] and collaboration-based [Sma99] designs in C++ (see section 4.4.1). In the latter technique, mixin-layers are implemented with classes which can be stacked by parameterized inheritance. At the same time, these classes nest other classes which form a collaboration. So by nesting several classes (roles) are embraced in a collaboration and in conjunction with templates these are parameterized in a uniform way. Figure 13 illustrates the mixin-based implementation technique for TestCase and TestResult, listing 5 shows an implementation.

Listing 5:
Test component implementation using mixin layers

```
class CollabBase
{ // static interface for TestResult and TestCase
public:
  class TestResult {};
  class TestCase {
  public:
    void run(TestResult&) {}
  };
};

template<class SUBCOLLAB=CollabBase>
class CollabWithTestCount:public SUBCOLLAB
{ // TestResult/TestCase parts only concerned with counting tests
public:
  class TestResult:public SUBCOLLAB::TestResult {
  public:
    void incrementTestCount() { ... }
   };
  class TestCase:public SUBCOLLAB::TestCase {
  typedef typename SUBCOLLAB::TestCase Base;
  public:
    void run(TestResult& tr) {
      tr.incrementTestCount();
      Base::run(tr);
    }
  };
};
```

Listing 5:
(continued)

```
template<class SUBCOLLAB=CollabBase>
class CollabWithFailures:public SUBCOLLAB
{ // TestResult/TestCase parts only concerned with handling failures
public:
  class TestResult:public SUBCOLLAB::TestResult {
  public:
    void addFailure() { ... }
  };
  class TestCase:public SUBCOLLAB::TestCase {
  typedef typename SUBCOLLAB::TestCase Base;
  public:
    void run(TestResult& tr) {
      Base::run(tr);
      if(failureDetected)
      tr.addFailure();
    }
  };
};

void main() {
  CollabWithTestCount<>::TestCase tc;
  CollabWithTestCount<>::TestResult trc;
  tc.run(trc);

  CollabWithFailures<>::TestCase tf;
  CollabWithFailures<>::TestResult trf;
  tf.run(trf);

  // both with test count and failures
  CollabWithFailures<CollabWithTestCount<> >::TestCase tcf;
  CollabWithFailures<CollabWithTestCount<> >::TestResult r;
  tcf.run(r);
}
```

This approach has the following advantages: single concerns like test counting and failure handling are encapsulated in a uniform way, the elements of the predominant composition (TestCase and TestResult) are preserved, OR-variations can be implemented easily, and new concerns (e.g. error handling) can be introduced without changing existing code.

6.2.5 Aspect-Orientation

As in the mixin technique, by combining the polymorphism techniques with nesting, even aspect-oriented programming can be performed in C++ without additional tools (cf. [CME01a-c]). Note that in this case namespaces, not classes, are used to embrace several elements (in this case aspects). Because namespaces can be nested, they can also be used to implement components in different levels of scale.

An aspect-oriented implementation of the test component is shown in listing 6.

Listing 6:
Aspect-oriented implementation of the test component

```
namespace original {
  class TestResult {};
  class TestCase { public:
    void run(TestResult&) {
      // perform test
    }
  };
}

namespace aspects {
  typedef original::TestResult TestResult;
  typedef original::TestCase   TestCase;
  class TestResultWithTestCount:public TestResult {
    public:    void incrementTestCount() { ... }
  };
  class TestCaseWithTestCount:public TestCase {
  public:
    void run(TestResultWithTestCount& tr) {
      tr.incrementTestCount();
      TestCase::run(tr);
    }
  };
  class TestResultWithFailures:public TestResult {
  public:
    void addFailure() { ... }
  };
  class TestCaseWithFailures:public TestCase {
  public:
    void run(TestResultWithFailures& tr) {
      TestCase::run(tr);
      if(failureDetected)
      tr.addFailure();
    }
  };
}

void main() {
  aspects::TestCaseWithTestCount tc;
  aspects::TestResultWithTestCount trc;
  tc.run(trc);

  aspects::TestCaseWithFailures tf;
  aspects::TestResultWithFailures trf;
  tf.run(trf);
}
```

Listing 6:
(ontinued)

```
void main() {
  aspects::TestCaseWithTestCount tc;
  aspects::TestResultWithTestCount trc;
  tc.run(trc);

  aspects::TestCaseWithFailures tf;
  aspects::TestResultWithFailures trf;
  tf.run(trf);
}
```

The aspect-oriented and collaboration-based implementations and benefits are similar. However, the single concerns (test counting, failure handling) are not explicitly encapsulated, and aspect combinations (e.g. both test counting and failure handling) cannot be accomplished without code duplication.

6.3 Component Implementation in Java

Java was designed to be object-oriented from the ground up and removed many of the complexities of the language C++ [Fla96]. However, many of these “complexities” offer different alternative kinds of variability mechanisms. As a result, only a limited amount of variability can be expressed through Java’s built-in mechanisms. For expressing the others additional tools have to be used.

Lacking a preprocessor, Java does not offer the macro capability for statically including optional or alternative code parts. But at least two alternatives exist: Mimicking macros via public static final boolean constants or using a frame processor like fp [FP] or the XVCL-based one [XVCL].

At the time of this writing, Java provides only a limited variety of polymorphism mechanisms ([CE00], p.178).
As an object-oriented language, it can express run-time variabilities through subtype polymorphism in a similar way as C++ (see section 6.2.2).
Currently, Java does not yet support parametric polymorphism. Using subtype polymorphism instead leads to those performance and combinability problems mentioned in section 4.3.1. However, there are tools for adding parametric polymorphism are Generic Java [GJ] and the Jakarta Tool Suite [JTS].
In Java refinements cannot be implemented with built-in language mechanisms, but the following tools exist as language extensions:

Collaboration-based designs can be implemented with the Jakarta Tool Suite JTS [JTS]. A popular mature implementation tool for Aspect-Oriented Programming is AspectJ [AspectJ]. Other well-known AOP tools are HyperJ [HyperJ] or DemeterJ and DJ [Demeter]. Further tools can be found at [Aosd-Net].

7 Summary and Outlook

This report dealt with one aspect of implementation technologies for software product lines: the programming language view.

7.1 Summary

Software product line concepts were introduced and the need for implementation level support was motivated in order to introduce them in practice. After a survey of contemporary industry programming languages, product line implementation work products were characterized as generic component implementations. What distinguishes them from conventional component implementations is adaptability, which is handled by variability mechanisms; and a classification of these was presented.

Language mechanisms for implementing these optional, alternative or coexisting parts were presented in a language-independent way, ranging from the more traditional, well-known ones like conditional compilation, different kinds of polymorphism and combinations thereof, to less-know ones like collaboration-based mechanisms or aspect-orientation. The strengths and weaknesses of these mechanisms were evaluated, both with regard to complexity and level of support for product line implementation.

An incremental process for introducing these programming language mechanisms was suggested, followed by a test component case study which showed how the aforementioned variability mechanisms can be applied in two of the most common general-purpose programming languages at the time of this writing, C++ and Java.

7.2 Outlook

As this report only serves as an introduction to programming language product line implementation technologies, many of the topics mentioned here were not dealt with in full detail and should be refined.

For example, the classification of variability mechanisms could be extended, and the list of specific well-known and less well-known variability mechanisms could be completed. On the one hand, this could be done at the concrete level by combining mechanisms or focussing on software generators, promising reuse mechanisms like frame technology, or configurability [Som99]. On the other hand, an abstraction over the different means of variability implementations could be developed, for example based on their symmetries and symmetry-breaking characteristics.

Furthermore, the criteria for determining the advantages and disadvantages could be extended, and the process should be performed in practice and be refined.

The case studies could be extended both for the two presented languages and for others, for example functional or dynamically typed (scripting) languages.

8 References

[ABB+01] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel. Component-Based Product-Line Engineering with UML. Addison-Wesley, 2001

[AG01] M. Anastasopoulos, C. Gacek. Implementing Product Line Variabilities. Proceedings of the 2001 Symposium on Software Reusability (SSR'01), Toronto, Canada, 2001

[Ale01] A. Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001

[Ale02] C. Alexander. The Nature of Order. Book One: The Phenomenon of Life. Oxford University Press, 2002

[AM02] C. Atkinson, D. Muthig. Enhancing Component Reusability through Product Line Technology, to appear

[Aosd-Net] Aspect-Oriented Software Development: http://aosd.net

[AspectJ] AspectJ: www.aspectj.org

[BA01] L. Bergmans, M. Askit. Composing Crosscutting Concerns Using Composition Filters. Communications of the ACM, October 2001

[Bas96] P. G. Basset:. Framing Software Reuse: Lessons from the Real World. Prentice-Hall, 1996

[BCS99] D. Batory, R. Cardone, Y. Smaragdakis. Object-Oriented Frameworks and Product-Lines. 1st Software Product Line Conference, 1999

[BFK+99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In Proceedings of the Symposium on Software Reuse, 1999

[BG97] D. Batory, B. J. Gerraci. Composition Validation and Subjectivity in GenVoca Generators. IEEE Transactions on Software Engineering, February 1997

[Blitz] Blitz++ Library: www.oonumerics.org.blitz

[Boost] Boost Library: www.boost.org

[Bos00] J. Bosch. Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, 2000

[Bro] Brockhaus - Die Enzyklopädie in 24 Bänden, 20. Auflage, F.A.Brockhaus 1996-99

[Byte] byte.com/documents/s=1271/byt20010831s001

[CB89] W. Cunningham, K. Beck. Construction Abstractions for Object-Oriented Applications. Journal of Object-Oriented Programming, July 1989

[CE00] K. Czarnecki, U. W. Eisenecker. Generative Programming: Methods, Tools and Applications. Addison-Wesley, 2000

[Cetus] Cetus Links: www.cetus-links.org/top_languages.html

[CME01a-c] K. Czarnecki, L. Dominick, U. W. Eisenecker. Aspektorientierte Programmierung in C++. iX Magazin, 8-10/2001

[Cop95] J. O. Coplien. Curiously Recurring Template Patterns. C++ Report, February 1995

[Cop99] J. O. Coplien. Multi-Paradigm Design for C++. Addison-Wesley, 1999

[Cop99a] J. O. Coplien. Reevaluating the Architectual Metaphor: Toward Piecemeal Growth. IEEE Software Sept./Oct.1999

[Cop00] J. O. Coplien. Pattern Languages. Maturing Beyond the Design Patterns. Slide Notes for Presentation at COMPSAC 2000

[Cop01] J. O. Coplien. The Future of Language: Symmetry or Broken Symmetry. Slide Notes forKeynote at VSLive! 2001 Conference

[Demeter] Demeter: Aspect-Oriented Development: www.cs.neu.edu/research/demeter

[DGD01] A. Duret-Lutz, T. Geraud, A. Demaille. Design Patterns for Generic Programming in C++. In Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), 2001

[Dmoz] Dmoz Open Directory Project: dmoz.org/Computers/Programming/Languages

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999

[FC++] FC++ Library: www.cc.gatech.edu/~yannis/fc++

[Fla96] D. Flanagan. Java in a Nutshell, 2nd Edition. O'Reilly, 1996

[FP] Simple frameprocessor fp: frameprocessor.sourceforge.net

[FPR01] M. Fontoura, W. Pree, B. Rumpe. The UML-F Profile for Framework Architectures. Addison-Wesley, 2001

[FY00] B. Foote, J. Yoder. Big Ball of Mud. In N. Harrison, B. Foote, H. Rohnert (eds.). Pattern Languages of Program Design 4. Addison-Wesley, 2000. Available at www.laputan.org/mud

[Gab96] R. P. Gabriel. Patterns of Software: Tales from the Software Community. Oxford University Press, 1996

[Gab96a] R. P. Gabriel. Christopher Alexander: The Search for Beauty. Keynote at the Current Object Practice and Experience (COPE) Conference, 1996. Available at www.dreamsongs.com/NewFiles/AlexanderPresentation.pdf

[Gab98] R. P. Gabriel. The Nature of Order: The Post-Pattern World. Presentation at Brown University, 1998. Available at www.dreamsongs.com/NewFiles/NatureOfOrder.pdf

[Gab01] R. P. Gabriel. Lessons From the Science of Nothing At All. Keynote at the Product Development and Management Assiciation Conference (PDMA), October 2001. Available at www. dreamsongs.com/LessonsFromNothing.html

[Gam98] E. Gamma. Extension Object. In R. C. Martin, D. Riehle, F. Buschmann (eds.). Pattern Language of Program Design 3. Addison-Wesley, 1998

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[GJ] GJ - A Generic Java Language Extension: www.research.avayalabs.com/user/wadler/pizza/gj

[Google] Google Directory: directory.google.com/Top/Computers/Programming/Languages

[Gulp] gulp.de/kb/mk/chanpos/javagegenc.html

[HyperJ] HyperJ: Multi-Dimensional Separation Of Concerns in Java: www.reaearch.ibm.com/hyperspace/HyperJ/HyperJ.htm

[Infoworld] infoworld.com/articles/hn/xml/01/10/24/011024hnitspend.xml

[IP] Intentional Programming: www.omniscium.com/nerdy/ip

[iX00] Ix Magazin für Professionelle Informationstechnik, November 2000

[iX02] Ix Magazin für Professionelle Informationstechnik, November 2002

[Jac01] M. A Jackson. Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley, 2001

[JF88] R. E. Johnson, B. Foote. Designing Reusable Classes. Journal of Object-Oriented Programming, June/July 1988

[JTS] Jakarta Tool Suite (JTS): www.cs.utexas.edu/users/schwartz/software.html

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. Getting Started with AspectJ. Communications of the ACM, October 2001

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. V. Lopes, J.-M. Loingtier, J. Irwin. Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1997

[Lak96] J. Lakos. Large-Scale C++ Software Design. Addison-Wesley, 1996

[Lambda] Lambda Library: http://lambda.cs.utu.fi

[Lie96] K. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing, 1996

[LOO01] K. Lieberherr, D. Orleans, J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. Communications of the ACM, October 2001

[Mar96] R. C. Martin. Patterns for Dealing with Dual Inheritance Hierarchies. 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS), 1996

[Mar96a] R. C. Martin. The Open-Closed Principle. C++ Report, February 1996. Available at www.objectmentor.com/resources/articles.ocp.pdf

[Mar02] J. J. Marciniak (ed.). Encyclopedia of Software Engineering (2nd ed.). John Wiley & Sons, 2002

[MAL+02] D. Muthig et. al. Technology Dimensions of Product Line Implementation Approaches, IESE Technical Report, No. 051.02/E, 2002

[MNS00] B. McNamara, Y. Smaragdakis. Static Interfaces in C++. Generative and Component-Based Software Engineering (GCSE) Conference, 2000. Available at www.oonumerics.org/tmpw99/manamara.pdf

[MNS00a] B. McNamara, Y. Smaragdakis. Functional Programming in C++. Functional Programming in C++. International Conference on Functional Programming (ICFP) 2000

[MS96] D. R. Musser, A Saini. STL Tutorial and Reference Guide. Addison-Wesley, 1996

[MTL] Matrix Template Library: www.osl.iu.edu/reaearch/mtl

[Mut02] D. Muthig. A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines. PhD Thesis, University of Kaiserslautern, 2002

[Mye95] N. C. Myers. Traits: A New and Useful Template Technique. C++ Report, June 1995

[OT01] H. Ossher, P. Tarr. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. Communications of the ACM, October 2001

[Pre95] W. Pree. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995

[Ree96] T. Reenskaug. Working with Objects: The OOram Software Engineering Method. Manning Publications, 1996

[Rie00] D. Riehle. Framework Design: A Role Modeling Approach. PhD Thesis, ETH Zürich, 2000. Available at www.riehle.org/computer-science-research/dissertation

[RJ98] D. Roberts, R. Johnson. Patterns for Evolving Frameworks. In R. C. Martin, D. Riehle, F. Buschmann (eds.). Pattern Language of Program Design 3. Addison-Wesley, 1998

[SB02] Y Smaragdakis, D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. To appear in ACM Transactions on Software Engineering and Methodology

[Sim95] C. Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. Nato Science Committee Conference, 1995

[Sma99] Y. Smaragdakis. Implementing Large-Scale Object-Oriented Components. PhD Thesis, University of Texas, Austin, 1999

[Som99] P. Sommerlad. Configurability. 4th European Conference on Pattern Languages of Programming and Computing (EuroPLoP), 1999

[SourceForge]www.sourceforge.net

[Spirit] Spirit Parser Library: spirit.sourceforge.net

[Str94] B. Stroustrup. The Design and Evolution of C++. Addison-Wesley.1994

[SVGB00] M. Svahnberg, J. van Gurp, J. Bosch. A Taxonomy of Variability Realization Techniques. Available at segroup.cs.rug.nl/publications/variabilitytaxonomypaper.html

[TOH+99] P. Tarr, H. Ossher, W. Harrison, S. M. Sutton. N Degrees of Separation: Multi-Dimersional Separation of Concerns. In Proceedings of the 21st International Conference on Software Engineering (ICSE), 1999

[VG98] T. M. Veldhuizen, D. Gannon. Active Libraries: Rethinking the Role of Compilers and Libraries. In Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98), SIAM Press, 1998. Available at http://osl.iu.edu/~tveldhui/papers/oo98.ps

[VGBS01] G. van Gurp, J. Bosch, M. Svahnberg. On the Notion of Variability in Software Product Lines. In Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA), 2001. Available at segroup.rug.nl/publications/notionOfVariability.html

[VH97] M. VanHilst. Role-Oriented Programming for Software Evolution. PhD Thesis, University of Washington, Seattle, 1997

[Woo98] B. Woolf. The Null Object. In R. C. Martin, D. Riehle, F. Buschmann (eds.). Pattern Language of Program Design 3. Addison-Wesley, 1998

[XVCL] XVCL and its frame processor: www.comp.nus.edu.sg/labs/software/xvcl.html

Document Information

Title:	Product Line Implementation Technologies - Programming Language View
Date:	October 31, 2002
Report:	IESE-057.02/E
Status:	Final
Distribution:	Public