Fraunhofer Institut
Experimentelles
Software Engineering

# Product Line Implementation with Frame Technology: A Case Study


**Authors:**
Thomas Patzke
Dirk Muthig

Funded by Stiftung Rheinland-Pfalz für Innovation, Project #15202-386261/51: "Entwicklung und Erprobung eines Methodenleitfadens für Software-Produktlinien-Implementierungstechnologien"

IESE-Report No. 018.03/E
Version 1.0
March 18, 2003

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

Software development today has to meet various demands, such as reducing cost, effort, and time-to-market, increasing quality, handling complexity and product size, or satisfying the needs of individual customers. This is why many software organizations today do not just develop and maintain single, separate software systems, but a set of systems, a product line. Given the fact that the systems are typically in the same application domain, it would be inefficient to develop them independently of each other in an ad-hoc way, as this usually leads to high development and maintenance effort.

However, a means for systematically developing and maintaining similar products is offered by software product line engineering, as covered by the PuLSE™ approach[1] developed at the Fraunhofer IESE. To implement such a product line approach in practice, special technologies are required for effectively identifying reusable artifacts, as well as capturing and controlling their commonalities and variabilities. To do the latter efficiently, the PoLITe project (Product Line Implementation Technologies) explores techniques at the implementation level for managing variability.

Frame technology is one of these implementation techniques that, as this report shows, offers advantages for implementing product lines over object-oriented techniques. This report highlights reuse principles and explains the major concepts underlying frame technology. The core concepts are extracted and realized in a plain frame processor. This tool will be used in the following case study, where the evolution of a software system from a single system into a product line will be discussed, each time comparing the frame approach with a conventional object-oriented solution.

**Keywords:** software product lines, product line implementation technologies, generators, frame technology

1 PuLSE is a registered trademark of Fraunhofer IESE

# Table of Contents

# 1 Introduction

Software development today has to meet various demands, such as reducing cost, effort, and time-to-market, increasing quality, handling complexity and product size, or satisfying the needs of individual customers. This is why many software organizations today do not just develop and maintain single, separate software systems, but a set of systems, a product line. Given the fact that the systems are typically in the same application domain, it would be inefficient to develop them independently of each other in an ad-hoc way, as this usually leads to high development and maintenance effort.

However, a means for systematically developing and maintaining similar products is offered by software product line engineering, as covered by the PuLSE™ approach[1] developed at the Fraunhofer IESE. To implement such a product line approach in practice, special technologies are required for effectively identifying reusable artifacts, as well as capturing and controlling their commonalities and variabilities. To do the latter efficiently, the PoLITe project (Product Line Implementation Technologies) investigates techniques at the implementation level for managing variability. The PoLITe project is presented in Section 1.1.

Programming languages are one technique applied to implementing. Section 1.2 maps the fundamental product line concepts such as variation points and variability types to them.

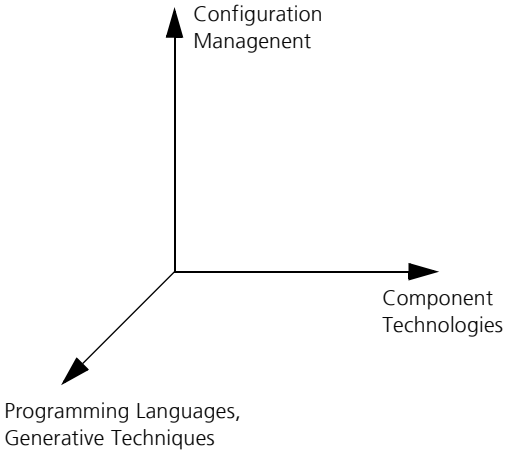
One product line implementation technique coming with its own special set of mechanisms is frame technology. Its underlying reuse principles are presented in Section 1.3. Section 1.4 outlines the remainder of this report, which introduces and investigates product line implementation with frame technology in detail.

## 1.1 The PoLITe Project

The PoLITe project is organized according to three dimensions of product line implementation technologies, as shown in Fig. 1: Component technologies, configuration management and generative features of programming languages including generators. Each dimensions is investigated separately in detail.

1 PuLSE is a registered trademark of Fraunhofer IESE

Figure 1:
Product Line Implementation Technology Dimensions

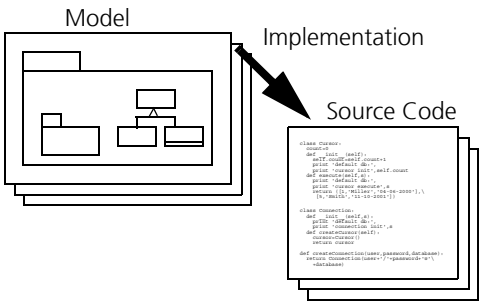



The first report in the programming language dimension [PM02] presents an overview of currently available implementation techniques. The subsequent section summarizes this before we focus on frame technology as an emerging generative technique for developing and maintaining software product lines.

## 1.2 Product Line Implementation: Programming Language Dimension

In a traditional top-down software development process, a single system is implemented by transferring a single model into a single block of source code. This is shown in Fig. 2. This process must be performed again for each single system variant, as indicated by the third dimension in the figure.

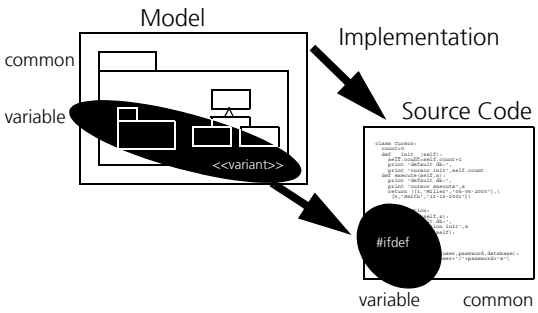
Figure 2:
Implementing single systems




To limit the growing number of individual implementations, in case of a product line the set of similar programs should be developed and maintained together. Similarity means that all systems have much in common (commonalities) but also significant differences. Fig. 3 illustrates the different issues arising when developing a software product line: the model consists of elements which are common for all products, and variable elements which are unique to a specific

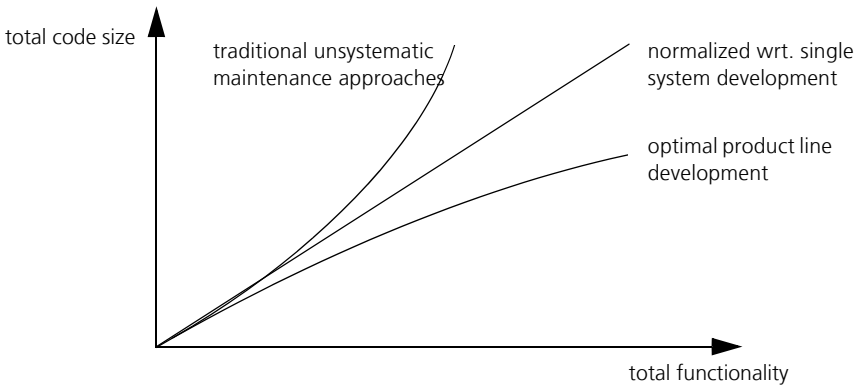
product. As in the single system development process, implementation means transferring the model into source code, so that in the product line approach the common and variable parts of the model are reflected in common and variable parts of the source code. Note that in contrast to the single-system case, the whole process does not have to be repeated anew for each non-trivial system variant. Instead, the common part is created only once for the entire product line, and only the variable parts are altered to obtain a specific product.

Figure 3:
Implementing Product Lines




As a result, the size of the code base during development and maintenance is typically smaller for a product line approach then for a single system development approach offering the same functionality [MS02]. This is illustrated in Fig. 4.

Figure 4:
Code size vs. covered functionality




Maintaining traditional systems typically leads to a code base that is larger than necessary because added functionality is not fully integrated into the existing (similar) code base, or removed functionality is not completely deleted from the code base. Such a development is avoided by a product line approach where the common parts are created only once for the whole set of products, and only the variable parts are modified for each individual system, which results in a lower code base.

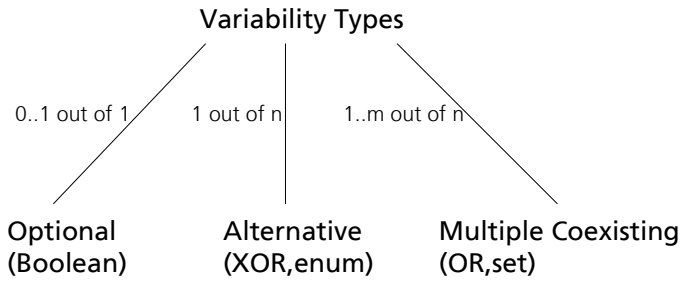
There are different ways to realize such a modification.

On the one hand, the modification can be accomplished by selecting from a set of plug-compatible components with alternative and fixed functionality. In the PoLITe project, this aspect is covered in the component technology dimension (see Fig. 1). The drawback of this approach is that it is often not flexible and scalable enough when not the exact capabilities, but only similar ones are needed.
On the other hand, individual variable parts can be obtained by providing a means to adapt a fixed default functionality, which offers significant reusability advantages that will be described in Section 1.3.

As indicated in Fig. 4, variability represents gaps within the common parts, at both the model and the source code level, which have to be filled to obtain a specific product. Three basic kinds of these gaps, often referred to as variation points or hot spots, can be distinguished (see Fig. 5):

In the simplest case, only a single possibility exists which can either be included or excluded. This represents an optional variability or a boolean variability point. An example of this often encountered in practice is that a debugging facility should be provided during development but excluded in production code.
We speak of an alternative variability or an "exclusive or" (XOR) variation point when exactly one out of n possibilities has to be selected. A common example of this is when an individual member of a software product line should support exactly one out of several kinds of hardware, operating system, or database.
When several possibilities can be selected simultaneously, we call this multiple coexisting possibilities or an OR variation point. For example, the support for multiple file formats in a software system represents such an OR variation point.

Figure 5:
Basic kinds of variabilities




Behavioral variabilities like those mentioned above can be expressed by a number of programming language mechanisms. Table 1 gives an overview of these language mechanisms and lists their individual advantages and disadvantages. In [PM02] the advantages and disadvantages of each of these major variability mechanisms are discussed in detail.

Table 1: 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 product line implementation | generally not well-known, might require tools |
| Aspect-Orientation | good support for product line implementation | often requires tools, market caution |
| Frame Technology | good support for product line implementation | not well-known, additional tool support required |

Conditional compilation is a simple variability mechanism in which macros are used to mark points of variation; the macros enclose optional sections of source code which, at compile time, is included into a specific product if the macro is defined, otherwise the code is excluded.

Frame Technology, developed by Paul Basset since the 80s as the result of his research in software reuse [Bas96], shares many characteristics of conditional compilation, but avoids its scalability drawbacks by ordering its physical units hierarchically.

A result of the report [PM02] is that frame technology offers good support for product line implementation. This report will demonstrate that product line implementation efforts can be facilitated with frame technology. The next section begins to illustrate this by introducing the general concepts of reuse underlying frame technology: use-as-is and adaptation.

## 1.3 Use vs. Reuse

When talking about software reuse it is helpful to go into more detail to distinguish reuse from use explicitly.

A widespread view is that software reuse means using a component of fixed software properties (data and functions) several times, like using identical copies of physical parts several times when building physical products. This "Lego block" approach is simple, and it is useful if the context in which the component is used never changes during the entire lifecycle of the product, as is usually the case in the physical world.

However, software requirements tend to change frequently during the lifecycle of a software system. In this context it is very likely that such fixed components of a software product cannot be used-as-is anymore. As traditional "Lego block" components are not general enough and cannot be adapted easily in each context, traditional reuse becomes ineffective.

As a simple example often encountered in practice, consider a software system A that initially uses the services of some fixed component B by invoking its function f(), as depicted in Fig. 6a. As the system A evolves into system A', it requires a similar functionality with an additional parameter, f(x). As Fig. 6b shows, this cannot be accomplished by the fixed component B anymore. In order to still use the functionality of component B, an additional wrapper B' has to be introduced (Fig. 6c). In an object-oriented context the Adapter design pattern is commonly used for resolving such interface mismatches, or the Decorator for introducing additional functionality.

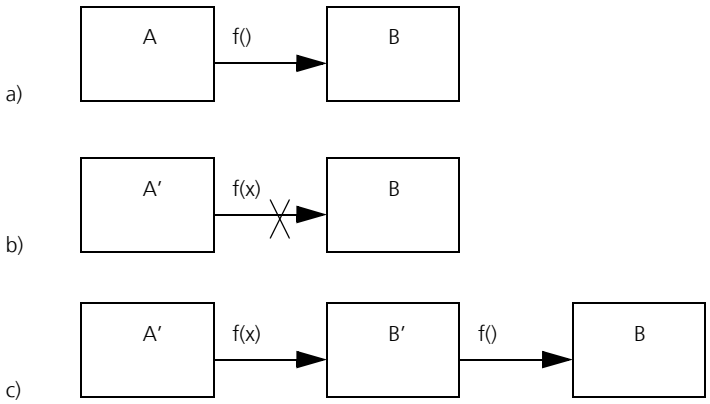



Figure 6:
a) Component B is used by system A
b) Component B cannot be used by a similar system A'
c) Wrapper B' has to be introduced between A' and B

However, wrapping introduces an additional level of indirection, which complicates the design and might reduce performance. For simple functionalities, an alternative would be to rewrite B from scratch, not using the original component at all, which leads to a proliferation of similar components with negative impacts on maintainability.

To overcome these drawbacks, a generalized concept of software reuse has been proposed beyond the simple notion of use-as-is. This concept explicitly distinguishes between use (as-is) and reuse, where use is only the simplest form of reuse. In this sense, reuse means the ability to change the properties of a software component (data and functions) as required in a specific context. This adaptability is the major feature that distinguishes these reusable components from traditional, unadaptable ones.

To continue the example: When, as depicted in Fig. 6b, the component B cannot be used as-is anymore, it should be made adaptable, resulting in the compo-

nent C shown in Fig. 7. Then, by performing an adaptation to a property of the component, e.g. changing the signature of the function f, the component C can be made usable for both systems A and A'.

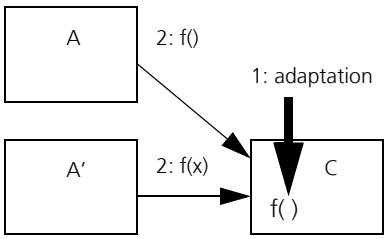



Figure 7:
The reusable component C can be made usable for both systems A and A' by adaptation

Note that there is a sequence, as indicated by the numbers in Fig. 7: If C's defaults are not suitable in a specific context, an adaptation has to be performed first, and then the component can be used (the function can be called).

Also note that adaptation and use appear in different dimensions (again cf. Fig. 7): while adaptation takes place when the component is implemented (at construction time), use occurs while the system is executed (at run-time).

To summarize, there is a duality between use and reuse; use stresses sameness (fixed data and functions), whereas reuse emphasizes similarness (adaptation of the data and functions which will be used afterwards), i.e. sameness with exceptions. Component reuse in this general sense is the process of adapting a generalized component to various contexts of use [Bas96, p.14].

As explained in the previous section, a software product line consists of several systems offering similar functionality, and Basset's notion of reuse emphasizes similarness as well. Consequently, his approach for representing reusable assets and the resulting implementation technique are suitable for implementing product lines.

In the product line context, consider the previous example again (see Fig. 6 and Fig. 7). On the one hand, the system A might evolve into system A' over time, but A and A' might just as well represent two similar systems existing at the same time, e.g. two customer-specific systems in the same application domain. In both cases, they can be seen as two members of a software product line. The generic component C shown in Fig. 7 offers a construction time variability that manifests in a variation point both in the model and in the implementation of the component. During the implementation, the variability mechanism of adaptation is used to generate a specific product.

The reuse ideas presented in this section led to the development of a technology for implementing reusable software: frame technology.

## 1.4 Outline

The remainder of this report is organized as follows:

In Section 2, the major concepts of frame technology will be explained, and a program generator utilizing these core concepts, a plain frame processor, will be introduced. This will be used in the following case study to support a product line implementation.

An example software system which, as a single system, will serve as the basis for a software product line in the case study, will be presented in Section 3.

In the following sections, four typical evolution scenarios for the single product into a product line will be discussed. These represent reactive approaches to introducing or changing a product line. Section 4 shows how related alternative behavior crosscutting several modules can be introduced in one place only. In Section 5, the change of contracts (combinations of new preconditions and postconditions) is evaluated. Section 6 deals with negative variability, i.e. moving a formerly common feature of the product line out into a specific product only. In contrast to this, Section 7 describes how a single-system feature can be included into the product line. In each of these scenarios the frame approach is compared with a conventional object-oriented solution.

Chapter 8 will discuss the results gained in the aforementioned steps, and will give a summary and an outlook on future research activities.

To keep the source code examples compact in chapters 3 to 7, the programming language Python is used there. For readers unfamiliar with this language, the same examples are shown in C++ in Appendix A.

As an alternative to using the frame processor, some examples can be implemented using a C preprocessor. This is sketched in Appendix B.

During the case study a number of reoccurring sequences of steps had to be performed with frame technology. As a starting point for collecting these interrelated idioms, seven small-scaled frame technology idioms are listed in Appendix C.

# 2 Frame Technology and the Plain Frame Processor

The reuse ideas presented in the previous section led to the development of frame technology as a technology for implementing reusable software. In this chapter, core concepts of frame technology will be discussed, and it will be shown that frame technology supports the implementation of product lines. A tool supporting these core concepts will be presented, which will be used in the case study in the following chapters to implement a product line.

## 2.1 Concepts of Frame Technology

Inspired by ideas from artificial intelligence, skeleton code, macros and generators, frame technology has been developed as a technique to support reuse in practice. Some of the major characteristics of frame technology are summarized in Fig. 8 and discussed in the remainder of this section.

Figure 8: Characteristics of Frame Technology

- similar physical organization as in conventional programming languages
- facilitates general reuse
- makes variation points explicit
- enables non-trivial adaptation
- is similar to multiple inheritance in object-orientation
- makes levels of reusability explicit
- is similar to macros in preprocessing
- performed at construction time
- independent of any programming language
- tool-based
- generative
- supports major variability types

In frame technology, the physical implementation units, called frames, have the same appearance as those in any major programming language: they form a group of symbols (source code) that can be consistently referenced as a unit.

However, whereas conventional physical units contain only the source code corresponding to the specific programming language (meant to be used as-is), frames contain both program source code and frame-specific code (providing adaptation) which enables general reuse.

Frame specific code consists of frame commands or frame variables, which resemble macros and will be discussed later. The simplest frame commands do not require frame variables and fall into the three categories listed in Fig. 9:

Figure 9:
Three simple categories of frame commands

1 mark variation points

2 adapt frames

3 alter default frame text

a) leave default text unchanged

b) replace default text

Frames consist of source code text areas which stay the same among different members of a product line, and those that vary among individual systems. These common and variable text areas are distinguished by marking the variable parts. The first category of frame commands is used to mark those parts of the source code where this construction time variability can occur, making variation points explicit. The (source code) text enclosed by this frame command serves as a default for other frames.

A frame's variation point serves as a compile-time hook for extension or modification. As this alteration should take place without changing the existing frame, the alteration will be expressed within a separate frame. This frame refers to the reused one with an adapt frame command, which resembles the #include or import commands in the C preprocessor or in Java. In fact, in the trivial case of adaptation where no alteration is performed, adapting corresponds to including or importing.

A non-trivial alteration can either be an extension to the default provided within the variation point (positive variability), or a modification of the default (negative variability; see Cop99). These correspond to the two subcategories of frame commands mentioned in Fig. 9. Extension occurs by inserting text, and this insertion is either before or after the default text enclosed by the variation point. On the other hand, the marked part is modified by replacing it by alternative code or by removing it altogether. As a result, these alteration commands can be used to express the three basic variability types (Fig. 5). Note, however, that multiple coexisting possibilities should not be expressed by replacing the entire default text, as this introduces duplication for each combination.

One single frame can adapt several other frames at the same time, thereby combining their capabilities. This is similar to multiple inheritance in object-orientation. However, in object-orientation the features (data, methods) can only be added in a "same as, plus" way, but not removed entirely in a "same as, except" way. Consider data: a derived class contains all the data members of its base class. You cannot remove them via inheritance. On the other hand, frames allow both adding and unrestricted removal of base properties like data or parameters.

Frame adaptation leads to hierarchies of frames that have a special property: they allow the frames to be categorized according to their level of reusability. This is illustrated in Fig. 10 which shows a graphical representation of a frame hierarchy. Each circle represents one frame, and each arrow means "is adapted by" or "is reused by". A frame in the lower part of this hierarchy is reused (adapted and eventually changed) by those frames above it. As a result, the lower frames are more generic, more context free and ultimately more reusable than those above. In practice, the lowest frames should contain basic technical functionality like I/O or vendor-supplied functionality; those above them might encompass technical corporate standards, those frames higher up general domain-specific (business) rules, and finally the top frames contain the unique business rules [Bas96, p.95]. Thus the highest frames in the hierarchy are the most context-specific and the lest reusable ones; they are called specification frames (SPCs) and determine the specific product.

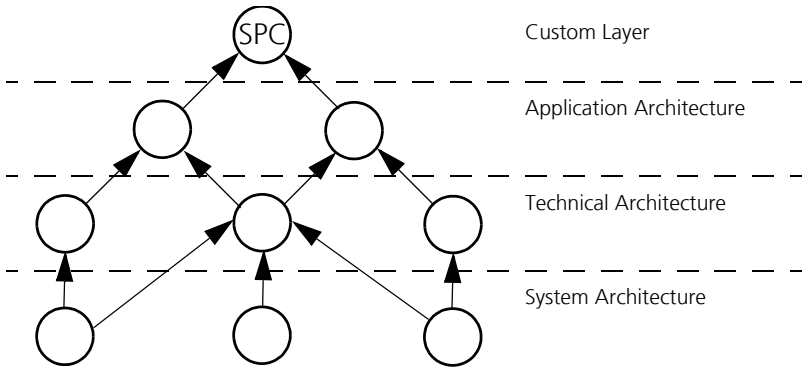



Figure 10: Graphical representation of a layered frame hierarchy

Frame variables are similar to macros in conventional preprocessing environments and can be used as in the variability mechanism of conditional compilation. In addition to this, frame variables are more local than conventional macros, and thus safer to use. In addition to these closed kinds of parameters that are limited to a fixed predefined set of options, frames support open parameters with an unlimited set of options. Thus, like conditional compilation, the mechanisms provided by frame technology work at construction time.

Frames can contain source code in any programming language, or generally any group of symbols that does not belong to the set of frame commands. This is why frame technology is independent of any programming language.

In frame technology, an advanced preprocessor (frame processor) is used as to automatically process the frame hierarchy, thereby executing the frame commands and generating physical units containing program source code only, which can then be translated into executables via compilers or interpreters. Thus, it is a generative technique.

Finally, besides offering the general variability mechanism of adaptation and the explicit support for the product line concept of variation points, frame technology supports expressing all major variability types: optional, alternative as well as multiple coexisting possibilities (see Fig. 5).

For more details about the differences to macros and inheritance, see [Bas96 pp.117ff., p.53 and pp.143ff.].

## 2.2 A Plain Frame Processor

A frame processor is a tool supporting frame technology, synthesizing run-time modules from generic ones. Besides the original frame processor developed by Paul Basset, which is part of the Netron Fusion product, two other frame processors using the techniques described above exist: An XML-based one using the XVCL language [XVCL] and the plain frame processor fp, which will be characterized in this section and used in the case studies in the following chapters.

Fp was developed to be as simple as possible, while still offering the functionality that distinguishes frame processors from other generative tools, as summarized in Fig. 8. Hence, frame variables and their associated frame commands are not supported, as they are similar to conventional open macro parameters and thus do not contribute to the distinct core functionality of frame processors. Frames are files containing program source code and frame commands. Fp's frame command set is restricted to those three categories mentioned in Fig. 9; all frame commands provided by fp are listed in Fig. 11.

Figure 11:
Fp command reference

| category | purpose | fp syntax | example |
|---|---|---|---|
| 1 | mark a point of variation | VP *vp-name*<br>*default-text (opt.)*<br>VP_END | VP myVp<br>printf("common part");<br>VP_END |
| 2 | refer to a frame | ADAPT *frame-name*<br>*insert_before-, insert_after-, or replace-commands (opt.)* | ADAPT example.frame |
| 3a | insert text before a VP | INSERT_BEFORE *vp-name*<br>*text to be inserted* | INSERT_BEFORE myVp<br>printf("before"); |
| 3a | insert text after a VP | INSERT_AFTER *vp-name*<br>*text to be inserted* | INSERT_AFTER myVp<br>printf("after"); |
| 3b | replace the text of a VP | REPLACE *vp-name*<br>*text to be replaced (opt.)* | REPLACE myVp<br>printf("instead"); |
| | name the output file | OUTFILE *output-file* | OUTFILE main.c |

The frame command VP is used for marking a variation point within a frame. As a single frame should be able to contain several different variation points, each VP frame command is followed by a unique identifier naming the variation point. Moreover, the variation point is one-dimensional only in the trivial case where the default frame text is empty. Otherwise, a variation range exists, so that fp provides the aforementioned VP command for marking the beginning of a variation range, and VP_END for marking its end. Note that even for empty default text the two commands have to be used to mark one variation point.

The fp command ADAPT is used to refer to a frame. This command is followed by the frame's file name, which uniquely identifies each frame. Note that in fp frames can have arbitrary file names; especially, they are not distinguished based on a file name extension. Also note that in contrast to the VP command, it is not necessary to explicitly mark the end of an ADAPT command, as it is implicitly ended by a following ADAPT command or the end of the file.

The ADAPT frame command is typically followed by frame instructions for altering the default frame text. The commands INSERT_BEFORE and INSERT_AFTER extend the default frame text by prefixing or suffixing it, the REPLACE command overrides the default text. All these three frame commands are followed by the identifier naming the variation point, as assigned in the VP command. The following lines contain the altering (source code) text. Only for the REPLACE command it makes sense to leave this text empty, in which case the default text of the adapted frame is erased. Note that in contrast to the VP command, and like all other fp frame commands, the altering commands do not require an end command. In this case each command is implicitly ended by the beginning of another altering or adapting command, or the end of the file.

A command not fitting into the categorization of the general frame commands (Fig. 9) is the fp command OUTFILE. It is commonly used at the beginning of a frame and specifies the file name to be generated. Note that the OUTFILE command must not specify the frame name of the frame which contains this command, as this would lead to overwriting itself. Fp issues a warning in this case and does not generate the file.

The use of fp will be illustrated in chapters 4ff., where we will show how the implementation of a single software system can evolve into a product line. This single software system, the basis for the product line, will be presented in the following chapter.

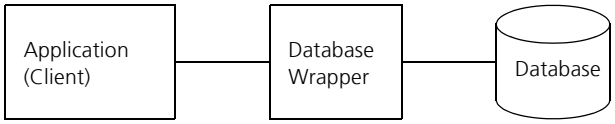
# 3 Example Component: Database Wrapper

In this chapter, we introduce a database wrapper software component which serves as a running example in this case study. First its high-level architecture is sketched. Then we present what requirements the component has to fulfil in a specification model showing the externally visible properties of the component. After that, we will show how the component fulfills these requirements with realization models. Both the specification model and the realization model [ABB+01] describe the logical component in a technology-independent way as UML diagrams. Finally, we show a simple implementation of this component, driven by unit-tests [Bec99].

The initial system which is described in this chapter embodies a single system, which will evolve into a software product line in the chapters to follow. This corresponds to a situation commonly encountered in practice[1], where an existing single system is taken as the basis for a product line.

When interfacing an application with a database, the DB API is often encapsulated in a simple DB facade in order to decouple the application logic from the database (layering), as depicted in Fig. 12.

Figure 12: High-level picture of a database wrapper




In many cases, these wrappers provide similar services to applications, as illustrated in the structural specification in Fig. 13. The database layer provides the classes `Connection` and `Cursor`. The effects of their externally visible methods are described in the operational specifications. The `Connection` constructor is responsible for access control to the database (Table 2). Moreover, `Connection` offers its clients to retrieve one or several `Cursor` instances through the method createCursor (Table 3). A query can be issued to the database by calling the `Cursor`'s execute method (Table 4).

1 This approach for migrating to product line engineering isoften referred to as the reactive or evolutionary transfer strategy [MS01]. The complimentary approach is the proactive or revolutionary tranfer strategy.

Figure 13: Structural specification of a DB wrapper

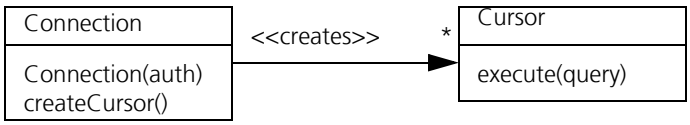



Table 2: Operation specification of the Connection constructor

| | |
|---|---|
| Name | Connection() |
| Description | Establishes the connection to the database |
| Receives | The user's authentification, e.g. a user id and a password |
| Result | A connection is established for a valid authorization, otherwise, an error is issued |

Table 3: createCursor() operation specification

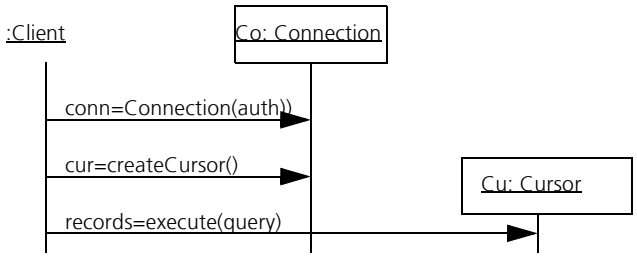
| | |
|---|---|
| Name | createCursor() |
| Description | Creates a cursor object for querying the database |
| Returns | A new cursor instance |
| Rules | A usable cursor is only returned for a valid Connection |
| Result | A new cursor is returned for a valid Connection |

Table 4: execute() operation specification

| | |
|---|---|
| Name | execute() |
| Description | Executes a query on the database |
| Receives | A query, e.g. an SQL statement |
| Returns | A query result |

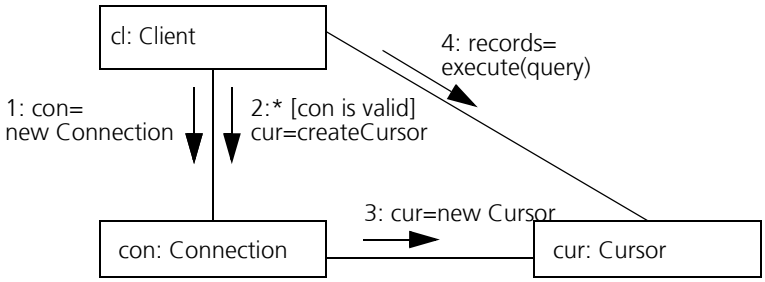
The sequence of a query session is depicted in Fig. 14. The client establishes a connection by invoking the `Connection` constructor, then requests a `Cursor` by invoking createCursor on the `Connection`, and finally issues a query by calling execute on the `Cursor`, receiving a result set.

Figure 14: Sequence diagram for issuing a database query



The interaction between the three objects in this operation is shown in the collaboration diagram in Fig. 15. Note that a client can request several `Cursors` for issuing multiple queries in parallel.

Figure 15: Collaboration diagram for client usage of DB wrapper




The following implementations will be shown in Python code, as it is generally more compact and readable than the more common programming languages like Java or C++.

Consider this simple Python test code for such a database facade, representing a surrogate of the application performing a query sequence (cf. Fig. 14):

Listing 1: Test code for the database facade

```
from db import *

def test():
  conection=createConnection('user','password','database')
  cursor=connection.createCursor()
  records=cursor.execute('SELECT * FROM table WHERE id<9')
  print 'execute returned:',records

if __name__=='main':
  test()
```

After creating a `Connection` for a specific database, authenticating with a user name and a password, one `Cursor` is created. This `Cursor` is used for executing a query, and the result is printed.

The following Mock Object pair represents the database facade for a specific default database. Instead of interfacing with a real database, the text 'default db:' is printed, and each call of a DB API service is mimicked by outputting an appropriate message like 'cursor execute' or 'connection init'. Executing a database command is illustrated by printing the query, and a dummy result consist-

ing of two data sets is returned. To make the example more interesting, the number of open `Cursors` is counted and printed while opening each new one.

Listing 2: Initial implementation db.py of the database facade

```
class Cursor:
  count=0
  def __init__(self):
    self.count=self.count+1
    print 'default db:',
    print 'cursor init',self.count
  def execute(self,s):
    print 'default db:',
    print 'cursor execute',s
    return ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])

class Connection:
  def __init(self,s):
    print 'default db:',
    print 'connection init',s
  def createCursor(self):
    cursor=Cursor()
    return cursor

def createConnection(user,password,database):
  return Connection(user+'/'+password+'@'+database)
```

When running this example in Python the following output is produced:

```
> python test.py

default db: connection init user/password@database
default db: cursor init 1
default db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

The following sections describe different scenarios in which the database facade evolves into several similar products. The implications on the above implementation is sketched, utilizing two kinds of implementation techniques for handling variability: On the one hand, traditional language-independent OO techniques, and on the other frame technology as employed in the simple frame processor fp. By comparing these two techniques, their suitability in the context of product line implementation is evaluated. Language-dependent techniques or paradigms like multiple inheritance or parametric polymorphism were not evaluated. Excerpts from frame implementations in a more complex programming language (C++) and preprocessor implementations can be found in the appendix.

Throughout all the scenarios, the test driver, representing the existing client application, should remain unchanged. Furthermore, the number of changes in existing code or frames should be kept to a minimum, the execution order should be maintained, and the solutions should be kept as simple as possible.

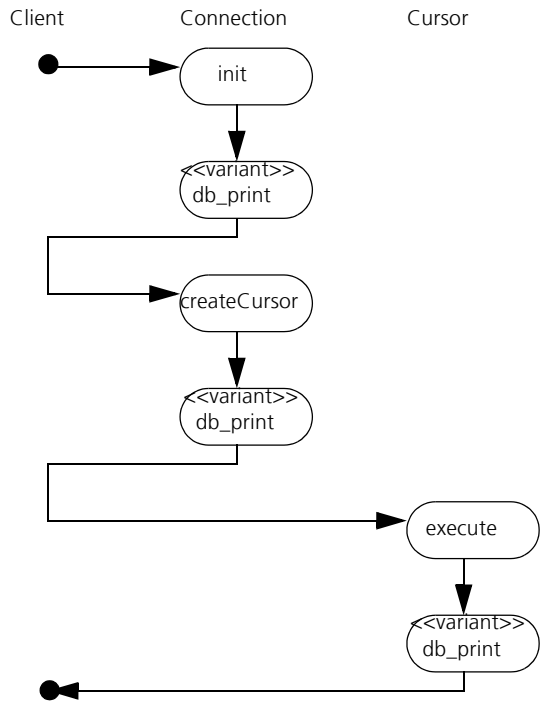
# 4 Scenario #1: Introducing Alternative Behavior

In this first evolution scenario, a new database is required to be accessed by the existing application. In the example implementation, this shall be reflected by printing ‘new db’ instead of ‘default db’ before executing each method (a change in a part of the tracing aspect), as shown in Fig. 16 and in the following output (in this and the outputs to follow, variants to the previous scenario are written in bold):

```
> python test.py

new db: connection init user/password@database
new db: cursor init 1
new db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

Figure 16:
Activity diagram for
issuing a database query

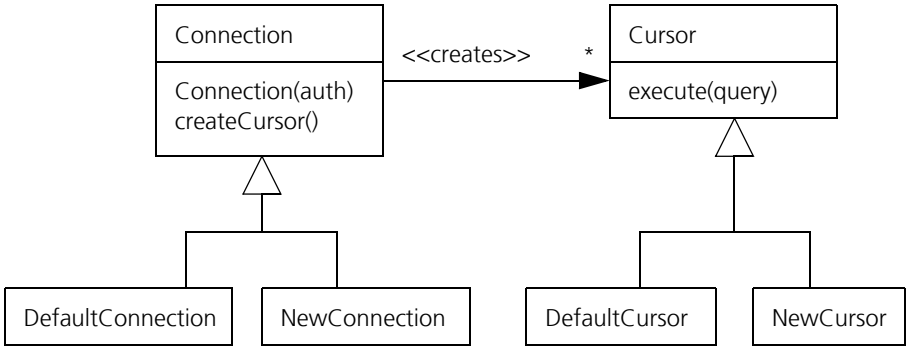


## 4.1 Conventional Solution

A simple conventional solution would be to introduce new subclasses for both `Connection` and `Cursor`, and change createConnection to return the new connection. However, as the XOR-variable functionality of printing the database name is performed in the constructors, this solution is not viable because both 'new db' and 'default db' would be printed when instantiating the subclasses.

Thus common base classes have to be introduced, making the default and new `Connections` and `Cursors` siblings. Figure 17 illustrates this solution.

Figure 17:
Class diagram of conventional solution to scenario 1




Listing 2 shows the resulting implementation, with the new classes introduced in the same file as the old ones, in order not to change the client's file dependency. As in the listings and outputs to follow, the added or changed parts are printed in bold, and the differences in the different product implementations are underlined.

The method body of createConnection has been changed at one place to create the `NewConnection` class. In order to use the `DefaultConnection` instead, this has to be substituted at exactly one place.

Cursor.execute() implements the behavioral variability in the Template Method db_print, implemented in the respective child classes. The same applies for the constructors (__init__).

Note that in this solution, the bold part, representing change during evolution, is large, whereas the underlined part, indicating variability across products (i.e., non-duplicates), is very small, but spread throughout the code. A large amount of duplicated code has been introduced by nearly all of the concrete classes (all the bold text there except for the underlined parts). Moreover, this solution has a performance overhead due to the subtype polymorphism introduced by the TemplateMethod, and a space overhead because of the additional classes.

Listing 2: Conventional implementation db.py of scenario 1 (compare to Listing 1)

```
class Cursor:
  count=0
  def __init__(self):
    self.count=self.count+1
    self.db_print()
    print 'cursor init',self.count
  def execute(self.s):
    self.db_print()
    print 'cursor execute',s
    return ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])

class DefaultCursor(Cursor):
  def db_print(self):
    print 'default db:',

class NewCursor(Cursor):
  def db_print(self):
    print 'new db:',

class Connection:
  def __init__(self,s):
    self.db_print()
    print 'connection init',s

class DefaultConnection(Connection):
  def createCursor(self):
    cursor=DefaultCursor()
    return cursor
  def db_print(self):
    print 'default db:',

class NewConnection(Connection):
  def createCursor(self):
    cursor=NewCursor()
    return cursor
  def db_print(self):
    print 'new db:',

def createConnection(user,password,database):
  return NewConnection(user+'/'+password+'@'+database)
```

## 4.2 Frame Solution

In the frame solution, exactly the same classes and methods are used after introducing the new database, and no additional classes, hierarchies, methods, call chains or parameters are necessary.

Instead, the following process is performed:

a) convert those files into frames which are supposed to change
b) introduce variation points at the respective locations
c) create an adapting frame which manipulates the variation points

The detailed process is as follows:

a) Convert the existing files (in this case, db.py) as-is into frames. This can be done manually (by prepending OUTFILE db.py to the source code in the file db.py and storing this as db.frame) or automatically (by calling mkf db.py db.frame). The OUTFILE frame command will instruct the frame processor to generate a file of the given name.
Backup the original files (db.py), as they will be overwritten.

Test recreating and running it:

creating: fp db.frame
running: python test.py

or both: fpy db.frame test.py

Exactly the same db.py is created (which can be checked via its file length or tools like diff), and the same output is produced when running it (which could be checked automatically with a unit test framework).

b) Instrument the existing frame with variation points:

Mark one position where the replacement shall occur with VP and VP_END (in this case, the first occurrence of print 'default db:'), and recreate and run the application as mentioned before. If necessary, repeat this for each other place where the same variation will occur, introducing one variation point at a time (here, in two iterations for each other occurrence of print 'default db:'). Because converting into frames and introducing variation points does not change the source code to be generated by the frame processor, the application must run exactly as before (if not, an error was introduced during the last iteration, for

example, by mistyping VP/VP_END, or leaving out the VP's name or VP_END). The resulting frame db.frame is shown in Listing 3:

Listing 3:
Reuseable frame db.frame in scenario 1 (compare to Listing 1)

```
OUTFILE db.py
class Cursor:
  count=0
  def __init__(self):
    self.count=self.count+1
VP db_print
    print 'default db:',
VP_END
    print 'cursor init',self.count
  def execute(self,s):
VP db_print
    print 'default db:',
VP_END
    print 'cursor execute',s
    return ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])

class Connection:
  def __init__(self,s):
VP db_print
    print 'default db:',
VP_END
    print 'connection init',s
  def createCursor(self):
    cursor=Cursor()
    return cursor

def createConnection(user,password,database):
  return Connection(user+'/'+password+'@'+database)
```

c) Overwrite the text in the variation point(s):

Create a new frame (newdb.frame), adapting the frame created above (with the frame command ADAPT db.frame). Replace the text embraced by the VP (through the frame command REPLACE db_print), followed by the new text. Newdb.frame is depicted in Listing 4:

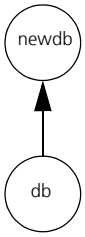
Listing 4:
SPC newdb.frame in scenario 1

```
ADAPT db.frame
REPLACE db_print
    print 'new db:',
```

The new database facade can now be generated by calling the frame processor with the new frame (fp newdb.frame), and the original one by calling it with the

frame produced in step b (fp db.frame). The frame hierarchy is shown in Fig. 18, with db.frame as the more general frame and newbd.frame as the SPC:

Figure 18:
Frame Hierarchy in scenario 1




Note that in the frame solution, the bold parts of the code, representing change during evolution, is considerably smaller than in the OO solution. Moreover, the differences between the products is clearly localized in the text enclosed by VP/ VP_END (for the original product), and in the text following the REPLACE (for the new product). And, unlike in the OO solution, no code duplication has been introduced during this step, and there is no space or performance overhead.

# 5 Scenario #2: Changing Contracts

In this evolution scenario, an additional action has to be optionally performed before, after or both before and after creating the `Cursor` in createCursor (e.g. some synchronization or changing the `Cursor` after creating it, see Fig. 19 and Fig. 20).

Figure 19: Collaboration diagram for the default DB wrapper (compare with fig. 2)

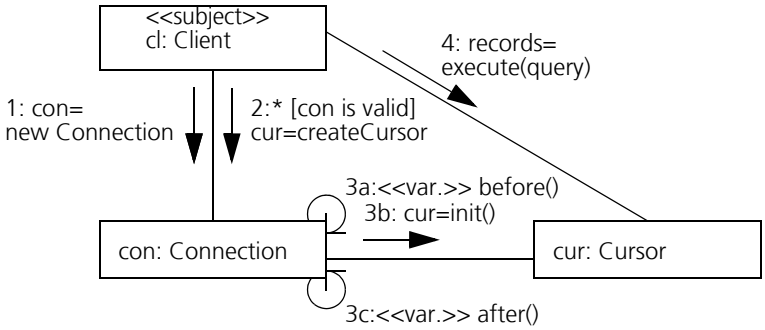
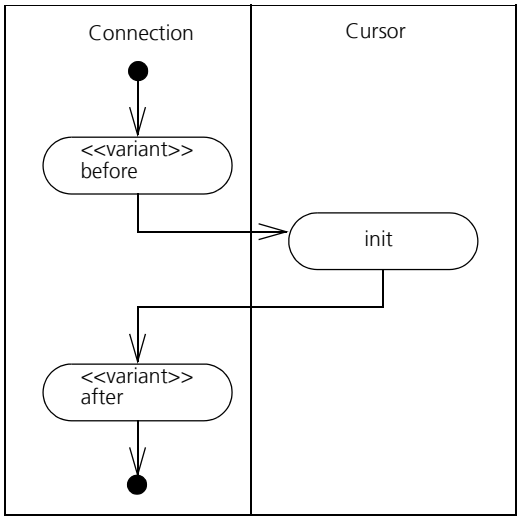



Figure 20: createCursor() activity diagram




However, only the original product, the default database wrapper, should exhibit this behavior, but not the new one, so that five different products can be obtained: The default database without additional actions, with before, after, or before and after actions, and the new database without additional actions, as captured in the decision model in Table 1. It illustrates that in scenario 2 not only variants are resolved (changing pre- and postconditions), but also interdependencies while maintaining several products within a product line are evaluated.

Table 5: Decision table for scenario 2

| ID | Description | Depends on ID | Resolution | Effect |
|---|---|---|---|---|
| 1 | Is the default DB used? | | yes | consider IDs 2 and 3 |
| | | | no | remove 3a and 3c (Fig. 19) |
| 2 | Should an action be performed before creating the cursor? | 1 | yes | remove <<var.>> at 3a (Fig. 19), remove <<var.>> for before (Fig. 20) |
| | | | no | remove 3a (Fig. 19), remove before (Fig. 20) |
| 3 | Should an action be performed after creating the cursor? | 1 | yes | remove <<var.>> at 3c (Fig. 19), remove <<var.>> for after (Fig. 20) |
| | | | no | remove 3c (Fig. 19), remove after (Fig. 20) |

In this case study, the additional actions before and after are reflected by printing '-> action before creating cursor' and '-> action after creating cursor', respectively. For the default database with both of these actions, the output will look like this:

```
> python test.py

default db: connection init user/password@database
-> action before creating cursor
default db: cursor init 1
-> action after creating cursor
default db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

## 5.1 Conventional Solution

A simple solution is to include or exclude the additional action(s) depending on the value of an additional parameter in DefaultConnection.createCursor(). In order not to change the signature of createCursor (which would require changes in the application), this parameter has to be a class attribute, initialized in the constructor of `DefaultConnection`. This means that a `DefaultConnection` constructor has to be introduced, and its factory createConnection has to be changed for each of the before/after combinations (again configuring

the product takes place in the factory). Listing 5 shows the resulting implementation:

Listing 5: Conventional implementation of scenario 2 (changed part of Listing 2)

```
...
class DefaultConnection(Connection):
  nothing=0
  before=1
  after=2
  before_and_after=3
  def __init__(self,s,mode):
    Connection.__init__(self,s)
    self.mode=mode
  def createCursor(self):
    if self.mode==1 or self.mode==3:
      print '-> action before creating cursor'
    cursor=DefaultCursor()
    if self.mode==2 or self.mode==3:
      print '-> action after creating cursor'
    return cursor
  def db_print(self):
    print 'default db:',

...
def createConnection(user,password,database):
  return DefaultConnection(user+'/'+password+'@'+database,\
                           DefaultConnection.before_and_after)
```

In this solution, the change caused by evolution (the bold part) is rather localized, affecting only one class (`DefaultConnection`) and its factory, but not the other product (`NewConnection` or `NewCursor`). However, most of the difference between the products (the underlined parts) is concerned with configuring, the use of these magic constants is problematic for comprehensibility and affects space and performance negatively, and the configuration requires knowledge of the class implementation internals.

## 5.2 Frame Solution

The frame solution will require the following process:

a) introduce a variation point at the respective location
b) create two adapting frames for the before and after action
c) create a wrapper frame for the before/after combination

In detail:

a) Mark the point in db.frame where the `Cursor` is created with VP create_cursor and VP_END (Listing 6):

Listing 6: Reusable frame db.frame in scenario 2 (changed part of Listing 3)

```
...
class Connection:
  def __init__(self,s):
VP db_print
    print 'default db:',
VP_END
    print 'connection init',s
  def createCursor(self):
VP create_cursor
    cursor=Cursor()
VP_END
    return cursor

...
```

b) Create two frames, before.frame and after.frame, ADAPTing db.frame. For adding text before the VP create_cursor, use the frame command INSERT_BEFORE create_cursor, followed by the additional text, and for adding text afterwards, use the frame command INSERT_AFTER similarly. The two frames will look like this:

Listing 7: Specification frame before.frame

```
ADAPT db.frame
INSERT_BEFORE create_cursor
    print '-> action before creating cursor'
```

Listing 8: Specification frame after.frame

```
ADAPT db.frame
INSERT_AFTER create_cursor
    print '-> action after creating cursor'
```

Test-run each of them:

```
> fpy before.frame test.py
...
default db: connection init user/password@database
-> action before creating cursor
default db: cursor init 1
default db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

```
> fpy after.frame test.py
...
default db: connection init user/password@database
default db: cursor init 1
-> action after creating cursor
default db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

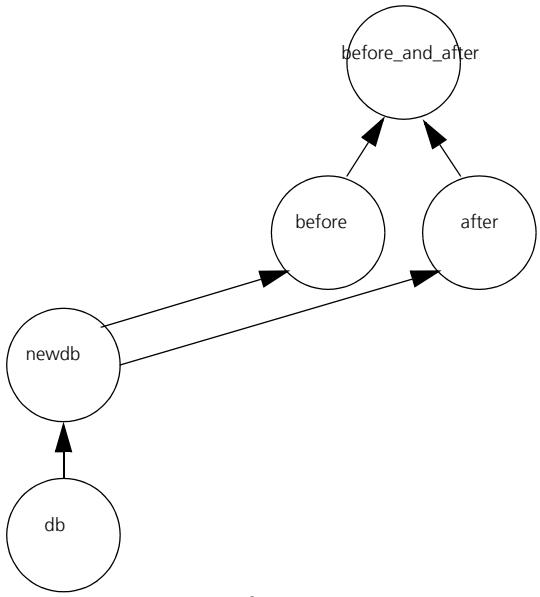
c) For performing both before and after actions, create a frame before_and_after.frame, which only contains ADAPT commands, and no INSERTs. This kind of frame is called a wrapper frame. ADAPT both before.frame and after.frame, as shown in Listing 9,

Listing 9: SPC before_and_after.frame

```
ADAPT before.frame
ADAPT after.frame
```

and test-run it.Figure 21 shows the frame hierarchy for the new database and the default one. Each frame can serve as the SPC for one concrete product, with db.frame being reused by all other frames.

Figure 21: Frame Hierarchy in scenario 2




Again, the changes in the frame solution are small and well-localized; as opposed to the OO solution, the information for each of the before, after and before/after actions is located in separate physical units, and thus changing a product only requires calling the frame processor with a different frame. There are no code changes as in the conventional solution, and mode constants are avoided. As in the OO solution, the new database is not affected by providing for new variants in the default one.

# 6 Evolution #3: Cancelling Behavior

In the third scenario, only the cursor of the newly introduced database wrapper should handle counting, but not the default db cursor (see Fig. 22 and Table 2).

Figure 22: Cursor.init() activity diagrams

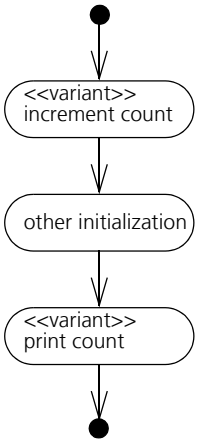



Table 2: Decision table for scenario 3

| Description | Resolution | Effect |
|---|---|---|
| Is the default DB used? | yes | remove both variant activities (Fig. 22) |
| | no | remove <<var.>> from both variant activities (Fig. 22) |

As in the previous scenario, the commonality between the two products evolves into a similarity, and a change to only a single product, not to both, is required. However, in contrast to that case, a behavior of an existing product shall not be added, but be removed. Furthermore, the resolution does not result in a change in one location, but in several ones simultaneously.

The outputs will be:

```
> python test.py

default db: connection init user/password@database
default db: cursor init
default db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

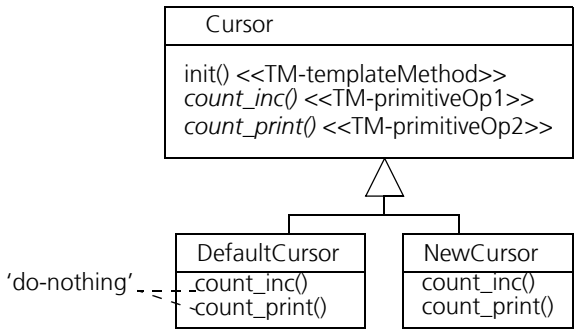
```
> python test.py

new db: connection init user/password@database
new db: cursor init 1
new db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

## 6.1 Conventional Solution

In the conventional case, the commonality of counting has been implemented in a base class. One solution if the commonality turns out to be broken to a small degree will be to introduce negative variability, which turns out to be difficult to implement in many programming languages and is generally discouraged. A cleaner solution is to remove the commonality from the base class altogether, and reimplement the functionality in derived classes. In this case, the base class counting behavior is refactored into a TemplateMethod, with the counting behavior moved into the class `NewCursor`, and a Null behavior is implemented in the `DefaultCursor` class. Figure 23 and Listing 10 illustrate this solution.

Figure 23:
Class diagram of conventional solution for scenario 3



Listing 10: Conventional implementation of scenario 3 (changed part of Listing 2)

```
class Cursor:
  def __init__(self):
    self.count_inc()
    self.db_print()
    print 'cursor init',
    self.count_print()
    print
  def execute(self.s):
    self.db_print()
    print 'cursor execute',s
    return ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])

class DefaultCursor(Cursor):
  def db_print(self):
    print 'default db:',
  def count_inc(self):
    pass
  def count_print(self):
    pass

class NewCursor(Cursor):
  count=0
  def db_print(self):
    print 'new db:',
  def count_inc(self):
    self.count=self.count+1
  def count_print(self):
    print 'cursor init',self.count

...
def createConnection(user,password,database):
  return DefaultConnection(user+'/'+password+'@'+database,\
    DefaultConnection.before_and_after)
```

Although this solution is not hard to implement, there is some effort required to come up with this solution, to introduce and call four new methods, and to ensure that the refactoring has not changed the behavior of class `NewCursor`. Moreover, the change is not localized, so that all classes in the class hierarchy have to be changed, and redundancy has been introduced (the method signatures count_inc and count_print are equal for `DefaultCursor` and `NewCursor`).

## 6.2 Frame Solution

The following steps have to be performed for a frame solution:

a) refactor the code at the respective locations so that each line contains either code common to all products or code which is variable across products, but not both intermixed in the same line.
b) introduce VPs for those lines containing variable code.
c) adapt the frame for each of the different products, REPLACE text for devia-

tions which break the commonalities (i.e. introduce negative variabilities, where necessary)
d) optionally, refactor the frames so that the REPLACEs are substituted by INSERT_BEFOREs and INSERT_AFTERs, thus turning the negative variabilities into positive ones.

After each step, test-run the code to ensure that no errors have been introduced.

In detail:

a) The only line where both common and variable code exists is "print 'cursor init',self.count" in the method Cursor.__init__. This has to be refactored as follows (the comma means line continuation; the single print means end line):

```
print 'cursor init',
print self.count,
print
```

A similar process has also been applied in the OO solution, when the lines "self.count_print()" and "print" were introduced.

Generate and test-run both products (fpy db.frame test.py and fpy newdb.frame test.py).

b) Three lines in the `Cursor` implementation deal with counting. Listing 11 shows the frame after introducing the three VPs:

Listing 11: Reusable frame db.frame in scenario 3 (compare to Listings 3)

```
...
class Cursor:
VP cursor_attributes
  count=0
VP_END
  def __init__(self):
VP count_inc
    self.count=self.count+1
VP_END
VP db_print
    print 'default db:',
VP_END
    print 'cursor init',
VP count_print
    print self.count,
VP_END
    print
  def execute(self,s):
VP db_print
    print 'default db:',
VP_END
    print 'cursor execute',s
    return ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])

...
```

Again, generate and test-run both products.

c) Because the new database should provide for counting, nothing in its specification frame newdb.frame has to be changed. For the default database, introduce a frame defaultdb.frame which adapts db.frame and clears the text within the three variation points (Listing 12):

Listing 12: Intermediate frame defaultdb.frame in scenario 3

```
ADAPT db.frame
REPLACE cursor_attributes
REPLACE count_inc
REPLACE count_print
```

Test-run the products (fpy defaultdb.frame test.py and fpy newdb.frame test.py).

Note that the before/after-aspects cannot be taken into account in this solution without changing all their ADAPTs from ADAPT db.frame to ADAPT defaultdb.frame!

d) In order to enhance later extensibility, the three frames might optionally be refactored, so that the common frame only contains the commonalities and

defaults of all products and becomes more reusable. Additionally, the before/after-aspects can be taken into account again without changing them!

In this example, move the code lines within the three VPs from db.frame into INSERT_AFTER sections (or INSERT_BEFORE; it does not matter which one is used, but I generally prefer INSERT_AFTER) in newdb.frame (Listings 13 and 14). The frame defaultdb.frame will become obsolete and can be erased.

Listing 13: Refactored reusable frame db.frame in scenario 3 (compare to Listing 11)

```
OUTFILE db.py
class Cursor:
VP cursor_attributes
VP_END
  def __init__(self):
VP count_inc
VP_END
VP db_print
    print 'default db:',
VP_END
    print 'cursor init',
VP count_print
VP_END
    print
  def execute(self,s):
VP db_print
    print 'default db:',
VP_END
    print 'cursor execute',s
    return ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])

...
```

Listing 14: Refactored SPC newsb.frame in scenario 3 (compare to Listings 4 and 12)

```
ADAPT db.frame
REPLACE db_print
    print 'new db:',
INSERT_AFTER cursor_attributes
  count=0
INSERT_AFTER count_inc
    self.count=self.count+1
INSERT_AFTER count_print
    print self.count,
```

The frame hierarchy will look like the one presented in the first evolution (Fig. 18). Test-run it with "fpy db.frame test.py and fpy newdb.frame test.py.

Like in the OO solution, the class `Cursor` is changed by adding and removing lines in several places. However, it is easier to ensure that the changes are behavior-preserving (which is always the case when just introducing VPs), the changes to the specification frame are well-localized in the three INSERT_BEFORE parts, and there is no code duplication (nothing is underlined), increasing comprehensibility and maintainability.

# 7 Evolution #4: Changing Alternative to Optional Behavior

In this final evolution scenario, both the default database Cursor and the new one should have the optional counting capability (see Fig. 22), which means that existing single-product functionality should become an optional part of the product line. The resulting decision table is depicted in Table 3:

Table 3: Decision table in evolution 4

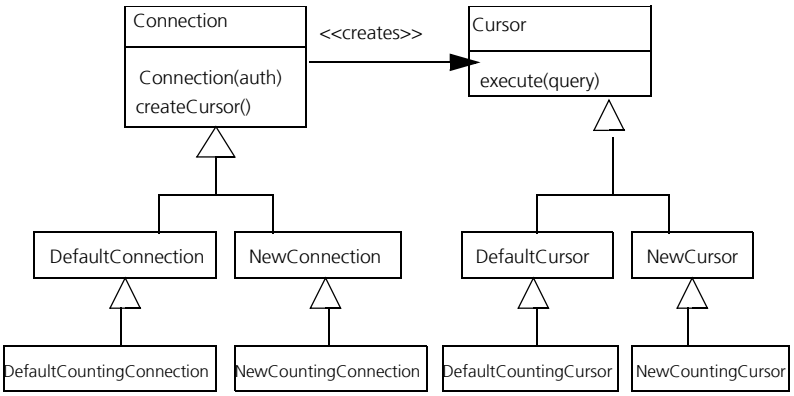
| Description | Resolution | Effect |
|---|---|---|
| Should the Cursors be able to count? | yes | remove both variant activities (fig. 21) |
| | no | remove <<var.>> from both variant activities (fig. 21) |

Ten products can be produced: The four kinds of default database (without, with before, after and before and after actions), all without or with counting, and the new database without or with counting. As in Section 5, the effort required to introduce feature combinations is evaluated here as well.

## 7.1 Conventional Solutions

There are two kinds of conventional solutions, based on the two adaptation principles of unification and separation. The simplest solution for the unification principle is to introduce additional subclasses for counting, resulting in the following dual inheritance hierarchy (Fig. 24):

Figure 24: Class diagram of conventional solution for scenario 4, unification principle




(An even cleaner, but more complex solution not considered here is would be to make the classes for counting and for not counting siblings, i.e. to introduce additional intermediate classes as introduced in Section )

Listing 15 shows the resulting implementation.

Listing 15: Conventional implementation of scenario 4 (unification; changed part of Listings 5 and 10 )

```
...
class DefaultCountingCursor(DefaultCursor):
  count=0
  def count_inc(self):
    self.count=self.count+1
  def count_print(self):
    print 'cursor init',self.count

class NewCursor(Cursor):
  def db_print(self):
    print 'new db:',
  def count_inc(self):
    pass
  def count_print(self):
    pass

class NewCountingCursor(NewCursor):
  count=0
  def count_inc(self):
    self.count=self.count+1
  def count_print(self):
    print 'cursor init',self.count
...
class DefaultCountingConnection(DefaultConnection):
  nothing=0
  before=1
  after=2
  before_and_after=3
  def __init__(self,s,mode):
    DefaultConnection.__init__(self,s)
    self.mode=mode
  def createCursor(self):
    if self.mode==1 or self.mode==3:
      print '-> action before creating cursor'
    cursor=DefaultCountingCursor()
    if self.mode==2 or self.mode==3:
      print '-> action after creating cursor'
    return cursor
  def db_print(self):
    print 'default db:',
...
class NewCountingConnection(NewConnection):
  def createCursor(self):
    cursor=NewCountingCursor()
    return cursor

def createConnection(user,password,database):
  return NewCountingConnection(user+'/'+password+'@'+database)
```

As indicated by the bold parts, the change during evolution is large, and the amount of non-duplicates is very small and spread (the underlined parts). In other words, large amounts of very similar code has to be introduced. More-

over, understandability, changeability and maintenance is complicated by introducing the additional classes with similar functionality.

Another simple solution would be to use the separation principle and introduce a third inheritance hierarchy, as sketched in Fig. 25:

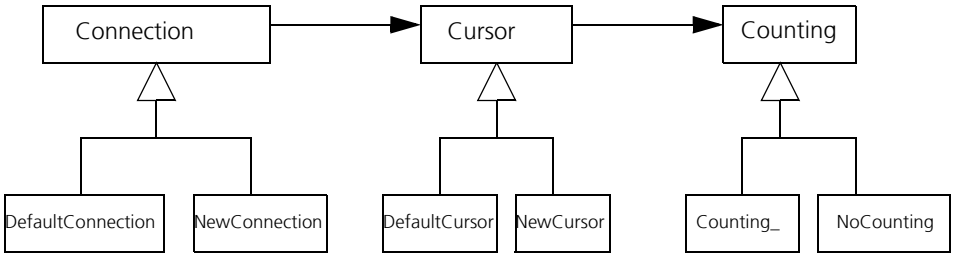



Figure 25:
Class diagram of conventional solution for scenario 4, separation principle

According to Gamma et al., this ought to be the preferred solution ("Favor object composition over class inheritance", [GOF, p.20), resulting in is less new code and less duplication, but at the cost of another level of indirection and more fine-granular changes in each class of the existing hierarchies.

Listing 16 sketches the resulting implementation:

Listing 16: Conventional implementation of scenario 4 (separation; compare Listing 15)

```
class Counting:
  pass

class Counting_(Counting):
  count=0
  def count_inc(self):
    self.count=self.count+1
  def count_print(self):
    print 'cursor init',self.count

class NoCounting(Counting):
  def count_inc(self):
    pass
  def count_print(self):
    pass
...

class Connection:
  def __init__(self,s,c):
    self.counting=c
    self.db_print()
    print 'connection init',s

class DefaultConnection(Connection):
  nothing=0
  before=1
  after=2
  before_and_after=3
  def __init__(self,s,c,mode):
    Connection.__init__(self,s,c)
    self.mode=mode
  def createCursor(self):
    if self.mode==1 or self.mode==3:
      print '-> action before creating cursor'
    cursor=DefaultCursor(self.counting)
    ...
class NewConnection(Connection):
  def createCursor(self):
    cursor=NewCursor(self.counting)
    ...
```

## 7.2 Frame Solution

The following process should be performed for a frame solution:

a) ensure that the behavior to be moved up the frame hierarchy is in an INSERT_BEFORE or INSERT_AFTER part of the adaptee frame, not in a REPLACE part
b) create a new adapting frame for counting, and move the counting behavior into this frame
c) create wrapper frames for combinations

In detail:

a) This process can be performed with less effort if no negative variabilities are concerned, as mentioned in Section 6.2. Thus, the optional step 6.2.d) should have been performed in order to turn the negative variability into a positive one. For example, one advantage of this step is that the number of frames has been reduced, so that in the following process less frames have to be taken into account. Another reason is that interdependencies between INSERT_BEFORE and INSERT_AFTER frames tend to be more manageable than those of REPLACE frames because the former just perform one action (add), whereas the latter performs two actions (replace means remove and add). A third reason is that in fp's implementation, REPLACE commands in different frames are processed in the order the frames were adapted, so that a generated product would be dependent on the ADAPT order. This is not the case for INSERT_BEFORE and INSERT_AFTER.

b) Move the INSERT_AFTERs concerned with counting (cursor_attributes, count_inc, count_print) from newdb.frame (Listing 17) into their own frame counting.frame (Listing 18):

Listing 17:
SPC newdb.frame in scenario 4 (compare to Listing 14)

```
ADAPT db.frame
REPLACE db_print
    print 'new db:',
```

Listing 18:
SPC counting.frame in scenario 4 (compare to Listing 14)

```
ADAPT db.frame
INSERT_AFTER cursor_attributes
  count=0
INSERT_AFTER count_inc
    self.count=self.count+1
INSERT_AFTER count_print
    print self.count,
```

The two databases without counting and the default one with counting can now be test-run with db.frame, newdb.frame and counting.frame:

```
> fpy db.frame test.py
...
default db: connection init user/password@database
default db: cursor init
default db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

```
> fpy newdb.frame test.py
...
new db: connection init user/password@database
new db: cursor init
new db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

```
> fpy counting.frame test.py
...
default db: connection init user/password@database
default db: cursor init 1
default db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])
```

c) For combining the new database with counting, write a wrapper frame (counting_newdb.frame, Listing 19) which adapts both newdb.frame (Listing 17) and counting.frame (Listing 18), and test-run it:

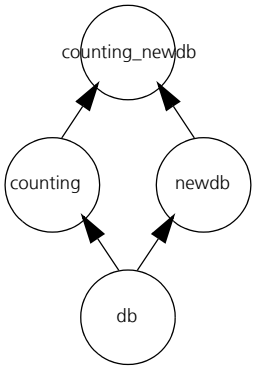
Listing 19: SPC counting_newdb.frame in scenario 4

```
ADAPT newdb.frame
ADAPT counting.frame
```

```
> fpy counting_newdb.frame test.py
...
new db: connection init user/password@database
new db: cursor init 1
new db: cursor execute SELECT * FROM table WHERE id<9
execute returned: ([1,'Mayer','04-06-2000'],[5,'Smith','11-10-2001'])
```

Figure 26 depicts the resulting frame hierarchy. As before, each frame can be used as the SPC for implementing a specific product.

Figure 26: Frame hiererchy in scenario 4



# 8 Analysis and Outlook

In this chapter, the results gained in the individual scenarios of this case study are compared, followed by a summary and an outlook of future research activities.

## 8.1 Analysis

The implementation effort for coping with the aforementioned evolution scenarios differs considerably for the conventional and the frame solutions. In order to compare these, we evaluated the following physical and logical entities:

At a coarse physical level, we measured the size of the python files in characters and the effective source lines of code (ESLOC). At the logical level, the number of classes and number of methods were considered, and at a finer physical level the number of added, similar and changed lines for each evolution scenario. An added line was considered similar for two products, if most of its contents and semantics were similar, e.g. in Section  the newly introduced lines cursor= DefaultCursor() and cursor=NewCursor() were counted as two similar lines.

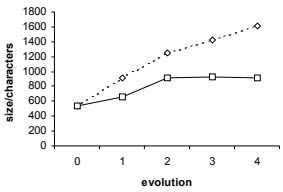
Table 4 depicts the results:

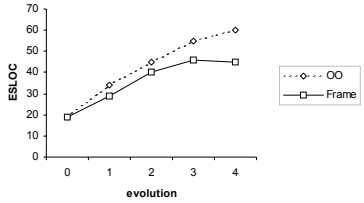
Table 4: Comparison of source code metrics during the evolutions

| evolution (ch.) | 0 (3) | 1 (4.1) | 1 (4.2) | 2 (5.1) | 2 (5.2) | 3 (6.1) | 3 (6.2) | 4 (7.1) | 4 (7.2) |
|---|---|---|---|---|---|---|---|---|---|
| size/characters | 536 | 910 | 659 | 1250 | 909 | 1427 | 933 | 1610 | 915 |
| ESLOC | 19 | 34 | 29 | 45 | 40 | 55 | 46 | 60 | 45 |
| # classes | 2 | 4 | 2 | 6 | 2 | 6 | 2 | 9 | 2 |
| # methods | 4 | 7 | 4 | 10 | 4 | 14 | 4 | 14 | 4 |
| # added lines | | 18 | 10 | 11 | 10 | 10 | 13 | 13 | 9 |
| # similar lines | | 10 | 1 | 2 | | 4 | | 6 | |
| # changed lines | | 3 | | | | 2 | | 7 | |

Figure 27 shows how the size of the source code and the ESLOC evolved for the two techniques. The frame technique results in a significantly lower size of code, and especially for the later evolution scenarios the code size growth is well below the one employed conventionally.

Figure 27:
Evolution of coarse-grained physical parameters




A statistical analysis of thousands of real-world software projects [PM92] derived the following equation for a software productivity index:

$$PI = \log_{1,272}\left[\frac{ESLOC}{t^{4/3} \times \left(\frac{effort}{B}\right)^{1/3}}\right] - 26,6$$

In this formula, effort means staff months, and B is an ESLOC related skill factor.

According to the equation, if a project is performed with the same parameters for PI, effort and B, its duration is:

$$t_1 = t_0 \times \left(\frac{ESLOC_1}{ESLOC_0}\right)^{3/4}$$

This results in the following time-savings for the frame implementation in the four scenarios, compared to the conventional object-oriented solution:

| evolution | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| time saving $1-(t_{frame}/t_{oo})$ | 11% | 8% | 13% | 19% |

As a result, the frame-based solution in this case study provides an average time-saving of 13% compared to an object-oriented one, with a standard deviation of 4%.

Figure 14 shows the evolution of the logical entities. During evolution the conventional solution normally results in a growth both in the number of classes as well as the number of methods, whereas in the frame solution both of these parameters can be kept constant, i.e. at the same level as in the initial version.

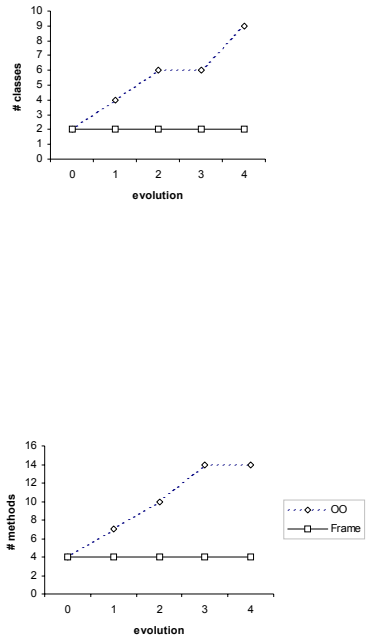
Another observation is that frame technology minimizes the number of possibly non-behavior-preserving changes in existing frames (only VPs are introduced which are guaranteed to be behavior-preserving). Changes can often be limited to changing just the adapting frame, and not the adaptee.

On the other hand, conventional evolution implementations often result in pairs of changes, e.g. extracting some partial functionality will result in creating a new method and calling it. Also, for coping with the same evolution scenario, a wider range of conventional refactorings (like Add Parameter, Introduce Subclass or Form Template Method) is necessary compared to frame technology.

With frames, non-intrusively changing contracts (i.e. pre- or postconditions) can be accomplished easily by inserting a VP and adapting it, but without introducing e.g. wrapper objects.

In the context of product lines frames offer the advantage that the number of products which can be built can be obtained easily by counting the specification frames. On the other hand, this is not that easy in the conventional case, because there can be a mix of static and dynamic configurations at the same time (e.g. class hierarchies and configuration parameters), with dual inheritance hierarchies complicating this issue.

Figure 28:
Evolution of logical parameters




## 8.2 Summary and Outlook

This report presented a case study of frame technology for implementing software product lines. In the remaining sections, we will summarize the major results gained in the case study and give an outlook on outstanding issues in this area of product line implementation technologies.

### 8.2.1 Summary

Software product line engineering presents a means for systematically developing and maintaining similar products. At the implementation level, three technological dimensions can be distinguished: component technologies, configuration management as well as programming languages including generators [MKLP02].

In this report, we evaluated the emerging generative approach of frame technology in a case study. We compared four evolutions of a database wrapper component using conventional object-oriented techniques with applying the

generative reuse technique. Throughout the case study, a plain frame processor developed for handling only the essential frame technology concepts was used.

In each evolution, the frame solutions were illustrated as a sequence of steps, and the difficulties in using the two techniques for solving the particular problems were sketched.

In order to compare the overall implementation efforts of both approaches, metrics of both physical and logical entities were taken. The result of this case study is that frame technology, even in the minimalistic way employed here, offers significant advantages for implementing product lines over conventional object-oriented techniques

### 8.2.2 Outlook

In this report, we started evaluating generative techniques for implementing product lines by means of a frame technology case study. Both on a concrete and abstract level, more work remains to be done.

On a concrete level, the frame technology approach should be performed in practice in order to evaluate its fitness to real-world problems and its scaleability. In this context, we could also examine if the simple frame processor lacks important features or introduces unnecessary complexity.

Moreover, it should be explored how frame technology relates to or combines with other programming language techniques for implementing variability like conditional compilation, collaboration-based design or aspect-orientation, or even other the product line implementation technologies like component technologies or configuration management.

On a more abstract level, general idioms employed in frame technologies should be collected, resulting in more general sequences of steps than those described in this paper so far. Such a categorization is sketched in Appendix C. Moreover, this could also be done for other generative programming techniques, resulting in a technique-independent pattern-language.

Finally, this approach could be evaluated with regard to existing studies of generativity in other scientific areas like mathematics [Ley01] or architecture [Ale03].

# A C++ Implementations of Database Wrapper

Appendix A shows C++ implementations of the database wrapper offering the same functionality as the Python code presented before. Note that only those files are shown here that differ from the aforementioned ones, so that the language-independent SPCs (Listings 9 and 19) are not shown again. Also, implementations of intermediate states (like those in Listings 11, 12 and 15) are not presented.

As indicated in chapter 3, the reason for having chosen Python code in the running examples is that is more compact and readable than e.g. C++ code, which becomes obvious when comparing the size and structure the corresponding implementations.

Listing 20:
Test driver main.cpp
(see Listing 1)

```
#include "db.h"

void test() {
  cout<<endl;
  Connection* connection=createConnection("user","password","database");
  Cursor* cursor=connection->createCursor();
  string records=cursor->execute("SELECT * FROM table WHERE id<9");
  cout<<"execute returned: "<<records<<endl;
  cout<<endl;
}

int main()
{ test(); }
```

Listing 21:
Initial implementation
db.h (see Listing 2)

```
#ifndef DB_H
#define DB_H

#include<string>
using namespace std;

class Cursor {
  static int count;
public:
  Cursor() {
   count++;
   cout<<"default db: ";
   cout<<"cursor init "<<count<<endl;
  }
  string execute(string s) {
    cout<<"default db: ";
    cout<<"cursor execute "<<s<<endl;
    return "1,Miller,04-06-2000\t5,Smith,11-10-2001";
  }
};

class Connection {
public:
  Connection(string s) {
    cout<<"default db: ";
    cout<<"connection init "<<s<<endl;
  }
  Cursor* createCursor() {
    Cursor* cursor=new Cursor;
    return cursor;
  }
};

Connection* createConnection(string user,string password,string database) {
  return new Connection(user+"/"+password+"@"+database); }

#endif
```

Listing 22: Conventional implementation db.h of scenario 1 (see Listing 3)

```
#ifndef DB_H
#define DB_H

#include<string>
using namespace std;

class Cursor {
  static int count;
public:
  void init() {
    count++;
    db_print();
    cout<<"cursor init "<<count<<endl;
  }
  virtual void db_print()=0;
  string execute(string s) {
    db_print();
    cout<<"cursor execute "<<s<<endl;
    return "1,Miller,04-06-2000\t5,Smith,11-10-2001";
  }
};

class DefaultCursor:public Cursor {
public:
  DefaultCursor()
  { init(); }
  void db_print()
  { cout<<"default db: "; }
};

class NewCursor:public Cursor {
public:
  NewCursor()
  { init(); }
  void db_print()
  { cout<<"new db: "; }
};

class Connection {
public:
  void init(string s) {
    db_print();
    cout<<"connection init "<<s<<endl;
  }
  virtual void db_print()=0;
  virtual Cursor* createCursor()=0;
};

class DefaultConnection:public Connection {
public:
  DefaultConnection(string s)
  { init(s); }
  Cursor* createCursor() {
    Cursor* cursor=new DefaultCursor;
    return cursor;
  }
  void db_print()
  { cout<<"default db: "; }
};
```

Listing 22 (continued)

```
class NewConnection:public Connection {
public:
  NewConnection(string s)
  { init(s); }
  Cursor* createCursor()
  {
    Cursor* cursor=new NewCursor;
    return cursor;
  }
  void db_print()
  { cout<<"new db: "; }
};

Connection* createConnection(string user,string password,string database) {
  return new NewConnection(user+"/"+password+"@"+database); }
```

Listing 23:
Reusable frame
db.frame in scenario 1
(see Listing 3)

```
OUTFILE db.h
#ifndef DB_H
#define DB_H

#include<string>
using namespace std;

class Cursor {
  static int count;
public:
  Cursor() {
    count++;
VP db_print
    cout<<"default db: ";
VP_END
    cout<<"cursor init "<<count<<endl;
  }
  string execute(string s) {
VP db_print
    cout<<"default db: ";
VP_END
    cout<<"cursor execute "<<s<<endl;
    return "1,Miller,04-06-2000\t5,Smith,11-10-2001";
  }
};

class Connection {
public:
  Connection(string s) {
VP db_print
    cout<<"default db: ";
VP_END
    cout<<"connection init "<<s<<endl;
  }
  Cursor* createCursor() {
    Cursor* cursor=new Cursor;
    return cursor;
  }
};

Connection* createConnection(string user,string password,string database) {
  return new Connection(user+"/"+password+"@"+database); }

#endif
```

Listing 24:
SPC newdb.frame in
scenario 1 (see Listing 4)

```
ADAPT db.frame
REPLACE db_print
    cout<<"new db: ";
```

Listing 25: Conventional implementation (part) of scenario 2 (see Listing 5)

```
...
class DefaultConnection:public Connection
{
public:
  enum insert_type { nothing,before,after,before_and_after } type;
  DefaultConnection(string s,insert_type t)
    :type(t)
  { init(s); }
  Cursor* createCursor() {
    if(type==before || type==before_and_after)
      cout<<"-> action before creating cursor"<<endl;
    Cursor* cursor=new DefaultCursor;
    if(type==after || type==before_and_after)
      cout<<"-> action after creating cursor"<<endl;
    return cursor;
  }
  void db_print()
  { cout<<"default db: "; }
};

...
Connection* createConnection(string user,string password,string database) {
  return new DefaultConnection(user+"/"+password+"@"+database,
    DefaultConnection::before_and_after); }
...
```

Listing 26: Reusable frame db.frame (part) in scenario 2 (see Listing 6)

```
...
class Connection {
public:
  Connection(string s) {
VP db_print
    cout<<"default db: ";
VP_END
    cout<<"connection init "<<s<<endl;
  }
  Cursor* createCursor() {
VP create_cursor
    Cursor* cursor=new Cursor;
VP_END
    return cursor;
  }
};
...
```

Listing 27: SPC before.frame (see Listing 7)

```
ADAPT db.frame
INSERT_BEFORE create_cursor
    cout<<"-> action before creating cursor"<<endl;
```

Listing 28: SPC after.frame (see Listing 8)

```
ADAPT db.frame
INSERT_AFTER create_cursor
    cout<<"-> action after creating cursor"<<endl;
```

Listing 29: Conventional implementation (part) of scenario 3 (see Listing 10)

```
...
class Cursor {
public:
  void init() {
    cursor_init();
    db_print();
    cout<<"cursor init ";
    count_print();
    cout<<endl;
  }
  virtual void db_print()=0;
    string execute(string s) {
    db_print();
    cout<<"cursor execute "<<s<<endl;
    return "1,Miller,04-06-2000\t5,Smith,11-10-2001";
  }
  virtual void cursor_init()=0;
  virtual void count_print()=0;
};

class DefaultCursor:public Cursor {
public:
  DefaultCursor()
  { init(); }
  void db_print()
  { cout<<"default db: "; }
  void cursor_init()
  {}
  void count_print()
  {}
};

class NewCursor:public Cursor {
  static int count;
public:
  NewCursor()
  { init(); }
  void db_print()
  { cout<<"new db: "; }
  void cursor_init()
  { count++; }
  void count_print()
  { cout<<count; }
};

...
Connection* createConnection(string user,string password,string database) {
  return new DefaultConnection(user+"/"+password+"@"+database,
    DefaultConnection::before_and_after); }
...
```

Listing 30:
Reusble frame db.frame in scenario 3 (see Listing 13)

```
...
class Cursor {
VP cursor_attributes
VP_END
public:
  Cursor() {
VP cursor_init
VP_END
VP db_print
  cout<<"default db: ";
VP_END
  cout<<"cursor init ";
VP count_print
VP_END
  cout<<endl;
  }
  string execute(string s)
  {
VP db_print
    cout<<"default db: ";
VP_END
    cout<<"cursor execute "<<s<<endl;
    return "1,Miller,04-06-2000\t5,Smith,11-10-2001";
  }
};
...
```

Listing 31:
SPC newdb.frame in scenario 3 (see Listing 14)

```
ADAPT db.frame
REPLACE db_print
    cout<<"new db: ";
INSERT_BEFORE cursor_attributes
    static int count;
INSERT_BEFORE cursor_init
    count++;
INSERT_BEFORE count_print
    cout<<count;
```

Listing 32: Conventional implementation (part) of scenario 4 (separation, see Listing 16)

```
...
class Counting {
public:
  virtual void inc_count()=0;
  virtual void count_print()=0;
};

class Counting_:public Counting {
  static int count;
public:
  void inc_count()
  { count++; }
  void count_print()
  { cout<<count; }
};

class NoCounting:public Counting {
  void inc_count()
  {}
  void count_print()
  {}
};
...

class Connection {
protected:
  Counting* counting;
public:
  void init(string s,Counting* c) {
    counting=c;
    db_print();
    cout<<"connection init "<<s<<endl;
  }
  virtual void db_print()=0;
  virtual Cursor* createCursor()=0;
};

class DefaultConnection:public Connection {
public:
  enum insert_type { nothing,before,after,before_and_after } type;
  DefaultConnection(string s,Counting* c,insert_type t)
    :type(t)
  { init(s,c); }
  Cursor* createCursor() {
    if(type==before || type==before_and_after)
      cout<<"-> action before creating cursor"<<endl;
    Cursor* cursor=new DefaultCursor(counting);
    if(type==after || type==before_and_after)
      cout<<"-> action after creating cursor"<<endl;
    return cursor;
  }
  ...
};
```

Listing 32 (continued)

```
class NewConnection:public Connection {
public:
  NewConnection(string s,Counting* c)
  { init(s,c); }
  Cursor* createCursor() {
    Cursor* cursor=new NewCursor(counting);
    return cursor;
  }
  ...
```

Listing 33:
SPC newdb.frame in scenario 4 (see Listing 17)

```
ADAPT db.frame
REPLACE db_print
  cout<<"new db: ";
```

Listing 34:
SPC counting.frame in scenario 4 (see Listing 18)

```
ADAPT db.frame
INSERT_BEFORE cursor_attributes
  static int count;
INSERT_BEFORE cursor_init
  count++;
INSERT_BEFORE count_print
  cout<<count;
```

# B Preprocessor Implementations of Database Wrapper

Appendix B shows alternative implementations of some of the Python frame examples using Python and the C preprocessor. Compared to the frame solution, the code presented here has a comparable size, but is split into more files, resulting in higher maintainance effort.

The file db shown in Listing 35 can be preprocessed by the C preprocessor, resulting in a Python source file db.py. For a specific product, the file (or symbolic link) db_print is then exchanged by one of the files shown in Listing 36 or 37.

Listing 35: Implementation db of scenario 1 using a preprocessor (compare to Listing 4a)

```
class Cursor:
  count=0
  def __init__(self):
    self.count=self.count+1
#include "db_print"
    print 'cursor init',self.count
  def execute(self,s):
#include "db_print"
    print 'cursor execute',s
    return ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])

class Connection:
  def __init__(self,s):
#include "db_print"
    print 'connection init',s
  def createCursor(self):
    cursor=Cursor()
    return cursor

def createConnection(user,password,database):
  return Connection(user+'/'+password+'@'+database)
```

Listing 36: File default_db_print

```
    print 'default db:'
```

Listing 37: File new_db_print

```
    print 'new db:'
```

Listing 38: Implementation of scenario 2 using a preprocessor (compare to Listing 6)

```
class Cursor:
  count=0
  def __init__(self):
    self.count=self.count+1
#include "db_print"
    print 'cursor init',self.count
  def execute(self,s):
#include "db_print"
    print 'cursor execute',s
    return ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])

class Connection:
  def __init__(self,s):
#include "db_print"
    print 'connection init',s
  def createCursor(self):
#include "before_create_cursor"
    cursor=Cursor()
#include "after_create_cursor"
    return cursor

def createConnection(user,password,database):
  return Connection(user+'/'+password+'@'+database)
```

Here, the files before_create_cursor and after_create_cursor can be empty, or have the implementation shown in Listing 39 and 40, respectively:

Listing 39: File before_create_cursor

```
print '-> action before creating cursor'
```

Listing 40: File after_create_cursor

```
print '-> action after creating cursor'
```

Listing 41:
Implementation of scenario 3 using a preprocessor (compare to Listing 13)

```
class Cursor:
#include "cursor_attributes"
  def __init__(self):
#include "cursor_init"
#include "db_print"
#include "count_print"
  def execute(self,s):
#include "db_print"
    print 'cursor execute',s
    return ([1,'Miller','04-06-2000'],[5,'Smith','11-10-2001'])

class Connection:
  def __init__(self,s):
#include "db_print"
    print 'connection init',s
  def createCursor(self):
    cursor=Cursor()
    return cursor

def createConnection(user,password,database):
  return Connection(user+'/'+password+'@'+database)
```

Listing 42:
File cursor_attributes

```
      count=0
```

Listing 43:
File cursor_init

```
      self.count=self.sount+1
```

Listing 44:
File count_print

```
       print self.sount
```

# C A Collection of Frame Technology Idioms

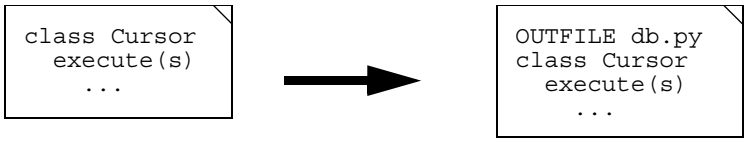
In this appendix we list several small-scale idioms which were employed in the case study in section 4 to 7, as the start of a catalog of frame technology idioms. These will be presented in a similar format as Martin Fowler's refactorings [Fow99], consisting of the following six parts:

- Each idiom will have a name, describing which action is performed to perform the idiom. The name will resemble those of the refactorings.
- Next, we will give a short summary, indicating in which situation the idiom should be used and what it does. This is often illustrated by a diagram.
- The motivation section will describe why the idiom is useful and when it should not be used.
- The mechanics give a stepwise description of how the idiom is performed.
- The example section will refer to those chapters in this document where the idiom was applied.
- Finally, related idioms will hint at other related idioms in this catalog.

## C.1 Convert Physical Unit into Frame

A physical unit should evolve into a product line.

*Create a frame which contains the name of the original physical unit as the output file, and copy the contents of the original physical unit into the frame.*




db.py db.frame

**Motivation:**

You have a physical unit of a single system which is going to evolve into several similar, slightly different (e.g. customer-specific) units. Maintaining single systems individually or using traditional techniques for accomplishing this like macros or polymorphism turn out to be too complicated to manage the unit's concurrent evolution.

Make sure that the physical unit should not just be used-as-is, which would only require variations to occur at run-time, e.g. by mechanisms such as subtype polymorphism.

**Mechanics:**

- Create a new physical unit, name it differently than already existing units.
- Indicate which original physical unit is going to be regenerated. In FP, this is done with the OUTFILE command in the frame's first line.
- Copy the contents of this original unit into the new one.
- Backup the original physical unit.
- Call the frame processor for the newly created frame, test-run the system.

**Example:** Ch. 4.2 step a

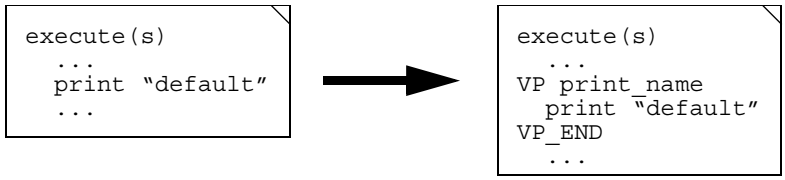
**Related Patterns:**

- To distinguish commonalities and variabilities in the newly created frame, use Introduce Variation Point
- To handle several frames, use Introduce Wrapper Frame

## C.2 Introduce Variation Point

Common and variable text areas within a frame have to be distinguished.

*Mark the varying area with frame commands indicating the start and end of the variation point.*




**Motivation:**

You applied Convert Physical Unit into Frame in order to evolve your single system into a product line of slightly different systems. Or, you already have a framed system and during its evolution certain new parts turn out to become variant among the systems. Hence, you need a way to distinguish between those parts which stay the same among different products and those that vary. This is done by explicitly marking the variable, exceptional text parts[1].

Do not introduce VPs for run-time variations or those that are only related to single systems, but not to different products. Also, do not clutter the frame with too-fine-gained VPs; try to merge those VPs belonging to related variations in this case.

**Mechanics:**

- (fp-specific:) If a line contains both common and variable parts, split it either by refactoring or by introducing line separation characters, like the '\' in C. Test-run the system.
- Insert a VP/VP_END pair of lines before and after the variant section, which can be empty. Name the variation point appropriately, and do not overlap several VPs.
- Test-run the system.

**Examples:** Ch. 4.2 step b, ch. 5.2 step a, ch. 6.2 step b

1 A variable text part, called default frame-text in [Bas96], will often be called *the default* in the remainder of this section.

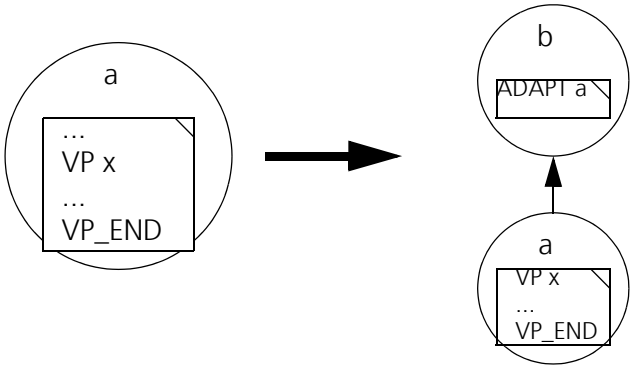
**Related Patterns:**

- In order to prepare for changes in the variation points, use Introduce (Parent) Frame.
- The text associated with the VP serves as the default. If it later turns out to be used in a more general context, Push Default into Child Frame.

## C.3 Introduce Parent Frame

Specific changes to the variable text areas have to be stored without altering the existing text.

*Create a new frame which refers to the frame that contains the variable text area.*




**Motivation:**

You applied Introduce Variation Point for distinguishing a text area within a frame which is going to vary across different products from other text areas which remain common for all products. Or, you want to introduce a new product based on an already existing framed one. Thus, you need to introduce a new location containing the deviations in the new product.

Only introduce a parent frame if there is already an existing VP to be altered; otherwise, apply Introduce Variation Point first. For already existing frames, make sure that no ancestor frame exists which already covers the intended alteration. In these cases, consider using that frame as-is, or introduce a parent frame for that one.

In order to extend an existing product, only introduce parent frames, not child frames, as this would require changing existing frames, and more general, but not more concrete alterations would be introduced.

**Mechanics:**

- Create a new physical unit, name it differently than already existing units.
- Reference the frame containing the variable text. In fp, insert the frame command ADAPT *f-name*, with f-name naming the frame which contains the VP.
- Run the frame processor on the newly created frame, which should produce the same results as running it on the adapted one.

**Examples:** Ch. 4.2 step c, ch. 5.2 step b, ch. 6.2 step c, ch. 7.2 step b

**Related Patterns:**

- After applying Introduce Parent Frame, the ancestor frame can customize its adaptee's default in one of two ways: Add to Default or Replace Default
- Introduce Parent Frame is the simplest way of performing Extract Frame.
- If you want to integrate two or more independent frames without changing any of their variation points, use Introduce Wrapper Frame.
- Remove Frame has the opposite effect as Introduce Parent Frame.

## Intermission: Frame Concepts for Expressing Product Line Variabilities

As introduced in Section 1.2 (see Fig. 5), variabilities can be classified into three basic groups, depending on the range of choices offered and the number of possible simultaneous choices. The following discussion deals with how these kinds of variabilities can be expressed with frame technology. For the sake of simplicity, we discuss only behavioral variabilities.

In the simplest case of an optional behavior, one kind of behavior is offered, which can be included or excluded. Frame hierarchies are ordered according to reuse potential, with the more reusable frames at the bottom of the hierarchy (the adaptees), and the less reusable ones in adapting frames further up the hierarchy.

Thus, if the predominant default (for far more than 50% of the products) is that the behavior exists and it only needs to be cancelled in rare cases, then the default behavior should be expressed within a VP in some frame, and an adapting frame would cancel this by ADAPTing the frame and REPLACEing the VP with an empty body. This negative variability mechanism is mentioned in [Bas96, p.105] and in ch. 6.2, steps a - c, where the default behavior of counting is cancelled in the new Cursor.

As already mentioned in ch. 6.2, step d, the alternative is to leave the VP in the adaptee frame empty, and to push the behavior into the adapting frame. This is

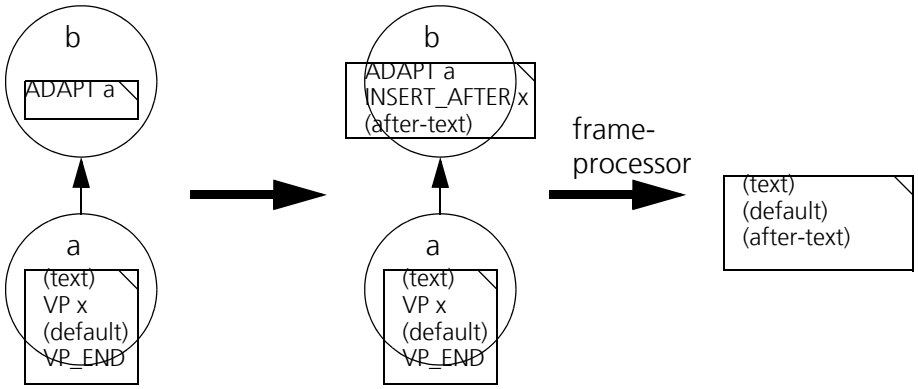
the preferred solution if it is not clear if some functionality should exist as a default for most (>50%) of the products.

The alternative types of variability offer n kinds of behavior, and exactly one (XOR) or 1 or more (OR) can be chosen. The original Netron frame processor offers the SELECT mechanism to handle surplus properties as alternatives in one unit [Bas96, p.172]. In fp, these n kinds of behavior have to be provided in a less localized way: in n different frames.

## C.4 Add to Default

Some text has to be added to a variable text area without altering the existing text.

*In an adapting frame, indicate by a frame command that and for which VP text should be inserted, and include this text.*




**Motivation:**

You have an existing frame, with a variable text marked as a VP, and an adapting frame as a location for more specific alterations. For a specific product, adding customizations have to be introduced without changing the existing default.

Do not perform Add to Default for changing defaults; use Replace Default instead.

**Mechanics:**

- Decide on whether the text to add should be before or after the variation point. With fp, use INSERT_BEFORE in the former case and INSERT_AFTER in the latter.

- Reference the variation point whose default should be added to, in fp by INSERT_BEFORE/AFTER *vp-name*.
- Append the additional text.

**Examples:** Ch. 5.2 step b, ch. 7.2 step b

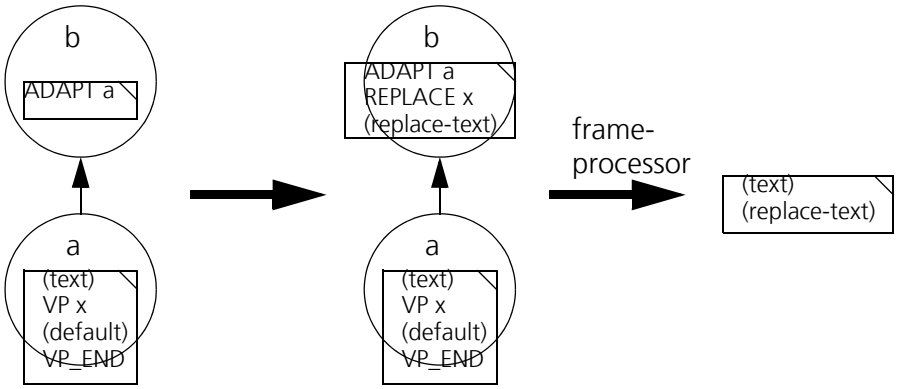
**Related Patterns:**

- Add to Default leaves the original variable text area as-is. However, if the variable text area itself has to be changed, use Replace Default or Remove Default.

## C.5 Replace Default

Variable text has to be substituted without altering the existing text.

*In an adapting frame, indicate by a frame command that and for which VP text should be substituted, and include the replacing text.*




**Motivation:**

You have an existing frame, with a variable text marked as a VP, and an adapting frame as a location for more specific alterations. For a specific product, most or all of the default for the VP needs overwriting without changing the existing default.

Note that OR-variations cannot be expressed with Replace Default, i.e. in contrast to Add to Default, this pattern can only be applied once for a variation point.

Do not perform Replace Default for adding to an existing default, which would involve duplicating most of it; use Add to Default instead.

**Mechanics:**

- Reference the variation point whose default should be replaced, in fp by REPLACE *vp-name*.
- Append the replacing text.

**Example:** Ch. 4.2 step c

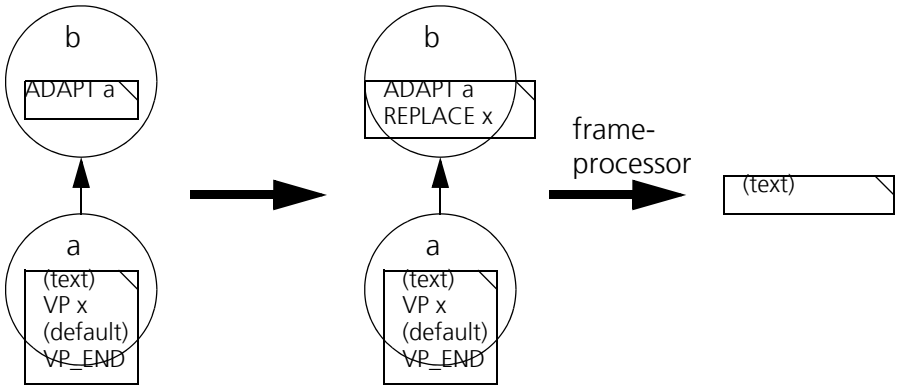
**Related Patterns:**

- If the replacing text is empty, this pattern becomes Remove Default.
- Use this negative variability mechanism sparingly. Instead, Change Negative Variability into Positive Variability, and then apply Add to Default.

## C.6 Remove Default

Variable text has to be removed without altering the existing text.

*In an adapting frame, indicate by a frame command that and for which VP text should be removed, and include no replacing text.*




**Motivation:**

You have an existing frame, with a variable text marked as a VP, and an adapting frame as a location for more specific alterations. For a specific product, all of the default for the VP has to be removed without changing the existing default.

**Mechanics:**

- Reference the variation point whose default should be replaced, in fp by REPLACE *vp-name*.
- Append nothing. If necessary later, start with a new frame command immediately afterwards.

**Related Patterns:**

- Remove Default is the trivial case of Replace Default when there is no replacing text.

## C.7 Remove Frame

A product is not required anymore.

*Remove its SPC.*

**Motivation:**

A product expressed by an SPC becomes obsolete. By removing the SPC, other products will not be affected.

**Example:** Ch. 6.2 step d

**Related Patterns:**

- Introduce Parent Frame has the opposite effect as Remove Frame.

# 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

[Ale03] C. Alexander. The Nature of Order. Oxford University Press, 2003

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

[Bec99] K. Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999

[Fow99] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999

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

[Ley01] M. Leyton. A Generative Theory of Shape. Springer-Verlag, 2001

[MS02] D. Muthig, K. Schmid. Balancing Evolution with Revolution to Optimize Product Line Development. Submitted to NATO Symposium, 2002

[MKLP02] D. Muthig, S. Kettemann, R. Laqua, T. Patzke: Technology Dimensions of Product Line Implementation Approaches. IESE Technical Report No. 051.02/E, September 2002

[PM92] L. H. Putnam, W. Myers. Measures for Excellence: Reliable Software on Time, Within Budget. Yourdon Press, 1992

[PM02] T. Patzke, D. Muthig. Product Line Implementation Technologies. Programming Language View. IESE Technical Report No. 057.02/E, October 2002

[XVCL] Homepage of XVCL: http://fxvcl.sourceforge.net

# Document Information

| | |
|---|---|
| Title: | Product Line Implementation with Frame Technology: A Case Study |
| Date: | March 18, 2003 |
| Report: | IESE-018.00/E |
| Status: | Final |
| Distribution: | Public |