

University of Magdeburg  
Department of Computer Science



## Master's Thesis

### **Optional Composition - A Solution to the Optional Feature Problem?**

Author:

Constanze Adler

Matriculation Number:

184594

December 17th, 2010

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake  
Dr.-Ing. Christian Kästner

Institute of Technical & Business Information Systems

**Adler, Constanze:**

*Optional Composition - A Solution to the Optional Feature Problem?*

Master's Thesis, University of Magdeburg, 2010.

# Acknowledgements

At this point, I would like to thank Christian Kästner for advising this thesis. Thank you for the discussions about this topic and helpful suggestions. Special thanks for reading all the drafts of this thesis and the critical view, which helped to improve this thesis.

I would like to thank my fellow student Christian Becker for the fantastic time during the study. Furthermore, I thank him for his compiler and the introductive help with the handling of it. Also, I would like to thank for the review of drafts of this thesis, which was much helpful.

Many thanks to Thomas Thüm, who provided the  $\text{\LaTeX}$ -template for this thesis and the help with the feature models in FeatureIDE.

Special thanks to my husband, Simon Adler, who supported me a lot after the birth of our little daughter, so I was able to finish this thesis. He also read drafts of this thesis and his criticism was very helpful for the improvement of this thesis.



# Contents

<b>List of Figures</b>	<b>viii</b>
<b>List of Tables</b>	<b>ix</b>
<b>List of Code Listings</b>	<b>xii</b>
<b>List of Acronyms</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>5</b>
2.1 Software Product Lines . . . . .	5
2.2 Software Product Line Implementations . . . . .	8
2.2.1 Preprocessor Approach / Conditional Compiling . . . . .	9
2.2.2 Feature-Oriented Programming . . . . .	10
2.2.3 Aspect-Oriented Programming . . . . .	14
2.2.4 Feature Interaction . . . . .	15
2.3 Compiler . . . . .	16
<b>3 Problem Statement</b>	<b>19</b>
3.1 Optional Feature Problem . . . . .	19
3.2 Software Derivatives . . . . .	20
3.3 Preprocessor . . . . .	24
3.4 From Optional Weaving to Optional Composition . . . . .	26
3.4.1 Optional Weaving . . . . .	26
3.4.2 Optional Composition . . . . .	27
3.4.2.1 Explicit Optional Composition . . . . .	28
3.4.2.2 Implicit Optional Composition . . . . .	29
<b>4 Compiler Extension</b>	<b>31</b>
4.1 FOP Compiler . . . . .	31
4.2 Extension for Explicit Optional Composition . . . . .	33
4.2.1 Design . . . . .	33
4.2.2 Implementation . . . . .	34
4.3 Extension for Implicit Optional Composition . . . . .	36
4.3.1 Design . . . . .	36

4.3.2	Implementation . . . . .	36
<b>5</b>	<b>Case Study</b>	<b>41</b>
5.1	Expression Problem . . . . .	41
5.1.1	Software Derivatives . . . . .	43
5.1.2	Preprocessor approach . . . . .	47
5.1.3	Optional Composition . . . . .	49
5.2	Chat SPL . . . . .	53
5.2.1	Software Derivatives . . . . .	55
5.2.2	Preprocessor Approach . . . . .	60
5.2.3	Optional Composition . . . . .	62
<b>6</b>	<b>Discussion</b>	<b>65</b>
6.1	Comparison . . . . .	65
6.1.1	Separation of Concerns . . . . .	66
6.1.2	Code Replication . . . . .	67
6.1.3	Variability . . . . .	68
6.1.4	Additional Effort . . . . .	69
6.1.5	Summary . . . . .	71
6.2	Suggestion . . . . .	72
<b>7</b>	<b>Conclusion</b>	<b>75</b>
	<b>Bibliography</b>	<b>79</b>

# List of Figures

2.1	Feature model . . . . .	6
2.2	An overview of domain and application engineering adopted from [Käs10]	7
2.3	Example collaboration model . . . . .	11
2.4	AHEAD composition . . . . .	12
2.5	Superimposition of FSTs . . . . .	13
2.6	Interacting features [Käs07] . . . . .	16
2.7	Phases of a compiler adopted from [ALSU08] . . . . .	16
3.1	Interaction of <i>Logging</i> and <i>Encryption</i> . . . . .	19
3.2	Feature interaction: <i>Gui - Contactlist</i>   <i>Console - Contactlist</i> . . . . .	20
3.3	Software derivatives adopted from [KAuR <sup>+</sup> 09] . . . . .	22
3.4	Optional Composition . . . . .	22
3.5	Optional Weaving [Käs07] . . . . .	26
3.6	Optional Composition . . . . .	28
4.1	Scheme of the FOP Compiler [Bec10] . . . . .	32
4.2	FOP Compiler changes for explicit optional composition . . . . .	33
4.3	FOP Compiler changes for implicit optional composition . . . . .	36
5.1	Feature model of SPL expression problem . . . . .	42
5.2	Feature model of SPL expression problem - Software Derivatives . . . .	44
5.3	Collaboration model of SPL expression problem - Software Derivatives	45
5.4	Collaboration model of SPL epression problem - Optional Composition	50
5.5	Feature model chat SPL similar to [Sch09] . . . . .	53
5.6	Feature model of chat - Software Derivatives . . . . .	56

5.7	Collaboration model of chat - Software Derivatives . . . . .	57
5.8	Collaboration model of chat - Optional Composition . . . . .	62



# List of Tables

5.1	Feature interactions of SPL expression problem . . . . .	42
5.2	Abbreviations from the feature model . . . . .	43
5.3	Feature interactions of chat SPL . . . . .	54
6.1	Characteristics of the case study expression problem . . . . .	67
6.2	Characteristics of the case study chat . . . . .	67
6.3	Comparison of the approaches . . . . .	71



# List of Code Listings

2.1	An extract of class Message from Chat in object-oriented design . . . . .	9
2.2	An extract of class Message from Chat in preprocessor design . . . . .	10
2.3	An example for the order of feature composition . . . . .	14
2.4	Message - basis implementation . . . . .	15
2.5	Aspect for Encryption . . . . .	15
3.1	Class Gui from feature <i>GUI</i> . . . . .	21
3.2	Class Gui from feature <i>Contactlist</i> . . . . .	21
3.3	<i>Contactlist</i> feature . . . . .	23
3.4	Derivative of <i>GUI</i> and <i>Contactlist</i> . . . . .	23
3.5	<i>GUI</i> Feature - Preprocessor approach . . . . .	25
3.6	<i>Contactlist</i> Feature - Preprocessor approach . . . . .	25
3.7	Aspect Contactlist - Optional Weaving . . . . .	27
3.8	Class Gui from feature <i>Contactlist</i> - Explicit Optional Composition . .	28
3.9	Class Gui from feature <i>Contactlist</i> - Implicit Optional Composition . .	29
4.1	Excerpt from the Scanner file Keywords.flex . . . . .	34
4.2	AST additions for explicit optional composition . . . . .	34
4.3	Parser additions for explicit optional composition . . . . .	34
4.4	Class Program, sequence for explicit optional composition . . . . .	35
4.5	Front end with extension of compiler switch . . . . .	37
4.6	Class Program, sequence for implicit optional composition . . . . .	38
5.1	Interface Exp, feature Core (CK) - Software Derivatives . . . . .	46
5.2	Class Test, Feature Core (CK) - Software Derivatives . . . . .	46
5.3	Interface Exp, Feature Print (CP) - Software Derivatives . . . . .	46
5.4	Class Test, Feature Print (CP) - Software Derivatives . . . . .	46
5.5	Class Test - Preprocessor approach . . . . .	48
5.6	Class Plus - Preprocessor approach . . . . .	49
5.7	Class Plus (feature <i>Plus</i> ) - Optional Composition . . . . .	51

5.8	Class Plus (feature <i>Print</i> ) - Optional Composition . . . . .	51
5.9	Class Plus (feature <i>Eval</i> ) - Optional Composition . . . . .	51
5.10	Class Test (eature <i>Plus</i> ) - Optional Composition . . . . .	51
5.11	Class Gui (Derivative GUI/Encryption) - Software Derivatives . . . . .	58
5.12	Class Gui (Derivative GUI/Logging) - Software Derivatives . . . . .	58
5.13	Class Gui (Derivative GUI/Logging/Encryption) - Software Derivatives	58
5.14	Class Gui (feature <i>Encryption</i> ) - Preprocessor . . . . .	61
5.15	Class Gui (feature <i>Logging</i> ) - Preprocessor . . . . .	61
5.16	Class Gui (feature <i>Encryption</i> ) - Optional Composition . . . . .	63
5.17	Class Gui (feature <i>Logging</i> ) - Optional Composition . . . . .	63

# List of Acronyms

AHEAD	Algebraic Hierarchical Equations for Application Design
AOP	Aspect-Oriented Programming
AST	Abstract Syntax Tree
FOP	Feature-Oriented Programming
FST	Feature Structure Tree
GUI	Graphical User Interface
IDE	Integrated Development Environment
IRC	Internet Relay Chat
ITMD	Inter-Type Member Declaration
LOC	Lines of Code
OOP	Object-Oriented PArogramming
SoC	Separation of Concerns
SPL	Software Product Line
SSL	Secure Sockets Layer



# 1. Introduction

Any types of software like software applications and computer games are developed from providers for several platforms. According to the platforms, different requirements have to be met. Using a personal computer, high resolutions can be used and a lot of primary storage is available. But, porting this software to a smart phone, less hardware resources are available. So functionalities must be deleted or for other platforms functions must be added, e.g. when another input device like a touch pad might be used.

Therefore, software developers have to implement a tailored solution, which just contain the functionalities for the correspondent platform. From the view of an economist, the effort of the adaption to all the required platforms should be minimized. A disadvantage of these tailored software solutions is, that existent source code cannot be completely reused, because the functionalities or concerns are not implemented in software modules. So these functionalities cannot be changed without further effort by other functionalities, for example the control of a keyboard as input device cannot simply be changed into the control of a touch pad.

An approach, which facilitates the generation of variants out of a common code base, is the approach of [Software Product Lines \(SPLs\)](#) [[BCK05](#), [PBvdL05](#)]. So, software can be offered as tailored solutions with several different functionalities and concurrently the source code can be reused efficiently. For the development of such a [SPL](#), an implementation strategy is necessary, which modularizes the concerns, to make the functionalities optional.

Such an implementation technique is the [Feature-Oriented Programming \(FOP\)](#). Here, the concerns are implemented in enclosed modules, called features [[Pre97](#), [BSR03](#)]. These features are the main aspect of [FOP](#). The basic idea of [FOP](#) is the decomposition of a software system in terms of the features it provides [[AK09](#)]. The aim of this decomposition is to create a well-structured software, that can be tailored to the requirements of the user and the application scenario. Typically, from a set of features, many different variants can be generated. These variants share common features, but differ in other features. Let us compare this with the offer at the fast food restaurant

Subway. They are selling sandwiches, with a base of roll covering. The bread can be selected out of 4 varieties. These are the mandatory features. The sorts of salad, like tomatoes and paprika, are arbitrary optional features. So there can be generated some variants, due to the needs of the customer.

An important problem of implementing SPLs is, that features, although being conceptual independent, may interact in their implementation which is namely the optional feature problem, when the interacting features are optional ones. A feature interaction is a situation where two or more features exhibit an unexpected behavior that does not occur when these features are used in isolation [AK09]. Because paprika and tomatoes from the previous example do not interact, let us take the standard example for this problem: a phone with two features *call waiting* and *call forwarding*, besides basic functionalities. Using these features in isolation, they work fine, but when used in combination, it is unclear what to do with an incoming call on a busy line. This call is either forwarded or announced; in either case, the expected behavior of one of the features is compromised [CKMRM03]. There are several solutions to dissolve these dependencies and to recover the volitional variability.

One quite new approach to solve the optional feature problem is the optional weaving approach [LARS05, Käs07], which is the base of the optional composition approach developed in this thesis. During compilation, the optional weaving skips parts of the implementation of a feature, if necessary, when the interaction feature was not selected for generating the variant. This approach tries to solve these dependencies between optional features somewhat automatic.

The unacknowledged question is, whether the optional composition is a solution to the optional feature problem or not and if it has benefits compared to other approaches, that require a higher manual effort. In this thesis, we develop a prototyping tool for the optional composition, evaluate and discuss the results of the optional composition to the other approaches on the basis of case studies. Finally, we compare the optional composition to the other approaches and we discuss the general suitability and try to answer the question whether it is a solution or not and give recommendations for the usage of the selected approaches and future work.

## Structure of the Thesis

First of all, we present the backgrounds of SPLs and their implementations, in Chapter 2. There, we go into detail of the conditional compiling, the FOP and the Aspect-Oriented Programming (AOP) for the implementations of SPLs. Furthermore, we present the backgrounds of the feature interactions and the necessary technical terms for compilers.

In Chapter 3, we go into detail on the optional feature problem. Moreover, we focus on the solutions to this problem and introduce the approach of software derivatives, the preprocessor approach, the optional weaving and the optional composition.

Chapter 4 deals with the necessary technical extensions, we made to the compiler, to make the optional composition possible. We present the compiler and the necessary



extensions by concept and the technical implementation, which are important for the evaluation.

In [Chapter 5](#), we introduce two case studies, the expression problem and a chat software. They are chosen, because they are different in their feature characteristics. The expression problem has a repeating pattern of feature interactions where each feature has a limited amount of functionality. The chat is in contrast a more practical example where the features have to interact in a defined way for the desired functionality of the application. This case is more practical driven. Each approach was implemented for each case, so, at the end we can present reasonable advantages and disadvantages of each approach that could be identified. Here, we focus on the different outcomes of the approaches themselves. In [Chapter 6](#) we discuss and compare the approaches by using objective but also subjective attributes for the comparison, because clarity and error correction in the source code are essential for software development.

We conclude this thesis in [Chapter 7](#) and review the future work.



## 2. Background

In this chapter, we discuss the backgrounds for the understanding of the optional composition. First of all, we present [Software Product Lines \(SPLs\)](#), which are a mean in software development techniques for reusability and to generate customized software variants. Secondly, we introduce implementations for [SPLs](#) where we focus on [Feature-Oriented Programming \(FOP\)](#) and [Aspect-Oriented Programming \(AOP\)](#). At the end we introduce some terms of compiler construction which are needed to understand the compiler extensions we made in this thesis for optional composition.

### 2.1 Software Product Lines

Traditionally, software is developed for one customer, so we can say there is one system one time. So, every software development starts with a requirements analysis regarding to the customer. After designing, implementing and testing phases this development process results in a single software product [[Käs10](#)]. In contrast, [SPL](#) development focuses on multiple similar software systems in one domain with a common code base [[BCK05](#), [PBvdL05](#)]. With such an approach it is possible to generate software solutions which are tailored for different customer needs. Such a tailored solution, generated from a [SPL](#) is called *variant*. Therefore [SPLs](#) provide the opportunity of reusability of code and to generate several variants for several customers within the domain. Thus, the [SPLs](#) gain more and more importance in the last years [[BCK05](#), [PBvdL05](#)].

Let us explain the most relevant terms for [SPLs](#) with the help of the sandwiches of the fast food restaurant Subway which fit a product line. Tim and Mary are going to Subway for eating a chicken fajita sandwich. Tim wants the cheese-oregano bread and Mary likes the honey oat one. After selecting a bread, which is one of the selectable, but mandatory features, they have to make a choice which salad ingredients, also selectable, but optional, features, they would like to have on their sandwich. Mary takes salad, tomatoes, olives, cucumber and paprika whereas Tim leaves out the olives but takes the pepperonis. At the end they have to choose the dressing - Mary fancies honey mustard dressing and Tim the hot Mexican southwest one. So, they get two variants of the

chicken fajita sandwich after finishing their order. To make it clear: The ingredients of the sandwich are the features and the composition of the selected features (ingredients) is the variant (the submarine sandwich) and the domain are submarine sandwiches.

The development of a *SPL* is divided into two categories, the problem space and the solution space [CE00]. The problem space is responsible for the domain analysis and the transformation of the customers requirements into features, whereas the solution space deals with the implementation and the generation of the variants.

## Problem Space

In the problem space the domain of the *SPL* will be defined and analyzed. Therefore, domain specific knowledge is needed to find adequate features. We use *feature* as a single unit of a *SPL* which is a function or characteristic of a software system visible for the user. In literature several definitions for the term feature can be found. Kang et al. [KCH<sup>+</sup>90] define a feature in the following way:

A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.

A list of further definitions can be found in [AK09].

According to the domain engineering within the problem space, features are defined and their dependencies to each other are pointed out. For modeling these features and their dependencies feature models are popular [KCH<sup>+</sup>90, CE00, Bat05].

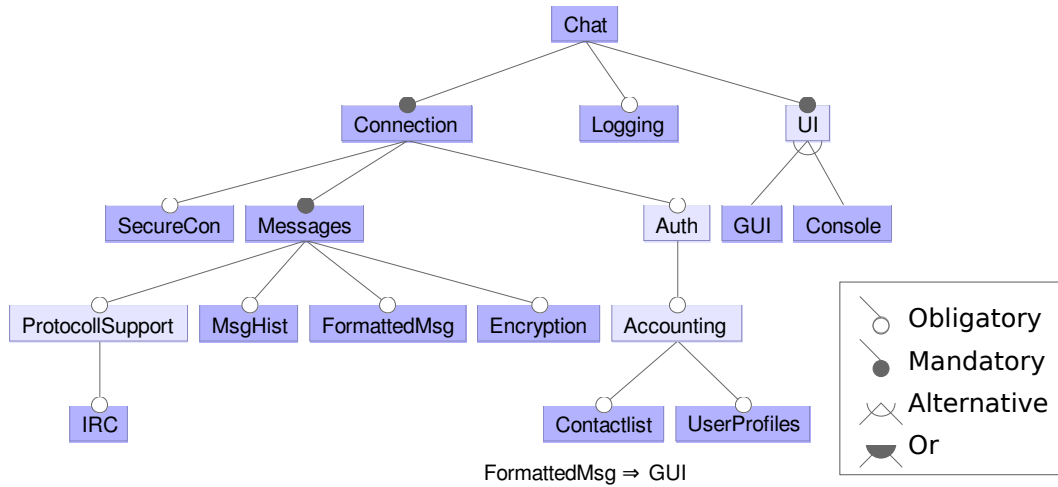


Figure 2.1: Feature model

We show an example of such a feature model in Figure 2.1. The features *Chat*, *Connection* and *UI* are mandatory features, so they are always required. *Logging*, *SecureCon*, *Auth* and all others with a non-colored full circle on top are obligatory features, i.e. these features are optional ones. As user interface (feature *UI*) a *Graphical User Interface* (*GUI*) (feature *GUI*) or a console (feature *Console*) can be chosen alternatively. Some

dependencies can not be mapped to the tree, therefore the tree can be extended with boolean expressions. In this example, the feature *FormattedMsg* which is a colored text message implies the feature *GUI* ( $FormattedMsg \Rightarrow GUI$ ). Features modeled with the darker blue represent an implementation unit, containing the source code of the feature and features modeled with the lighter blue represent organization units for structuring the feature model. Although this example is small sized with its just eight selectable features, 384 variants can be generated.

Besides graphical representation, feature models can be mapped to logic formulas or grammars (e.g. GUIDSL as a grammar [Bat05]) which can be used easily for further processing.

To generate a variant, a choice out of features from the feature model must be made. The feature model can help the customer to make his choice or to ask for more features, which can be added at any time.

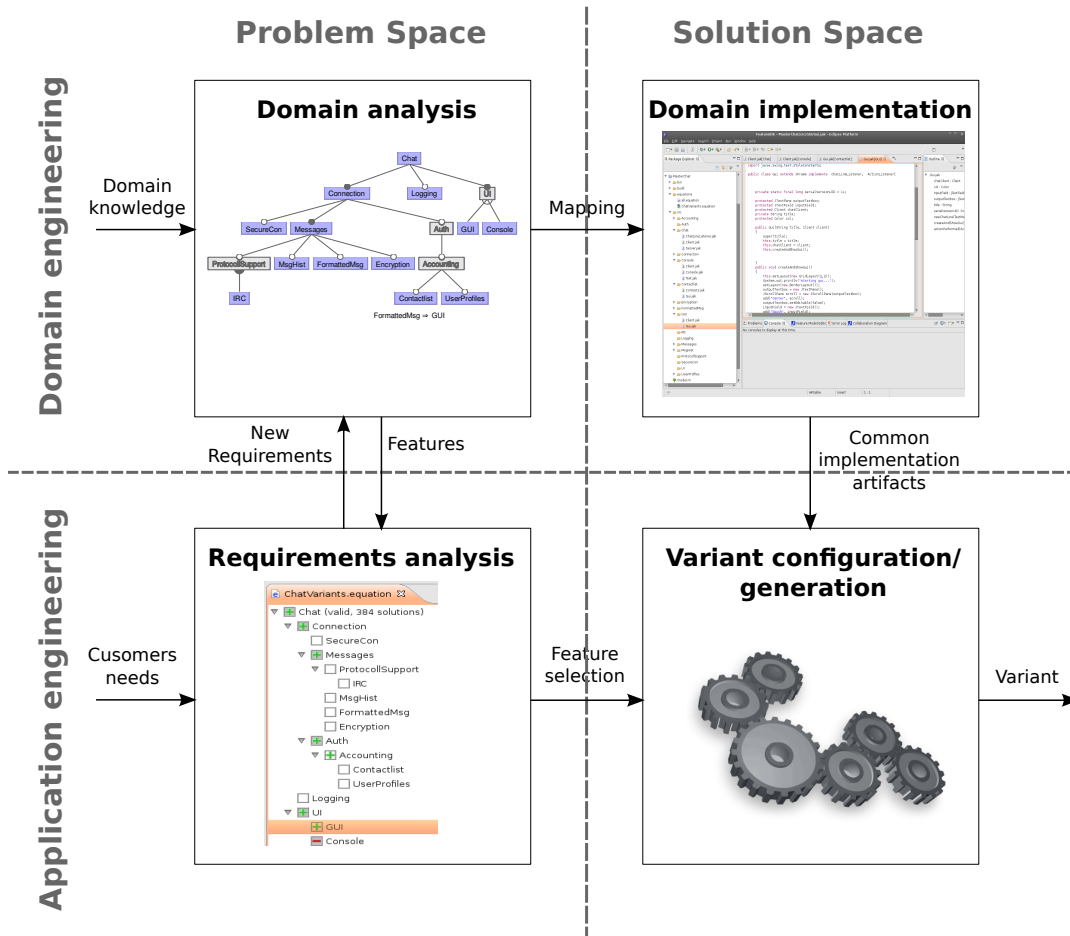


Figure 2.2: An overview of domain and application engineering adopted from [Käs10]

## Solution Space

In the solution space the itemized features will be implemented. Therefore various techniques are possible, which will be introduced in Section 2.2.

To illustrate the separation into problem and solution space, domain and application engineering, we show [Figure 2.2](#).

In the domain engineering within the problem space, the domain knowledge will be transformed during the domain analysis into a feature model. This model will be changed by the requirements analysis in the problem space of application engineering when customers needs are respected. Here, the features according to the needs of the customer are selected. The domain implementation in the solution space of domain engineering is done after the mapping of the domain analysis to it. In this part, features are implemented with the corresponding languages and tools. After the selection of the features within the requirements analysis, the customer specific variant can be generated from the common implementation artifacts in the solution space of the application engineering.

## 2.2 Software Product Line Implementations

In software development, large software is divided into several smaller units called modules. These modules alleviate the implementation and maintainability of the resulting software. This procedure is called [Separation of Concerns \(SoC\)](#) in the literature which was introduced by Parnas and Dijkstra [[Par72](#), [Dij82](#), [Dij97](#)]. The SoC is the major goal of [SPL](#) implementations.

First of all, the object-oriented software design could be used for implementing [SPLs](#), because it is established and has techniques (classes and inheritance) to write modular software [[Fla02](#)]. But it is not always possible to modularize all concerns with the object-oriented approach simultaneously. Kiczales et al. [[KLM<sup>+</sup>97](#)] and Tarr et al. [[TOHS99](#)] show the limitations of the SoC. Mostly concerns belonging to the core functionality of the program cannot be modularized. Furthermore, there are concerns appearing on many points within the program, so that they crosscut it, the so called *crosscutting concerns* [[KLM<sup>+</sup>97](#)]. An example for these crosscutting concerns is the logging functionality of a program.

Let us look at a little example to illustrate the stated problems. The class `Message` in [Listing 2.1](#) contains not only the core functionality of the chat for sending Messages, but also the features *History* and *Encryption*. So, this class contains code of multiple features which is called code tangling [[KLM<sup>+</sup>97](#)]. Furthermore, it is most likely that the encryption algorithms will be used at other points in the software, too. This is called code scattering. These two problems may not be seen in this little example, but looking on more complex programs classes implement much more features and these features are shared with a lot of other classes. A typical example is the feature *Transaction* in a database. There the localization of code belonging to one feature (the feature traceability) can become expensive. So object-oriented software development is not suitable for implementing [SPLs](#), because optional features cannot be implemented optional with such a design.

We have shown, that for implementing [SPLs](#) we need techniques to build up optional features. Therefore, code belonging to a feature must be traceable, which is made difficult by code tangling and scattering. So, we focus on techniques which minimize these

```

1 public class Message{
2     String content;
3     ArrayList history = new ArrayList(); //for logging
4     Encryption enc;
5     //more functionality
6     public void sendMessage() {
7         history.add(content);           //logging
8         content = enc.encryptMessage(content); //encryption
9         sendContent(content);
10    }
11 }

```

Listing 2.1: An extract of class Message from Chat in object-oriented design

problems. Kästner et al. [KAK08] divides the implementation into two strategies: annotative approach and compositional approach. Annotative approaches mark the code and delete it before compilation and can be implemented with preprocessors [Käs10]. Compositional approaches encapsulate the code of features in single modules, compose it to a selected variant and can be implemented via Frameworks [JF88], Components [SGM02], FOP or AOP.

Because the discussed approach to solve the optional feature problem was originally inspired of AOP and we transferred to FOP we focus on these two implementation strategies in the following sections. However, parts of our solution are inspired by preprocessors, so we will now start with this approach.

### 2.2.1 Preprocessor Approach / Conditional Compiling

The preprocessor approach or conditional compiling is a representative of annotative approaches. For example, in programming languages like C/C++ it is possible to annotate the code with preprocessor statements like `#ifdef` x `#endif` and so to compile the program conditionally [HJ95].

Listing 2.2 demonstrates the use of preprocessor statements for the implementation of SPLs. It shows the same implementation as Listing 2.1, but the singular features are framed by the `#ifdef` statements. So this parts of code are just compiled, when the features (macros) *Logging* and/or *Encryption* are defined with a `#define` statement. Otherwise this code is left out.

This approach uses well-known techniques and is therefore accepted among programmers. Simple preprocessors are available for almost all programming languages. Another advantage is, that an existing program can be refactored with these annotations to a SPL easily. But within research this approach is seen critical [Spe92, KS94]. Implementing crosscutting features is difficult, because the annotations for one feature are necessary in many classes (problem of code tangling and scattering). Furthermore, the usage of annotations can be error-prone. Having many annotations can result in confusing source code, that complicates the maintenance of software. So preprocessor annotations can be used interlaced and fine-grained annotations allow to make singular tokens optional.

```

1  public class Message{
2      String content;
3      #ifdef Logging
4          ArrayList history = new ArrayList();
5      #endif Logging
6      #ifdef Encryption
7          Encryption enc;
8      #endif Encryption
9      //more functionality
10     public void sendMessage() {
11         #ifdef Logging
12             history.add(content);
13         #endif Logging
14         #ifdef Encryption
15             content = enc.encryptMessage(content);
16         #endif Encryption
17         sendContent(content);
18     }
19 }

```

Listing 2.2: An extract of class Message from Chat in preprocessor design

These problems can be reduced by using disciplined annotations [KA09], i.e. limiting the expressive power of annotations by just allowing the annotation of classes, methods and statements. This prevents the risk of making errors by annotating tokens like just a closing bracket, which is left out, when a feature is not selected.

Having introduced the approach of preprocessors for SPLs here, we go further to another approach in the next section, the FOP.

## 2.2.2 Feature-Oriented Programming

Feature-oriented software development is a paradigm for the construction, customization and synthesis of large-scale software systems. [AK09]

As quoted above from Apel and Kästner, FOP is a paradigm to implement SPLs. As stated in Section 2.1, features are the modules of a SPL; they are the most important parts in FOP. The idea of FOP is to separate a software system into the features it provides, so that it can be tailored to the customer needs and be composed to several variants. Because of the decomposition of the software into features a 1:1 mapping between the feature model and the implementation can be made. This is an advantage in contrast to the object-oriented approach. Therefore code belonging to a feature can be traced, modified and maintained easily and the problem of code tangling and scattering is minimized [Pre97, BSR03].

FOP uses object-oriented design as basic structure, because it is an established and common concept of programming. But we have shown in Section 2.2 that the object-oriented approach cannot map features to implementations with a 1:1 mapping. Therefore FOP extends the object-oriented design with the opportunity of splitting classes



to make the 1:1 mapping possible. Features implemented from a set of classes is called **collaboration** and the part of a class implementing one feature is called **role**. Figure 2.3 displays the context of such a collaboration model. This model is a simplified model of the Chat. The class `Server` plays roles in the features `Core` and `SecureCon`. The feature `GUI` consists of the collaboration of the classes `Gui` and `Client`.

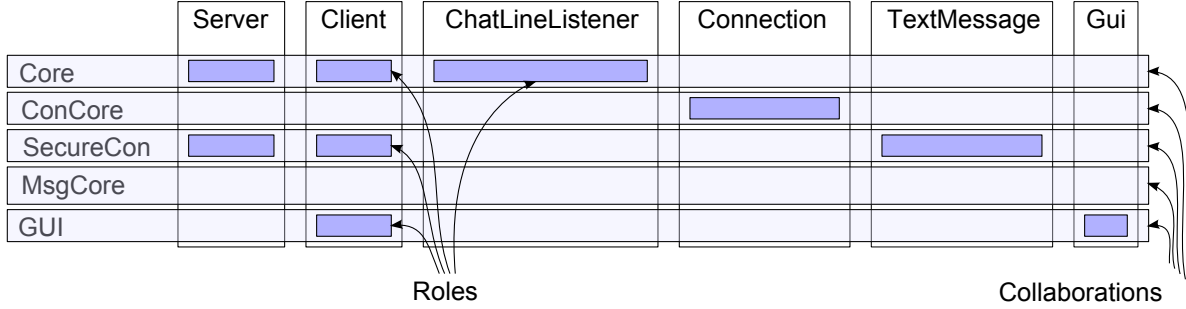


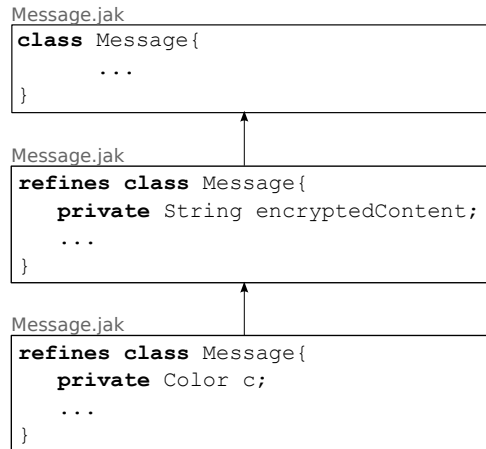
Figure 2.3: Example collaboration model

For **SPL** development two prominent representatives of **FOP**, namely **FeatureHouse** and **Algebraic Hierarchical Equations for Application Design (AHEAD)**, are used. But within this thesis we use a **FOP** Compiler for Java prototyped by Becker [Bec10], because we could extend this compiler for our purposes which we explain in Chapter 4. This compiler implements both approaches, **AHEAD** and **FeatureHouse** in parallel, so, they can be used within one tool. The **FOP** Compiler and **AHEAD** are representatives for the Java group, **FeatureHouse** is in use for several languages (Java, C# and XML) and can be easily extended for other languages [AKL09]. There exist also representatives for other programming languages, namely **FeatureC++** [ALRS05] for the C++ group and **Xak** for XML, which we will leave out here, because we concentrate on the Java language representatives.

## AHEAD

**AHEAD** is developed by Batory et al. [BSR03]. They use stepwise refinement to implement the **FOP** approach. A basic implementation is extended stepwise by the functionality of the selected features. **AHEAD** uses an extension of Java called *Jak* for implementing the features within refinements. The individual features are grouped into blocks via the **layer** keyword. The classes within the feature, if they refine classes from other features, are marked with the **refines** keyword. The same for this classes constructors. For method refinements a **Super()** call invokes the original method which should be refined. For composing the features **AHEAD** uses the mixin approach [SB02]. This approach generates one class per role with the appropriate hierarchy, renames the classes so that the last incoming refinement gets the original name and the **Super()** is replaced by a **super** call. Figure 2.4 illustrates the procedure of **AHEAD**. The class `Message` is refined by the features *Encryption* and *FormattedMsg*. In the figure an example is shown representative for class and method refinements. Additionally the resulting code after composition of the *Jak* files to one *Jak* file are shown with the resulting class hierarchy.

## Class Refinement



## Method Refinement

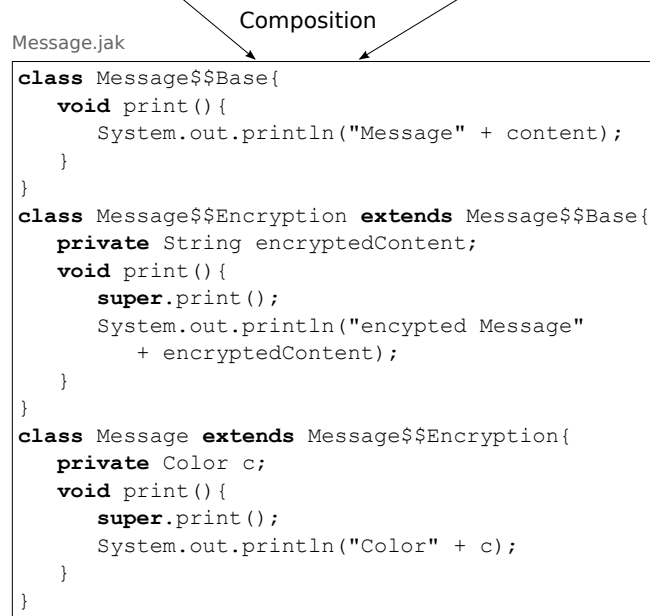
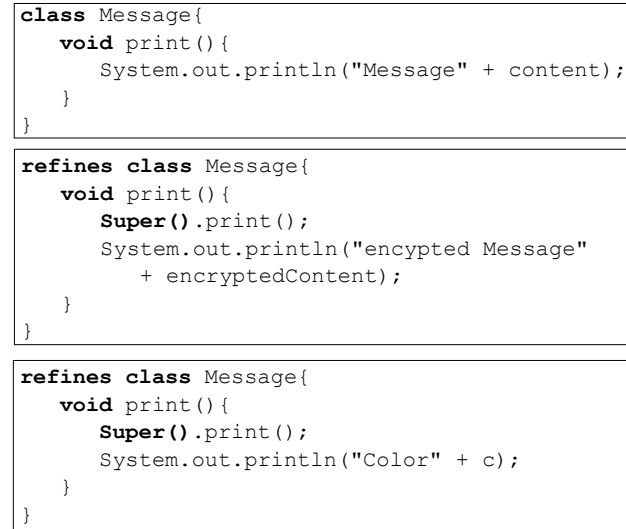


Figure 2.4: AHEAD composition

## FeatureHouse

FeatureHouse [AKL09] is another approach for the implementation of FOP. In contrast to AHEAD this approach does not use additional keywords, but a formalization developed by Apel et al. [ALMK08]. This formalization, called feature algebra, is used to analyze the similarities of the several techniques to implement the singular roles. It also shows that FOP can be adopted for other software artifacts, not just for programming languages.

The basic idea of FeatureHouse is to superimpose Feature Structure Trees (FSTs) [AKL09]. Within these FSTs, features are modeled as trees reflecting the structure of their software artifacts. A FST can have terminal and non-terminal nodes, where non-terminals are the inner nodes of the tree including the root, they are transparent, can have children, have a name and a type, but do not contain any content. So they can be superimposed easily. Whereas terminals are the leaves of the tree, also having a name and a type, but contain further content, therefore, their superimposition is not trivial.

Figure 2.5 shows a small example of the superimposition of FSTs. The basis class Message has the children nodes content and print and the class Message to superimpose has the children nodes encryptedContent and print (red colored). The result is a FST of the basic Message with added encryptedContent and an superimposed print.

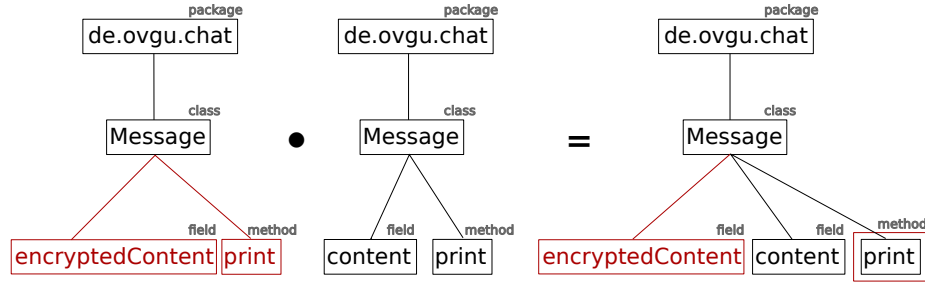


Figure 2.5: Superimposition of FSTs

## Order of Composition

Within feature composition the order of features play an important role, which counts for both approaches, AHEAD and FeatureHouse. Apel et al. show that [ALMK08] the feature algebra mentioned above the composition of features is not commutative, i.e. compositing feature *A* with feature *B* must not result in the same as the composition of *B* with *A*. Let us illustrate this with a small example in Listing 2.3. A basis is composed with two features *A* and *B*. According to the order of composition, the resulting output will be "green blue text" or "blue green text".

For the definition of the feature composition order, both approaches use a file where the selection of features and the order of composition is defined. AHEAD uses an equation file and FeatureHouse an expression file. This order is also relevant for optional features. When feature *A* refines feature *B* and feature *B* will be composed after feature *A*, it

```

1 public class MyColor{                               /* Basis Feature*/
2     void paint() {
3         System.out.print("text");
4     }
5 }
6 refines public class MyColor{                       /* Feature A*/
7     void paint() {
8         System.out.print("green_");
9         Super().paint();
10    }
11 }
12 refines public class MyColor{                      /* Feature B*/
13     void paint() {
14         System.out.print("blue_");
15         Super().paint();
16    }
17 }

```

Listing 2.3: An example for the order of feature composition

comes to a clash. This will be shown within the case study of the expression problem in [Section 5.1](#).

In this section, we presented the [FOP](#) as a method to implement [SPLs](#). Furthermore, we introduced two prominent representatives of this programming paradigm. In the next section, we will show how [AOP](#) can be used for [SPL](#) development.

### 2.2.3 Aspect-Oriented Programming

As we stated at the beginning of this chapter, the original optional weaving approach was inspired by [AOP](#), so we will now introduce it very briefly.

[AOP](#) is another programming paradigm which can be used to implement [SPLs](#). It aims at the modularization of crosscutting concerns which we introduced in [Section 2.2](#). Kiczales et al. [[KLM<sup>+</sup>97](#)] present the idea of [AOP](#). They suppose to implement crosscutting concerns as aspects to eliminate code tangling and scattering. The software basis is implemented with traditional programming concepts like [Object-Oriented Programming \(OOP\)](#). Additional features are implemented with the concept of pointcuts and advice. An aspect weaver builds the program consisting of the aspects representing additional features and the software basis.

As [AOP](#) aims at the separation of crosscutting concerns, it allows static and dynamic extensions, where homogeneous extensions are achieved by quantification. Now, let us go into detail, what an aspect is. An aspect manipulates class hierarchies, can add methods and fields into a class, can extend methods with additional code and catches events like method calls or field accesses and then executes additional or alternate code. Static extensions are done by [Inter-Type Member Declarations \(ITMDs\)](#), i.e. adding a method *X* to a class *Y*, whereas dynamic extensions are done by the joint point model. The joint point model comprehends three elements: joint point, pointcut and advice. A joint point is an event, e.g. a method call or field access, during program execution.

A pointcut selects joint points and an advice is the code which will be executed when a pointcut selected a joint point. So, if a pointcut matches a joint point the advice with the additional or alternate code will be executed. Furthermore, let us explain the quantification, because we will need it later on, when reflecting the optional weaving in [Section 3.4.1](#). Quantification is an important characteristic of pointcuts. Pointcuts quantify joint points by declaration and can choose several joint points. For example an advice  $D$  is always executed, when a method  $X$  in class  $Y$  is called. Or an advice  $E$  is executed when any field in class  $Y$  is accessed. Another possibility is to execute an advice  $F$ , when anywhere in the system a public method is called and a method  $X$  was called before.

```

1 class Message{
2     void print() {
3         System.out.println("Message_" + content);
4     }
5 }

```

Listing 2.4: Message - basis implementation

```

1 public aspect Encryption {
2     pointcut encMessage(Object o) : call(* Message.print());
3     void after() : print() {
4         System.out.println("encrypted_Message_" + encryptedContent);
5     }
6 }

```

Listing 2.5: Aspect for Encryption

In [Listing 2.4](#) we show a basic implementation for the print method of the class Message. [Listing 2.5](#) implements the feature *Encryption* within the pointcut model of AspectJ which is a prominent example for AOP in the Java group. First of all the aspect is defined. In line 2 of [Listing 2.5](#) we define the pointcut where the advice (line 3 to 6) shall operate, namely when the method print() of Message is called. The keyword **after** within the advice indicates that this code will be executed after the original method print().

Now, having seen how SPLs can be implemented with preprocessors, FOP and AOP we need to introduce another fact which occurs among feature implementation, when features are going to interact.

## 2.2.4 Feature Interaction

Mainly from the telecommunication industry the occurrence of feature interactions is known [[CKMRM03](#)]. A *feature interaction* is a situation where two or more features behave unexpected which does not occur when they are used in isolation. The standard example is a phone with the additional two features *call forwarding* and *call waiting*: Using them in isolation everything works fine, but when using in combination it is unclear what to do when an incoming call meets a busy line. The call is either forwarded or signaled, so in either case one of the feature's behavior is compromised.

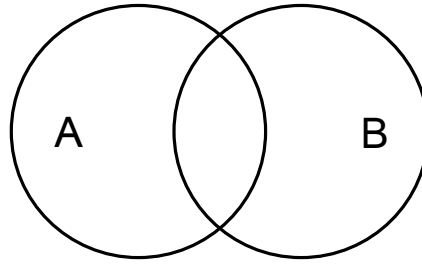


Figure 2.6: Interacting features [Käs07]

A feature interaction can be illustrated like in Figure 2.6. This image represents the code base of two features  $A$  and  $B$ , where the interaction is illustrated with the intersection of both. When two features interact, it is an interaction of first order and when three features interact, it is an interaction of second order [LBL06]. If there are optional features interacting with each other these interactions lead to the optional feature problem which we discuss in detail in Chapter 3. Further, the solutions to this problem are reviewed there: software derivatives, preprocessor, optional weaving and optional composition.

## 2.3 Compiler

In this section we briefly introduce some terms of compiler construction which we use within Chapter 4 where we explain necessary compiler extensions for the optional composition.

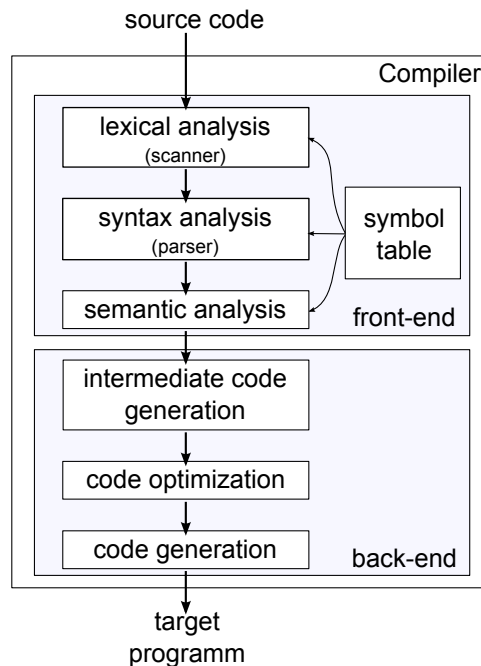


Figure 2.7: Phases of a compiler adopted from [ALSU08]

Figure 2.7 shows the two singular phases of a common compiler, the front-end and back-end [ALSU08]. The front-end is responsible for the analysis of the source code and the back-end for the synthesis analysis. First of all, in the front-end the source code comes through the scanner where the lexical analysis takes place. The result is altered in the syntax analysis, where it is sent through the parser. The resulting **Abstract Syntax Tree (AST)** proceeds the semantic analysis where the attention is turned on the type and error checking. In the synthesis phase the **AST** comes through a generator where intermediate code is created. This code will be optimized in the phase of code optimization. At the end, the final code is generated and results in the target program.

The extension of the compiler, made in this thesis, is limited to the front-end. Therefore we concentrate within explanation on the phases within the front-end in the following and leave out the phase of the back-end.

### Scanner

In the lexical analysis, the scanner reads the source code in and organizes the characters into meaningful sequences, called lexemes. Each lexeme will be assigned to a token, where a token consists of a token name and optional attributes. These tokens are handed over to the next phase into the parser.

### Parser

The parser uses the tokens from the scanner to generate a tree-like structure, which exhibits the grammatical structure of the token stream. This generated **AST** is characterized by inner nodes which represent the operations. The children of these inner nodes represent the arguments of these operations. The **AST** is then transferred to the semantic analysis. Furthermore, the parser generates the symbol table used in the semantic analysis.

### Semantic Analysis

Within the semantic analysis the **AST** and symbol table are checked if they are semantic consistent with the help of the language specification. This phase collects type information and saves them within the **AST** or in the symbol table, so it can be used for the construction in the phases of the back-end. An important component in this phase is the type checking, i.e. the compiler assures that each operator executes the adequate operation. Another point is the type conversion which is done in this phase, according to the language specification.





## 3. Problem Statement

In this thesis, the question of a solution for the optional feature problem by the optional composition approach shall be answered. Therefore, we explain the optional feature problem and possible solutions to this problem in this chapter. As solutions software derivatives, preprocessor approach, optional weaving and the in this thesis developed optional composition are introduced here.

### 3.1 Optional Feature Problem

As stated in Section 2.2.4, feature interactions are a key part of feature-oriented designs. If these interactions take place between optional features they lead to the optional feature problem [LARS05, Käs07, KAUR<sup>+</sup>09]. This problem appears when two or more interactions of optional features occur and these features show another behavior than being used in isolation.

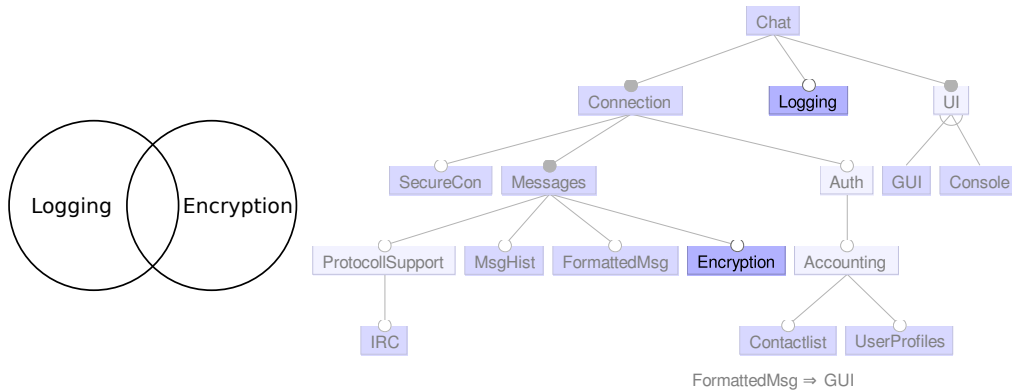


Figure 3.1: Interaction of *Logging* and *Encryption*

Let us illustrate the stated problem with an appropriate example. For security reason, the chat needs some encryption and logging, so messages are not transferred as plain

text. The logging is for collecting status information, which are printed out at the console. Now, let us take a look at the problem space. The features *Encryption* and *Logging* in the chat interact with each other (see Figure 3.1). Here the feature *Logging* means that all methods have to print out their status. So that means for the solution space that the method `send` from class `Client` is refined by a print of the status, which will be something like “message will be sent” to the console. The feature *Encryption* also refines this method by encrypting the message before sending. That is, seeing both features in isolation, no problem. When both features are selected the status message is not modified that an encrypted message was sent.

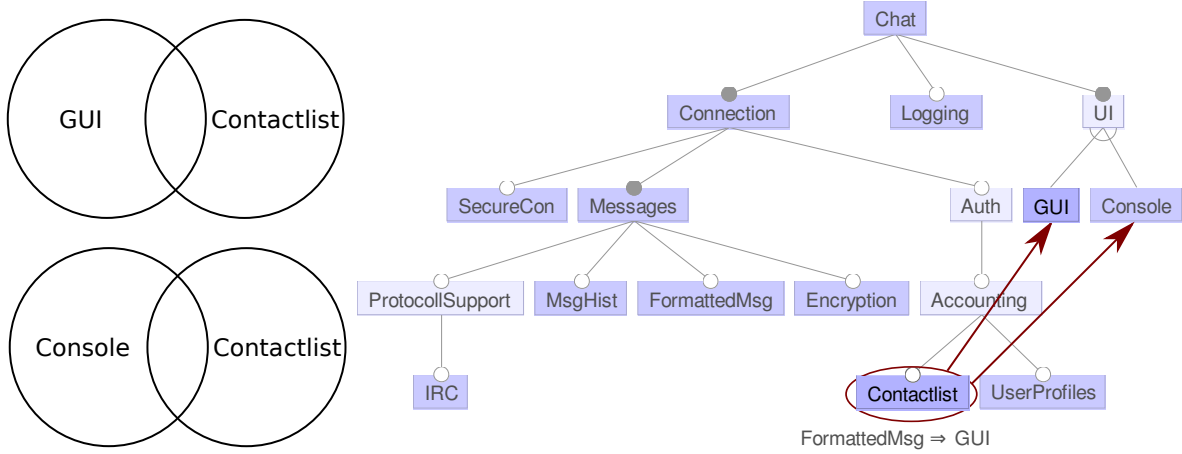


Figure 3.2: Feature interaction: *Gui - Contactlist* | *Console - Contactlist*

Let us take a closer look at two other interacting features: *GUI* and *Contactlist*. The contact list of the chat user must be displayed within the **Graphical User Interface (GUI)**. Therefore, the feature *Contactlist* refines the `createAndShowGui` method (see Listing 3.2) of the main class of the *GUI* feature. But, *GUI* and *Console* are alternate features (see Figure 3.2). So, *Contactlist* contains a refinement for the main class of the *Console* feature, too. What will happen if *Contactlist* refines *GUI*, but instead of *GUI*, *Console* was selected to generate the variant of the chat? With **Feature-Oriented Programming (FOP)** this variant could not be generated, because the refinement of an absent class will result in an error for the **Software Product Line (SPL)**. Always, the selection of *Contactlist* will result in an error with FOP, unless splitting it into two separate features - one *Contactlist* for *GUI* and one for *Console*. Another opportunities will be opened by the software derivatives, the preprocessor approach, optional weaving and optional composition.

We will introduce four approaches to overcome this problem in the next sections. First of all, the approach of software derivatives will be presented

## 3.2 Software Derivatives

The first approach we present is the approach of software derivatives. The basic idea of this approach is to break up the dependencies of the features and to encapsulate the code

```
1 layer GUI;
2 public class Gui extends JFrame implements ChatLineListener,
  ActionListener{
3   /* some fields */
4   public Gui(String title, Client client) {
5       //initializing
6   }
7   public void createAndShowGui(){
8       // gui will be created here
9   }
10  public void newChatLine(TextMessage tm) {
11      // this method gets called every time a new message is received
        (observer pattern)
12  }
13  public void actionPerformed(ActionEvent evt) {
14      // sending message if button pressed
15  }
16 }
```

Listing 3.1: Class Gui from feature *GUI*

```
1 layer Contactlist;
2 import javax.swing.JList;
3 public refines class Gui {
4   public void createAndShowGui() {
5       Super().createAndShowGui();
6       Contacts contacts = new Contacts();
7       JList contactList = new JList(contacts.getContactList());
8       add("West",contactList);
9       this.repaint();
10  }
11 }
```

Listing 3.2: Class Gui from feature *Contactlist*

implementing the interactions in separate modules. This idea was first published by Prehofer [Pre97]. He “lifted” the interactive code out of the features and therefore, called these code fragments *lifter*. Based on this idea, Liu et al. [LBN05, LBL06] developed the approach of software derivatives and an associated algebra. This approach suggests to swap the code, responsible for the interaction, into an additional derivative feature. It is called derivative because it is derived from the features it connects. We use the naming convention also used by Kästner et al. [KAuR<sup>+</sup>09]  $A/B$  for a derivative feature which was derived from the features  $A$  and  $B$ . We show a visual representation of this approach in Figure 3.3.

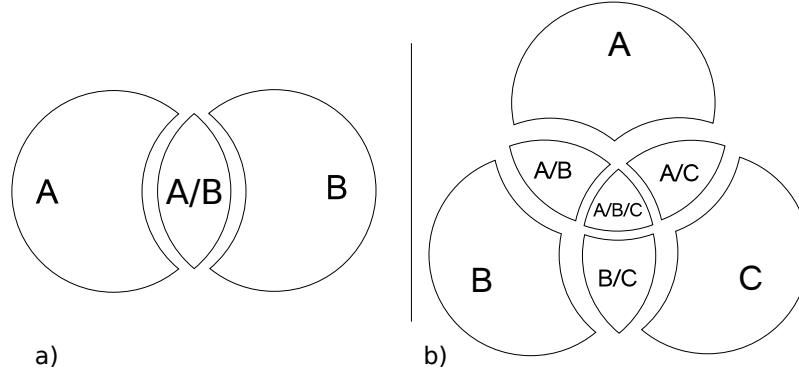


Figure 3.3: Software derivatives adopted from [KAuR<sup>+</sup>09]

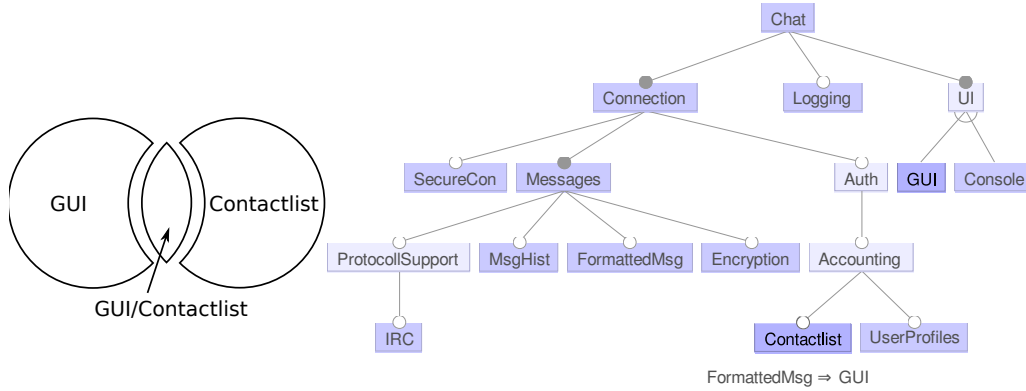


Figure 3.4: Optional Composition

Let us explain this approach with an example. We take again the features *GUI* and *Contactlist* into account (see Figure 3.4). We said in the section before, that we need a mechanism to map the code where both features are selected. *GUI* is implemented like in the section before, but we split *Contactlist* into the feature *Contactlist* and the derivatives for the features *GUI* and *Console*. So, we have now the possibility to swap the interactive code into the derivative feature called *Gui/Contactlist*, where we can add the contacts to the *GUI* (see Listing 3.4).

We show in Figure 3.3 that this approach does not scale concerning the number of derivative features. This was pointed out by Liu et al. [LBN05] and approved by

```
1 layer Contactlist;
2 public class Contacts {           //Feature Contactlist
3     private String[] contacts;
4     public Contacts() {
5         contacts = new String[4];
6         contacts[0] = "Tim";
7         contacts[1] = "Mary";
8         contacts[2] = "Martin";
9         contacts[3] = "Judith";
10    }
11    public String[] getContactList() {
12        return contacts;
13    }
14 }
```

Listing 3.3: *Contactlist* feature

```
1 layer Contactlist;
2 import javax.swing.JList;
3 public refines class Gui {       // Derivative Gui/Contactlist
4     public void createAndShowGui() {
5         Super() .createAndShowGui();
6         Contacts contacts = new Contacts();
7         JList contactList = new JList(contacts.getContactList());
8         add("West", contactList);
9         this.repaint();
10    }
11 }
```

Listing 3.4: Derivative of *GUI* and *Contactlist*

Kästner [Käs07] within a case study. A quadratically growth of the maximum number of derivatives were suggested with the number of features. We will prove this within our own case study in [Chapter 5](#). However, the encapsulation of interactions in derivatives is expensive according to the number of features and their interactions, which we discuss in detail in [Chapter 5](#) and [Chapter 6](#). So, we would like to introduce another approach to overcome the stated optional feature problem, in the next section.

### 3.3 Preprocessor

Now, we would like to introduce the preprocessor approach. We use the [FOP](#) implementation as base. The feature interactions are integrated via nested preprocessor statements, as introduced in [Section 2.2.1](#). So, we combine the annotative approach with the compositional approach as suggested from [KA08]. For this reason, the interaction code is just used when both features are selected. Let us illustrate this with the features *GUI* and *Contactlist* from our chat. The main class from *GUI* is the same like the other approaches, except the surrounding `#ifdef` tag (see [Listing 3.5](#)). The interactive code in [Listing 3.6](#) is surrounded by the preprocessor statement for selecting the *GUI* and the *Contactlist* feature to include this code just when both features are chosen.

```
1 #ifndef GUI;
2     public class Gui extends JFrame implements ChatLineListener,
        ActionListener{
3         /* some fields */
4         public Gui(String title, Client client) {
5             //initializing
6         }
7         public void createAndShowGui() {
8             // gui will be created here
9         }
10        public void newChatLine(TextMessage tm) {
11            // this method gets called every time a new message is
                received (observer pattern)
12        }
13        public void actionPerformed(ActionEvent evt) {
14            // sending message if button pressed
15        }
16    }
17 #endif Gui
```

Listing 3.5: *GUI* Feature - Preprocessor approach

```
1 #ifndef Contactlist
2 #ifndef GUI
3     import javax.swing.JList;
4     public refines class Gui {
5         public void createAndShowGui() {
6             Super().createAndShowGui();
7             Contacts contacts = new Contacts();
8             JList contactList = new JList(contacts.getContactList());
9             add("West", contactList);
10            this.repaint();
11        }
12    }
13 #endif GUI
14 #endif Contactlist
```

Listing 3.6: *Contactlist* Feature - Preprocessor approach

## 3.4 From Optional Weaving to Optional Composition

In this section, we discuss another approach to resolve the optional feature problem, the optional composition. We transferred the original optional weaving approach, which is implemented with the means of [Aspect-Oriented Programming \(AOP\)](#), to [FOP](#) with the means of [FOP](#), so we decide to rename the approach to *optional composition*. The transfer was done to see if it is possible to implement this approach with [FOP](#). First of all, we introduce the original optional weaving approach and then bring our optional composition approach in.

### 3.4.1 Optional Weaving

The first idea for the optional weaving approach was published by Leich et al. [[LARS05](#)]. Their idea was to leave the interaction code of two features in one of these features and just weave it with the means of [AOP](#) concepts when both features are selected. [Figure 3.5](#) illustrates this idea of having the interaction code of feature *A* and *B* in an optional part of feature *B*. The approach of Leich et al. [[LARS05](#)] uses wild cards defined by join points in pointcuts which are robust against changes of features and composition. However Ceasar [[MO04](#)] and FeatureC++ [[ALRS05](#)] combine [FOP](#) and [AOP](#), the idea is that the programmer is free to decide using aspects or mixins, but this “looks like a hack” [[LARS05](#)]. Therefore, Leich decided to declare methods within mixins with the keywords **before**, **after** and **around** as optional, i.e. if there is no method to refine, these refinements will be ignored. So this approach is implicitly defined by the signature of the refined method.

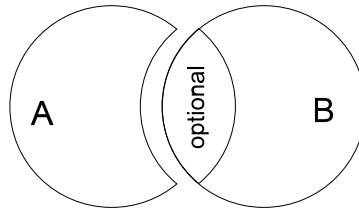


Figure 3.5: Optional Weaving [[Käs07](#)]

Now, let us come to the example, for the optional weaving. We again take features *GUI* and *Contactlist* from our chat into consideration. For class *Gui* from feature *GUI*, we have always the same implementation (see [Listing 3.1](#) in [Section 3.1](#)). The aspect *Contactlist* from feature *Contactlist* contains the interaction code of *GUI* and *Contactlist* and of *Console* and *Contactlist*. The pointcut, which matches the *Gui* joint point is executed after the call of the *createAndShowGui* method, when the feature *GUI* belongs to the feature selection, otherwise the advice is left out. The matching pointcut of the *Consoles* constructor is called after the constructors call. Here, the advising method is left out as well, if the feature *Console* is not selected.

We introduced the optional weaving approach here, and now, we would like to present the results of a case study from literature with this approach, which brings us to the optional composition approach.



```

1 aspect Contactlist{
2     after() : call (* Gui.createAndShowGui() ){
3         Contacts contacts = new Contacts();
4         JList contactList = new JList(contacts.getContactList());
5         add("West",contactList);
6         this.repaint();
7     }
8     after() : call (Console.new()) {
9         Contacts contacts = new Contacts();
10        String[] list = contacts.getContactList();
11        for(int i = 0 ;i< list.length; i++)
12            System.out.println(list[i]);
13    }
14 }

```

Listing 3.7: Aspect Contactlist - Optional Weaving

## Discussion

In this part, we bring out the results of a case study for the optional weaving approach implemented with AspectJ, done by Kästner [Käs07]. In comparison to the approach of software derivatives, introduced in Section 3.2, for the optional weaving approach it is not necessary to create a derivative feature. But the disadvantages were significant. He points out, that it is not possible to reference optional classes, methods or member variables in the optional advice statements, so that code replication becomes necessary. Secondly, he states, that optional weaving is possible for advice statements but not for *Inter-Type Member Declarations (ITMDs)*. The target for the ITMDs must exist and so they cannot be optional. So all extensions belonging to the interaction are woven anyway, if they do not belong to the optional advice. The last disadvantage directs the scope problem which hinders the advising methods in optional classes. To define an advice the target class must be in the scope of the aspect, i.e. in Java style it must be imported by the **import** statement. When this class is removed from the compilation an error occurs within the aspect at the import statement. Finally, he states that the optional weaving is no solution to the optional feature problem because of these lacks at this time.

However, these lacks lie in the character of AspectJ and other representatives of the AOP which lead us to the approach within FOP which we developed during this thesis and call optional composition and introduce in the next section.

### 3.4.2 Optional Composition

Our approach of optional composition is derived from the optional weaving approach, introduced in Section 3.4.1. For technical reasons, the approach of optional weaving is just applicable with AspectJ with restrictions, so we transferred it to Jak, the FOP extension of Java. Further, we would like to see, if this approach is portable to FOP. We distinguish between two approaches, the implicit and the explicit optional composition, which we will introduce in the following sections. We show the visual representation of

this approach for our example for both, the explicit and implicit optional composition in Figure 3.6.

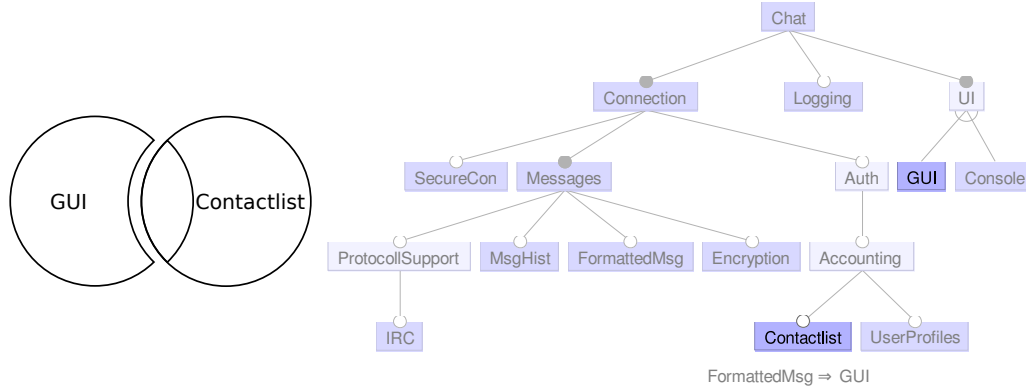


Figure 3.6: Optional Composition

### 3.4.2.1 Explicit Optional Composition

The first approach of the optional composition is the explicit optional composition. In order to solve the optional feature problem, we make methods and classes, belonging to the interactions of the features, explicitly optional by using the keyword `optional`. When this feature is in the compilation unit, the optional code will be compiled, otherwise not. For this approach the intersection of a feature *A* and a feature *B* must be manually marked by the optional keyword to strictly delimit the interaction code from the code of the several features. Therefore, this approach is comparable to a preprocessor behavior, introduced in Section 3.3, that just operates at functions and not at simple statements, because code cluttered with preprocessor statements is difficult to read and maintain, according to the Linux kernel coding style<sup>1</sup>. Another point is, that the programmer must always have the feature model in mind, to place the optional tags with the accordant feature to the appropriate place, because the order of composition plays an important role.

```

1 import javax.swing.JList;
2 optional GUI public refines class Gui {
3     optional GUI public void createAndShowGui() {
4         Super() .createAndShowGui();
5         Contacts contacts = new Contacts();
6         JList contactList = new JList(contacts.getContactList());
7         add("West", contactList);
8         this.repaint();
9     }
10 }
```

Listing 3.8: Class Gui from feature *Contactlist* - Explicit Optional Composition

<sup>1</sup>see /Documentation/SubmittingPatches in the Linux source

Let us come again to the example of the feature interaction between *Gui* and *Contactlist*. The original class *Gui* from feature *GUI* is the same implementation as in [Section 3.1](#) (see [Listing 3.1](#)). This class *Gui* is introduced, when the feature *GUI* is selected. The refinement of this class *Gui* (see [Listing 3.8](#)) by the feature *Contactlist* is introduced, when *Contactlist* is selected. The marking of this class by the optional keyword operates, that this class is compiled, when the feature *GUI* is selected, otherwise this refinement is ignored and not compiled.

The explicit composition is similar to the preprocessor approach, because the interactions are framed by the optional keyword instead of preprocessor statement. Therefore, we developed a more automatic method to realize the optional composition, which we call implicit optional composition and describe in the next section.

### 3.4.2.2 Implicit Optional Composition

In this part, we introduce the implicit optional composition approach, which is in contrast to the explicit optional composition an automatic approach. It is characterized by the ignorance of class or method refinements by the compiler, when the original class, the one to refine, is not present. This means, that the code of the interaction of two features is automatically left out when just one of the features is selected, and the original class is absent, because the feature introducing this class is not selected. So, the compiler decides, whether to include or exclude the class refinement. Thus, the programmer does not have to be aware of the feature model, in contrast to the explicit optional composition, because he does not have to specify the interacting feature, to which the refined class belongs.

```

1 import javax.swing.JList;
2   public refines class Gui {
3       public void createAndShowGui() {
4           Super() .createAndShowGui();
5           Contacts contacts = new Contacts();
6           JList contactList = new JList(contacts.getContactList());
7           add("West",contactList);
8           this.repaint();
9       }
10  }
```

Listing 3.9: Class *Gui* from feature *Contactlist* - Implicit Optional Composition

We again take the example of the interaction features *Gui* and *Contactlist* into consideration (see [Figure 3.6](#)). The implementation of class *Gui* in feature *GUI* is the same as well as in [Listing 3.1](#) in [Section 3.1](#). The implementation of the intersection between *GUI* and *Contactlist* is just a normal refinement (see [Listing 3.9](#)), although these features are optional. That is why the compiler decides, when *GUI* is within the feature selection, to include the refinement of class *Gui* in the compilation, or when its not selected the compiler leaves this refinement out.

In this chapter, we explained the optional feature problem and introduced three approaches to solve this problem. We illustrated the approach of software derivatives, the preprocessor approach and the optional composition approach with its two characteristics. In [Chapter 5](#) we take a closer look to these approaches within case studies and will answer the question, whether the optional composition can solve the optional feature problem or not and compare it to the other approaches. In the next chapter, we present the compiler extensions for the explicit and implicit optional composition approaches.

## 4. Compiler Extension

In the previous chapter, we introduced the optional composition approach. Because the evaluation of this approach is the major goal of this thesis and to make a realistic evaluation in Chapter 6 with the case studies in Chapter 5, we need a tool to implement the optional composition approach. Therefore, we extend a compiler, that supports this approach. So, this chapter describes, which technical changes we had to make the compiler, to realize the optional composition. Thus, we introduce the native FOP Compiler from Becker [Bec10] in this chapter which we extended, and we present the extensions we had to make for the explicit and implicit optional composition.

### 4.1 FOP Compiler

For the reason of not starting from scratch to implement a compiler, we built on an existent implementation. The FOP Compiler from Becker [Bec10] was implemented within the JastAdd [SS07, Ekm06] compiler framework. The compiler works with Java and its FOP extension Jak. We use Jak as FOP extension, so that we can use existent projects, that we just extend. Another compiler build with the JastAdd compiler framework is Fuji<sup>1</sup> ([AKL<sup>+</sup>11]), developed at the university of Passau. Other compilers like Algebraic Hierarchical Equations for Application Design (AHEAD) or FeatureHouse use a two-tier method for compilation, i.e. the FOP source code is composed into native source code, which is then compiled by a standard compiler. Therefore, not all FOP specific errors can be detected, so we decided to extend Becker's compiler, because this compiler leaves out the intermediate step, i.e. the program is compiled from the FOP source code directly. In principal it does not matter if we use the Fuji compiler or the compiler from Becker, because both use the JastAdd Compiler Framework and implemented the AHEAD and FeatureHouse approach. But, we decided to use Becker's compiler, because he works in the same research group and is present for call backs.

This native FOP compiler combines the ideas of FeatureHouse and AHEAD. Let us explain the construction of this compiler. In Figure 4.1, the scheme of this compiler is

---

<sup>1</sup>see <http://www.fosd.de/fuji>

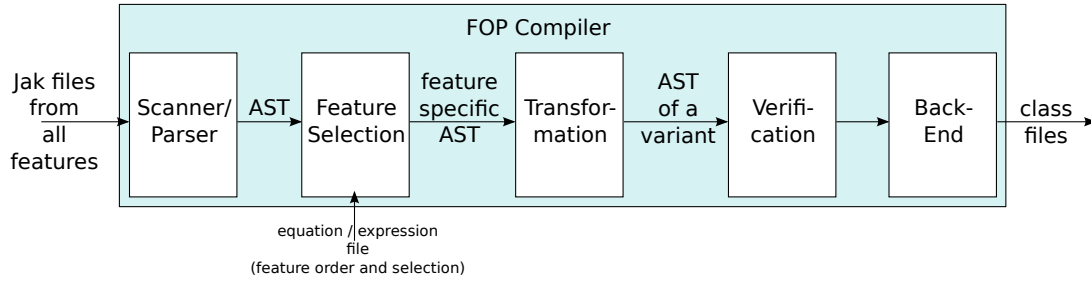


Figure 4.1: Scheme of the FOP Compiler [Bec10]

shown to illustrate the singular steps, that we will bring up briefly. The compiler scans all features within the scanner component and then the parser generates an [Abstract Syntax Tree \(AST\)](#) from all features of the project. An additional scan of the equation or expression file, representing the order and the selection of the features, will be done at the beginning of the feature selection part. The information from these files are used to decide which parts of the [AST](#) can be left out or deleted, because of belonging to a non-selected feature. These files are also used to determine the order of the composition of the selected features. So a feature specific [AST](#) is handed over to the transformation part.

In the transformation component the features are composed by using the concept of superimposition of [Feature Structure Trees \(FSTs\)](#) (see [Section 2.2.2](#)). As a result the [AST](#) of a variant is accomplished. This variant [AST](#) is then tested for semantic errors in the verification component. Finally the variant [AST](#) is transformed into byte code within the default back-end of the compiler framework.

Here, we introduced the FOP compiler, which we modified for the purposes of optional composition. We start with the explanation of the explicit optional composition, because the implicit solution then is easier to understand. We need both solutions because of the fact, that more than two features can interact with each other, but this is discussed later on in chapters [Chapter 5](#) and [Chapter 6](#). In the next section we describe the extension of this compiler for the explicit optional composition.

## 4.2 Extension for Explicit Optional Composition

In this part, we describe the necessary changes to provide the explicit optional composition. The goals of the optional composition are to eliminate derivative features and hold the feature interaction inside one of the interacting features, but for all that the feature interaction is encapsulated. So, just the necessary source code is compiled in the compiler. This approach shall be simply integrated in existent source code and the work flow.

### 4.2.1 Design

In this part, we specify the design of the compiler extension for the explicit optional composition. We show in Figure 4.2 that the changes to the compiler take effect for the explicit optional composition in the scanner and parser component. We introduce a new

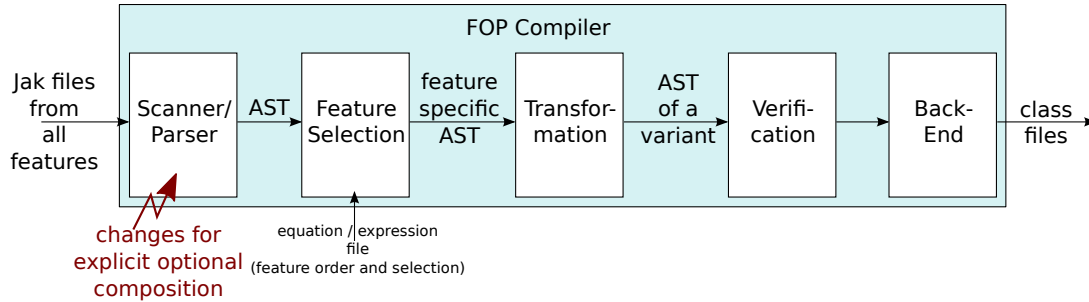


Figure 4.2: FOP Compiler changes for explicit optional composition

keyword, **optional**, because the existent keywords are not usable to realize the optional behavior. We use the optional keyword as preposition in combination with the feature name to mark classes and methods being optional. For this reason, this approach is comparable to the software derivative approach. Being not finer in granularity than the methods level is more clearly on the code level according to the Linux Guidelines<sup>2</sup> and achieves the encapsulation of the feature interaction. For using the optional keyword, we have to introduce it to the scanner component of the compiler. The scanner converts the optional keyword with the feature name into a token during the lexical analysis, so the scanner must be modified. Further, the new token is processed in the parser, where the token of the optional statement is transferred to a node in the *AST*. This node is a marker for the syntax branches in the *AST*. If the feature behind the optional keyword is not selected, the node representing the optional statement will be deleted and their children as well. But, if the feature is selected, the node with the optional statement is deleted, but their children take the place of the optional node in the *AST*.

<sup>2</sup>see /Documentation/SubmittingPatches in the Linux source

## 4.2.2 Implementation

In this part, we explain the implementation details of the compiler extension for the explicit optional composition. For the realization of the explicit optional composition, we added the new optional keyword to the scanner. In [Listing 4.1](#) this optional keyword is introduced and a terminal node with the name is generated.

```
1 "optional" { return sym(Terminals.OPTIONAL); }
```

Listing 4.1: Excerpt from the Scanner file Keywords.flex

For the changes to the parser, we must define, where in the [AST](#) this optional nodes can be placed. First, we made methods optional, because we have just cases within our case studies, where the interaction is placed in methods. So, we created an [AST](#) node representing the optional statement for methods. These [AST](#) nodes are generated by the compiler framework and therefore, they must be described in a JastAdd specific grammar (see [Listing 4.2](#)). We extend the method declaration with the optional keyword and the feature name, so we can fall back to the method declaration, when interpreted the statement in front of the method declaration. Thus, we built a language construction, consisting of the keyword optional followed by the name of the feature concerning this code, followed by a native method declaration.

```
1 OptWeav_Stmt : BodyDecl ::= <Feature:String> MethodDecl;
```

Listing 4.2: AST additions for explicit optional composition

Then, the parser must be modified. Lines 1 - 4 of [Listing 4.3](#) show the position of the optional statement in the [AST](#). The following lines describe the production rule for the optional method declaration, where the keyword optional is followed by an identifier and the method declaration.

```
1 BodyDecl class_member_declaration
2     = opt_method_declaration.o
3     {: return o; :}
4 ;
5
6 BodyDecl opt_method_declaration
7     = OPTIONAL IDENTIFIER method_declaration.m
8     {: return new OptWeav_Stmt (IDENTIFIER, m); :}
9 ;
```

Listing 4.3: Parser additions for explicit optional composition

The last step is the adaption of class `Program`, where the [AST](#) nodes are processed. We distinguish between refine nodes and class nodes [[Bec10](#)]. Class nodes represent the class in the [AST](#) and refine nodes represent the refinements within the [AST](#). In line 10 of [Listing 4.4](#), we extract the feature behind the optional keyword so, we can compare



it to the given feature list and decide whether the method must be deleted or not. In line 20 of Listing 4.4, we delete the optional statement from the AST and insert the method declaration to the AST when the given feature was in the feature list (see lines 21 to 23, Listing 4.4)

```

1 public class Program extends ASTNode<ASTNode> implements Cloneable {
2     public void addRefinesStmts(ClassDecl originalclassdecl,
        ReferenceType refinesNode) {
3         List<BodyDecl> refinesStmts = null;
4         List<BodyDecl> copyRefinesStmts = null;
5         /*more code*/
6         for (ASTNode child : refinesStmts){
7             if (child instanceof OptWeav Stmt){
8                 OptWeav Stmt weave = (OptWeav Stmt) child;
9                 String feat = weave.value.toString();
10                boolean mustWeave=false;
11                for (String feature: featureList){
12                    String[] args = feature.split("[ " + System.
                        getProperty("file.separator")+"]");
13                    if (args[args.length-1].equals(feat)){
14                        mustWeave = true;
15                        break;
16                    }
17                }
18                int i = copyRefinesStmts.indexOfChild(child);
19                copyRefinesStmts.removeChild(i);
20                if (mustWeave){
21                    copyRefinesStmts.insertChild(weave.getMethodDecl(), i
                        );
22                }
23                i = refinesNode.indexOfChild(copyRefinesStmts);
24                if (i>-1) refinesNode.removeChild(i);
25                ((Refine_Class) refinesNode).setBodyDeclList(
                        copyRefinesStmts);
26            }
27        }
28        for (ClassDecl decl : class_decl){
29            /* analogous to loop for refine statements*/
30        }
31        return refinesNode;
32    }
33    /*more functionality*/
34 }

```

Listing 4.4: Class Program, sequence for explicit optional composition

In this section, we presented the extension of the compiler concerning the explicit optional composition approach. But introducing a new keyword and the preprocessor like behavior result in additional expense for programmers. Therefore, we introduce the compiler extension for the implicit optional composition, that does without a new keyword, in the next part.

### 4.3 Extension for Implicit Optional Composition

In this section, we show, how we changed the compiler to afford the implicit optional composition. This approach can be seen as extension to the explicit optional composition, that is why the goals are the same as listed in [Section 4.2](#). But, for this approach the keyword is not necessary, because it is an automated approach, so the manual effort is lower than with the explicit optional composition approach.

#### 4.3.1 Design

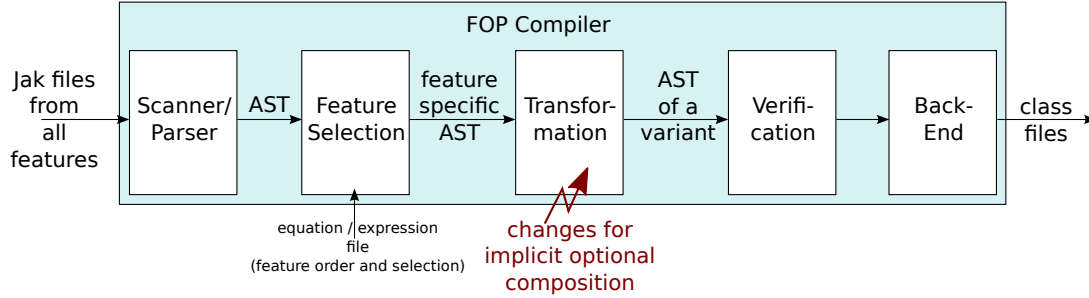


Figure 4.3: FOP Compiler changes for implicit optional composition

[Figure 4.3](#) illustrates that the changes take effect in the transformation component. So, the scanner and parser are not modified. For **FOP**, the class refinement is composed with the original class from the feature, which introduced this class. Therefore, it is necessary, that this original class exists in the **AST**, otherwise this code cannot be composed. But, to automate the optional composition we look up the **AST** for the original class and when it is present, we compose the original class with the refinement, otherwise we throw the refinement away.

#### 4.3.2 Implementation

In contrast to the explicit version, where we introduced a keyword, we implement a compiler switch *optional* to the compiler's front end ([Listing 4.5](#), line 24) for the implicit optional composition approach. We use this switch to eliminate some error messages, so refinements can be left out when the original class is missing without throwing an error. This denotes, that the interactive code, embedded in one of the two interacting features, can be left out, when just one of the features is selected. Further this interactive code will be used when the second feature is selected, because the class to refine will exist then.

[Listing 4.6](#) shows the sequence of class `Program` where the transformations for the implicit optional composition will be done. First of all, a checking of whether the original class is present or missing will be done. When the original class is present, the optional switch will not take effect. But when the original class is missing the optional switch will throw the refinement away and leave it out of the compilation unit (see lines 9 - 10 and lines 20 - 21 of [Listing 4.6](#)).

```
1 public class Frontend extends java.lang.Object {
2     public boolean process(String[] args, BytecodeReader reader,
3         JavaParser parser) {
4         program.initBytecodeReader(reader);
5         program.initJavaParser(parser);
6         initOptions();
7         processArgs(args);
8         /* process compiler switches */
9         if(program.options().hasOption("-featurehouse")) {
10             System.out.println("FeatureHouse");
11             program.fstComposer();
12         }
13         else{
14             program.searchinASTforClassandRefinesStmt();
15             program.transformAST(program.options().hasOption("-optional"
16                 ));
17         }
18         /*process errors...*/
19         return true;
20     }
21     protected void initOptions() {
22         Options options = program.options();
23         options.initOptions();
24         /* several options added*/
25         options.addKeyOption("-featurehouse")
26         options.addKeyOption("-optional");
27     }
28     /* more code */
29 }
```

Listing 4.5: Front end with extension of compiler switch

```
1 public class Program extends ASTNode<ASTNode> implements Cloneable {
2     public void transformAST(boolean optional) {
3         boolean flag = false;
4         Refine_Class refClass=null;
5         for (Refine_Class ref : refines_stmt) {
6             /* check original class is present...*/
7             if (!flag && optional && refClass!=null){
8                 flag = optional;
9                 CompilationUnit cu = (CompilationUnit) refClass.
10                    getParent().getParent();
11                 cu.setFromSource(false);
12             }
13             /* set error if !flag and !optional */
14         }
15         for (Refine_Interface ref : refine_interfaces) {
16             /* check original interface is present...*/
17             if (optional && !flag)
18             {
19                 flag = optional;
20                 CompilationUnit cu = (CompilationUnit) ref.getParent().
21                    getParent();
22                 cu.setFromSource(false);
23             }
24             /* set error if !flag and !optional */
25         }
26         /*more functionality*/
27     }
```

Listing 4.6: Class Program, sequence for implicit optional composition

---

In this chapter, we presented the details of the prototyping tool for the optional composition. We have shown our changes for the implicit and explicit optional composition, so, we can do our case studies to evaluate whether the optional composition is a solution for the optional feature problem or not. Therefore, the next chapter deals with the case studies.



## 5. Case Study

In this chapter, two case studies are presented to compare the different approaches, introduced in [Chapter 3](#), for the implementation of possible solutions for the optional feature problem. The implementations are done with the software derivatives, preprocessor and optional composition. The two case studies are representative examples for feature interactions and are suitable to compare advantages and disadvantages of the different approaches. First of all, we introduce the expression problem as case study, which is a simple example as usual and it demonstrates the problems well, in a small area. Because the expression problem is more a generic example, a chat implementation will be introduced afterwards as a more concrete and practical case. The different facets of a chat are features for an application. Former case studies have shown the complexity of this domain [[Sch09](#)]. We will see, that these features also may interact, which are resolved with the different approaches as well. At the end of each case study, we discuss in short the advantages and disadvantages of solving the optional feature problem. In the following [Chapter 6](#), we use the case studies as a basis to abstract the advantages and disadvantages of the approaches in general and try to make a recommendation for their usage.

### 5.1 Expression Problem

The expression problem is a well-known problem [[Rey94](#), [KFF98](#), [Coo90](#)] in programming languages to introduce new methods and data types in a type safe manner. The expression problem is described as fundamental problem of variability in [Software Product Lines \(SPLs\)](#)<sup>1</sup>.

Here, we introduce the expression problem, which is a general concept in programming tasks to compose complex expressions out of smaller and less complex expressions. We focus on mathematical problems as an obvious example, where a complex equation is composed out of smaller expressions like a couple of additions and subtractions. Each

---

<sup>1</sup>see <http://www.cs.utexas.edu/~schwartz/ATS/EPL/index.html>

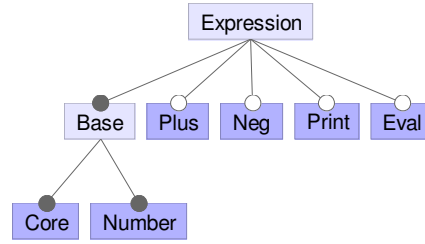


Figure 5.1: Feature model of SPL expression problem

of these operations are expressions implemented as features. For a business variation of a calculator just a small subset of operations is required, whereas a scientific variation requires a lot more operations and consequently more features. Figure 5.1 shows an example of a feature model of the expression problem applied as base for the case study. We focus on the binary mathematical operation addition (*Plus*), the unary mathematical operation negation (*Neg*), the evaluation operator (*Eval*) and a print statement to present the expression (*Print*). These features are optional ones and sufficient to demonstrate the interaction problems and the methods to solve them, because the implementation of other operators will be straightforward. Indeed, beside these operators we also need terminal symbols for the operations, so additionally we introduce a feature *Number*. The main application with some test cases is implemented in a feature called *Core*. *Core* and *Number* are mandatory features.

	Core	Number	Print	Eval	Plus	Neg
Core	-	-	-	-	-	-
Number	x	-	-	-	-	-
Print	x	x	-	-	-	-
Eval	x	x	-	-	-	-
Plus	x	-	x	x	-	-
Neg	x	-	x	x	-	-

Table 5.1: Feature interactions of SPL expression problem

The features *Plus* and *Neg* must be created by the base application, therefore, they interact with the *Core* feature. The features *Print* and *Eval* add a print method and an evaluation method to the features *Number*, *Plus* and *Neg*. And because these methods must be called, the features *Print* and *Eval* interact with the *Core*. Table 5.1 summarizes all interactions of the SPL. These interactions will be resolved with the approaches introduced in the previous chapters.

We use the implementation of the derivative approach from Batory<sup>2</sup>, the implementation of the preprocessor approach from Kästner and our own implementation of the optional composition approach. In the next section, we start with the software derivative approach.

<sup>2</sup>see <http://www.cs.utexas.edu/~schwartz/ATS/EPL/index.html>



### 5.1.1 Software Derivatives

In this section, we introduce the software derivative implementation of the expression problem. This *SPL* contains five features: *Print*, *Eval*, *Num*, *Plus* and *Neg*. These features are structuring nodes in the feature model in Figure 5.2. The source code is implemented within the software derivatives in the branch of the *All* feature. These derivatives cannot be selected by the user, they are automatically chosen, when the feature requires an accordant derivative. The names of the derivatives are abbreviated

Abbreviation	Derivative of
CK	Core and Kore
CE	Core and Eval
CP	Core and Print
BK	BaseNumber and Kore
BE	BaseNumber and Eval
BP	BaseNumber and Print
NK	Neg and Kore
NE	Neg and Eval
NP	Neg and Print
PK	Plus and Kore
PE	Plus and Eval
PP	Plus and Print

Table 5.2: Abbreviations from the feature model

(Figure 5.2), so the abbreviations and the acceptations of them are listed in Table 5.2. All features from the left branch with the nodes *Ops* and *Str* interact with any feature of the middle branch with the nodes *Structs* and *Op*. Batory dissolved all interactions into the derivatives, so there is no feature implementing source code except the derivative features. For each feature the depending functionality has to be implemented in an own derivative. The *Core* feature has a core functionality and is directly interacting with *Plus* and *Eval*. Because of the subdivision into derivatives, a separate implementation for core is required, which is named *Kore* to avoid name clashes, for all these possible interactions. Just for our small expression example with just two expressions, 12 derivatives had to be implemented.

#### Implementation

The derivative *CK* implements the core functionality, i.e. the base test class containing the main method. The derivative *CE* contains the evaluation and the derivative *CP* includes the print functionality for the core feature. Analog to this the derivatives *BK*, *BE*, *BP*, *NK*, *NE*, *NP*, *PK*, *PE* and *PP* implement core, evaluation and printing functionalities to base number (*Num*), negation (*Neg*) and addition (*Plus*), respectively.

These explained derivative features has been implemented within five classes: *Test*, *Num*, *Exp*, *Neg* and *Plus*. The roles which these classes play in the several features are

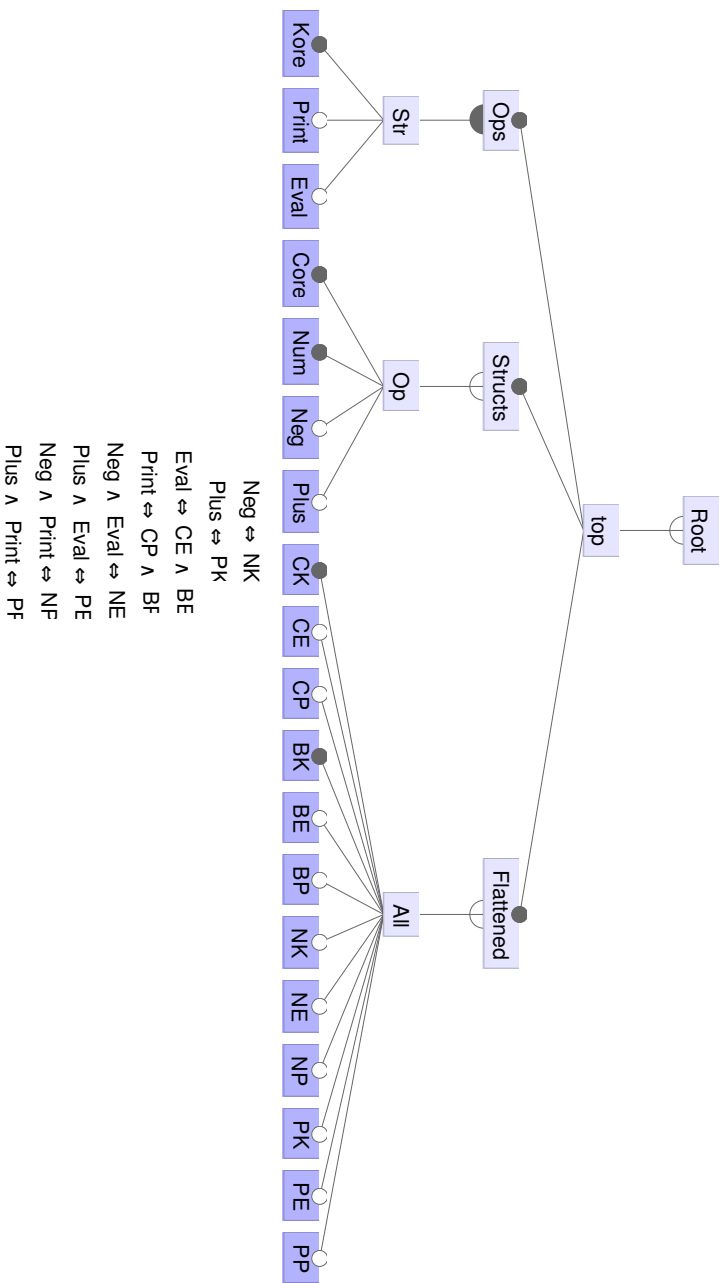


Figure 5.2: Feature model of SPL expression problem - Software Derivatives

listed in the collaboration model in Figure 5.3. Test represents the main class, Num represents the base number, Exp stands for expression, Neg implements the negation and Plus implements the addition.






















	Exp	Test	Plus	Num	Neg
CK					
CE					
CP					
BK					
BE					
BP					
NK					
NE					
NP					
PK					
PE					
PP					

Figure 5.3: Collaboration model of SPL expression problem - Software Derivatives

This implementation of the SPL has just feature interactions of first order, i.e. two features interacting with each other. Thus, we introduce one example to illustrate the implementation of this approach. Let us take a look at the features *Core* and *Print*, i.e. we take the derivatives *CK* and *CP* into consideration. *CK* contains the interface *Exp* and the class *Test*. *Exp* is the interface for all expressions that appear in this SPL. Within this derivative this interface is introduced without any abstract methods (see Listing 5.1).

The class *Test* from *CK* (see Listing 5.2) adds a class declaration and introduces class variables and the empty main method declaration. The derivative *CP* also contains the interface *Exp* and the class *Test*. So, it refines the interface *Exp* by adding the abstract *toString* method (see Listing 5.3). *CP* changes the class *Test* by refining the main method by adding a call of the *printtest()* method and adding the *printtest()* method (see Listing 5.4).

## Evaluation

An advantage of this approach is that the feature interactions are dissolved without redundancy. Furthermore the **Feature-Oriented Programming (FOP)** is usable for this approach without any changes. For this SPL five features are mapped into twelve software derivatives, which is manageable for the introduced SPL. But to come back to the initial variation of the business calculator, three additional expressions (subtraction, multiplication and division) might be added. Per expression three derivatives are generated, because any expression has to be evaluated printed and implemented to the core. So nine additional derivatives would be necessary to represent this features. Because a scientific variant would have more expressions, the number of derivatives would

```
1 layer CK;  
2 interface Exp {  
3 }
```

Listing 5.1: Interface Exp, feature Core (CK) - Software Derivatives

```
1 layer CK;  
2 class Test {  
3     static Exp e;  
4     public static void main(String args[]) {  
5         }  
6 }
```

Listing 5.2: Class Test, Feature Core (CK) - Software Derivatives

```
1 layer CP;  
2 refines interface Exp {  
3     String toString();  
4 }
```

Listing 5.3: Interface Exp, Feature Print (CP) - Software Derivatives

```
1 layer CP;  
2 refines class Test {  
3     public static void main(String args[]) {  
4         Test.printttest();  
5         Super(String[]).main(args);  
6     }  
7  
8     static void printttest(){  
9         }  
10 }
```

Listing 5.4: Class Test, Feature Print (CP) - Software Derivatives

increase dramatically. Here, the feature model (see Figure 5.2) is confusing, because of the derivative features and the structural features, but for presentation and discussions they can be hidden. Nevertheless, the derivatives have to be implemented and maintained. A more detailed discussion is done in Chapter 6.

In this part, we described the implementation of the software derivative approach for the expression problem SPL. In the next section, we introduce the implementation of preprocessor approach for this SPL.

### 5.1.2 Preprocessor approach

In this section, we present a method to resolve feature interactions with a preprocessor. In some high level programming languages like C++ and C a preprocessor step is a common and necessary step during compilation, thus other languages like Java does not require a preprocessor, but it can be used as an extension to the language. In languages where the preprocessor is already available, it would be an obvious way to resolve feature interactions and to assure that unnecessary parts of the source code will not be compiled into the resulting executable. The preprocessor approach uses preprocessor statements to mark the features, as we explained in Section 2.2.1 and Section 3.3.

#### Implementation

As example for the preprocessor implementation, we use the same feature model of the expression problem case study as shown in Figure 5.1. A code detail of the implementation of the class Test is shown in listing Listing 5.5, while the detail of implementation of the class Plus is shown in listing Listing 5.6. The *Plus* feature needs an evaluation, a printing and a call in the core. So, it interacts with the three features *Eval*, *Print* and *Core*. The implementation of the interaction with *Eval* and *Print* is shown in lines 21-28 and lines 42-50 of Listing 5.5 and in Listing 5.6 We have shaded the several features to not losing the track of the features.

#### Evaluation

An advantage of the preprocessor approach is, that a preprocessor is a well-known and accepted programming tool and is an inherent part of the compilation process in some programming languages. The functionality of a preprocessor can also be integrated in languages where preprocessors are not inherent like Java, where we were using an extension. So, existent systems can be extended with the preprocessor approach. Listing 5.5 shows a disadvantage of the preprocessor approach: The interactive code is tangled in this class. So, the Separation of Concerns (SoC), the major goal of SPL implementations, is not achieved. The features are not modularized, because they are marked with preprocessor statements in the main class. Colors may be used to support the interpretation of the source code, but the higher the number of expressions, the more preprocessor statements and consequently macros are needed. Furthermore, the preprocessor macros can not overlap (see line 21-28 Listing 5.5), which results in extreme cases, when more than two features interact. All these problems are not fatal for this special case, but the class Test becomes more and more confused, if more example code

```

1  class Test {
2      static Exp e;
3      public static void main( String args[]){
4  #ifdef PRINT
5          Test.printttest();
6  #endif PRINT
7  #ifdef EVAL
8          Test.evaltest();
9  #endif EVAL
10     }
11 #ifdef EVAL
12     static void evaltest(){
13 #ifdef NUM
14         e=new Num(1);
15         System.out.println("eval(1)␣␣" + e.eval());
16 #endif NUM
17 #ifdef NEG
18         e=new Neg(new Num(1));
19         System.out.println("eval(Neg(1))␣␣" + e.eval());
20 #endif NEG
21 #ifdef PLUS
22         e=new Plus(new Num(1),new Num(2));
23         System.out.println("eval(1+2)=" + e.eval());
24 #ifdef NEG
25             e=new Neg(new Plus(new Num(1),new Num(2)));
26             System.out.println("eval(-(1+2))=" + e.eval());
27 #endif NEG
28 #endif PLUS
29     }
30 #endif EVAL
31 #ifdef PRINT
32     static void printttest(){
33 #ifdef NUM
34         e=new Num(3);
35         System.out.println("print(3)␣␣" + e);
36 #endif NUM
37 #ifdef NEG
38         e=new Neg(new Num(5));
39         System.out.println("print(Neg(5))␣␣" + e);
40 #endif NEG
41 #ifdef PLUS
42         e=new Plus(new Num(5),new Num(7));
43         System.out.println("print(5+7)␣␣" + e);
44 #endif PLUS
45     }
46 #endif PRINT
47 }

```

Listing 5.5: Class Test - Preprocessor approach

```

1 package tmp;
2 #ifndef PLUS
3 class Plus implements Exp {
4     Exp x;
5     Exp y;
6     Plus( Exp x, Exp y) {
7         this.x=x;
8         this.y=y;
9     }
10 #ifndef EVAL
11     public int eval() {
12         return x.eval() + y.eval();
13     }
14 #endif EVAL
15 #ifndef PRINT
16     public String toString() {
17         return x + "⌋+⌋" + y;
18     }
19 #endif PRINT
20 }
21 #endif PLUS

```

Listing 5.6: Class Plus - Preprocessor approach

will be added. We discuss the advantages and disadvantages of this approach in more detail in [Chapter 6](#) during comparison with the other approaches.

In this part, we introduced the implementation of the preprocessor approach for the case study of the expression problem and in the next section, we show the approach of the optional composition for this case study.

### 5.1.3 Optional Composition

In this section, we introduce the implementation of the optional composition approach for the case study of the expression problem. As we distinguished in previous chapters between explicit and implicit optional composition, we focus in this implementation on the implicit optional composition. But in some configurations we need the explicit version for implementation details, so we use a combination of the implicit and explicit optional composition. We use this hybrid approach, because the explicit optional composition is similar to the preprocessor approach and the implicit approach cannot dissolve all interactions. For example, an interaction between two optional features and a mandatory feature cannot be realized with the implicit approach, because the classes of the mandatory feature will always exist, and so the refinements cannot be ignored, when one of the optional features is selected. Therefore we use the hybrid approach for the case study.

## Implementation

Again we are implementing the expression problem with the feature model shown in Figure 5.1 The collaboration model in Figure 5.4 shows the roles the classes play in



















	Exp	Test	Plus	Num	Neg
Core					
Number					
Eval					
Print					
Plus					
Neg					

Figure 5.4: Collaboration model of SPL epression problem - Optional Composition

the collaborations (features) as it was explained in Section 2.2.2. As we stated in the section before, any mathematical expression has to be evaluated, printed and used in the main program. Therefore we show the implementation of the *Plus* feature and its interactions with *Print*, *Eval* and *Core*. As shown in Listing 5.7, the class *Plus* of feature *Plus* implements the constructor for *Plus* and class variables. The feature *Print* refines the class *Plus* if feature *Plus* is selected, by adding a `toString()` method returning the string of the expression (see Listing 5.8) to the code of base source code of the class *Plus* in Listing 5.7. Listing 5.9 shows, that the feature *Eval* refines the class *Plus* if feature *Plus* is selected, by adding an `eval()` method returning the result of the addition. Listing 5.10 shows the integration of the *Plus* feature into the main program, if feature *Print* and/or feature *Eval* was selected. In the method `printtest()` (lines 3-7 of Listing 5.10) the first call is the **Super**()- call, which calls the previous version of this method in order of the composition of the features. Then, an object of *Plus* is generated and printed to the console output. The method `eval()` in lines 8-12 of Listing 5.10 also contains a **Super**()- call to bind the other versions of this method into the variant during compilation. Here, an object of *Plus* is generated as well and the result of its `eval()` method is printed. The **optional** tags in line 3 and 8 are needed, because this class is introduced by feature *Core* and neither by *Print* nor by *Eval* so this methods would be compiled anyway if the **optional** tag was left out, no matter what features are selected except *Plus*.

Now, let us come to a problem which occurred during the straightforward implementation of this approach. For this case study this approach could not be compiled, because the order of composition hindered it. To explain this behavior, we list the order of composition: *Core*, *Num*, *Print*, *Eval*, *Plus*, *Neg*. And here is the trap: *Print* and *Eval* refine *Num*, *Plus* and *Neg*, but when *Print* comes along *Plus* and *Neg*, it does not refine them, because they do not exist at this moment of compilation. So, we decided to reorder and that was the new order: *Core*, *Num*, *Plus*, *Neg*, *Print* and *Eval*. But this is not the whole truth - it clashes during compilation again. The clash comes from the problem, that *Num*, *Plus* and *Neg* refine the `printtest()` method in the main class *Test*, which is introduced not till the refinement of *Print* takes effect. For the `eval()` method



```

1 layer Plus;
2 public class Plus implements Exp{
3     Exp x;
4     Exp y;
5     Plus (Exp x, Exp y){
6         this.x = x;
7         this.y = y;
8     }

```

Listing 5.7: Class Plus (feature *Plus*) - Optional Composition

```

1 layer Print;
2 public refines class Plus {
3     public String toString(){
4         return x + "⌋+⌋" + y;
5     }
6 }

```

Listing 5.8: Class Plus (feature *Print*) - Optional Composition

```

1 layer Eval;
2 public refines class Plus {
3     public int eval(){
4         return x.eval() + y.eval();
5     }
6 }

```

Listing 5.9: Class Plus (feature *Eval*) - Optional Composition

```

1 layer Plus;
2 public refines class Test {
3     optional Print static void printtest(){
4         Super().printtest();
5         e = new Plus( new Num(5), new Num(7));
6         System.out.println( "print(5+7)⌋=⌋" + e );
7     }
8     optional Eval static void evaltest(){
9         Super().evaltest();
10        e = new Plus( new Num(1), new Num(2));
11        System.out.println("eval(1+2)=" + e.eval());
12    }
13 }

```

Listing 5.10: Class Test (eature *Plus*) - Optional Composition

in the class *Test*, the problem is analog. We found a workaround for this case study, which can not be generalized, but works for this special case. The feature *Num* is a mandatory feature, *Print* is optional. So, we shifted the introduction of the `printtest()` method from the *Print* feature to the *Num* feature. We shifted the introducing `eval()` method from *Eval* to *Num* as well. So the case study was able to compile, which is important for the discussion later on in [Chapter 6](#).

## Evaluation

An advantage of the optional composition is that it works implicitly, i.e. the refinements are ignored, if the original class does not exist. This is an advantage, because the programmer does not have to have the feature model in mind during programming. But this leads also to a disadvantage. We had to switch off the error propagation to realize this implicit working, so debugging and maintaining might be hard even impossible, when more expressions are added to this [SPL](#), because the compiler cannot detect, whether this refinement should be ignored or the base class is missing based on an error. So, the explicit version of the optional composition can be seen advantageous with its optional keyword. But here a disadvantage can be seen as well: The annotations with the optional keyword on the method layer are similar to the `#ifdef` statements of the preprocessor, and so the source code can also become confusing, if the [SPL](#) of this case study would be extended with additional mathematical operations. We discuss the advantages and disadvantages of this approach more detailed in [Chapter 6](#).

Now, having presented the case study of the expression problem with the three approaches as solutions to the optional feature problem, we saw that the approach of software derivatives is the most expensive one with respect to the manual effort of resolving and encapsulating the feature interactions, but it separates the features from their interactions considerably. The preprocessor approach “pollutes” the source code with the preprocessor annotations and the optional composition approach did just work with a trick for this case study. Because of this awareness and the fact, that this case study was more or less a generic example, we will present a chat implementation as a case study, in the next section.

## 5.2 Chat SPL

In this part, we introduce the case study with the implementation of a chat SPL. There exist a lot of chat programs, like pidgin, trillian, skype, on several platforms. All of them have in common, that they provide a contact list and the user can chat with the people from this list. They use a protocol for the transfer of the messages, the messages can be colored and be decorated with emoticons. We adopted the chat SPL from Schulze [Sch09] after the domain analysis. We figured out the most requirements for the messages, the message history, the format of a message, the encryption of a message and the support of message protocols. Further requirements are authentication, logging, a secure connection and a selectable user interface. Logging achieves the crosscutting concern and leads to higher order feature interactions. We summarized the features in a feature model in Figure 5.5.

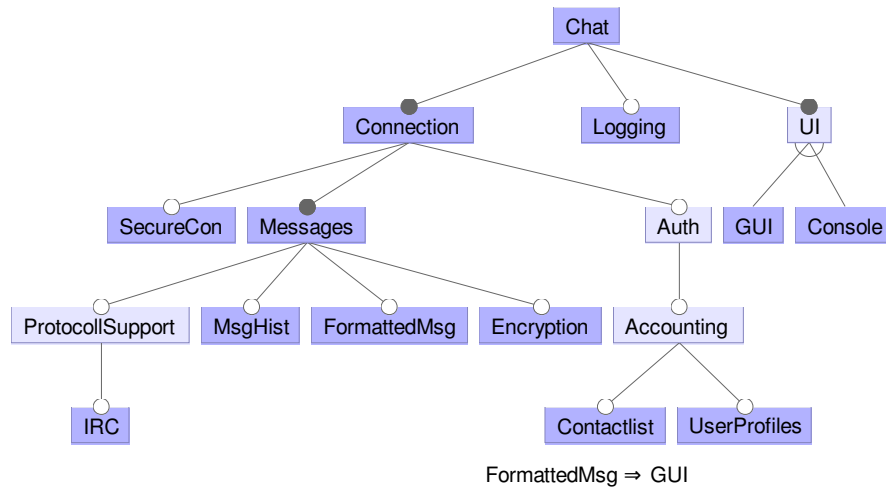


Figure 5.5: Feature model chat SPL similar to [Sch09]

The feature *Chat* represents the base application, which contains the client and server architecture. The implementation of the connection is held by the mandatory feature *Connection*. This connection can be secured with the optional feature called *SecureCon*. The mandatory feature *Messages* contains the text message implementation which can be extended by four optional features. The structural feature *ProtocolSupport* has just the feature *IRC* which stands for the chat protocol. Other alternative protocols like HTTP or FTP would be possible, but for the discussion of the principle feature interaction one protocol is sufficient. Another optional feature is the message history (*MsgHist*), which writes outgoing and incoming messages to a history file. Additionally, we have a feature to change the message format (*FormattedMsg*), so the user can change the color of the text. To secure the message transfer, some encryption would be necessary. Therefore, we added an optional *Encryption* feature. The last features belong to the branch of the node *Connection* and implement the authentication. The *Contactlist* feature contains the contacts and the *UserProfiles* feature implements the authentication of the user. Because a chat application is a distributed application and we have a lot of features manipulating the base program, we need some debugging mechanism to

	Chat	Connection	SecureCon	MsgHist	Messages	Console	GUI	IRC	Contactlist	FormattedMsg	UserProfiles	Encryption	Logging
Chat	-	-	-	-	-	-	-	-	-	-	-	-	-
Connection		-	-	-	-	-	-	-	-	-	-	-	-
SecureCon	x		-	-	-	-	-	-	-	-	-	-	-
MsgHist	x			-	-	-	-	-	-	-	-	-	-
Messages					-	-	-	-	-	-	-	-	-
Console	x					-	-	-	-	-	-	-	-
GUI	x						-	-	-	-	-	-	-
IRC	x							-	-	-	-	-	-
Contactlist						x	x		-	-	-	-	-
FormattedMsg							x			-	-	-	-
UserProfiles	x	x				x	x				-	-	-
Encryption				(x)	(x)	(x)	(x)	(x)		(x)	(x)	-	-
Logging	x	x	x	x	x	x	x	x	x	x	x	x	-

Table 5.3: Feature interactions of chat SPL

monitor the different stages of the program during processing. So, the optional feature *Logging* adds a console output to every method of any feature and therefore interacts with all features. The right branch contains the user interface, which is separated into two alternative features: the feature *GUI* is an implementation for the [Graphical User Interface \(GUI\)](#) and the feature *Console* for a console variant. Most interactions are between the features representing the user interface and the other features. We summarized the feature interactions in [Table 5.3](#). The “x” in the table represents the interactions between two features and the “(x)” implies interactions between three features. In our case the third feature is the *Logging*. The feature *SecureCon* interacts with the *Chat* feature (the base application), because the Client and Server class must implement the [Secure Sockets Layer \(SSL\)](#) protocol to guarantee the secure connection between the clients and the server. The *MsgHist* feature also interacts with the base application, because the messages are logged from the server and the client. The user interface features interact with the base feature as well, because they must be created by the application. The [Internet Relay Chat \(IRC\)](#) protocol must be implemented to the client and server, so that the feature *IRC* also interacts with the base feature *Chat*. The feature *Contactlist* interacts with both user interface features, *GUI* and *Console*, because the contacts have to be shown to the user. The feature *FormattedMsg* interacts just with *GUI*, because it colors the text messages, which is not possible in the console variant. The authentication is done by the feature *UserProfiles*. The user must log in to the server, so *UserProfiles* interact with *Chat*, the connection must be activated, so this feature interacts with the *Connection* feature. *UserProfiles* interact with the

user interface features as well, thus, the user can enter the user name and password for authentication. Now, let us come to the *Encryption* feature, which has all the interactions between three features. The third feature of this combination is always the *Logging* feature. Messages have to be encrypted and decrypted, so it is obvious, that *Messages*, *FormattedMsg* and *UserProfiles* interact with the *Encryption*. For displaying the messages to the user, the user interfaces need the opportunity to decrypt the text, and so they are also interacting with *Encryption*. As we already said, the *Logging* feature interacts with all features, for debugging purposes.

To dissolve the feature interactions we use the same approaches as in the first case study (see [Section 5.1](#)). Here we introduced the principle structure of the chat and the functional range, that each feature represents. In the following section, we describe the implementation of the feature interactions by the approach of software derivatives.

### 5.2.1 Software Derivatives

In this part we describe the implementation of the approach of software derivatives for the chat case study. The approach of software derivatives dissolves the feature interactions from the features into derivative features (see [Section 3.2](#)).

#### Implementation

As we have already seen in the expression problem case study ([Section 5.1.1](#)), the feature model for the software derivatives is more complex than for the other approaches. We already said, that the derivative features can be hidden, but this does not change the need of implementing them. We have dissolved the interactions into 28 derivatives shown in the feature model in [Figure 5.6](#). Because we have not only feature interactions of two features like in the expression problem case study, we divided the branch of the derivatives in the feature model into two branches. One branch for derivatives of two interacting features and the other branch for the derivatives of three interacting features.

In this model we count 22 interactions with two features and 7 interactions with three features. [Figure 5.7](#) shows the collaboration model of the software derivative version of this chat. We see that 15 classes play 62 roles in 42 features.

Now, let us introduce an example to illustrate a feature interaction of first and second order. We take a look at the features *GUI* (as one representative of a user interface), *Encryption* and *Logging*. When a chat has an encryption, the message must be encrypted when send from the input of the *GUI* to the client and decrypted when displayed to the user in the *GUI*. We dissolved this interaction in the *GUI\_Enc* derivative feature. This derivative contains the changes to the class *Gui* (see [Listing 5.11](#)). [Listing 5.12](#) shows the interaction between the *GUI* and the *Logging* feature. To any method of the class *Gui* we add a status message, which is printed at the console output. The derivative *Log\_Gui\_Enc* contains the interaction of all three features: *GUI*, *Encryption* and *Logging*. We added a console output to the *newChatLine()* method of class *Gui*, because the message is encrypted at this moment (see [Listing 5.13](#)).

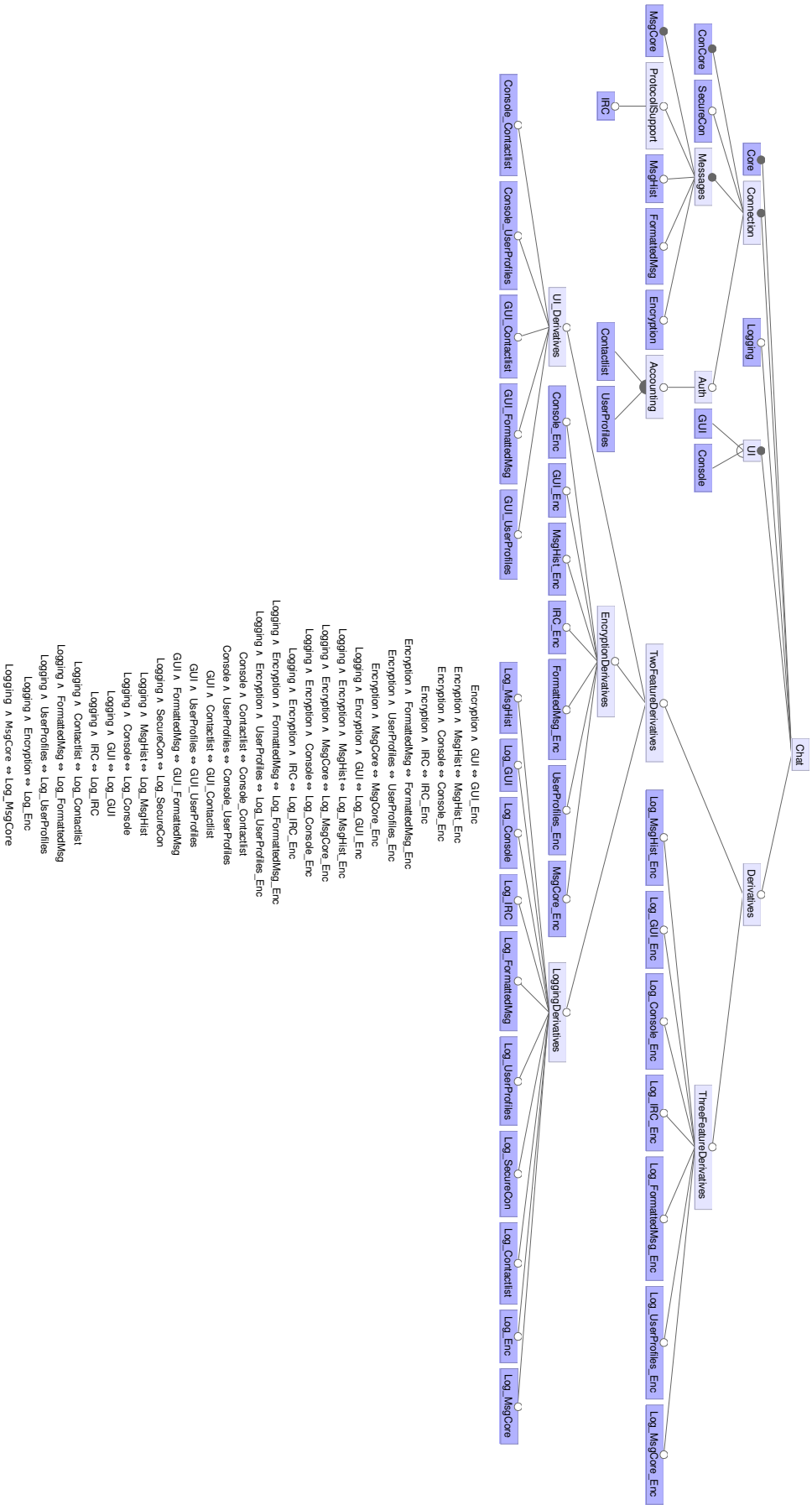


Figure 5.6: Feature model of chat - Software Derivatives

	AuthMessage	ChatLineListener	Client	Connection	Console	Contacts	Encryption	FormattedTextMessage	Gui	MsgLogger	Reverse	Rot13	Server	TextMessage	UserProfiles
Core															
ConCore															
SecureCon															
MsgCore															
Console															
GUI															
IRC															
FormattedMsg															
ContactList															
MsgHist															
UserProfiles															
Encryption															
Logging															
Console_Enc															
Console_UserProfiles															
Console_Contactlist															
FormattedMsg_Enc															
GUI_Contactlist															
GUI_Enc															
GUI_FormattedMsg															
GUI_UserProfiles															
IRC_Enc															
MsgCore_Enc															
MsgHist_Enc															
UserProfiles_Enc															
Log_Console															
Log_Contactlist															
Log_Enc															
Log_FormattedMsg															
Log_GUI															
Log_SecureCon															
Log_IRC															
Log_MsgCore															
Log_MsgHist															
Log_UserProfiles															
Log_Console_Enc															
Log_FormattedMsg_Enc															
Log_GUI_Enc															
Log_IRC_Enc															
Log_MsgCore_Enc															
Log_MsgHist_Enc															
Log_UserProfiles_Enc															

Figure 5.7: Collaboration model of chat - Software Derivatives

```
1 layer derGUI_Enc;  
2 refines class Gui{  
3     public void newChatLine(TextMessage tm) {  
4         //encrypt/decrypt with rot13 and reverse  
5         tm.setContent(tm.getContent());  
6         Super(TextMessage).newChatLine(tm);  
7     }  
8 }
```

Listing 5.11: Class Gui (Derivative GUI/Encryption) - Software Derivatives

```
1 layer derLog_GUI;  
2 refines class Gui{  
3     refines Gui(String title, Client client){  
4         System.out.println("Gui_created");  
5     }  
6     public void createAndShowGui() {  
7         System.out.println("Gui.createAndShowGui()");  
8         Super().createAndShowGui();  
9         this.repaint();  
10    }  
11    public void newChatLine(TextMessage tm) {  
12        System.out.println("Gui.newChatLine(TextMessage_tm)");  
13        Super(TextMessage).newChatLine(tm);  
14    }  
15    public void actionPerformed(ActionEvent evt) {  
16        System.out.println("Gui.actionPerformed(ActionEvent_evt)");  
17        Super(ActionEvent).actionPerformed(evt);  
18    }  
19 }
```

Listing 5.12: Class Gui (Derivative GUI/Logging) - Software Derivatives

```
1 layer derLog_GUI_Enc;  
2 public refines class Gui {  
3     public void newChatLine(TextMessage tm) {  
4         System.out.println("GUI:_new_chatline_with_encrypted_message");  
5         Super(TextMessage).newChatLine(tm);  
6     }  
7 }
```

Listing 5.13: Class Gui (Derivative GUI/Logging/Encryption) - Software Derivatives



## Evaluation

The feature model of this approach displays the manual effort of dissolving the feature interactions. The feature model increases from original 13 implementing features up to 42 features. This is more than a triplication of the number of features. Here the manual effort for a clear decomposition is quite high compared to the advantage of encapsulation of the feature interactions.

Here, we presented the chat case study for the part of software derivatives and in the next part we introduce the preprocessor approach for this case study.

## 5.2.2 Preprocessor Approach

In this section we present the implementation of the chat case study with the preprocessor approach. We saw in [Section 5.1.2](#) that the usage of preprocessor statements leads to confusing source code, when these statements are applied to single statements. Therefore, we decided to use the preprocessor in combination with the FOP suggested by Kästner [[KA08](#)] and described in [Section 3.3](#). So, we use the preprocessor statements to leave out methods and classes, but not for annotating fine granular statements. Thus, this approach becomes comparable to the other approaches.

### Implementation

For the implementation, we use the feature model introduced in [Section 5.2](#) and shown in [Figure 5.5](#). We implemented the chat with this approach, but we limit ourselves to describe one representative example. As this example, we discuss the preprocessor with the interaction between *GUI*, *Encryption* and *Logging* as in [Section 5.2.1](#). [Listing 5.14](#) shows the interaction code of *GUI* and *Encryption* which is located in the feature *Encryption*. This class refines the method `newChatLine()` with the encryption or decryption. The intersection of these two features is done by nested preprocessor statements (see lines 1-2 of [Listing 5.14](#)). The feature interaction of *GUI*, *Logging* and *Encryption* is shown in [Listing 5.15](#). In this source code we have the nested preprocessor statements as well. But we can see a specific characteristic in this listing. To realize the feature interaction of all three features, we had to implement the method `newChatLine()` twice, one with *Encryption* and the other without (see lines 12 - 23 [Listing 5.15](#)).

### Evaluation

As in the previous case study, we have the advantage that the preprocessor is an inherent part of programming languages and so existent systems can be extended easily. A disadvantage of this approach is the redundant code produced by a feature interaction of three features. This contradicts to the aim of the SPL to avoid redundant code. This disadvantage comes from the fact, that we limited the usage of preprocessor statements to the level of classes and methods as suggested in the Linux guidelines<sup>3</sup>, so that we cannot exclude single statements. This limitation we did to be comparable to the other approaches, but therefore, we produce code replication for interactions between more than two features. Nevertheless, we achieve with these restrictions a better SoC and modularity, because we apply the preprocessor statements to the FOP source code. We give a more detailed discussion in [Chapter 6](#).

In this section, we described the implementation of the preprocessor approach for this case study. In the next part, we will introduce the implementation with the optional composition approach for this chat case study.

---

<sup>3</sup>see /Documentation/SubmittingPatches in the Linux source

```

1
2 #ifdef Encryption
3 #ifdef GUI
4 refines class Gui{
5     public void newChatLine(TextMessage tm) {
6         //encrypt/decrypt with rot13 and reverse
7         tm.setContent(tm.getContent());
8         Super(TextMessage).newChatLine(tm);
9     }
10 }
11 #endif GUI
12 #endif Encryption

```

Listing 5.14: Class Gui (feature *Encryption*) - Preprocessor

```

1
2 #ifdef GUI
3 #ifdef Logging
4 refines class Gui{
5     refines Gui(String title, Client client){
6         System.out.println("Gui_created");
7     }
8     public void createAndShowGui(){
9         System.out.println("Gui.createAndShowGui()");
10        Super().createAndShowGui();
11        this.repaint();
12    }
13 #ifdef Encryption
14     public void newChatLine(TextMessage tm) {
15         System.out.println("Gui.newChatLine(TextMessage_tm)\n\
16         encryption on");
17         Super(TextMessage).newChatLine(tm);
18     }
19 #endif Encryption
20 #ifndef Encryption
21     public void newChatLine(TextMessage tm) {
22         System.out.println("Gui.newChatLine(TextMessage_tm)");
23         Super(TextMessage).newChatLine(tm);
24     }
25 #endif Encryption
26     public void actionPerformed(ActionEvent evt) {
27         System.out.println("Gui.actionPerformed(ActionEvent_evt)");
28         Super(ActionEvent).actionPerformed(evt);
29     }
30 #endif Logging
31 #endif GUI

```

Listing 5.15: Class Gui (feature *Logging*) - Preprocessor

### 5.2.3 Optional Composition

In this section, we describe the implementation of the optional composition approach for the chat case study. We use again the feature model shown in [Figure 5.5](#) and described

	AuthMessage	ChatListener	Client	Connection	Console	Contacts	Encryption	FormattedTextMessage	Gui	MsgLogger	Reverse	Rot13	Server	TextMessage	UserProfiles
Chat															
Connection															
SecureCon															
Messages															
Console															
GUI															
IRC															
FormattedMsg															
ContactList															
MsgHist															
UserProfiles															
Encryption															
Logging															

Figure 5.8: Collaboration model of chat - Optional Composition

in [Section 5.2](#). We use the combination of the explicit and implicit optional composition as well as in [Section 5.1.3](#). We use this hybrid approach, because the implicit variant can not dissolve all interactions and the explicit variant is similar to the preprocessor approach. To get an overview of the implemented classes and their refinements, we show in [Figure 5.8](#) the collaboration model for this approach. There, we have 15 classes and 13 features, and we can see the refinements the features apply to the classes. Here, 15 classes play 49 roles in 13 features. So, *Encryption*, for example, introduces the classes *Encryption*, *Rot13* and *Reverse* and refines the classes *AuthMessage*, *Client*, *Console*, *Gui* and *TextMessage*.

### Implementation

We implemented the whole *SPL* with this approach, but demonstrating all feature interactions here would go beyond the scope of this thesis. So, let us come to an implementation example, where we again take *GUI*, *Encryption* and *Logging* into consideration. The interaction between *GUI* and *Encryption* is represented by the refinement of the class *Gui* by the *Encryption* feature is shown in [Listing 5.16](#). The interaction

```
1 refines class Gui{  
2   public void newChatLine(TextMessage tm) {  
3     //encrypt/decrypt with rot13 and reverse  
4     tm.setContent(tm.getContent());  
5     Super(TextMessage).newChatLine(tm);  
6   }  
7 }
```

Listing 5.16: Class Gui (feature *Encryption*) - Optional Composition

```
1 import java.awt.event.ActionEvent;  
2 refines class Gui{  
3   refines Gui(String title, Client client){  
4     System.out.println("Gui_created");  
5   }  
6   public void createAndShowGui() {  
7     System.out.println("Gui.createAndShowGui()");  
8     Super().createAndShowGui();  
9     this.repaint();  
10  }  
11  public void newChatLine(TextMessage tm) {  
12    System.out.println("Gui.newChatLine(TextMessage_tm)");  
13    Super(TextMessage).newChatLine(tm);  
14  }  
15  public void actionPerformed(ActionEvent evt) {  
16    System.out.println("Gui.actionPerformed(ActionEvent_evt)");  
17    Super(ActionEvent).actionPerformed(evt);  
18  }  
19 }
```

Listing 5.17: Class Gui (feature *Logging*) - Optional Composition

of the features *GUI* and *Logging* is resolved by the refinement of class `Gui` shown in [Listing 5.17](#). To compare this to the preprocessor approach (see [Listing 5.15](#)) we see, that the method `newChatLine()` is just implemented once. The optional composition approach is not designed to handle more than one method in different occurrences in one class, so the implementation of a feature interaction of second order is impossible. We will discuss potential solutions of this problem in the discussion in [Chapter 6](#).

## Evaluation

In this part of the case study, we discovered, that with the optional composition approach, just feature interactions of two features can be dissolved. This is a disadvantage compared to the other approaches. The advantages and disadvantages introduced in [Section 5.1.3](#) hold for this case study as well. So, we discuss the advantages and disadvantages in detail in [Chapter 6](#).

In this chapter, we introduced a case study expression problem and a chat application. We gave a short evaluation for each approach and each case. We identified the conspicuous details of the approaches in significant code examples. The case studies are designed as small examples, that may show tendencies, if the functional range would be extended by additional features. We discussed the properties of each approach. The case study of the expression problem had shown a lot of advantages using feature-oriented approaches, because there are just interactions between two features, but we had additional discoveries in the case study of the chat, that should be considered if an approach will be used in routine. In the next chapter, we discuss the advantages and disadvantages of these three approaches and try to give a recommendation of when to use which approach.

## 6. Discussion

In this chapter, we discuss the advantages and disadvantages of the approaches to solve the optional feature problem to figure out whether the optional composition is a suitable solution for the optional feature problem or not. We compare the approaches to each other and try to make a recommendation for their usage.

### 6.1 Comparison

[Software Product Lines \(SPLs\)](#) are used to alleviate the development of software in various variants where different features are combined. One of the main advantages is, that features interact directly. To take into account, that the different variants may focus several platforms ranging from embedded systems to personal computers the resulting application should only contain the required software modules selected by the features. So, [SPLs](#) result in high variability of an application, with a high flexibility for the developer. That means, new features can be added easily, without copying the whole source code to build up a new application. To achieve these goals, additional effort is required for a clean [Separation of Concerns \(SoC\)](#) and to avoid code replication. On the other hand, the concepts for the implementation of [SPLs](#) and to solve the optional feature problem should be maintainable. The source code should be readable and clearly structured, but code tangling and scattering are influencing these requirements. Therefore, we compare the presented approaches in the following sections, focusing on the [SoC](#), code replication and variability, that can be measured. Furthermore, we discuss the additional effort, required to the implementation and during maintenance, in order that the developers needs are not neglected. This section is subjective, but presents tendencies between the approaches. In our comparison, we focus on the hybrid variant of the optional composition during the discussion, because we realized the case study with the hybrid approach. Another reason for this is, that the preprocessor approach and the explicit optional composition approach are similar to each other, when using preprocessor statements just on the level of methods. Further, the implicit variant of optional composition cannot dissolve interactions of two optional features

with a mandatory feature, because the classes of the mandatory feature would never be absent.

### 6.1.1 Separation of Concerns

The SoC is the procedure of dividing large software into modules, as introduced in Chapter 2.

#### Software Derivatives

An advantage of the software derivative approach is, that the major goal of SPLs, the SoC, is guaranteed, because each feature interaction is dissolved in a derivative feature. as an example, the interaction between the *GUI* and the *Contactlist* feature from the chat, shown in Section 3.2, is dissolved into three modules. One module for the Graphical User Interface (GUI), one module for the list of contacts and one module for the interaction between both. So, this is a strict SoC. But, this is also a disadvantage with respect to the manual effort of dissolving these interactions (see Section 6.1.4).

For the software derivatives a software engineer has to dissolve all feature interactions in the feature model explicitly and generate the derivatives. For each derivative a new folder is attached and the correspondent classes must be created. This means a high manual effort for the developer. We suspect, that for large projects, the approach of software derivatives may become confusing, although, the concerns are really separated from each other. This approach is feasible for projects with a limited amount of feature interactions.

#### Preprocessor Approach

For the preprocessor approach in Section 5.1.2, which uses the preprocessor statements on the level of statements, the SoC is not achieved (see class *Test* in Listing 5.5), because the source code is not modularized. In this class *Test*, all features (*Core*, *Number*, *Plus*, *Neg*, *Eval* and *Print*) are contained and include or exclude source code. The features are marked by preprocessor statements and so just one source code file of each class exists. For the preprocessor approach applied to the Feature-Oriented Programming (FOP) sources, which was used in Section 5.2.2, the SoC is also not achieved, even though the features are implemented with FOP, where the preprocessor statements are just attached to classes and methods, but the code of feature interactions is contained in the features and just marked by preprocessor statements. So, we say, that this approach achieves a reduced SoC.

Developers cannot trace the features in the code, when using the preprocessor approach, except they are using tools like CIDE[KAK08], which color the source code belonging to a feature. This problem exists more with the preprocessor approach working on the level of statements, than the preprocessor working on the level of methods. But in both cases the software engineer has to define macros and must not loose track. So, we suspect, that a larger SPL might be complicated.



## Optional Composition

We found out for the optional composition approaches as well, that the SoC is reduced, because of switching the interaction code into one of the features.

Because of having a reduced SoC for the optional composition and the implicit part hides the affiliation of source code to a feature with respect to the interaction code, the software engineer can also lose track. That means, that the interaction code is not marked in the implicit part, so it is unclear to which feature or feature interaction this code belongs. This can lead to errors, which results in a higher additional effort. Therefore, we discuss this in Section 6.1.4 in more detail.

### 6.1.2 Code Replication

We have shown in Section 5.1.1 and Section 5.2.1, that the feature interactions are dissolved without code replication with the approach of software derivatives. For the other approaches, code replication must be expected under different circumstances.

	Software Derivatives	Preprocessor Approach	Optional Composition
<b>number of features</b>	19 (12)	6	6
<b>binary size</b>	4.4kB	3.4kB	4.4kB
<b>Lines of Code (LOC)</b>	140	133 (62)	133
<b>code replication</b>	0%	0%	0%

Table 6.1: Characteristics of the case study expression problem

	Software Derivatives	Preprocessor Approach	Optional Composition
<b>number of features</b>	42 (29)	13	13
<b>binary size</b>	39.7kB	37.7kB	30.1kB
<b>LOC</b>	1134	1296 (230)	1026
<b>code replication</b>	0%	4%	0%

Table 6.2: Characteristics of the case study chat

## Preprocessor Approach

In contrast to this, we demonstrated in Section 5.2.2, that the preprocessor approach needs code replication to implement the feature interactions of higher order (above first order), because of the restrictions of applying the preprocessor statements on the level of methods. Let us remember the interaction of *Logging*, *Encryption* and *GUI*. In Listing 5.15 (Section 5.2.2) we had the method `newChatLine()` implemented twice. One

for the variant with *Encryption* (lines 13-18) and one for the variant without *Encryption* (lines 19-24). For the reason of the crosscutting *Logging* feature in the chat, we had 7 feature interactions of second order, which resulted in the fact, that 4% of the source code are replicated code lines (see Table 6.2). We do not have code replication for the preprocessor approach implementation in Section 5.1.2, because we did not restrict the usage that hard, we also allowed to surround statements, as supposed by Kästner in [KA09] for disciplined annotations. So, we did not count LOC of code replication for the expression problem case study (see Table 6.1).

## Optional Composition

The optional composition approach does not produce redundant code (see Table 6.1 and Table 6.2), but as we have shown in Section 5.2.3, it is just possible to implement interactions of first order. To solve this problem, a further extension of the compiler for the explicit optional composition would be conceivable to afford multiple variants of one method and the compiler decides, which to implement for the current feature selection. But, this would result in code replication as well as for the restricted preprocessor approach.

Replicated code is always a trap for the software engineer. Mostly, these LOC are produced by copy and paste. So, probable errors are copied as well and this increases the effort for the software engineer during debugging. In our chat case study we had a code replication of 4%, because of 7 feature interactions of second order. We suspect an increase of the code replication when the feature interactions of second order rise, or in other projects with more second order feature interactions.

### 6.1.3 Variability

The variability of a SPL describes the ability to restore all variants intended by the feature model. For the expression problem case study this goal is achieved with all approaches. For the chat case study it is just achieved for the software derivatives and the preprocessor approach, at the expense of code replication. So, the chat case study, related to practice, provided further results.

## Optional Composition

With the current implementation of the optional composition approach, regardless which variant, it is not possible to implement all feature interactions. So, the size of the binary files and the LOC of the optional composition approach is lower in the chat, because of the lack of representation of feature interactions of second order (see Table 6.2). If three or more optional features do interact, the intersection of all three cannot be resolved. This is limited because of the restrictions of *Object-Oriented Programming* (OOP), whereas no multiple implementation of an identical method can exist in a class. To implement these interactions a further extension of the compiler is necessary, so that more variants of a method can be implemented and computed. But, to come back to our chat example, we had the feature interaction of second order just with the *Logging* feature. For our point of view, it is not dramatically to loose the

debugging output for the methods refined by the *Encryption* feature in times of *Integrated Development Environments (IDEs)* with debugging environments, for this case. By the way, we implemented the *Logging* feature to show an interaction of second order. Mostly these interactions can be reduced to interactions of first order by changing the implementation. So, we believe, that interactions of second order are even rare.

For software engineers the reduced variability of the optional composition approach could mean more manual effort, when reducing feature interactions of second order to first order, to guarantee the full variability. When this is not possible they have to decide carefully how to implement these feature interactions of second order, if they cannot be avoided. For larger projects this could imply larger variants, because source code must be compiled into the variant, which is not necessary for the functionality of the product, because this feature was deselected, but for implementation details must be in.

### 6.1.4 Additional Effort

In this section, we discuss the impact of the manual effort of the approaches. We start with the approach of software derivatives.

#### Software Derivatives

For the approach of software derivatives we count the highest number of features in comparison to the other approaches in both case studies. In [Table 6.1](#) we display the number of features for the expression problem case study, where we counted 19 features, where 12 are required for the derivatives. For the chat we measured 42 features, including 29 derivatives (see [Table 6.2](#)). This documents the manual effort for this approach. We have shown in [Section 5.1.1](#) and [Section 5.2.1](#), that the number of derivative features increases with respect to the feature interactions, so that the feature model becomes confusing. We also said, that the derivatives can be hidden in the feature model, but nevertheless must be implemented and maintained. For our stage of experience, the software derivative approach is the most expensive one according to the manual effort, it took us three days to implement the chat with this approach. During implementation of the chat, we forgot to list several derivative features in the feature model and so we did not implement them, which resulted in hours of debugging. The chat worked fine, when Encryption was switched off, but turning on Encryption led to an unexpected behavior. Nevertheless, we believe, that this approach is the best one, when attaching great importance to the SoC.

We already said in [Section 6.1.1](#), that the software derivative approach means to the software developer, that he must dissolve all feature interactions at the level of the feature model and create the derivatives. For any derivative, a folder must be attached and the correspondent classes have to be created. For larger project, we suspect a very high manual effort for the dissolving of the feature interactions and for the reason of the modularization into folders, the project might become confusing.

## Preprocessor Approach

For the preprocessor approach we have the preprocessor annotations, which results in scattered and tangled code and more LOC in the chat case study than the other approaches (see Table 6.1 and Table 6.2). The preprocessor approach and the optional composition approach have the same number of LOC for the expression problem case study, but in this case, it is just a coincidence and not representable, whereas the number of preprocessor statements is almost the half of the source code (lines of preprocessor statements are in brackets behind the number of LOC). The preprocessor approach and the optional composition have the same number of features in both case studies, because the implementation was built on the same underlying feature model. For the research, we made with the preprocessor approach, we were bothered by the fact, that we had to use a further tool to use the preprocessor statements, which was up to the fact, that we used the FOP variant of Java to be comparable to the other approaches. Furthermore, it was annoying to annotate all the code with the macros representing the features.

The software engineer must define macros which correspond to the features. The developer uses these macros to annotate the source code belonging to the feature. This might be annoying, like in our case. For the debugging all variants must be generated to eliminate syntax errors, that might be created by dropping a feature. So, we suspect large projects have a lot of scattered and tangled code, which leads to confusion.

## Optional Composition

For the optional composition we had to extend the compiler, which can result in annotated code, when using the explicit optional composition with the optional keyword. So, it can be seen as an advantage of the software derivative approach in contrast to the two other approaches, that the FOP can be used without any extension, i.e. no extended compiler or preprocessor must be used. An advantage of the optional composition approach in contrast to the preprocessor is, that it works implicitly. This means, that no annotations are necessary, except in some special cases as shown in Section 5.1.3 and Section 5.2.3, where a class, which should be refined, was introduced by another not optional feature than the two interacting ones. But, this advantage leads to a disadvantage. Because of switching off the error propagation in the compiler to realize the implicit optional composition, the chance of producing errors is higher and the chance of detecting errors is reduced. Just for the cases, where a refinement is ignored, the original class has not been implemented. So, the program will show another behavior than the expected one. Such errors are hard to detect in large projects. And it is conceivable, that these errors just occur for certain variants of the SPL. The experience using the optional composition approach was, that it was very easy to implement the interactions, because the compiler decided, for the implicit part. For the explicit part, we had to decide carefully, which part making explicitly optional. If the order of composition is not in the right order, refinements are left out, and so the chat did not work as expected. So, the program variant compiles and is executable, but shows another behavior than expected. Therefore, the debugging is time-consuming. Nevertheless,

we recommend this approach, when it is for sure, that the **SPL** contains just feature interactions of first order and the order of composition is manageable.

The software developer can implement the feature interactions within one of the features without annotations, except special cases. The developer must not be fully aware of the feature model, because the compiler decides whether to implement the interaction code or not. The developer must look after the order of composition that no errors occur for the reason of the wrong feature order. This can simply lead to time-consuming errors during debugging. So, for larger projects we suggest a higher error rate according to the order of composition of the features, because refinement classes are left out without a warning, when the original class is absent. So the programmer does not get a feedback, except from the behavior of the software.

### 6.1.5 Summary

	Software Derivatives	Preprocessor Approach	Optional Composition
<b>SoC</b>	Yes (+)	reduced <b>SoC</b> (0)	reduced <b>SoC</b> (0)
<b>Number of features</b>	extremely high(−)	no additional features (+)	no additional features (+)
<b>Code Replication</b>	No (+)	No, for statement level (+) / Yes, when order >1 on method level (−)	Yes, when order >1 if approach would be extended(−)
<b>Variability</b>	all variants (+)	all variants (+)	less variants, if more than two features interact (−)
<b>Additional Effort</b>	high effort (−)	medium effort (0)	little effort (+)
<b>Maintainable</b>	high effort (−)	medium effort (0)	little effort(+)

+ good, 0 medium, − bad

Table 6.3: Comparison of the approaches

In this section, we summarize the comparison of the approaches. For this short overview we take [Table 6.3](#) into consideration. There, we listed the above discussed criteria and rated them from bad (−) to good (+) based on the discussion given above. The variability is fully achieved, for the software derivatives and the preprocessor approach, therefore, we rate them as good. The optional composition left out the intersection between all three, when three features interact, so the variability is limited, which is a negative aspect for us, so we rate as bad. According to the redundancies, we did not find code replication in the software derivative implementations and in the optional composition as well, but extending the approach for realizing interactions of higher

than first order, would result in code replication as we have shown for the preprocessor approach in the chat case study. The preprocessor just requires code replication, when limiting this approach to the level of methods. On the level of statements we did not find code replication. The additional effort is very high for the software derivatives, because of the derivative features. The preprocessor approach requires medium effort, because of the additional preprocessor statements. For the explicit optional composition the effort is comparably high to the preprocessor approach, but for the hybrid form of the optional composition the effort is lower. The number of features are extremely high for the software derivatives, which results in high effort for the maintenance of the SPL. Both other approaches do without additional features. Therefore, the effort for maintaining is lower, according to the additional effort.

In this section, we compared the approaches to each other. We discussed the power and weakness of each approach, so we can give a recommendation, when to use which approach in the next section.

## 6.2 Suggestion

Hence, none of the approaches is the odds-one favorite, we come to the conclusion that a combination of the approaches could be an improvement. First of all, the approach of optional composition can be extended according to its explicit variant, that more than one implementation of a method would be possible and the compiler decides with the help of the feature selection which method to include. This would result in the same issue for the code replication like the preprocessor approach with the restrictions of being used just for the level of methods. Another opportunity is the combination of the optional composition approach with the software derivative approach. Taking the optional composition for the feature interactions of first order and the software derivative for interactions of higher order, whereas it is questionable if there really exist interactions of higher order. In the expression problem we do not have interactions higher than first order and in the chat we have just one feature that causes these interactions of second order. In the chat we can argue if the *Logging* feature, which is the one causing the higher order feature interactions, is really necessary. We think, in times of powerful debugging tools such a debug log is not compulsory. We added it for demonstration purposes for the higher order feature interactions. For us it was more difficult than expected to provoke an interaction of second order. Another combination with the optional composition approach might also be possible. The combination of optional composition with the preprocessor approach, which is not restricted to the level of methods. We recommend the last variant with the preprocessor, if higher order interactions occur, because the combination with the software derivatives would result in a higher effort, because of the dissolving of the interactions in derivative features

We suggest the software derivative approach, when a strict SoC is desired and the manual effort is irrelevant. So, this approach is suitable for projects with few features and consequently few interactions.

The preprocessor approach, we recommend, when the SoC may be neglected and the number of features and the level of variability is important. Hence, we suggest the preprocessor for medium sized projects.

We vote for the optional composition approach, when the manual effort and the number of features should be minimized and no interactions above first order occur. For large projects with inevitable interactions of second order, we suggest the combination of the optional composition approach with the preprocessor approach on the level of statements. Finally, we come to the conclusion that the optional composition approach is just a partial solution to the optional feature problem, because of the lacks discussed above.

In our case studies, we have shown, that the optional composition is a suitable solution to the optional feature problem, even though there are limitations according to the interactions of higher order.

In this chapter, we discussed the advantages and disadvantages of the approaches of software derivatives, preprocessor and optional composition and recommended when to use which approach. In the next chapter, we conclude the thesis and review on future work.





## 7. Conclusion

In this thesis, we transferred the concept of optional weaving into the approaches of explicit and implicit composition. During our case studies, we used these approaches in combination as hybrid optional composition. We analyzed the optional composition approach as solution to the optional feature problem. We have shown, with the help of two case studies, that this approach is a partial solution to the optional feature problem at this stage of development.

First of all, we introduced the [Software Product Line \(SPL\)](#) as technique to realize software projects to generate software variants from a common code base. We presented the implementation strategies, where one of them was the [Feature-Oriented Programming \(FOP\)](#). This strategy is especially suitable for the implementation of [SPLs](#), because of summarizing source code, belonging to one feature, is stored in separate modules. So, a set of features can be combined into a variant.

Furthermore, we presented the optional feature problem, where conceptual independent features become dependent in cause of their implementation. As solution to this problem, we introduced the software derivative approach, where these interactions are dissolved by swapping it to derivative features, the preprocessor approach, where the interactions are resolved by using preprocessor statements, to include or exclude the interaction code, and the optional composition approach, which we distinguished between explicit and implicit, where the interaction was swapped into one of the interacting features. Followed by this, was our presentation of the extensions which we had to make to a compiler, so we could do the case studies on all this introduced approaches.

Additionally, we accomplished two case studies to evaluate the optional composition approach in contrast to the other approaches with respect to the optional feature problem. The first case study was a simple [SPL](#) of the expression problem, where we showed the advantages and disadvantages of the approaches. To present a practical and more comprehensive example, we introduced a chat [SPL](#), where we showed further advantages and disadvantages of the approaches to solve the optional feature problem.

In the general discussion, where we compared the software derivative approach, the preprocessor approach and the approach of optional composition to each other based on the two case studies, we came to the conclusion, that any of these approaches has its malices. We gave the advice to choose an approach accordant to the requirements. We recommend the software derivative approach, when the [Separation of Concerns \(SoC\)](#) is desired and the manual effort is irrelevant. The preprocessor approach can be used, when the [SoC](#) may be neglected and the number of features and the level of variability is important. When the manual effort and the number of features should be minimized, and no feature interactions above first order occur, then the optional composition can be used. We suggested a combination of the approaches, as well, but therefore, further case studies would be necessary. All in all the optional composition is just a partial solution of the optional feature problem.

Here, we concluded the thesis and emphasized the results of this thesis. In the next part we review on possible future work.

## Future Work

In this part, we suggest some probable future work to this topic. The optional composition approach cannot handle feature interactions of second order. So, the compiler extension for the optional composition can be more expanded, to realize these feature interactions of higher order with the explicit part. The optional construct of the compiler could be extended by boolean expressions, so that `optional A, !B doSomething()` will be possible, whereas  $A$  and  $B$  represent the features and the exclamation mark represents the negation. For our proposal to realize these interactions, code replication cannot be avoided. Therefore, further case studies might be necessary to survey the impact of the code replication.

Another point for the interactions of second order is, that we suspect, that these interactions are even rare. This could be explored in further, maybe industrial, case studies.

We suggested to combine the optional composition approach with the preprocessor or the software derivative approach in [Section 6.2](#). To figure out the benefit of these combinations in contrast to the singular approaches continuative case studies are necessary.

In this thesis, we constructed two [SPLs](#) for our case studies to analyze the approaches to solve the optional feature problem. We believe, that an industrial case study must be accomplished to see the benefit for practical cases in the real world.

To realize the implicit optional composition, we had to switch off the error propagation of the compiler, when a class which shall be refined is not present. Therefore, when an original class is absent which should be in compilation, no error or warning is thrown. Because the compiler loads the complete [SPL](#) into the [Abstract Syntax Tree \(AST\)](#) a comparison if the original class exists could be made. When the feature selection contains a refinement, where the original class is not present in the whole [SPL](#), an error could be thrown. Furthermore, a way to detect a wrong feature order might be useful, to eliminate the problem of the error-rate of the optional composition approach.

At this stage of development, the compiler which we used and extended in this thesis is not yet implemented in an [Integrated Development Environment \(IDE\)](#). So, a further step could be the implementation of this compiler into an Eclipse plug-in for the [FOP](#), like [FeatureIDE](#)<sup>1</sup>. The development of a debugger is also another opportunity. For the debugger, a look has to be taken which compositions should be debugged. Either just a variant, so the debugger must know the feature selection, or the debugger could check multiple variants at once.

---

<sup>1</sup>[http://wwwiti.cs.uni-magdeburg.de/iti\\_db/research/featureide/](http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/)



# Bibliography

- [AK09] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009. (cited on Page 1, 2, 6, and 10)
- [AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. FEATURE-HOUSE: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering*, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society. (cited on Page 11 and 13)
- [AKL<sup>+</sup>11] Sven Apel, Sergiy Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access Control in Feature-Oriented Programming. *Science of Computer Programming (Special Issue on Feature-Oriented Software Development)*, 2011. to appear; submitted 24 Mar 2010, accepted 29 Jul 2010. (cited on Page 31)
- [ALMK08] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An Algebra for Features and Feature Composition. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology*, pages 36–50, Berlin, Heidelberg, 2008. Springer-Verlag. (cited on Page 13)
- [ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer Verlag, September 2005. (cited on Page 11 and 26)
- [ALSU08] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, techniques, and tools. (Compiler. Prinzipien, Techniken und Werkzeuge.) 2nd ed.* Pearson Studium, 2008. (cited on Page vii, 16, and 17)
- [Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of Software Product Line Conference*, pages 7–20. Springer, 2005. (cited on Page 6 and 7)

- [BCK05] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston ; Munich [u.a.], 2005. (cited on Page 1 and 5)
- [Bec10] Christian Becker. Entwicklung eines nativen Compilers für Feature-orientierte Programmierung. Master's thesis, University of Magdeburg, June 2010. (cited on Page vii, 11, 31, 32, and 34)
- [BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-wise Refinement. In *Proceedings of the International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society. (cited on Page 1, 10, and 11)
- [CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. ACM Press/Addison-Wesley, 2000. (cited on Page 6)
- [CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41:115–141, January 2003. (cited on Page 2 and 15)
- [Coo90] William R. Cook. Object-Oriented Programming Versus Abstract Data Types. In *Foundations of Object-Oriented Languages*, pages 151–178. Springer-Verlag, 1990. (cited on Page 41)
- [Dij82] Edsger W. Dijkstra. *Selected writings on computing: a personal perspective*. Springer, New York, NY, USA, 1982. (cited on Page 8)
- [Dij97] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. (cited on Page 8)
- [Ekm06] Torbjörn Ekman. *Extensible Compiler Construction*. PhD thesis, Lund University, Schweden, 2006. (cited on Page 31)
- [Fla02] David Flanagan. *Java in a Nutshell, Fourth Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. (cited on Page 8)
- [HJ95] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Prentice Hall, 4th edition, 1995. (cited on Page 9)
- [JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988. (cited on Page 9)
- [KA08] Christian Kästner and Sven Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 35–40, Passau, Germany, October 2008. University of Passau. (cited on Page 24 and 60)

- [KA09] Christian Kästner and Sven Apel. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, September 2009. Refereed Column. (cited on Page 10 and 68)
- [KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering*, pages 311–320, New York, NY, USA, 2008. ACM. (cited on Page 9 and 66)
- [Käs07] Christian Kästner. Aspect-Oriented Refactoring of Berkeley DB. Master’s thesis, University of Magdeburg, 2007. (cited on Page vii, 2, 16, 19, 24, 26, and 27)
- [Käs10] Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, May 2010. (cited on Page vii, 5, 7, and 9)
- [KAuR<sup>+</sup>09] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proceedings of the International Software Product Line Conference, SPLC ’09*, pages 181–190, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. (cited on Page vii, 19, and 22)
- [KCH<sup>+</sup>90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990. (cited on Page 6)
- [KFF98] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing Object-Oriented and Functional Design to Promote Reuse. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 91–113. Springer, 1998. (cited on Page 41)
- [KLM<sup>+</sup>97] Gregor Kiczales, John Lamping, Anurag. Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc. Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, 1997. (cited on Page 8 and 14)
- [KS94] Maren Krone and Gregor Snelting. On the Inference of Configuration Structures from Source Code. In *Proceedings of the International Conference on Software Engineering*, pages 49–57, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. (cited on Page 9)
- [LARS05] Thomas Leich, Sven Apel, Marco Rosenmüller, and Gunter Saake. Handling Optional Features in Software Product Lines. In *Proceedings of International Conference on Object Oriented Programming, Systems, Languages and Applications Workshop on Managing Variabilities consistently in Design and Code*, San Diego, USA, 2005. (cited on Page 2, 19, and 26)

- [LBL06] Jia Liu, Don Batory, and Christian Lengauer. Feature-oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering*, pages 112–121, New York, NY, USA, 2006. ACM. (cited on Page 16 and 22)
- [LBN05] Jia Liu, Don Batory, and Srinivas Nedunuri. Modeling Interactions in Feature Oriented Software Designs. In Stephan Reiff-Marganiec and Mark Ryan, editors, *Proceedings of the international Conference of Feature Interactions in Telecommunications and Software Systems*, pages 178–197. IOS Press, 2005. (cited on Page 22)
- [MO04] Mira Mezini and Klaus Ostermann. Variability Management with Feature-oriented Programming and Aspects. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 127–136, New York, NY, USA, 2004. ACM. (cited on Page 26)
- [Par72] David L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Classics in software engineering*, 15(12):1053–1058, December 1972. (cited on Page 8)
- [PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, 2005. (cited on Page 1 and 5)
- [Pre97] Christian Prehofer. Feature-oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997. (cited on Page 1, 10, and 22)
- [Rey94] John C. Reynolds. *User-defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*, pages 13–23. MIT Press, Cambridge, MA, USA, 1994. (cited on Page 41)
- [SB02] Yannis Smaragdakis and Don Batory. Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-based Designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, 2002. (cited on Page 11)
- [Sch09] Andreas Schulze. Systematische Analyse von Feature-Interaktionen in Softwareproduktlinien. Diplomarbeit, University of Magdeburg, Germany, October 2009. (cited on Page vii, 41, and 53)
- [SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2nd ed. (15. november 2002) edition, 2002. ISBN-10: 0201745720 ISBN-13: 978-0201745726. (cited on Page 9)



- 
- [Spe92] Henry Spencer. `ifdef` Considered Harmful, or Portability Experience With C News. In *Proceedings of the Summer '92 USENIX Conference*, pages 185–197, 1992. (cited on Page 9)
- [SS07] Alexander Schütz and Dieter Schuller. Compilerframeworks - JastAdd Design und Implementierung moderner Programmiersprachen. Studienarbeit, Februar 2007. (cited on Page 31)
- [TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of the International Conference on Software engineering*, pages 107–119, New York, NY, USA, 1999. ACM. (cited on Page 8)



---

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 17. Dezember 2010