JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche Fakultät

# Feature-Based Composition of Software-Systems

## MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

Software Engineering

Eingereicht von:
Stefan Fischer

Angefertigt am:
Institute for Software Systems Engineering

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed M. Sc.

Mitwirkung:
Dr. Roberto Lopez Herrejon M. Sc.

Linz, Februar, 2014

# Contents

Danksagung iii

Kurzfassung v

Abstract vii

1 Introduction 1
1.1 Software Reuse . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.1.1 Clone-and-Own . . . . . . . . . . . . . . . . . . . . . . 1
1.1.2 Software Product Line Engineering . . . . . . . . . . 3
1.2 Goals of this Thesis . . . . . . . . . . . . . . . . . . . . . . . 5
1.3 Chapter Description . . . . . . . . . . . . . . . . . . . . . . 7

2 Background and Example 9
2.1 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2 Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.3 Associations . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3 Workflow & Principles 17
3.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 Extraction . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.3 Composition . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4 Technical Realization 23
4.1 Data structures . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 Extraction . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2.1 Module Calculation . . . . . . . . . . . . . . . . . . . . 27
4.2.2 Intersecting . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2.3 Sequencing . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3 Composition . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.3.1 Selecting Code . . . . . . . . . . . . . . . . . . . . . . 32
4.3.2 Tree Merging . . . . . . . . . . . . . . . . . . . . . . . 36
4.3.3 References . . . . . . . . . . . . . . . . . . . . . . . . 37
4.3.4 Code Order . . . . . . . . . . . . . . . . . . . . . . . . 39

**5 Tool Support** **41**
5.1 Demo Scenario . . . 41
5.2 Other Capabilities . . . 52

**6 Implementation** **57**
6.1 Data structure and Core Operations . . . 57
6.1.1 Features and Modules . . . 57
6.1.2 Nodes and Artifacts . . . 58
6.1.3 Composition . . . 59
6.1.4 Ordering . . . 61
6.2 Tool . . . 64
6.2.1 Project . . . 64
6.2.2 Representation . . . 66
6.3 Parsers . . . 67
6.3.1 EMF-Parser . . . 67
6.3.2 Java-Parser . . . 70
6.4 Printer . . . 70

**7 Evaluation** **71**
7.1 Case Studies . . . 71
7.1.1 Draw . . . 72
7.1.2 VOD . . . 72
7.1.3 ArgoUML . . . 72
7.1.4 Model Analyzer . . . 73
7.2 Evaluation Scheme . . . 74
7.3 Evaluation Metrics . . . 75
7.3.1 Composition Runtime . . . 75
7.3.2 Correctness . . . 76
7.3.3 Completeness . . . 77
7.3.4 Warnings . . . 79
7.4 Analysis . . . 81

**8 Threats to Validity** **83**

**9 Related Work** **85**

**10 Future Work** **87**

**11 Conclusion** **89**

**Lebenslauf** **97**

**Eidesstattliche Erklärung** **99**

# Danksagung

Ich möchte Univ.-Prof. Dr. Alexander Egyed M. Sc. und Dr. Roberto Erick Lopez-Herrejon M. Sc. für die hervorragende Unterstützung und Betreuung danken die sie mir während der Erstellung dieser Arbeit und auch darüber hinaus zukommen haben lassen.

Besonderer Dank gilt auch meiner Familie die es mir während meines Studiums sehr leicht gemacht hat und mich unterstützt hat wo immer es ging.

# Kurzfassung

Um in der Lage zu sein die Nachfrage nach maßgeschneiderten Software Systemen zu decken, verwenden Unternehmen in der Regel Techniken um Software wiederzuverwenden. Dadurch erhalten die Unternehmen ein Portfolio von gleichen, doch nicht identischen, Software Produkten. Über Jahre wurden verschiedene Arten von Techniken zur Wiederverwendung entwickelt. Trotz der großen Vorteile die solche Techniken mit sich bringen, haben sie auch Nachteile, wie hohe kosten oder sie unterstützen die Wiederverwendung nicht komplett und haben oft mangelhafte Unterstützung für die Weiterentwickling von Produkten. Dies führt dazu, dass viele Unternehmen vor der Verwendung dieser Techniken zurückschrecken. Wir konzentrieren uns im Wesentlichen auf die Wiederverwendungstechniken Software Produkt Linien (SPLs) und die weit verbreitete Technik die als Clone-and-Own bezeichnet wird. Software Produkt Linien sind dafür gedacht die gesamte Variabilität eines Produktportfolios zu modellieren. Das heißt sie müssen sorgfältig geplant werden, mit Rücksichtnahme auf alle möglichen Produkte die ableitbar sein sollen, was eine große Investition bedeutet, die sich viele Unternehmen nicht leisten können. Clone-and-Own andererseits ist mehr Ad-hoc. Es wird Code von verschiedenen Produkt Varianten kopiert und angepasst um den Anforderungen des Kunden zu entsprechen.

In dieser Arbeit stellen wir eine, durch ein Tool unterstützte, Methode vor welche sich die Vorteile der beider Wiederverwendungstechniken zunutze macht, jedoch versucht ihre Nachteile zu lindern. Mit unserer Methode selektiert der Entwickler lediglich die gewünschten Eigenschaften (Feature),

für welche unser Tool automatisch die Software Artefakte findet von denen sie implementiert werden. Das Programm kopiert diese Artefakte anschließend in ein neues Produkt und hilft weiters dem Entwickler bei der manuellen Fertigstellung des Produkts durch Hinweise auf eventuell fehlende oder noch nicht fertige Codeteile.

# Abstract

To keep pace with the increasing demand for custom-tailored software systems, companies usually use software reuse techniques. Therefore companies end up with a portfolio of similar, yet not identical, software products. Over the years different kinds of reuse techniques have emerged. Despite the great benefits of these techniques, they also have disadvantages, like they requiring large investments, or do not fully allow to facilitate reuse and often lack support for evolving products. Therefore many companies are wary to utilize these reuse techniques. We focus on the reuse techniques using Software Product Lines (SPLs) and also the wide spread technique referred to as Clone-and-Own. SPLs are intended to contain the entire variability of a product portfolio. This means they have to be planned meticulously and take all the possible product variants derivable into account, which requires a large upfront investment that companies often cannot afford. Clone-and-Own on the other hand is a more ad-hoc technique where code of product variants is copied and modified to fit the customers requirements.

In this thesis we introduce a tool supported approach which leverages the advantages of these two reuse techniques, yet tries to mitigate their respective disadvantages. With our approach a software engineer selects the desired features, for which the tool automatically finds the software artifacts implementing them. The approach then copies these artifacts into a new product and helps the software engineer during the manual completion by hinting which software artifacts may be missing or may need adaption.

# Chapter 1

# Introduction

This chapter provides an introduction to Software Reuse techniques, subsequently we describe the goal of this thesis and its chapter structure.

## 1.1 Software Reuse

It has become industrial practice to tailor software to the exact needs of customers. Therefore companies develop a portfolio of similar, yet not identical software products. Because of the high similarity in the products requirements, their code is reused in building new software variants. This allows the companies to develop products faster and therefore cheaper.

The term software reuse can have very different meanings, form the use of components of the shelf, down the reuse of application architectures [3]. We talk about the direct reuse of implementation artifacts on an arbitrary level of granularity, from reusing entire packages or classes even down to reusing single statements.

Next we will discuss the two main approaches for this type of code reuse.

### 1.1.1 Clone-and-Own

Clone-and-Own is a very cost efficient approach that allows software engineers to tailor software to customer needs [13] [16]. Using Clone-and-Own

the software engineer first has to build an initial product, potentially already tailored to a customers requirements. Afterwards the company will try to get new buyers for their product, which probably have slightly different requirements. Therefore the software engineer takes the existing product that is the closest fit to the customers needs and adapts it, with functionality respectively features to fit these new requirements. This adaption may consist of addition, modification or removal of code. The software engineer might also take code parts from other products in this product portfolio which already implement desired features.

**Advantages.**

- **Saves time and reduces costs:** It is more efficient to start with an already existing, tested and validated product, than to start from scratch. Most of the product is already existing and therefore does not have to be reimplemented.
- **No upfront investment:** There is no large upfront investment necessary for Clone-and-Own. Companies start of with a single product and sell it to their customers, as they would do without reuse.
- **Flexibility:** Clone-and-Own enables a high flexibility, because the products are adapted direct to the customers needs. It is not necessary to foresee all possibly needed product variants. Features can be added, as they are desired from the customers. Requirements of the product variant often emerge over time.

**Drawbacks.**

- **Feature maintenance:** If a change, for example a bug fix, is made to a feature, then these changes have to be propagated through all the products which implement this feature. The software engineers need to track every product which contains the changed feature and fix them as well, which also contains finding the features code in the product.

- **Choosing product variant to start:** It is often not clear which product variant should be used as a starting point. There is usually no infrastructure that tracks reuse opportunities, so it is mainly up to the software engineer to remember respectively find parts that can be reused. Moreover it can be extremely difficult to identify the implementation of features within the source code of products.

- **Lacks systematic methodology:** Clone-and-Own is discouraged in software engineering literature because it lacks a systematic methodology. Cloning is often done ad-hoc without planing and facilitating reuse for the future [12].

### 1.1.2 Software Product Line Engineering

In software product line engineering the product portfolio is designed complete, rather than building the product variants individually. The desired features in the product portfolio are therefore developed upfront. This means the whole range of variability is already contained in the software product line. Making it possible to configure a new product for a customer very quick.

*A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect different requirements.* [3] At the core there is a configurable system which allows to alter to code to fit the customers requirements.

**Advantages.**

- **Reduce time to market:** As soon as a product is needed the company can derive this product from the product line very quickly. Therefore a software product line can be a huge advantage for a company to satisfy their customers and to forestall the competition.

- **Reduce costs:** In the long run a software product line can reduce the costs for software, because new products can be generated very

quick.

- **Feature maintenance:** Because the features are implemented within the product line, they only exist once and therefore changes, like a bug fix, automatically involves all its member products. Nevertheless it can be complicated to maintain features like this, since changes in one feature can affect other features too.

**Drawbacks.**

- **Upfront investment:** The deign of a software product line is a complex and time consuming task. While the product line is still under development there are usually no complete products that could be sold. This implies a risk for the company that a competitor introduces a similar product first, or the market rejects the products and the investment is lost.

- **Variability specified at the beginning:** The majority of software product line engineering approaches assume that the company designs the complete product portfolio upfront that satisfies all future needs. This is often not possible, since the company can not foresee future developments. Therefore the product line has to be evolved, to contain more or different products, which causes a risky investment again.

- **Requires proper techniques and tool support:** Software product line engineering is nothing that can be done ad-hoc. It requires a lot of planing and know how of the engineers to design a product line in a proper way. Also tool support is required to allow to implement and maintain the product line and to generate its member products. These are reasons why many practitioners hesitate to use software product line engineering.

## 1.2 Goals of this Thesis

The first goal of this thesis is to develop an algorithm to compose product variants base on their features. This composition takes software fractions, which are already labeled with the feature they implement, and a set of desired features as input. From this information the algorithm should generate a product containing exactly these desired features.

With this we want to achieve the same advantages as with the reuse techniques discussed in the previous section and also minimize respectively eliminate some of their drawbacks:

- **Reduced Costs:** The tool will help companies reduce their development costs by reducing implantation time and also give them an advantage by reducing their time-to-market.
- **No upfront investment:** Companies using this tool will not have to make a large upfront investment. They just implement their first products like in clone-and-own and later use the composition to generate their products.
- **Flexibility:** Just like with clone-and-own the variability of the product portfolio is not specified completely at the beginning. This makes it easier to include new features in the future, because they will only be added to the required product variant. Therefore the features can be implemented for those cases and do not have to take possible constraints with not included features into account.
- **Feature maintenance:** Our tool supports the maintenance of features just secondarily. It can help the software engineer to find the products which implement certain features and also narrow down the search area for the implementation. In future work this can still be improved.
- **Configuring Products:** The implemented tool can configure prod-

uct variant by selecting the desired features it should have. It allows to re-compose existing products, which works fully automatic like in a product line. Also it is able to compose new products with feature combinations that did not exist before. These new products can then be used as a starting point for clone-and-own where all the required implementation of existing products, which eliminates the problem of finding the code manually.

To achieve this, in a usable manner, we have to tackle some more detailed subgoals:

- **Reference Implementation:** Providing a Java implementation of the composition.
- **Genericity:** The composition should be generic, so it can be used for different programming languages and also none-code artifacts (e.g. models, language, ...).
- **References:** Dealing with references in the code that cut across the regular hierarchy.
- **Ordering:** Calculate possible orderings for implementation artifacts.
- **Warnings:** Provide meaningful warnings for the software engineer, to narrow down the code that potentially needs manual adaption.

The second big goal of this thesis is to develop a tool support for the composition:

- **Include extraction:** Use the extraction algorithm of [4] to generate the necessary code fragments from product variants and include it into the tool.
- **Code representation extensible:** The tool should be fully extensible in the languages it can use, so it can utilize the genericity the composition provides.

- **EMF implementation:** Provide a reference code representation implementation for EMF (Eclipse Modeling Framework) [5], which itself already allows a high level of genericity. Also include JaMoPP [6] (Java Model Parser and Printer) to convert Java code into its EMF representation.

## 1.3 Chapter Description

The remainder of this thesis is structured as follows. Chapter 2 introduces the relevant background by means of an illustrative example. In Chapter 3 we discuss the proposed workflow for the presented tool support. Chapter 4 presents the technical process of the extraction [4] and the composition along with the used data structures. The tool support is demonstrated in Chapter 5. Chapter 6 gives an overview of the implementation of the data structures, the composition and the major tool parts. In Chapter 7 we show the results of the performed evaluation. Finally Chapter 8 talks about possible threats to validity. Chapter 9 gives an overview about related work. Chapter 10 gives an outlook on future work. Chapter 11 summarizes and concludes the thesis.

# Chapter 2

# Background and Example

This chapter introduces the necessary background and illustrates the running example that will be used throughout the remainder of this thesis.

## 2.1 Example

Consider a portfolio of drawing applications that has been created by applying clone-and-own.

Each variant supports a subset of the following features: the ability to handle a drawing area (`BASE`), draw lines (`LINE`) and rectangles (`RECT`), select a color to draw with (`COLOR`) and wipe the drawing area clean (`WIPE`). Let us assume that the portfolio consists of three variants, each providing a distinct set of features and each having its own distinct implementation. These three product variants and the features they implement are shown in Table 2.1.

A feature is a functionality of an application visible to the user. Features can describe functional and none-functional requirements. We denote features in upper case.

Each product variant consists of the implementation artifacts and the set of features implemented by the product. Artifacts can be anything from source code to models, test cases or requirements.

We use source code as example of the implementation artifacts of the

| **Products** | BASE | LINE | RECT | COLOR | WIPE |
|---|---|---|---|---|---|
| Product $P_1$ | ✓ | ✓ | | | ✓ |
| Product $P_2$ | ✓ | ✓ | | ✓ | |
| Product $P_3$ | ✓ | ✓ | ✓ | ✓ | |

Table 2.1: Initial Drawing Application Product Variants

product variants $P_1$, $P_2$ and $P_3$. Excerpts of their source code are shown in Figure 2.2. $P_1$ is shown from Lines 1 to 15 and includes a class `Canvas` that offers the method `wipe`, a class `Line` and a class `Main`. Next $P_2$ is displayed from Lines 16 to 28, which also includes the classes `Canvas`, `Line` and `Main` whose implementation differs slightly from the one of $P_1$, i.e. class `Canvas` provides the method `setColor` instead of method `wipe`. Moreover, the constructor of the class `Line` (see Line 21) receives the additional input parameter `color`. Method `initContentPane` in Line 24 adds an additional panel to the drawing application to choose colors from. Finally, $P_3$ is shown from Lines 29 to 46 offering the additional class `Rect`. As expected, the three variants' implementations have commonalities but also differences. For a better understanding, a screen shot of product $P_3$ can be seen in Figure 2.1.

The behavior of a single feature may depend on the presence or absence of other features, i.e. in $P_2$ feature `COLOR` offers the possibility to choose different colors for lines. In $P_3$ the behavior of this feature changes in the sense that it now also allows the selection of different colors for rectangles, which means that the behavior of feature `COLOR` changes in the presence (resp. absence) of feature `RECT`. The fact that features influence each other is referred to as feature interaction [2] and is a well-known problem in software reuse.

Next we illustrate the clone-and-own way of extending the product portfolio by creating a new product $P_4$ with features `BASE`, `LINE`, `RECT` and `WIPE`, whose final source code is shown in Figure 2.3. One would probably start off by copying product $P_1$ and adding the code for feature `RECT` after its extraction from product $P_3$. This can be done by identifying the code that

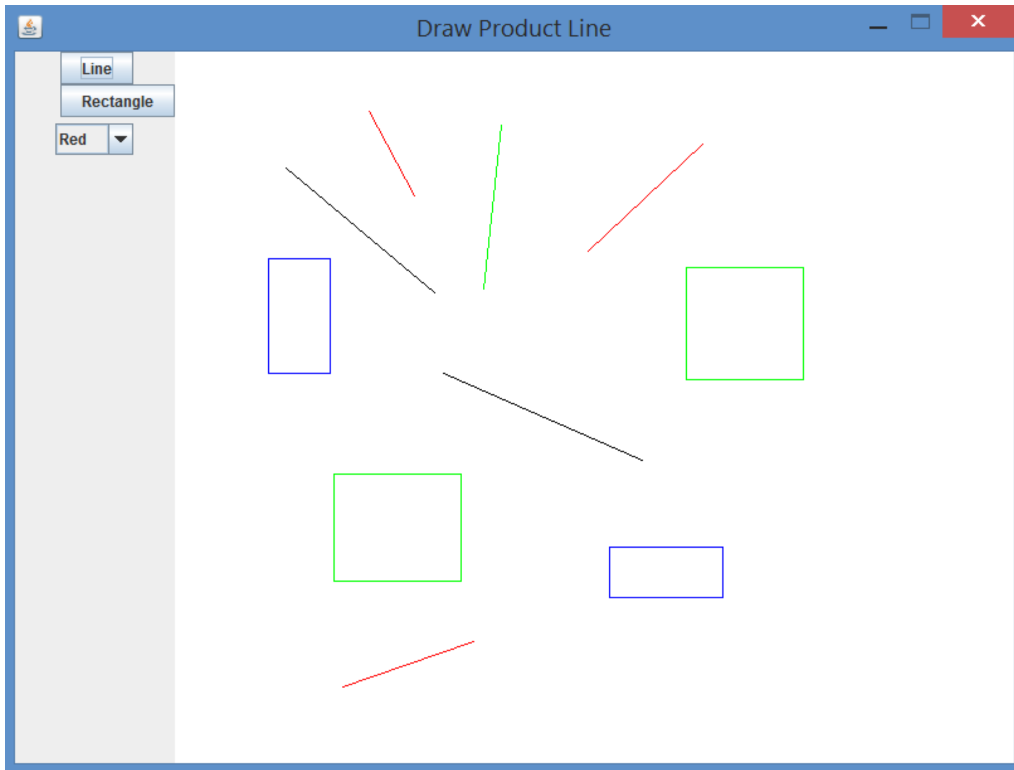



Figure 2.1: Product $P_3$

is only in product $P_3$ and not in $P_2$ as the only difference of features in these two products is feature RECT. This means that Lines 31, 37-39 and 43 have to be copied into the new product. The resulting product however will now contain a syntax error since feature RECT has only appeared in combination with feature COLOR and therefore expects a color parameter in its constructor (see Line 38 in Figure 2.2). So feature RECT without feature COLOR behaves differently, therefore this code will have to be manually removed and replaced with a constructor that does not have color as its parameter (see Line 59 of Figure 2.3). Additionally the semantics of the resulting product will also not be entirely correct. Since feature RECT has never appeared with feature WIPE before, the code responsible for the interaction between these features is missing. Indeed, without this feature interaction the new variant would fail to wipe rectangles. When wiping the drawing area clean only the lines will be wiped but not the rectangles (see Line 4 in Figure 2.2). Again the missing code for this feature interaction

will have to be manually added (see Line 52 of Figure 2.3).

Moreover, the software engineer would need to decide on the order of statements, consider for instance method `initContentPane()` in Line 62 of Figure 2.3. While it may be clear that the buttons associated for drawing lines and rectangles and for wiping the drawing area clean need to be added to the drawing area it is not clear in which order they shall appear as the feature interaction among features `LINE`, `RECT` and `WIPE` has not been present in any products of the product portfolio.

Now consider another product $P_5$ to be created with features `BASE`, `RECT` and `WIPE`. One could use the previously created product $P_4$ as a starting point and try to remove feature `LINE`. In Figure 2.4 we can see the source code excerpt for product $P_5$, which is now missing the class `Line` as well as the `lines` field in the class `Canvas`. The problem here is that feature `LINE` has never appeared without feature `BASE` in any product. This makes the identification of its code harder than simply looking at differences between products as was done previously for product $P_4$ with feature `RECT`.

The two newly created products are shown in Table 2.2.

| **Products** | BASE | LINE | RECT | COLOR | WIPE |
|---|---|---|---|---|---|
| Product $P_4$ | ✓ | ✓ | ✓ | | ✓ |
| Product $P_5$ | ✓ | | ✓ | | ✓ |

Table 2.2: New Drawing Product Variants

Apart from the challenge of creating new products there is also the task of maintaining the products currently available. It needs to be ensured that changes to individual features are propagated across all products that implement those feature. Imagine for instance a bug fix to be applied to feature `BASE`. This fix would have to be carried out in each of the five variants, because they all implement this feature.

## 2.2 Modules

To label the artifacts which implement features or feature interactions we use modules [14]. With these modules we follow a notation and terminology inspired by Liu et. al [15].

We distinguish modules of two kinds: `base` modules and `derivative` modules.

- **Base Modules** A base module labels artifacts that implement a single feature, i.e. base modules are independent of feature interactions. We refer to them with the feature's name in lowercase. For instance the module `line` represents all the code that implements feature `LINE`, without any interaction with other features that may occur.
- **Derivative Modules** A derivative module labels artifacts that implement feature interactions. Meaning code that is responsible for the collaboration of two or more features. We denote them as $\delta^{\mathtt{n}}(\mathtt{c_0}, \mathtt{c_1}, ..., \mathtt{c_n})$, where $c_i$ is `F` if feature `F` is selected or ¬`F` if `F` is not selected, and $n$ is the order of the derivative. A derivative module of order $n$ represents the interaction of $n + 1$ features. In our example we have seen such feature interactions for the features `RECT` and `WIPE`, where the software engineer needs to add the Line 52 of Figure 2.3 to product $\mathtt{P_4}$.

Modules can only add artifacts. To model the removal of artifacts when a feature combination is present we introduce negative features. A product contains the features it implements positively and those that it does not implement negatively. A feature not being present can influence the implementations of other features just as much as a feature that is present. However, a feature that is not present can not add code on its own, therefore there are no base modules of negative features. As a simple example feature `LINE` has its constructor as shown in Figure 2.2 at Line 8. When adding feature `COLOR` that constructor is removed and replaced by a new

constructor shown at Line 35. Since the first constructor is not always present when feature `LINE` is present it is not part of its base module `line`, and if it was then the derivative module $\delta^1(\texttt{line}, \texttt{color})$ could not remove it. So instead the first constructor without the color parameter is part of module $\delta^1(\texttt{line}, \neg\texttt{color})$.

## 2.3 Associations

To be able to express which modules correspond to which artifacts we next introduce associations. An association maps a set of modules to the artifacts implementing the feature and feature interaction represented by these modules. So an association consist of a set of modules and a number of artifacts. Alternatively, we say that these modules trace to these artifacts.

Using associations we can map aforementioned modules to the code of our drawing examples. For instance the base modules `line` will be mapped to the code fragments {... `List<Line> lines;` ...}. The derivative module $\delta^1(\texttt{line}, \texttt{color})$ will be mapped to the artifacts {... `Line(Color c, Point start)` ...}, respectively $\delta^1(\texttt{line}, \neg\texttt{color})$ is mapped to {...`Line(Point start)` ...}.

Product $P_1$ (BASE, LINE, WIPE):

```
1 class Canvas {
2   List<Line> lines;
3   void wipe() {
4     this.lines.clear();
5   } ...
6 }
7 class Line {
8   Line(Point start) {...} ...
9 }
10 class Main extends JFrame{
11   initContentPane() {
12     toolPanel.add(lineButton);
13     toolPanel.add(wipeButton);
14   } ...
15 }
```

Product $P_2$ (BASE, LINE, COLOR):

```
16 class Canvas {
17   List<Line> lines;
18   void setColor(String c){...}...
19 }
20 class Line {
21   Line(Color c, Point start){...}  ...
22 }
23 class Main extends JFrame{
24   initContentPane() {
25     toolPanel.add(lineButton);
26     toolPanel.add(colorsPanel);
27   } ...
28 }
```

Product $P_3$ (BASE, LINE, RECT, COLOR):

```
29 class Canvas {
30   List<Line> lines;
31   List <Rect> rects;
32   void setColor(String c){...}...
33 }
34 class Line {
35   Line(Color c, Point start){...} ...
36 }
37 class Rect {
38   Rect(Color c, int x, int y){...} ...
39 }
40 class Main extends JFrame{
41   initContentPane() {
42     toolPanel.add(lineButton);
43     toolPanel.add(rectButton);
44     toolPanel.add(colorsPanel);
45   } ...
46 }
```

Figure 2.2: Source Code Snippets for the initial Drawing Applications

Product $P_4$ (BASE, LINE, RECT, WIPE)

```
47 class Canvas {
48   List<Line> lines;
49   List <Rect> rects;
50   void wipe() {
51     this.lines.clear();
52     this.rects.clear();
53   } ...
54 }
55 class Line {
56   Line(Point start) {...} ...
57 }
58 class Rect {
59   Rect(int x, int y) {...} ...
60 }
61 class Main extends JFrame{
62   initContentPane() {
63     toolPanel.add(lineButton);
64     toolPanel.add(rectButton);
65     toolPanel.add(wipeButton);
66   } ...
67 }
```

Figure 2.3: Source Code Snippets for Product 4

Product $P_5$ (BASE, RECT, WIPE)

```
68 class Canvas {
69   List <Rect> rects;
70   void wipe() {
71     this.rects.clear();
72   } ...
73 }
74 class Rect {
75   Rect(int x, int y) {...} ...
76 }
77 class Main extends JFrame{
78   initContentPane() {
79     toolPanel.add(rectButton);
80     toolPanel.add(wipeButton);
81   } ...
82 }
```

Figure 2.4: Source Code Snippets for Product 5

# Chapter 3

# Workflow & Principles

This chapter will give an overview of the intended workflow, of the implemented tool support. Further it will illustrate how this workflow is useful in dealing with the problems introduces in Chapter 2. Finally, the chapter illustrates what the two active components of the workflow use as input and provide as output, to present a clear notion of what is performed in these steps.

## 3.1 Overview

Figure 3.1 shows an overview of the proposed workflow and highlights the parts which had to be implemented along this thesis.

The workflow starts with one or more existing product variants. Using the example we introduced in Chapter 2 our input products would be $\mathtt{P}_1$, $\mathtt{P}_2$ and $\mathtt{P}_3$. These products are parsed (highlighted green in Figure 3.1) into a generic data structure designed for this approach. The workflow is designed to be able to replace the parser according to the implementation artifacts used as input. For this thesis a parser for EMF had to be implemented.

Subsequently the parsed products serve as input for the extraction, implemented by [4]. This will calculate the contained associations, which map the implementation artifacts to the modules of the product variants. Afterwards these associations are stored in the database (DB).

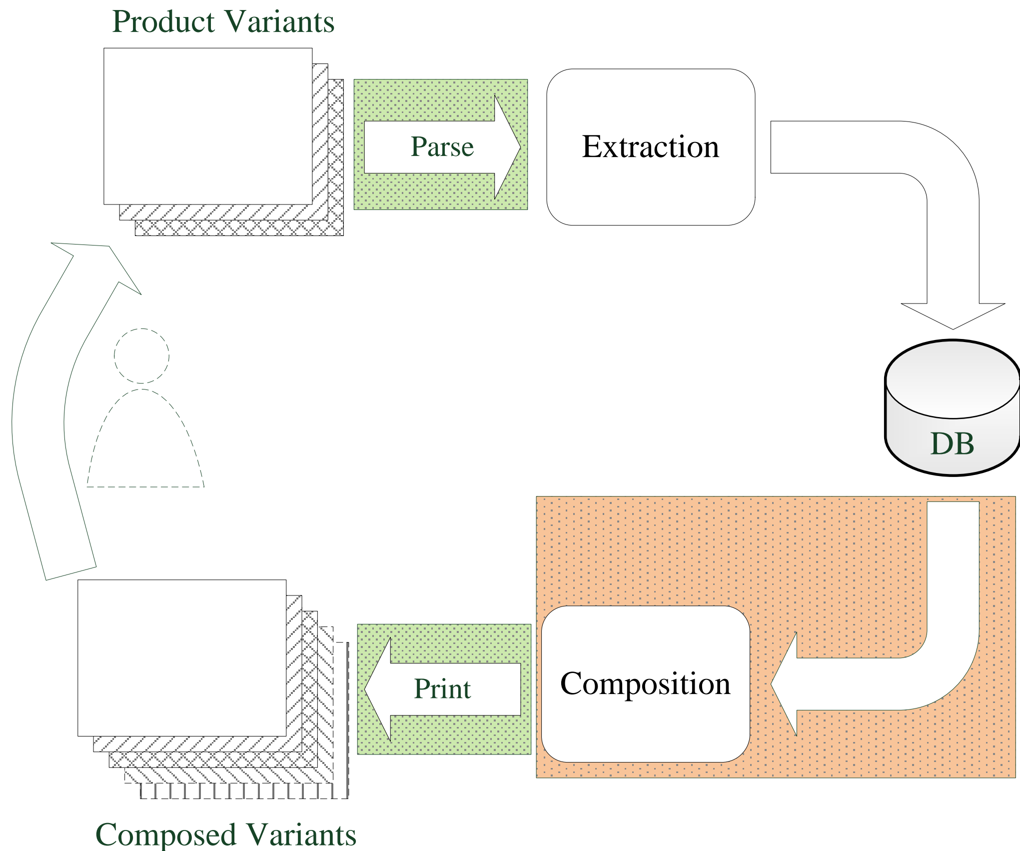



Figure 3.1: Overview

The composition was the main challenge of this thesis, therefore highlighted red in Figure 3.1. It uses the information from the database as input and can re-compose those existing input variants again, fully automated and complete. Moreover it is able to compose new product variants, with feature combinations that did not exist before. However these new product variants will have some information missing, like feature interactions of features that never occurred together before. Furthermore it can miss entire features which have not been implemented yet. Their implementation has to be provided by the software engineer to complete these products. To support the software engineer in filling those missing parts the composition generates warnings to guides the engineer to parts where information was missing from the database. We distinguish between different kind of warnings:

- **Missing Artifacts:** Modules required for the new product that never appeared in any of the input product variants, hint to artifacts that may be missing and whose implementation has to be added manually. For example this was the case for the module $\delta^1(\texttt{rect}, \texttt{wipe})$ in product `P4`, since the code for wiping rectangles never occurred before.

- **Surplus Artifacts:** Modules that where selected in addition to the modules intended, for the composition of this product. This can happen if artifacts are mapped to more than one module, because they could not have been distinguished from the existing input products. If only one of the modules is needed all the artifacts of the association will be used for the composition. Then this warning alerts the software engineer to take a look at these artifacts to make sure all of them belong in the new product. In our example this was the case in the creation of product `P5` where it was not clear if the artifact `List<Line> lines` belonged to the base module `line` or `base` or the derivative module $\delta^1(\texttt{line}, \texttt{base})$ as `P5` is the first product where the features `LINE` and `BASE` do not co-occur.

- **Order:** In some cases not only the structure of artifacts is relevant, but also the order of artifacts in the same hierarchy. For instance this is the case in Java for statements within a method, since the order of the code lines can change the semantics of the program and also make the application uncompilable. Therefore in the cases where such artifacts are combined from different product variant, i.e. never occurred together before, the order has to be determined by the software engineer. In our example this was the case for the statements of the method `initContentPane()` for Product `P4` (see Line 62 of Figure 2.3), where it could not be automatically derived in which order the buttons need to be added to the content pane.

The software engineer has to decide which of the warnings are important for the current product, since there can be too many warning for feature

interactions that never occurred before and do not actually have an implementation.

For the task of completing the composed product variants, the software engineer can use the printer (highlighted green in Figure 3.1) of the workflow. This will create the product with the intended representation of the implementation artifact (e.g. source code) again. Like the parser, the printer is fully replaceable and therefore can be individually provided for all different kinds of implementation artifacts.

After completing the new products, they can be added to the input of the workflow incrementally, which will further refine the information stored in the database. Therefore the quality of the newly composed product variants will improve over time.

## 3.2 Extraction

The extraction was developed by [4] and used for the purpose of this thesis. This section will give an overview on the extraction, since it is a key element to our workflow.

The extraction takes product variants as input which have to be parsed and converted into the generic data structure presented in Chapter 4. As output the extraction provides associations (see Section 2.3), which are stored in the database.

Figure 3.2 illustrates the associations the extraction provides for our example input products ($P_1$, $P_2$, $P_3$), every association with the mot significant modules and the corresponding implementation artifacts. For simplicity we will use the Java code of our example to illustrate the implementation artifacts of the associations.

The implementation parts with gray background are placeholder artifacts, which are just needed to be able to located their position in the code hierarchy, and do not actually belong into these associations.

## 3.3 Composition

The composition uses the associations stored in the database to construct them to products again. Therefore the it needs a set of features which the composed product should contain. Based on these features the composition selects the associations needed to construct the product variant and merges the implementation artifacts according to their hierarchy. As discussed before the so composed products might not be complete and need some adaption according to warnings the composition provides.

Using our example again, we now want to compose the Product $\mathtt{P_4}$ (`BASE`, `LINE`, `RECT`, `WIPE`). The composition will select the Associations `A1`, `A2` and `A3` and merge the implementation artifact contained in them. The result will be an incomplete version of $\mathtt{P_4}$ with the unknown lines 52 and 59 of Figure 2.3 missing and the line 41 in Figure 3.2 surplus.

Association A1 (base, line, $\delta^1$(line, base)):

```
1  class Canvas {
2    List<Line> lines;
3    ...
4  }
5  class Line {
6    ...
7  }
8  class Main extends JFrame{
9    initContentPane() {
10     toolPanel.add(lineButton);
11   } ...
12 }
```

Association A2 (wipe, $\delta^1$(line, wipe), $\delta^1$(line, $\neg$color), ...):

```
13  class Canvas {
14   void wipe() {
15     this.lines.clear();
16   }
17  }
18  class Line {
19   Line(Point start) {...}
20  }
21  class Main extends JFrame{
22   initContentPane() {
23     toolPanel.add(wipeButton);
24  }
25  }
```

Association A3 (color, $\delta^1$(line, color), ...):

```
26  class Canvas {
27   void setColor(String c){...}...
28  }
29  class Line {
30   Line(Color c, Point start){...}
31  }
32  class Main extends JFrame{
33   initContentPane() {
34     toolPanel.add(colorsPanel);
35  }
36  }
```

Association A4 (rect, $\delta^1$(rect, base), $\delta^1$(rect, color), ...):

```
37  class Canvas {
38   List <Rect> rects;
39  }
40  class Rect {
41   Rect(Color c, int x, int y){...}
42   ...
43 }
44  class Main extends JFrame{
45   initContentPane() {
46     toolPanel.add(rectButton);
47  }
48  }
```

Figure 3.2: Associations extracted from our Drawing Applications

# Chapter 4

# Technical Realization

This chapter describes the design of the major components (shown in Figure 3.1) in more detail. First the data structures used to represent the implementation artifacts are described. Subsequently the extraction algorithm is outlined briefly, to provide an idea on how the database is produced. Finally the composition algorithm is described in detail, since its design was a major part of this thesis.

## 4.1 Data structures

The data structure used for the extraction and composition components is a very generic tree-structure and not only applicable for a specific programming language, but for arbitrary implementation artifacts. We used it to represent Java code but it will work as well for other programming languages, as for other kind of artifacts, like models, test cases, documentation and so on. An example for the data structure is depicted in Figure 4.1 for the Product $\mathtt{P_1}$. The tree structure for class `Canvas` is shown in Figure 4.1a. The class is identified by its name, along with the filed contained in the class. Methods are identified by their Java method signature and statements simply are represented by their Java source code. Node `Canvas` is an unordered node with the ordered node `wipe`() as a child. The order of the statements in method `wipe` therefore matters. Statement `this.lines.clear()` refer-

ences the node for field `lines`, since it accesses it. References are denoted in dashed lines.

The classes `Line` and `Main` are depicted in Figure 4.1b respectively 4.1c.

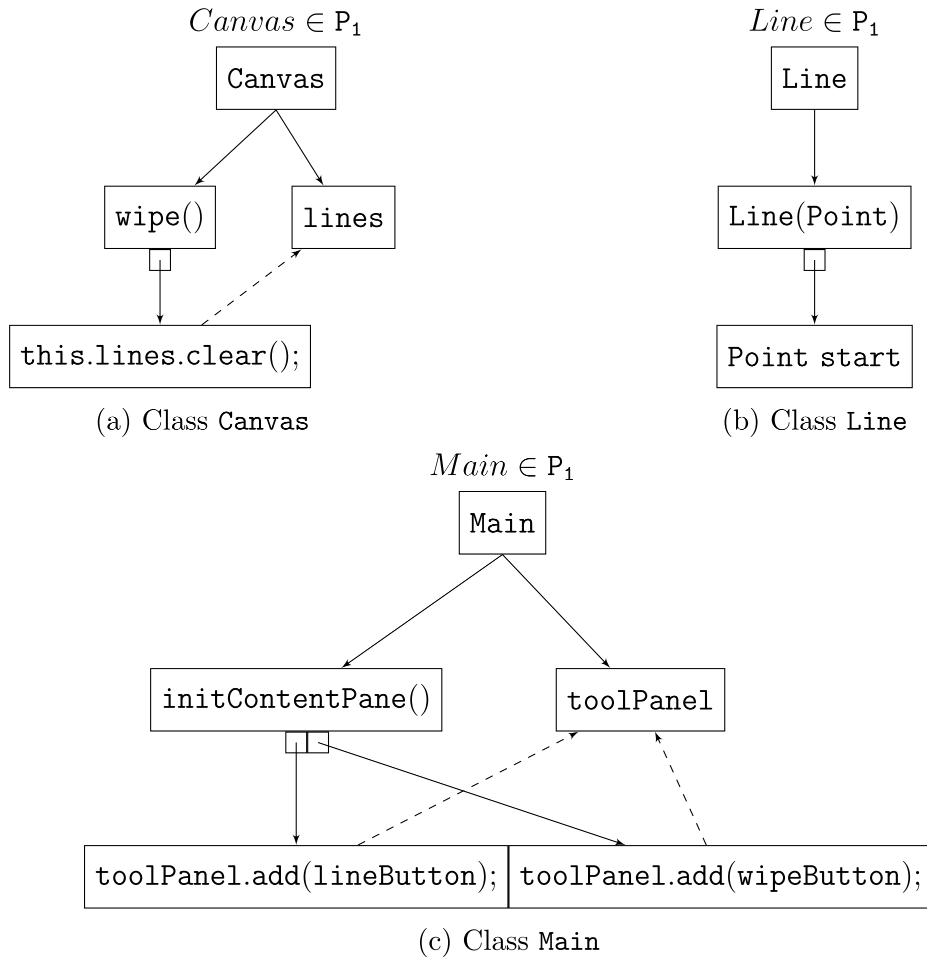



Figure 4.1: Data structure for Product $\mathtt{P}_1$ [4]

As discussed before, we deal with different kind of implementation artifacts. For genericity we defined artifacts with the following content:

- **Artifact Object:** An arbitrary object representing the artifact (e.g. a code line in the source code, a EMF-EObject,...).

- **Identifier:** A string identifying the artifact. This identifier should represent the artifact as good as possible, since it is used to distinguish the artifacts later on.

- **Uses and Used By references:** Artifacts can reference other artifact (e.g. a method call or field access in Java). Therefore each artifact has a uses property where it can reference other artifacts and also a used by property which consists of the artifacts referencing this artifact.

- **Containing Node:** Artifacts also have a reference to the node in the tree they are contained (as described in the following in this section).

The tree-structure itself consist of nodes. These nodes hold the implementation artifacts and moreover consist of the following parts:

- **Parent Node:** A reference to the parent node in the tree-structure.

- **Child Nodes:** An arbitrary number of children, as the next hierarchy level in the structure.

- **Ordered or Unordered:** Nodes can be ordered or unordered. For ordered nodes the order of the child nodes is relevant, for unordered nodes it is not.

- **Sequence Number:** All nodes initially have a sequence number, but only the children of ordered nodes get one assigned, serving as an extension to the identifier and indicating in what position the node should be in its parent. Therefore also nodes containing similar artifacts with similar identifiers can be distinguished.

- **Sequence Graph:** Ordered nodes also contain a sequence graph, which contains the information in which order the child nodes appeared in the input product variants.

- **Unique or Shared:** Moreover each node can either be unique or shared. This is of importance for the extraction and composition algorithms and will be described in more detail later on. For products all nodes are unique.

- **Atomic Flag:** A flag which gives more information about the structure. Nodes that are atomic are treated together with their child nodes as if they only would be one node. This is also important for the extraction and composition and will be covered in more detail there.

In Figure 4.1 we can see such nodes in their tree-structure containing their artifacts as source code fragments. The solid arrows point to the child nodes, where ordered nodes are depicted with a small square under them symbolizing the order of their child nodes. Further the dashed arrows symbolize references between artifacts, as in our example the statements in these referencing artifacts use the fields int the referenced artifacts.

As mentioned before each ordered node get a sequence graph assigned by the extraction. Figure 4.2 depicts an example sequence graph for our example drawing applications. In particular we are concerned about the order of the buttons in the application.

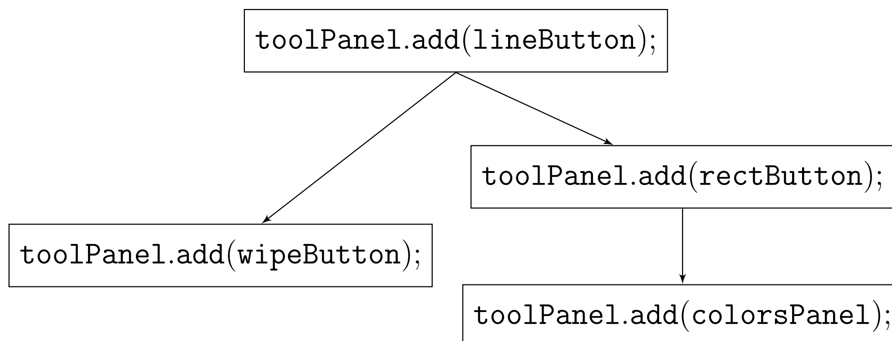



Figure 4.2: Sequence graph derived from Products $\mathtt{P}_1$, $\mathtt{P}_2$, $\mathtt{P}_3$

This sequence graph contains all the information of the order of nodes derivable from the input products. It tells us that the line `toolPanel.add(lineButton);` should always come first. Also the line `toolPanel.add(rectButton);` always comes before the line `toolPanel.add(colorsPanel);`. But the order of the line `toolPanel.add(wipeButton);`, `toolPanel.add(rectButton);` and `toolPanel.add(colorsPanel);` could not be determined, so

`toolPanel.add(wipeButton);` can be put before, in the middle or after the other two lines.

## 4.2 Extraction

The extraction provides the associations for the database. It was implemented as the thesis of [4] and will be summarized here, since it is a very important part of understanding this thesis. Artifacts in the associations are also contained in the data structure described above. In the following we will provide a short description of the extraction, to make clear how the data for the composition is generated and what its limitations are. For more detail on the extraction algorithm please read [4].

### 4.2.1 Module Calculation

Before the actual extraction can take place, the modules of the product variants have to be calculated using the list of features contained in the product. Therefore the list of features contained in the product is combined with the negated features not contained in the product. From this union the powerset is calculated. This gives us all the possible modules. However there are still not valid modules contained in this powerset. First the empty set has to be excluded. Subsequently all modules which consist only of negative features have to be excluded, since negative features can not add any implementation on their own. Negative features only can add artifacts due to interactions with positive features.

The result of this process is a set of sets of features, respectively a set of modules.

### 4.2.2 Intersecting

The first part of the extraction is to intersect the tree-structures parsed from the variant implementations and the implemented features. Therefore

creating a mapping between the artifacts and the modules calculated from the features, by mapping commonalities resp. differences in both. Thereby the modules that are in common can be mapped to the artifacts in common. And also the modules just in one product can be mapped to the artifacts that are unique to this product.

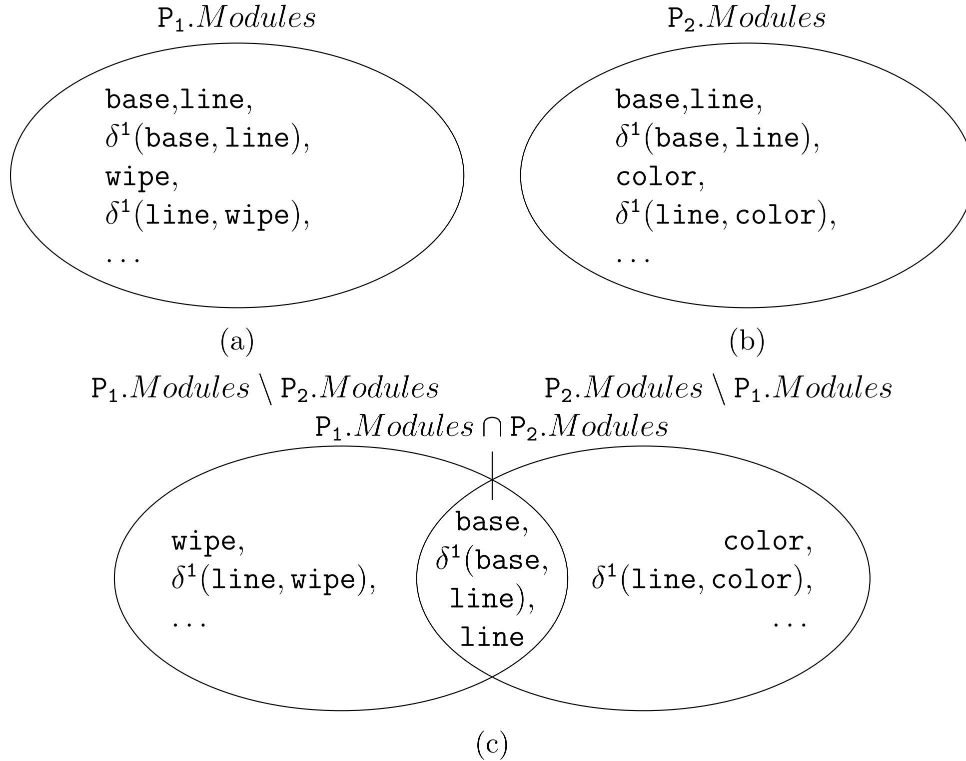



Figure 4.3: Intersecting modules of $\texttt{P}_1$ and $\texttt{P}_2$

Figure 4.3 illustrates the intersection of the modules of our three initial product variants. Complementary to this Figure 4.4 depicts the intersection of `class Canvas`. These intersections are then mapped to each other accordingly. The modules in $\texttt{P}_1.Modules \setminus \texttt{P}_2.Modules$ are mapped to the artifacts in $\texttt{P}_1.Artifacts \setminus \texttt{P}_2.Artifacts$, an modules in $\texttt{P}_1.Modules \cap \texttt{P}_2.Modules$ are mapped to the artifacts in $\texttt{P}_1.Artifacts \cap \texttt{P}_2$, and so on. The mapping is done by putting the modules and their respective implementing artifact into an association together. Therefore we will end up with a set of associations giving us the trace information we need to compose the artifacts later on.

Furthermore we can see in Figure 4.4 that `class Canvas` is in all three

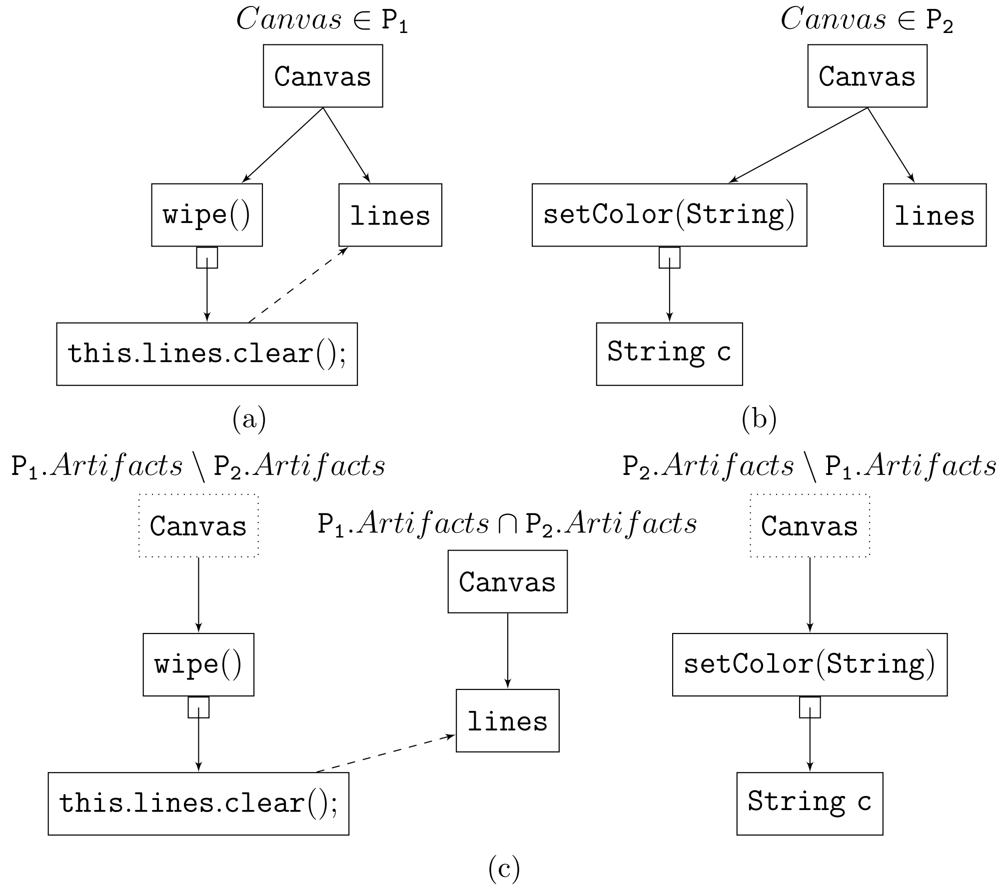



Figure 4.4: Intersecting code of $P_1$ and $P_2$

associations. However it only traces to the intersection, since both products contain this artifact. Nevertheless we need the node containing `class Canvas` as a placeholder, to know the location of its child artifacts. That is where the distinction between *unique* and *shared* nodes comes into account. In an associations only the *unique* nodes trace to the respective modules. The *shared* nodes are the placeholder nodes, which do not really trace there but are needed. In Figures 4.4 *shared* nodes are depicted as dotted and *unique* nodes as solid boxes.

Note that if a node was atomic it would be treated as a leaf node, together with all its child nodes. This means it would only be *unique* in on association and not occur in any other association.

As can be seen in Figures 4.3 and 4.4 is that not every modules can be completely traced to its implementing artifacts. For instance the modules

`base` and `line` can not be separated, since they occur in both input products together. If we would want to separate these modules we would have to provide a variant which does not implement feature `LINE`.

In some cases the extraction of artifacts is not that straight forward. For theses cases the associations also contain another set of modules, for non-unique traces, so where in the intersection no modules where left to trace to these artifacts. These modules are calculated as the set of all modules that where ever associated with the contained artifacts in any product variant, minus all the modules of the products that do not contain the artifacts. This is a more coarse mapping, and is only used if the extraction fails to uniquely trace artifacts.

Furthermore the artifacts stored in the nodes are reused. This means that the artifact is always the same for equal nodes. In our example all the nodes for `class Canvas` contain the same artifact object. Furthermore the artifacts reference to the node they are contained in, always points to the *unique* instance of this node.

The extraction works incremental, so we can add products as we go along. Therefore the extraction will use the already derived mappings and make the same intersection with the newly added products, to refine the mapping and gain more information about the implementation of certain features.

### 4.2.3 Sequencing

Now we want to discuss the sequencing of our tree-nodes. As illustrated before the possible orders of nodes (e.g. containing statements) are stored in a sequence graph, which tell us which ordering of statements are valid according to the input products. Therefore all the ordered nodes are *sequenced* and a sequence graph is assigned to them.

Figure 4.5 shows an examples of such a sequence graph for the content of the method `initContentPane()`. The parts 4.5a, 4.5b and 4.5c show the code snippet from the initial product variants. Afterwards we show how the

sequence graph grows with each product that gets added.

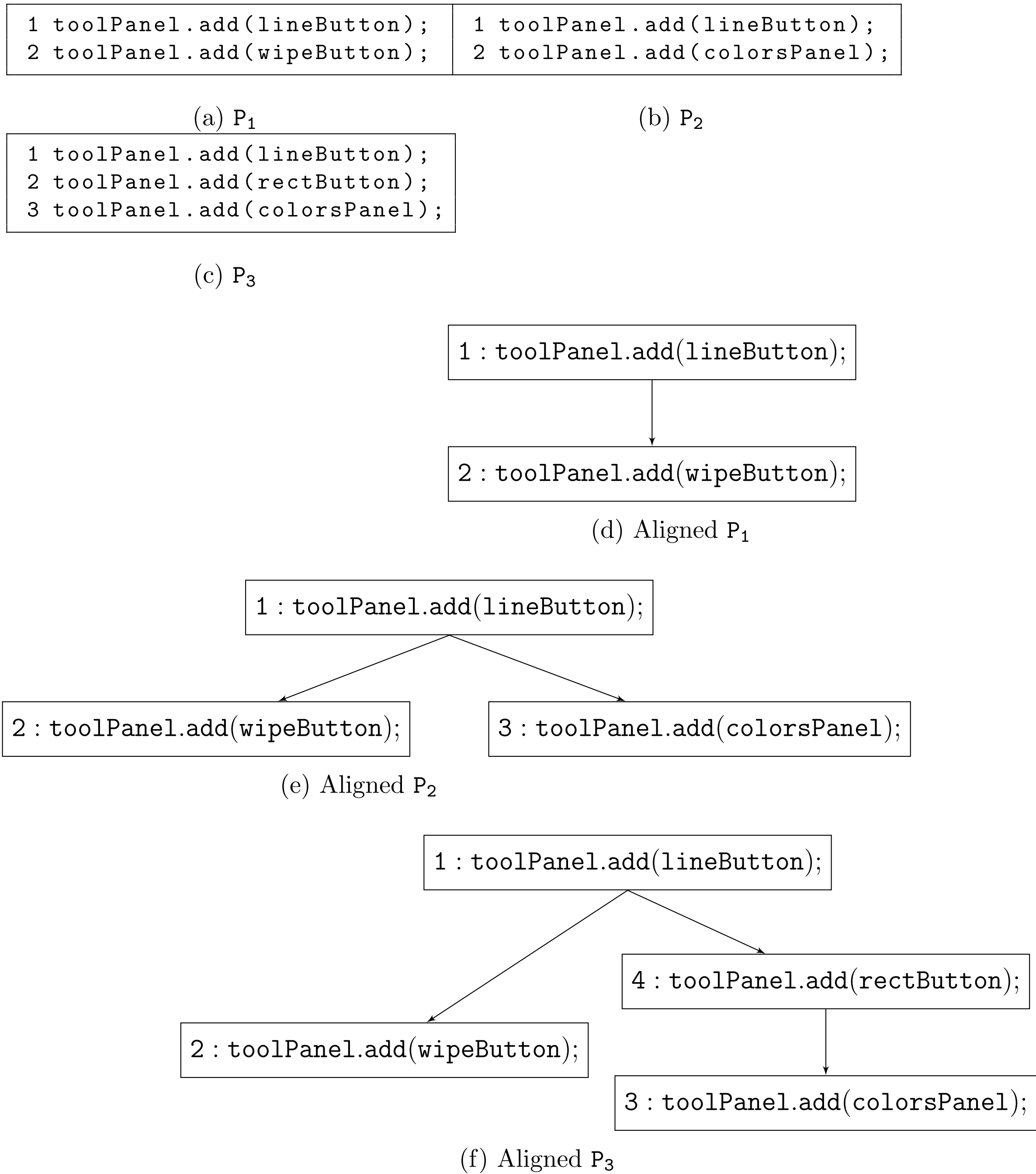



Figure 4.5: Sequence graph derived from Products $P_1$, $P_2$, $P_3$

In 4.5d only the product $P_1$ was added and we learned from it that the line `toolPanel.add(lineButton);` comes before line `toolPanel.add(wipeButton);`. Also sequence numbers have been assigned to these statements, depicted before the colon in the nodes. Subse-

quently product $P_2$ was added an the sequence graph extended (see 4.5e). There we learned that `toolPanel.add(lineButton);` is also before line `toolPanel.add(colorsPanel);`. The lines `toolPanel.add(wipeButton);` and `toolPanel.add(colorsPanel);` are parallel because they never co-occurred in a product and we therefore can not determine the ordering of these two statements. The newly added line `toolPanel.add(colorsPanel);` got the next sequence number (3) assigned. As the last product, $P_3$ is added, which tells us that line `toolPanel.add(rectButton);` is also after `toolPanel.add(lineButton);` but before `toolPanel.add(colorsPanel);`. This leaves us with the sequence graph depicted in 4.5f, with the newly added line `toolPanel.add(rectButton);` with the next sequence number (4). We still have no knowledge about where the line `toolPanel.add(wipeButton);` goes in the code, other than that it should be somewhere after `toolPanel.add(lineButton);`.

## 4.3 Composition

In this section we will describe how the composition of a product works using the before extracted associations. This part was implemented for this thesis and is part of our main contribution. We will discuss how the artifacts that should be put in the new product are selected, how they are restructured, including the resolving of references and also how the order of the artifacts is determined. Also there are some further configuration options to the composition we will discuss in this section.

Figure 4.6 shows an overview of the steps that will be described in detail in this section.

### 4.3.1 Selecting Code

In the first step of the composition the associations which are of relevance for the desired product have to be selected. Therefore the composition

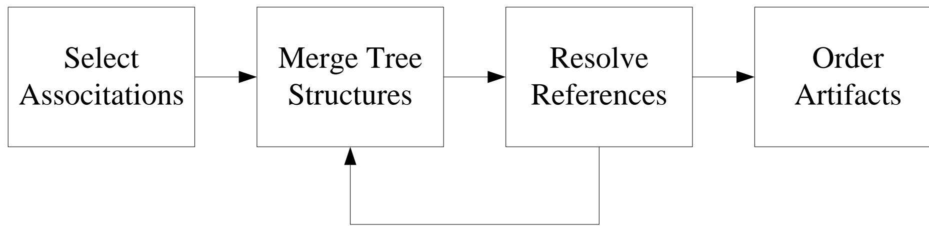



Figure 4.6: Composition Steps

needs a set of features as input, to specify which product variant should be composed. For example we want to compose product $\texttt{P}_4$ of our example after extracting all the trace information from $\texttt{P}_1$, $\texttt{P}_2$ and $\texttt{P}_3$ from our drawing applications portfolio. This means the composition will get the features BASE, LINE, RECT and WIPE as an input.

From these features the modules have to be calculated, using again the algorithm outlined in Section 4.2.1. Through the application of this algorithm we get the desired modules that we need to completely compose the product. Figure 4.7 depicts these desired modules for $\texttt{P}_4$.

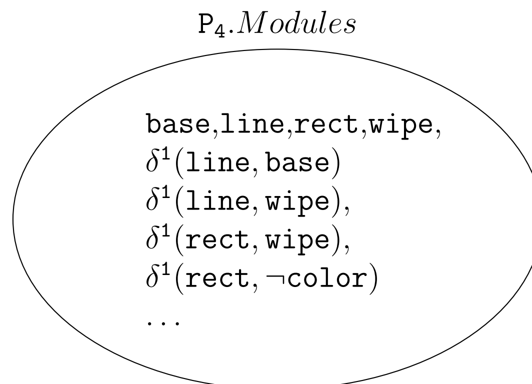



Figure 4.7: Desired modules for composing $\texttt{P}_4$

The selection is now done by intersecting these desired modules with the modules of every association separately. In the default settings the composition will select every association where the intersection of the modules is not empty, i.e. the association has at least one module that is needed to compose $\texttt{P}_4$. However this is in some cases not the ideal way of selecting artifacts for the product. Therefore the composition can be configured to refine this selection for different cases. The first configuration allows the

software engineer to assign a limit to the order of the modules, therefore only modules below this configured order are intersected. Secondly a percentage of the intersection can be configured, which will set a minimum overlap for the modules in the intersection in ratio to all the modules in the association. Therefore an association will only be selected if its modules are, to this percentage, needed to compose the product. This can help to rule out cases where a whole association, with maybe hundreds of modules, is selected just because one of its modules is needed, which can lead to errors.

As we have discussed earlier the associations also contain a second set of modules, in case the extraction could not find a unique trace for the contained artifacts. Is this the case then these associations are selected according to the same criteria as the others. Further the composition can be configured to use these non-unique traces in this case (as is set by default), or just to ignore associations that could not be traced precisely.

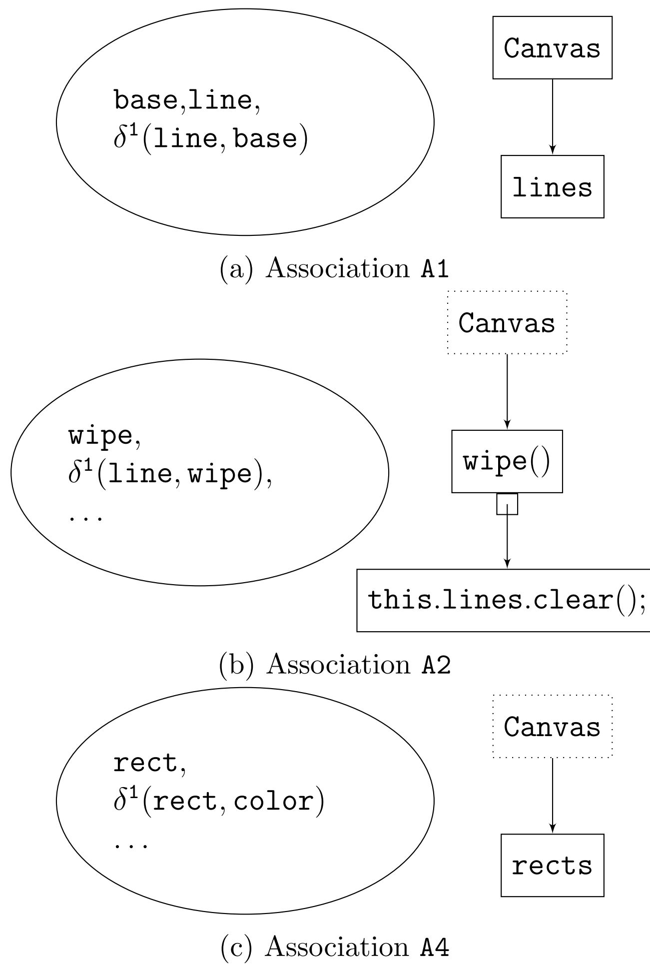
Figure 4.8 depicts the associations that are selected for the composition of product `P`$_4$. On the left hand side are the modules of the associations and their respective nodes of `class Canvas` on the right next to them.

If all of the before mentioned configurations still are not sufficient to select the desired associations, the composition also offers an interface for manual refinement of the selection. More on that in the tool description in chapter 5.

Furthermore while the selection is done, all the modules of the selected association are stored in a single set, as depicted in Figure 4.9.

Based on this set of selected modules (*SelectedModules*), the composition calculates warnings about modules that are missing and also modules that are surplus to the ones we needed for composing the product (see Figure 4.10).

- **Missing Modules:** These are all the modules that are not contained in any of the selected associations (in the default configuration not contained in any association), hinting to artifacts that might be miss-




(a) Association A1

(b) Association A2

(c) Association A4

Figure 4.8: Selected Associations for composing $\mathtt{P}_4$

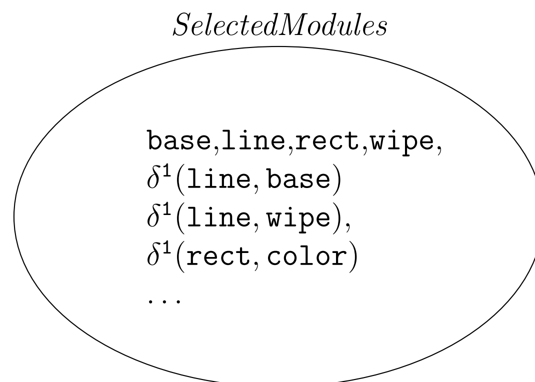



Figure 4.9: Modules of the selected associations

ing. In our example it is calculated by $\mathtt{P}_4.Modules \setminus SelectedModules$.

- **Surplus Modules:** These are modules that where contained in the selected associations, and therefore stored in the *SelectedModules* set,

but are not needed to compose the product. Therefore there might be artifacts in the selection that are not needed for the desired product, yet can not be filtered out. In our example theses are the modules in $\mathit{SelectedModules} \setminus \mathtt{P}_4.\mathit{Modules}$.

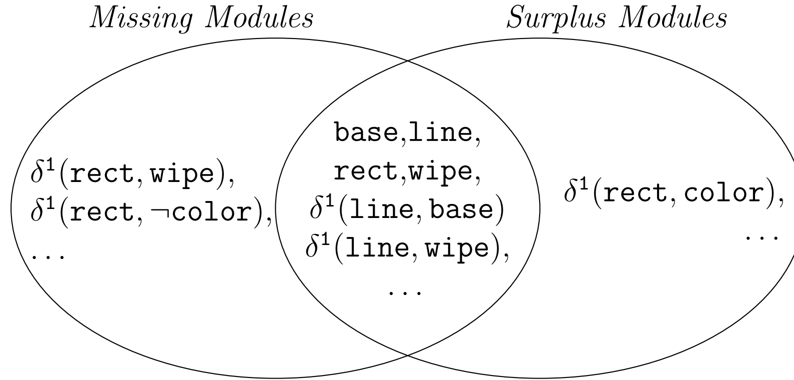



Figure 4.10: Calculating *Missing* & *Surplus Modules*

### 4.3.2 Tree Merging

Now the artifacts of the before selected associations are composed to a product. This is done by incrementally growing the new tree.

In particular the algorithm iterates over the selected associations and takes the contained tree-fragments. Each of theses tree-fragments are traversed and the nodes copied into a new structure, that is stored in the new product. Therefore the nodes in the associations remain unchanged. Moreover while traversing the trees the algorithm checks if the current subtree already was added to the products nodes and only copies the parts that are new to the product. Figure 4.11 illustrates how the tree-structure is restructured incrementally. On the left hand side the nodes of the association are shown and to the right the nodes of product $\mathtt{P}_4$ after adding these nodes. Note that the order in which the tree-fragments are added does not matter.

Further the algorithm remembers which of the inserted nodes where *unique* once and which nodes only occurred as *shared* in the selected associations.

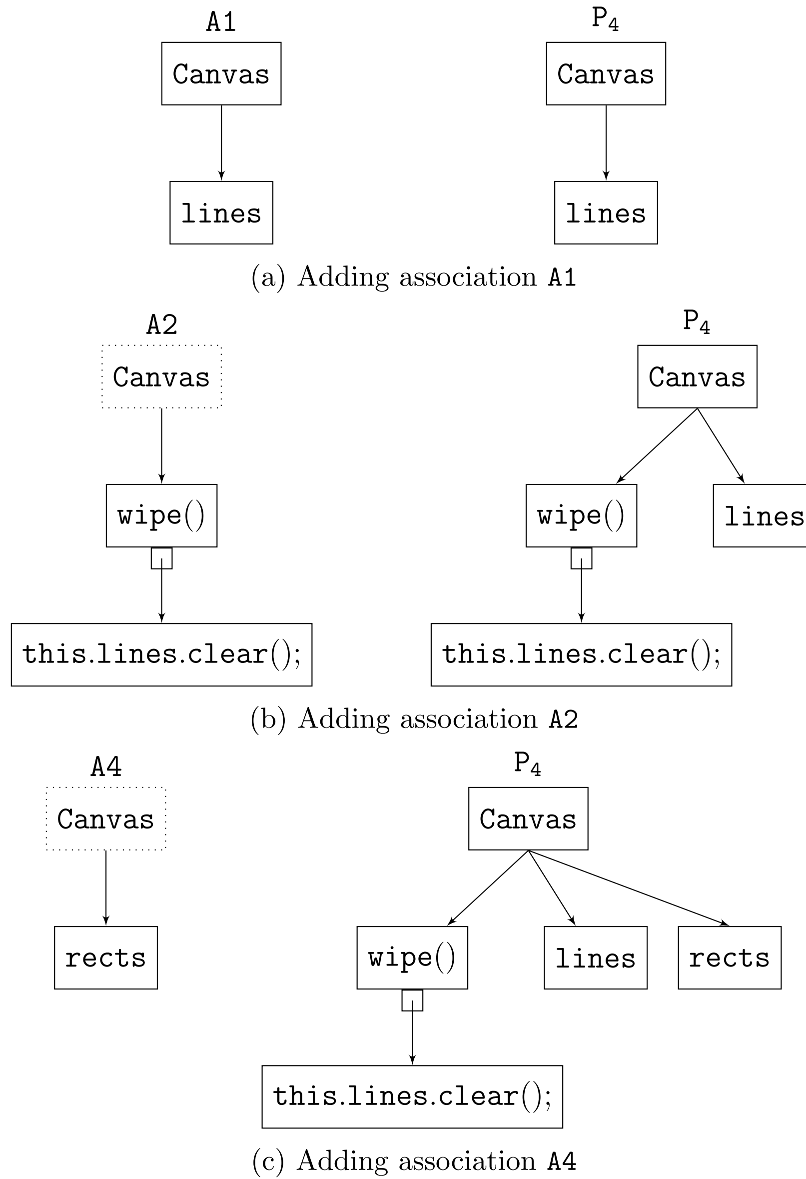



Figure 4.11: Merging Trees for composing $P_4$

Therefore it can issue warnings to tell the software engineer about placeholder nodes that had to be added to compose the product.

### 4.3.3 References

After the nodes are merged into the complete tree-structure again, the composition checks if all the used artifacts (i.e. referenced artifacts) are also contained in the new tree.

Therefore the algorithm traverses the composed tree-structure an checks

the contained artifacts for used references. If an artifact uses another artifact, the algorithm checks if this artifact is also contained in the tree, by traversing it again. Is the referenced artifact contained then nothing is to do. However if the referenced artifact can not be found in the composed product, the software engineer has to decide on what to do with this reference. It can be chosen from the following options:

- **Leave reference unresolved:** As default option the composition simply lets the unresolved reference stay in the code. This will raise a syntax-error in the product variant, which however can be useful as it marks a place in the source code where manual changes are required.

- **Remove referencing artifact:** Remove the node which holds the artifact that references the none-existent artifact (e.g. variable, field or method). If the reference is below an atomic node, then the atomic parent node is removed.

- **Add referenced artifact:** Add the node which holds the referenced artifact to the tree. Respectively its atomic parent node, if existent. Also the *unique* children of the referenced node have to be included, since they are important in most cases (e.g. type of a field). This will trigger another search for references if the added artifacts again reference other artifacts.

- **Add referenced association:** Add all the nodes of the association which holds the referenced artifact. Again all the contained artifacts have to be checked for references, which then can lead to more decisions that have to be made.

Figure 4.12 depicts the nodes from class `Canvas` of the composed product $\mathtt{P}_4$ after resolving the references.

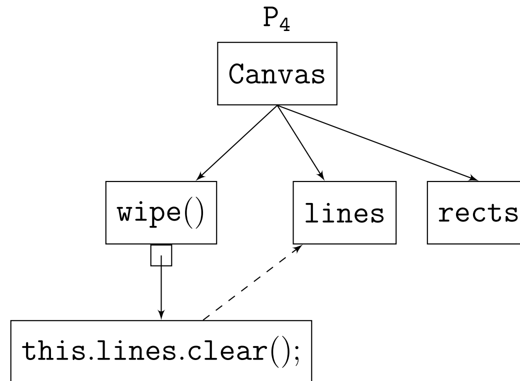



Figure 4.12: Resolved references in product $P_4$

### 4.3.4 Code Order

Until now the composition only restructured the nodes into a complete tree-structure, but did not consider the order in which the nodes should be put together. So as the last step of composing a product, the algorithm checks the order of the nodes in the tree.

The algorithm checks the sequence graph of every *ordered node* for possible orderings of its child nodes. Therefore it can determine all valid orders for these nodes. Is there only one possible order then the composition will put the child node in this exact order. If there are more than one orders contained in the sequence graph, the software engineer is notified and asked to select an order for the nodes.

Moreover it can be configured, if the order is not of great importance in composing a certain product, and therefore do not ask the software engineer for a decision but automatically chose the first valid order that conforms to the sequence graph. Also the composition allows to configure a threshold for the number of possible orders that should be generated. It will then limit the orderings from which the software engineer can chose.

From the sequence graph depicted in Figure 4.5 the algorithm can derive the two orders shown in Figure 4.13 for composing the product $P_4$.

| | |
|---|---|
| 1 toolPanel.add(lineButton);<br>2 toolPanel.add(wipeButton);<br>3 toolPanel.add(rectButton); | 1 toolPanel.add(lineButton);<br>2 toolPanel.add(rectButton);<br>3 toolPanel.add(wipeButton); |

Figure 4.13: Possible orders of Buttons in $\mathtt{P}_4$

# Chapter 5

# Tool Support

This chapter illustrates the tool support for the implemented workflow cycle. We will show an overview of the most important parts of the tool by means of a step by step demo scenario, that illustrates how the tool is intended to be used. Subsequently we will go into more detail of some parts that did not surface in the demo scenario.

The tool is implemented as a plug-in for the eclipse IDE (integrated development environment). Eclipse is one of the most used development environments and provides a good extendibility through plug-ins.

## 5.1 Demo Scenario

For this scenario we will use our illustrative example with the three drawing applications as input for the cycle.

Note that the tool includes a perspective in eclipse, which contains the needed views for working with the tool. It can be selected in *Window - Open Perspective - Other - Composition*.

At first the engineer has to create a new project in eclipse (*File - New - Project - Composition - Composition Project*), as shown in Figure 5.1.

By clicking on *Next* the name and location of the project has to be defined (see Figure 5.2). We used the name "Draw" for our drawing applications example.

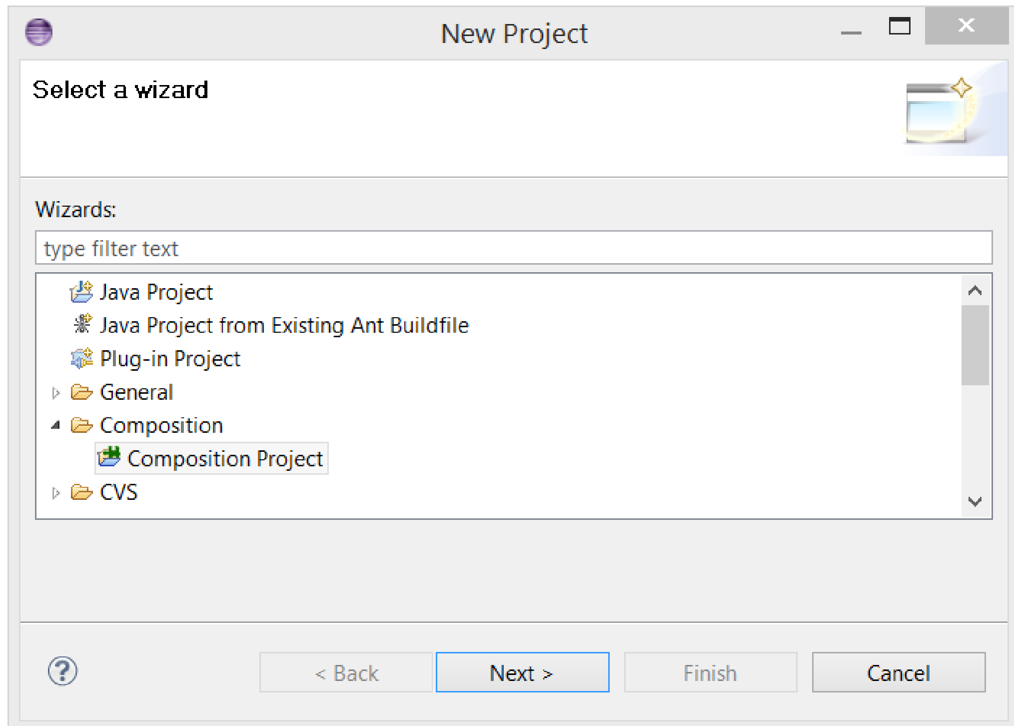



Figure 5.1: Create new Project

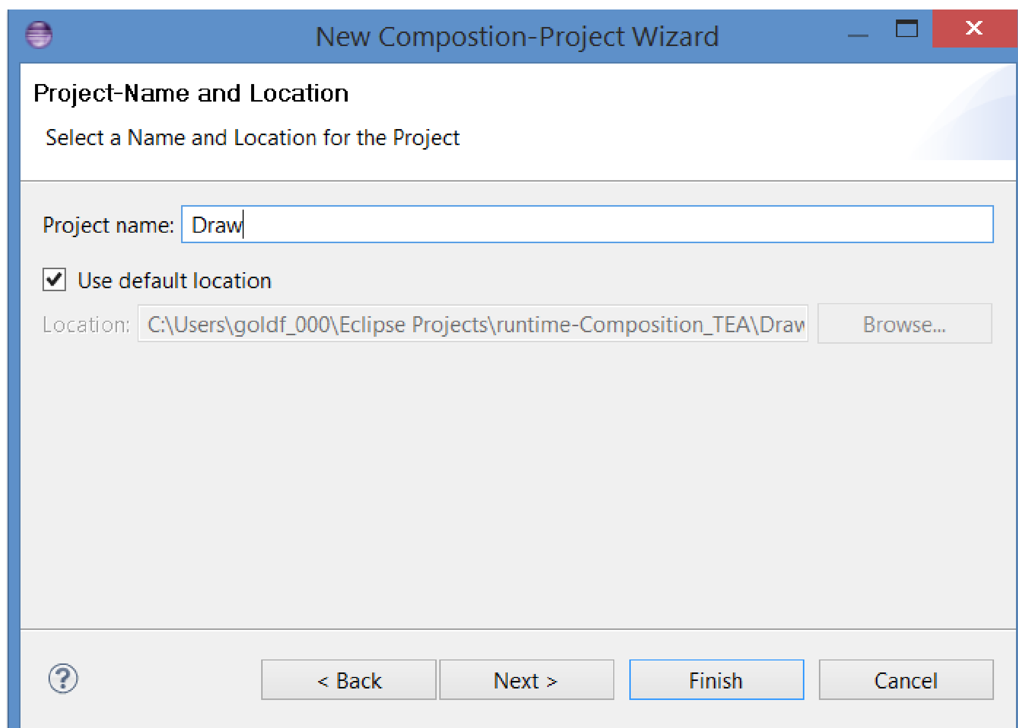



Figure 5.2: Define Name and Location

Subsequently Figure 5.3 shows the selection of a folder that holds the input product variants. Note that there is a convention on the format of

product variants to import them into the project like this. They have to be folders named with the variants name, containing a text file "features.txt" containing the list of features implemented and a folder "src" which contains the implementation (e.g. source code files).

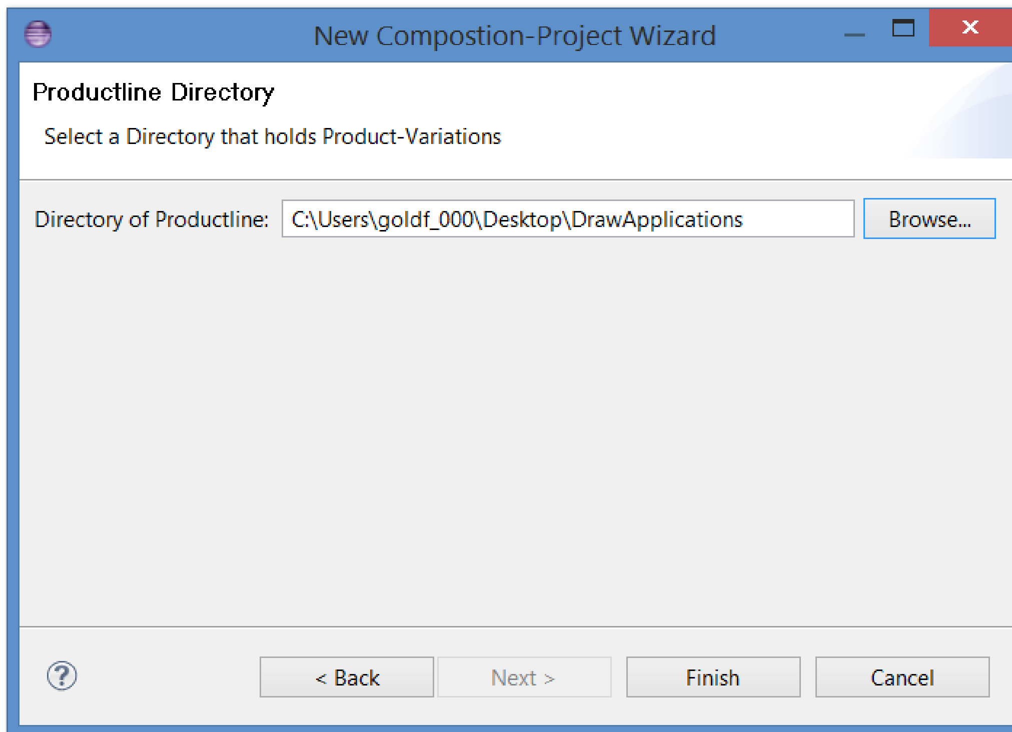



Figure 5.3: Select input Product Variants

The tool is extensible for the implementation artifacts it can work with. Moreover it is possible to define different ways the tool should handle the same artifacts in form of representation. Therefore if for an implementation there exist more than one way to interpret it, the tool will ask, once per project, how to handle these files, like in Figure 5.4 for Java files. They can be translated to EMF, using JaMOPP [6], and then handled as *EMFArtifacts*, or they can be directly parser by a Java parser that will generate our data structure with the code lines as strings (see Section 6.3). We will use EMF for our example.

The tool will now parse the products and display them in the navigator view on the left side of eclipse. As shown in Figure 5.5.

The parsed product variants are then inserted into the database where the

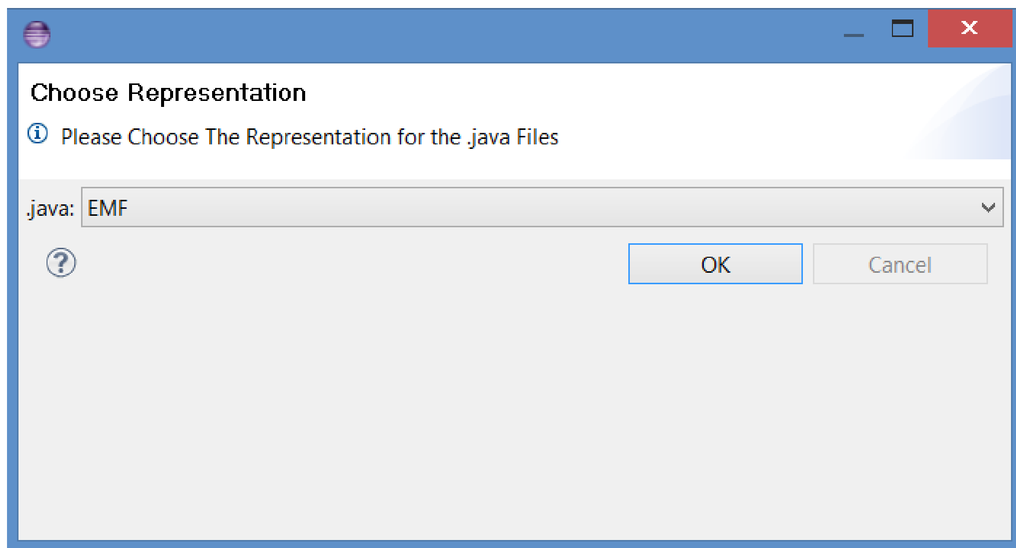



Figure 5.4: Choose a Representation for Java files

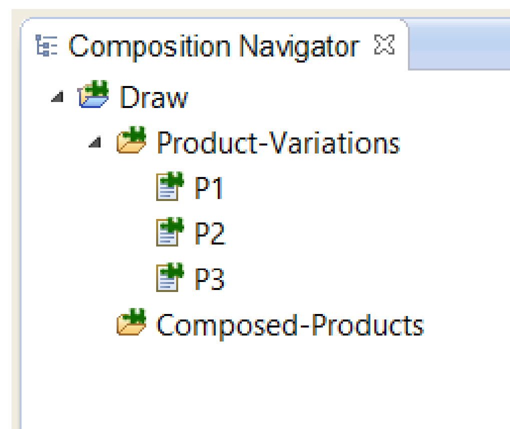



Figure 5.5: Parsed Product Variants

trace extraction is triggered. The resulting associations can also be viewed with the tool (Figure 5.6), where the software engineer can see the features contained in an association, the modules and of course the implementation artifacts. This view is opened by *Right Click on Product-Variations - Open* or by *Double Click on Product-Variations*. This view also includes an outline and properties view, which give more information about the location and characteristics of individual artifacts. The outline view also provides the option to switch to a more intuitive code view, which replaces the labels in the tree with the actual code line, as long as this is implemented for the used representation. Note also that only the artifacts that trace to an

association are displayed there, placeholder nodes are concealed, to avoid confusion.

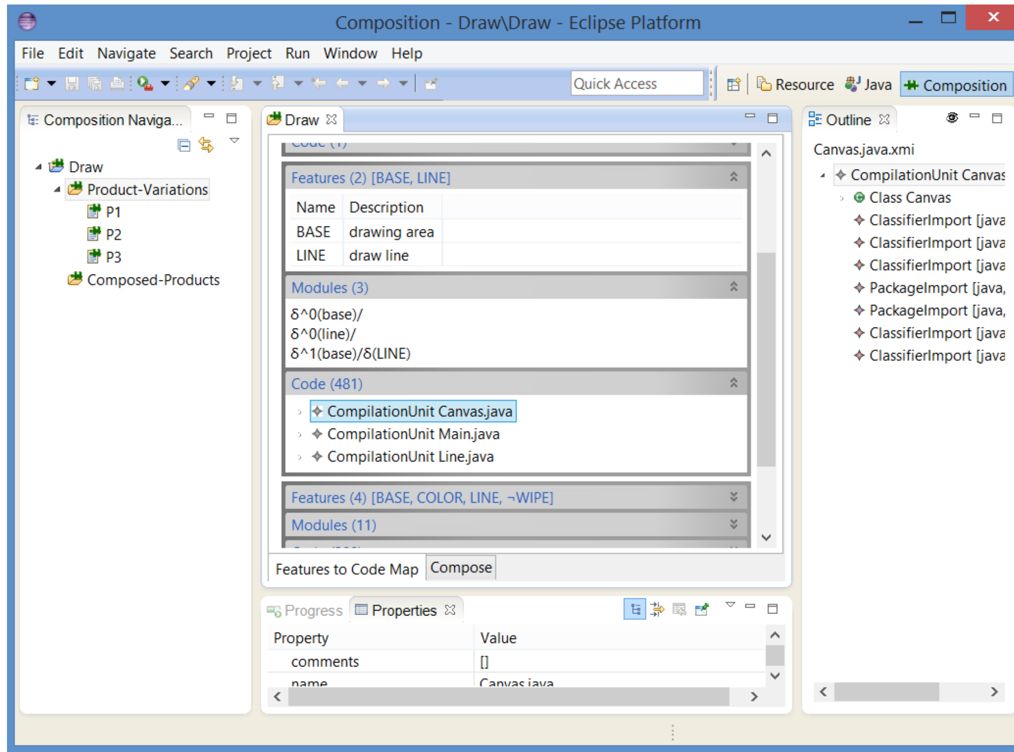



Figure 5.6: Database View

On the bottom the user can select a tab. By switching to the *Compose*-tab the form for composing a product appears. In this form a name for the desired product has to be entered, along with a selection of features which it should implement (see Figure 5.7).

By clicking on the *Compose*-button the composition gets initiated. Different kinds of warnings are generated, to support the software engineer, and viewed in the tool. The first warnings generated are usually addressing missing resp. surplus modules.

In Figure 5.8 we can see the warning for missing modules, that where not in the database. There it shows that the derivative modules that implement the feature interaction for $\delta^1(\texttt{rect}, \texttt{wipe})$ and $\delta^1(\texttt{rect}, \neg\texttt{color})$ are missing, which will probably affect our composed product.

Figure 5.9 on the other hand depicts the warning for surplus modules, that could not have been filtered out. For instance we have the module

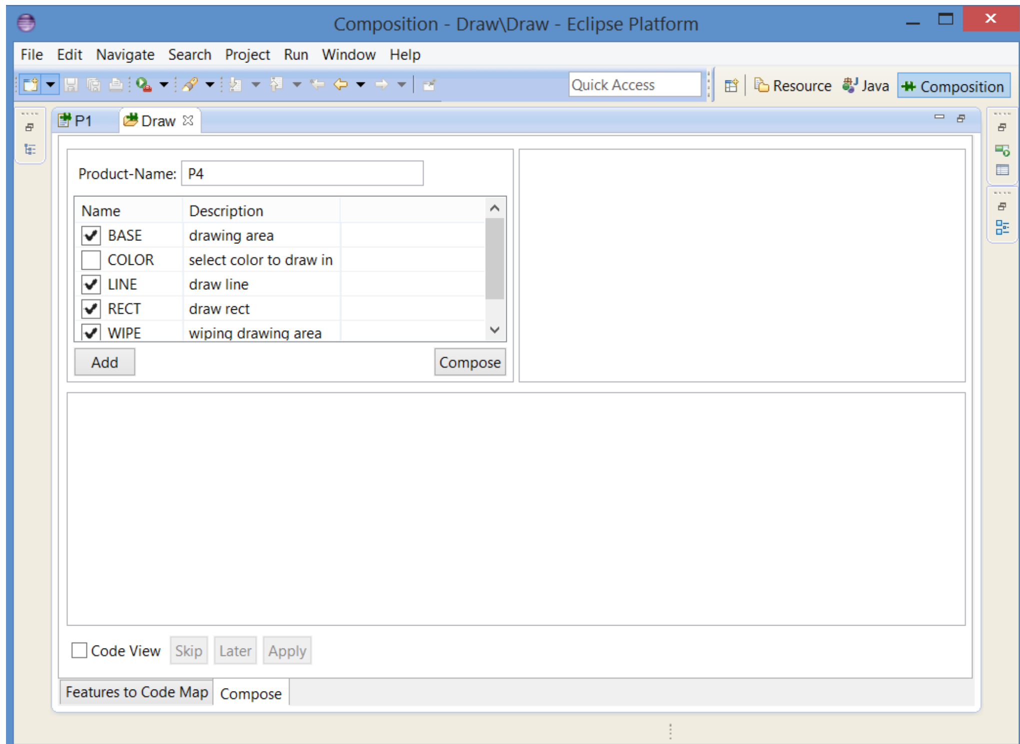



Figure 5.7: Compose Product $P_4$

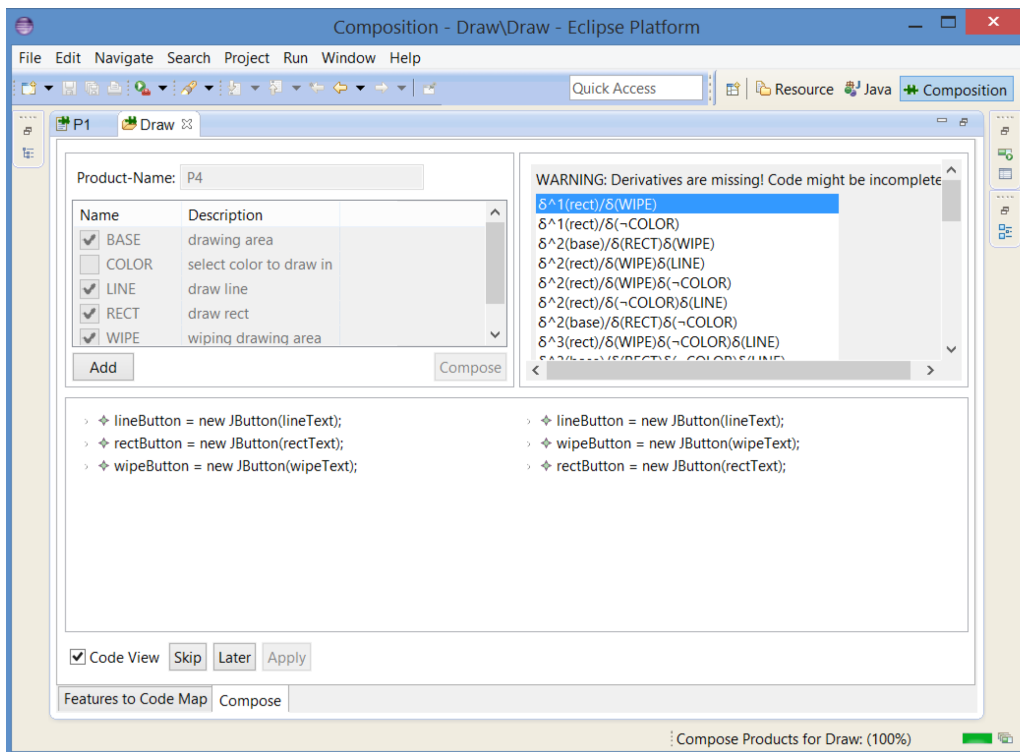



Figure 5.8: Missing Modules

$\delta^1$(rect, color) in the product, which we do not want, since the feature COLOR is not selected. When selecting one of these surplus modules the tool will display the artifacts contained in the respective association in the outline view, so the software engineer can directly check if there is code included that is not desired in the composed product.

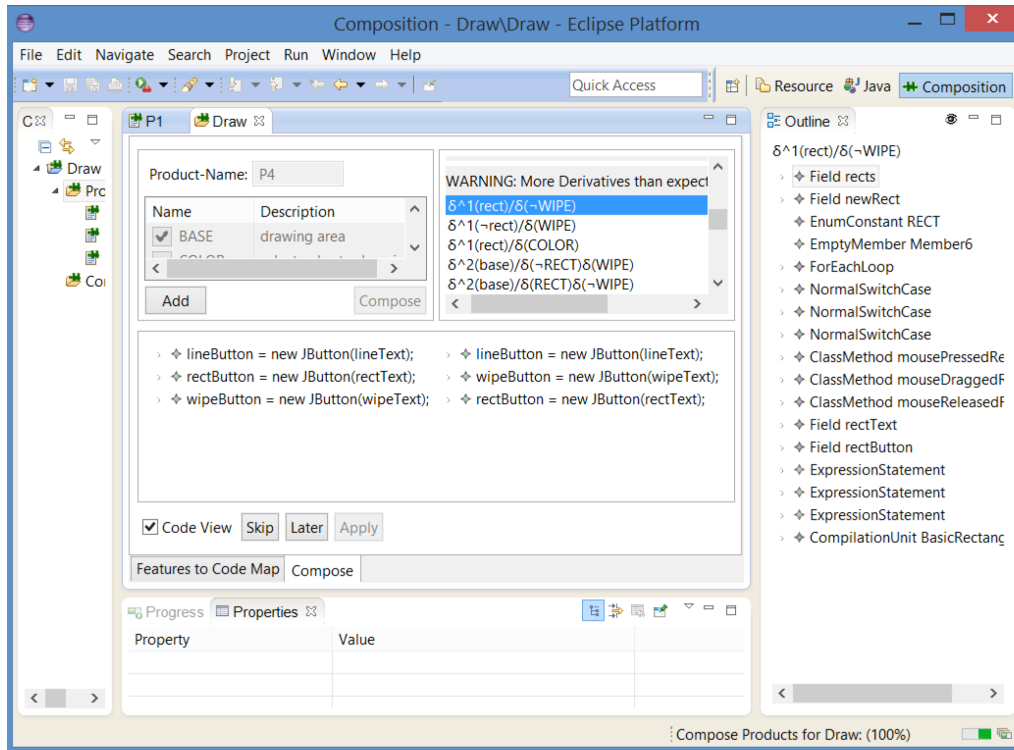



Figure 5.9: Surplus Modules

For the use references which targets could not have been found in the composed product a dialog will pop up and ask the software engineer to decide what should be done with this reference (see Figure 5.10). Moreover the dialog includes a check-box, which will store the decision made and automatically make the same decision for every unresolved reference in this composition. We choose to ignore theses unresolved references for now. This will later lead to compilation error in the code. However these errors mark the spots where manual adaption of the variants code is required.

As discusses in earlier chapters the composition checks if there are different valid orders in which artifacts can appear. If there exists more than one valid order the tool will ask the software engineer which one should be used

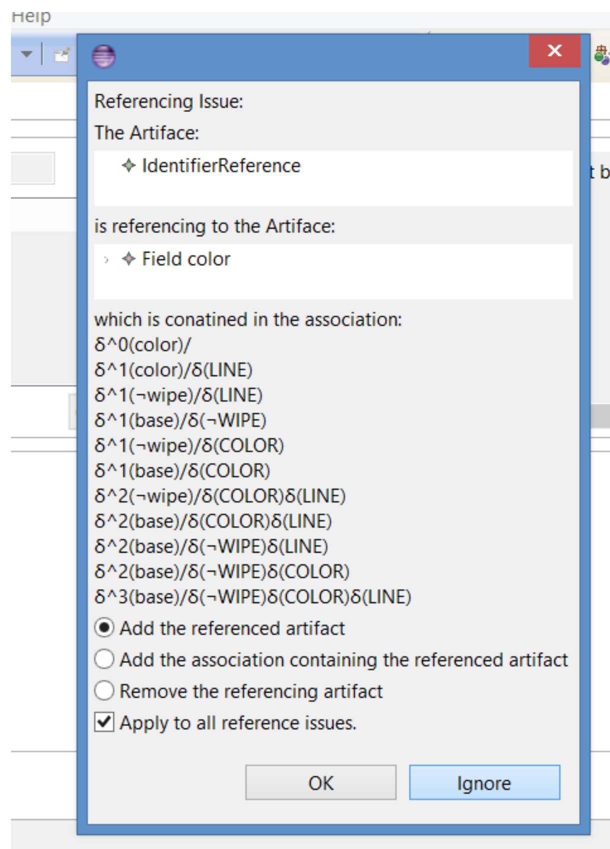



Figure 5.10: Resolve Use References

in the resulting product. In Figure 5.11 we can see this order warning, again on the example regarding the order of the buttons, that we used before to illustrate the order problem. Like in the outline view we also can switch to a code view, by selecting the check-box named *Code View*. For selecting an order the software engineer can either press *Skip*, which will simply order the artifacts according to the first option. Or he can select one of the options, by clicking on one of the artifacts of the option, and the press *Apply*. The tool will then order these artifact according to the selected, displayed with a gray background, order. Should the desired order for these artifacts not be contained in the options, then the engineer can determine his own order by simple drag and drop, and select this as the order the artifact should be put in the composed product. Furthermore the user can press the button *Later*, which will reschedule this order warning, so the software engineer can address other order issues first.

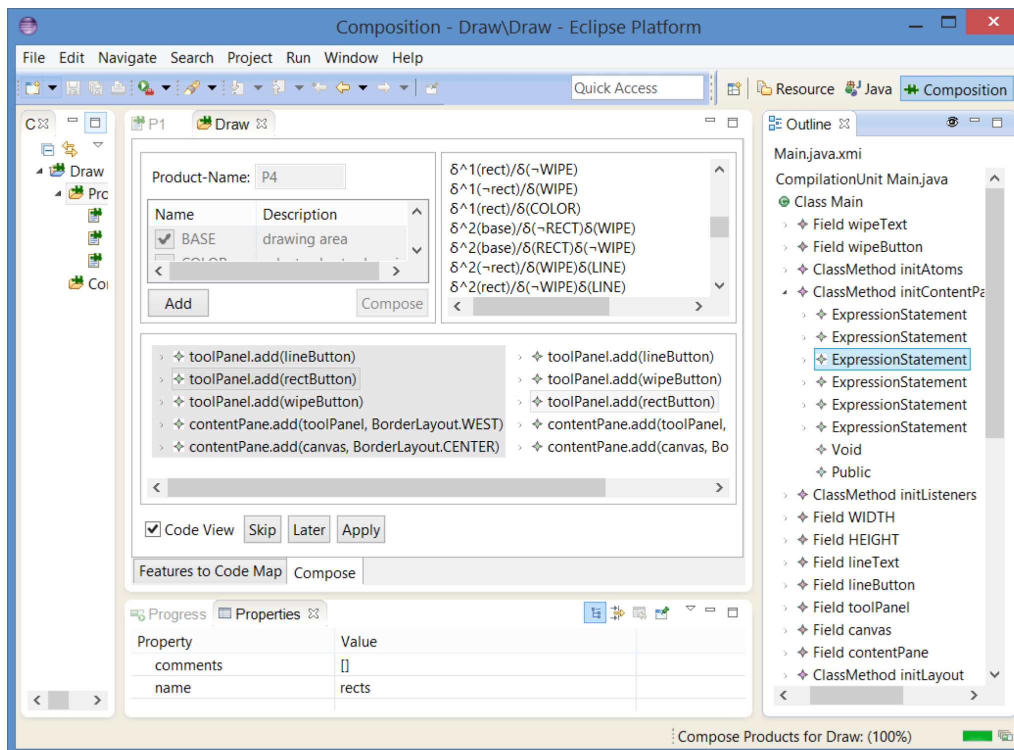



Figure 5.11: Choose Order of Statements

After all these choices have been made and the composition is complete the composed product will appear in the folder "Composed-Products" (Figure 5.12).

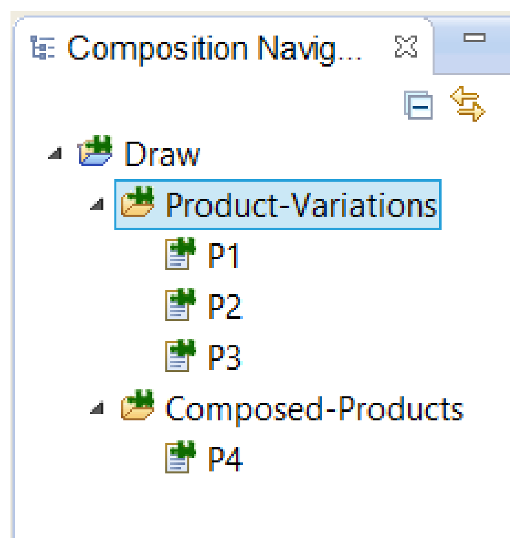



Figure 5.12: Composed Product $\mathtt{P}_4$

However as the warning seen above suggest the product variant is not

finished yet. There is still code missing resp. surplus that could not have been derived from the information in the database. Therefore the software engineer needs to manually complete the new product. Nevertheless the tool also supports the user fixing those issues. Through *Right Click on the new Product - Move to independent Project* the tool will create a new eclipse project, containing the code of the product. Therefore the software engineer can use the editing capabilities of eclipse to complete the product in a manner he is used to.

Note to change the perspective in eclipse, so the new projects are displayed. As we open this project, we can see that there are compilation errors in the code (see Figure 5.13). These errors are relating to the warning the composition triggered before. In particular this error was introduced by the decision to ignore the unresolved reference, during composition. Furthermore the composition triggered a warning that the module $\delta^1(\texttt{rect}, \texttt{color})$ could not have been filtered out and the module $\delta^1(\texttt{rect}, \neg\texttt{color})$ was entirely missing in the database. These warnings are also related to this error. Therefore we have to manually fix this, like discussed in chapter 2.

Additionally to compilation errors also semantical errors can occur, like if there is code missing. In our case the composition informed us that the module $\delta^1(\texttt{rect}, \texttt{wipe})$ was missing for $\texttt{P}_4$. This means that there is probably code missing for the feature interaction responsible to wipe the rectangles from the drawing area. Indeed when we start the application and used the wipe functionality, only the lines are cleared from the drawing area and the rectangles remain unchanged. Consequently we will have to fix this too. The tool offers support in this case, by helping the software engineer in finding where the involved features are implemented. As shown in Figure 5.14 we can find the associations which contain implementations of the desired feature to locate the responsible code parts. In our case the implementation of Feature `WIPE` is contained in a single association. We can find a method `wipe()` in there and also can see in the outline view that this method is located in `class Canvas`. After making some small adjustments

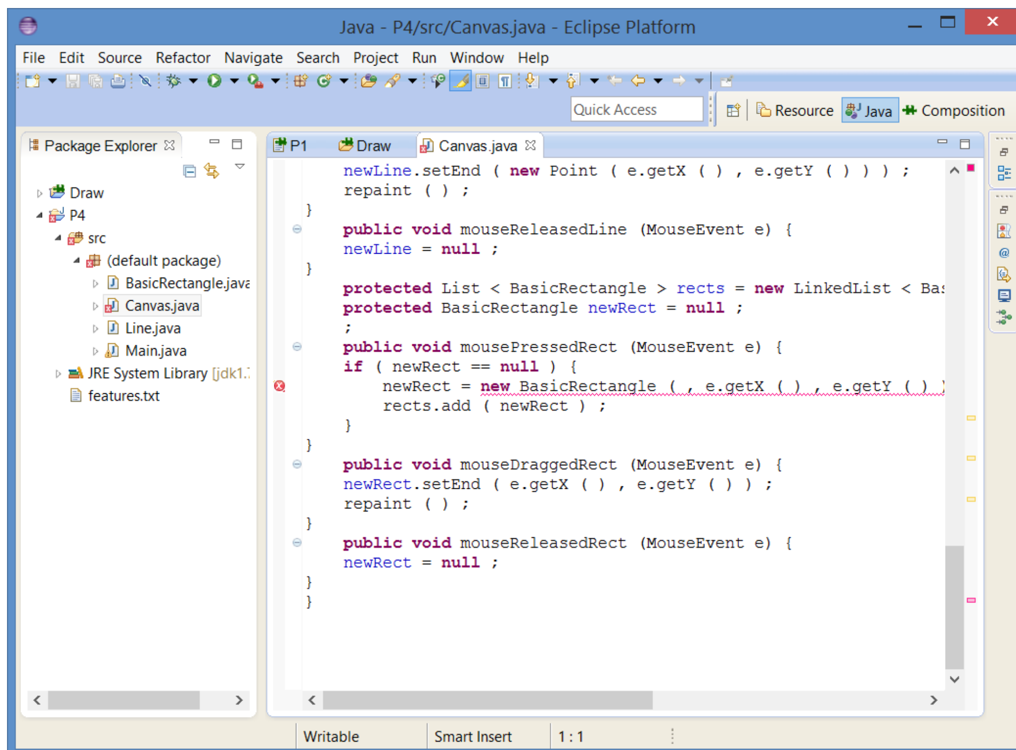



Figure 5.13: Java Project for Product $\mathtt{P}_4$

to this method, also the rectangles get cleared when wiping the drawing area. This mean the product variant is complete.

Now the next step is to parse these changes back into our data structure. Therefore the user switches back to the composition perspective, *Right Click on* $\mathtt{P}_4$ - *Update Code* and the code changes are applied to the product variant there.

Finally the now complete product is added to the input products, by *Right Click on* $\mathtt{P}_4$ - *Move to Product-Variations*. This will update the database, which will get refined with the information gained by this new product. Therefore the knowledge on how to handle certain feature interactions and the code implementing them is now available and the same problems will not have to be fixed again. For instance the module $\delta^1(\mathtt{rect}, \mathtt{wipe})$ is now in the database and therefore this interaction can be automatically composed in the future.

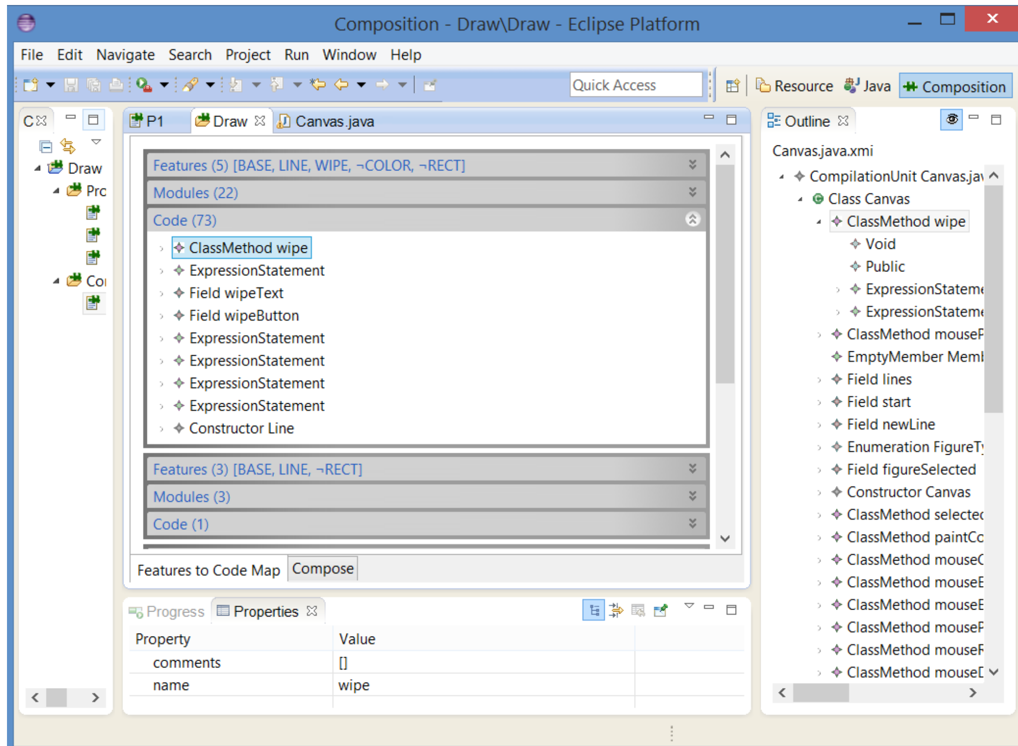



Figure 5.14: Locate Feature WIPE in the Code

## 5.2 Other Capabilities

In this section we will show tool parts that are important for the usage of the tool, yet where not covered in the intended workflow described in section 5.1.

Product variants can also be added after the project was created. By *Right Click on Product-Variations - New - Product-Variation* a dialog window opens, which lets the software engineer select a product to add (see Figure 5.15). It is either possible to specify the path to a product folder, that fits the conventions mentioned before, which will automatically display the products information in the dialog. Alternatively the software engineer can specify the product information, name, location of the implementation (e.g. code) and the implemented features manually.

Like shown in Figure 5.6 the associations contained in the database can be presented by the tool. Moreover also the product variants can be displayed. Both the features a product implements (see Figure 5.16) and the

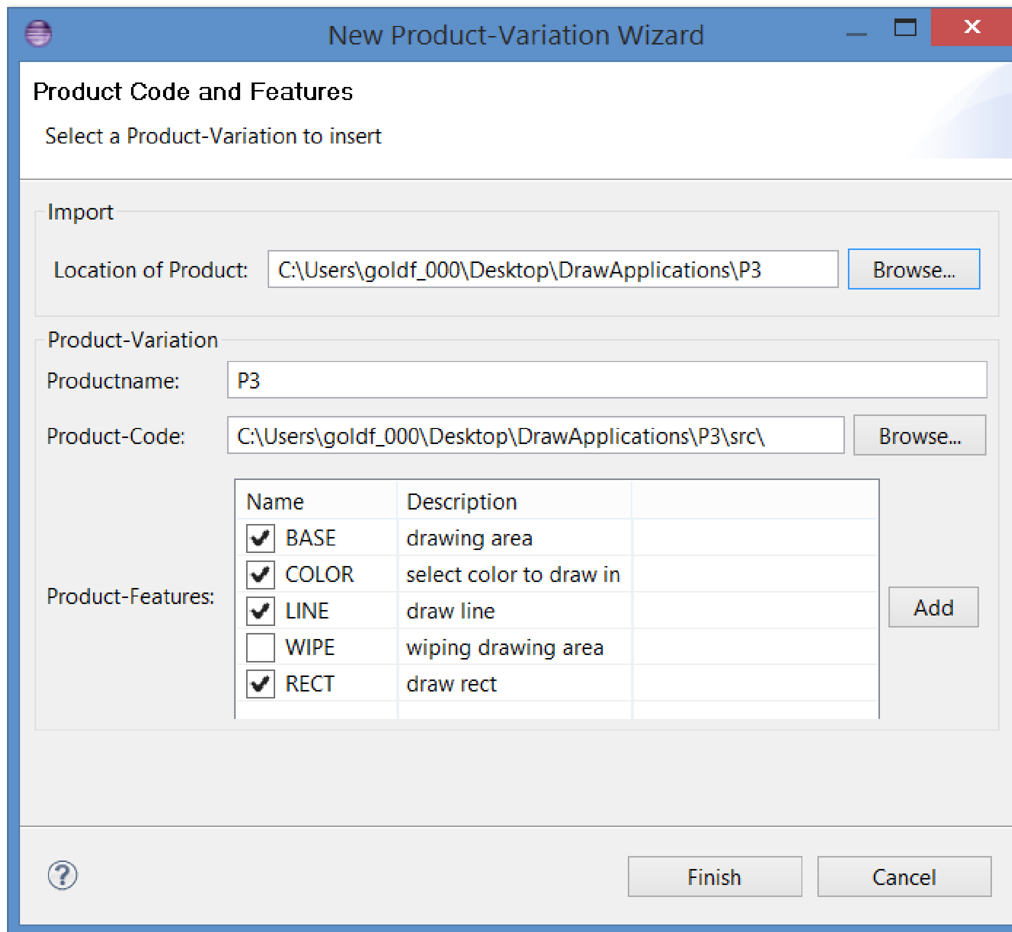



Figure 5.15: Add a Product Variant

implementation artifacts of the product (see Figure 5.17) can be viewed.

Features are presented in a table with their name and description. The implementation artifacts are displayed in their tree-structure.

As discussed in earlier chapters, it is possible to configure the composition. This is also supported by the tool. *Right Click on the Project - Properties - Composition Properties* will present the configuration abilities of the tool, as shown in Figure 5.18.

The first configuring capability allows to limit the order of modules that are displayed by the tool. Therefore higher order modules that most likely have no implementation can be hidden and the displayed information is more clear. The value zero means there is no limit.

Secondly the maximal order of modules for selecting the associations can

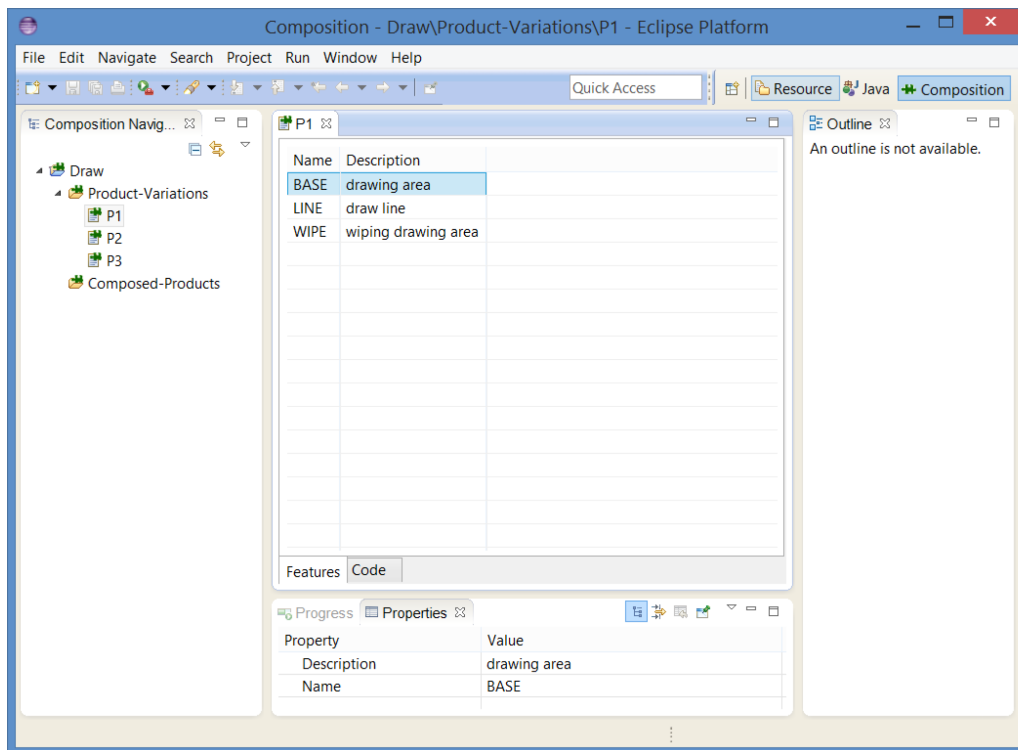



Figure 5.16: Features implemented in a Product Variant

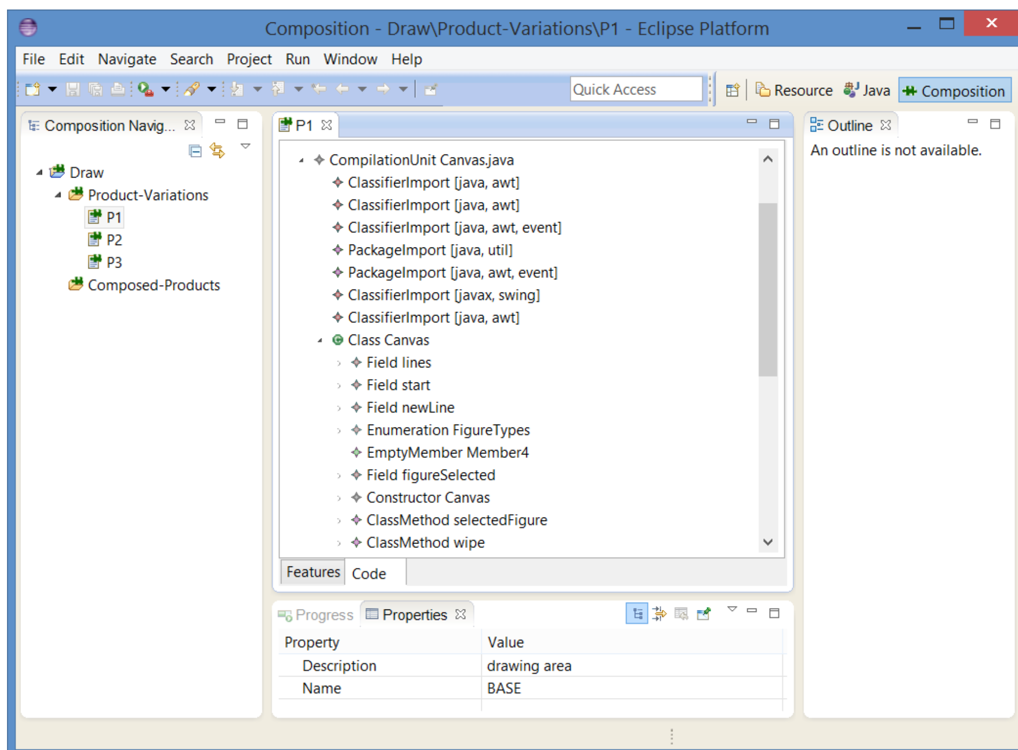



Figure 5.17: Implementation Artifacts of a Product Variant

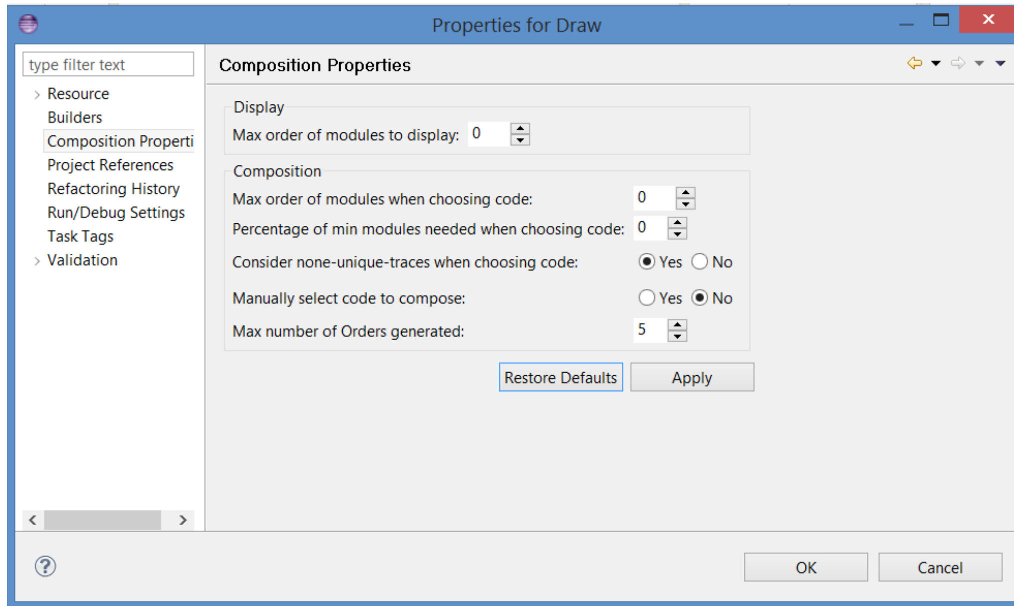



Figure 5.18: Configuring the Tool

be specified. Therefore no higher order modules are considered for the selection of an association. Again zero means there is no maximum.

Next the minimum percentage of modules needed in an association can be configured. This value can be from zero to 100, whereas zero means every association will be selected that has at least one required module. On the other extreme 100 means each of the modules in the association has to be required for composing a product, which is probably a to strict restriction in most cases. Nevertheless a value between these two extrema can be quite useful.

The first of the two boolean values specifies if the composition should take non-unique traces into account, if an association has no unique modules.

It can also be specified if the software engineer wants to see which associations have been selected and manually tweak this selection.

With the last value the user can control the maximal number of orders that are generated during composition. If the software engineer does not want to make any order decisions then he or she can set this value to one. Only one valid order will be determined and selected. Note this is not implicitly the correct order for these nodes in this case.

For the manual refinement of the selected associations there also exists a dialog window (see Figure 5.19). The composition will automatically select the associations as usual, and then lets the user refine the selection.

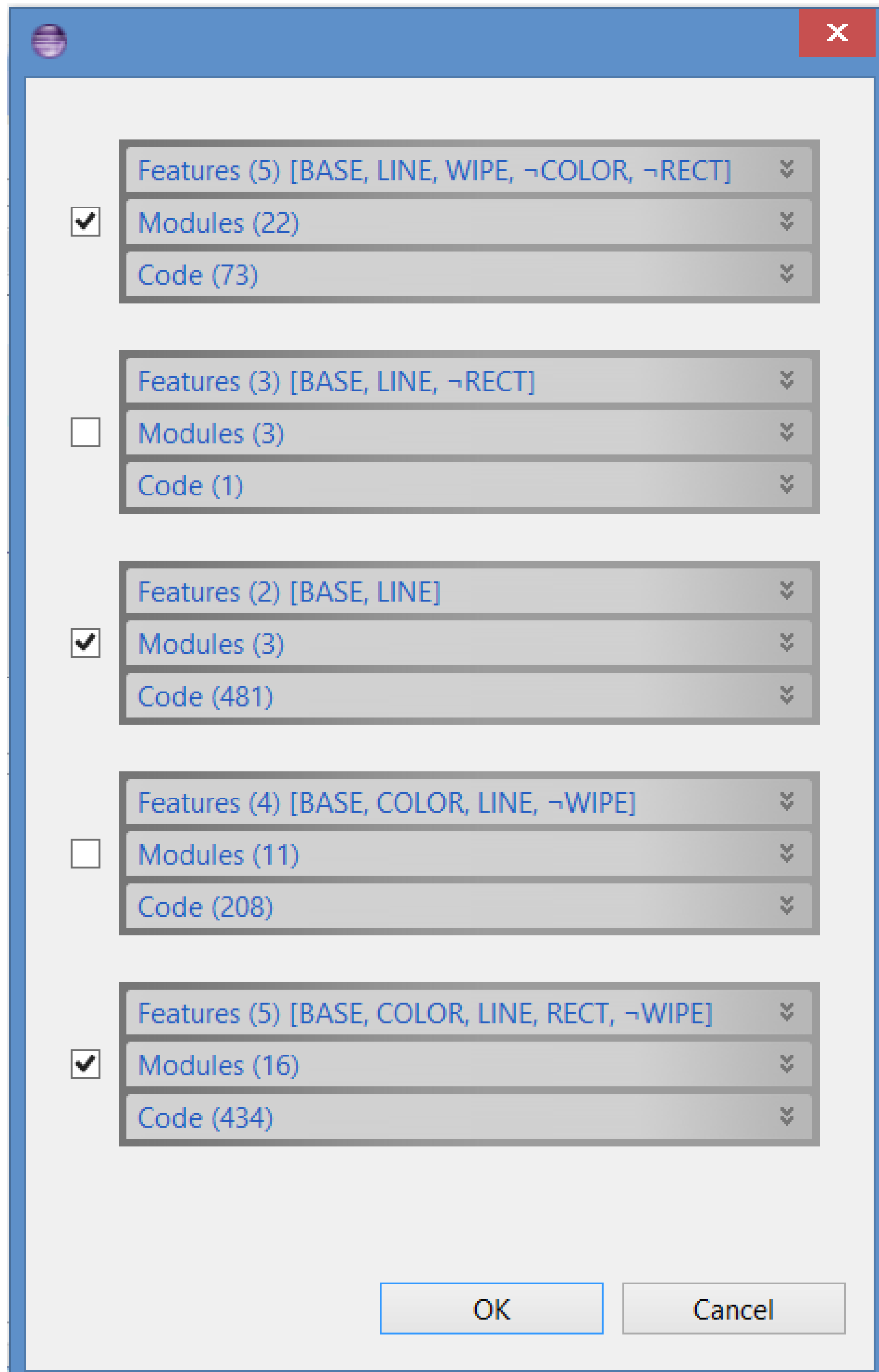



Figure 5.19: Manual select Associations

# Chapter 6

# Implementation

This chapter provides insight in the implementation of this thesis. First we outline the implementation of the data structures used and the composition. Subsequently we will provide an overview of the major implementation parts of the tool support for our approach. And finally we will discuss the parsing and printing of EMF.

## 6.1 Data structure and Core Operations

This section will illustrate the implementation of the core parts of the approach.

### 6.1.1 Features and Modules

Figure 6.1 shows the implementation of features and modules. `Features` are identified by their name, which is a simple Java String, that is used in the methods `equals` and `hashCode`. A `NegativeFeature` extends the class `Feature`. `FeatureSet` and `Module` are simply sets of features. They extend the class `HashSet` from the Java class library, with the generic type `Feature`. Therefore the there implemented set operations can be used for checking if features are contained resp. comparing two modules.

Moreover we can see in Figure 6.1 the `Database` as the core. It provides an

interface for adding products and triggers the intersection for these products base on already calculated associations. Therefore the contained associations get updated with every product added to the database.

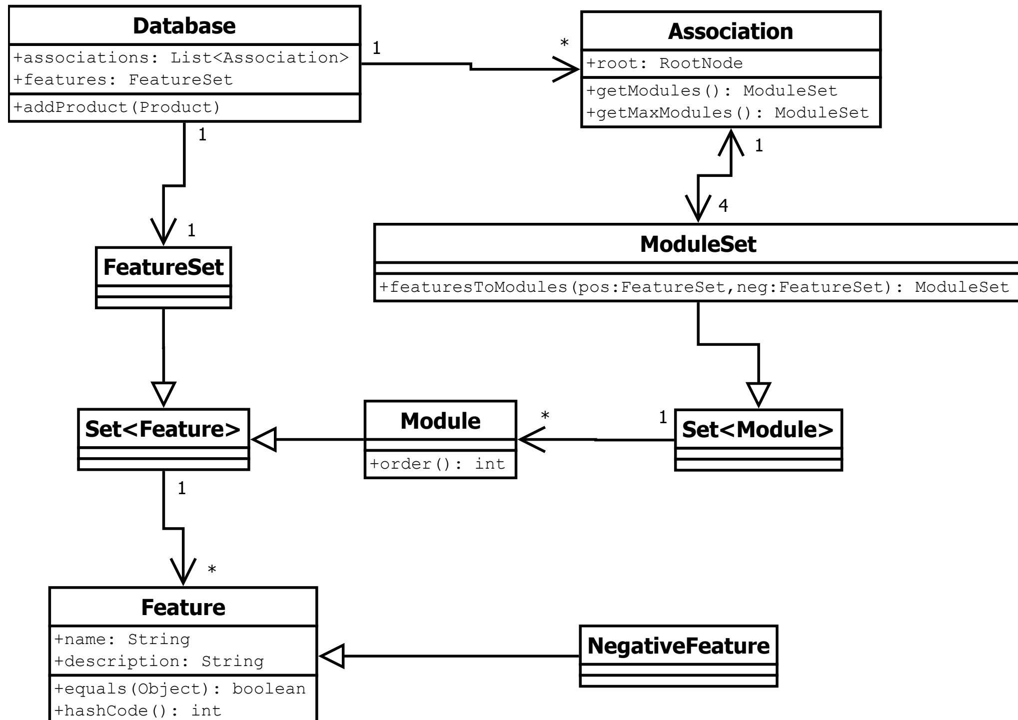



Figure 6.1: UML Class Diagram for Database, Association, Module and Feature

### 6.1.2 Nodes and Artifacts

The implementations of `Artifact` and `Node` are shown in Figure 6.2. The abstract base class `Artifact` provides the general content and interface for every artifact. We implemented three concrete kinds of artifacts:

- `EMFArtifact` for EMF source implementation. This artifact holds an EMF `EObject` which can represent anything from source code to different kinds of models, etc.

- `EMFFieldArtifact` for preserving the structure of EMF sources. This artifact is used to separate children to the different fields they occur in EMF, which is important to be able to reconstruct the EMF sources again from our data structure.

- `JavaStringArtifact` represents Java source code as a Java string (e.g. statement, method signature, ...).

Class `Node` contains the general data structures. It is also an abstract base class for three concrete kinds of nodes:

- `RootNode` is used for associations. It is the root of the tree structure and combines all the other nodes into a single root node. The `RootNode` itself does not contain any implementation artifacts.
- `UnorderedNode` is a simple implementation of `Node` without any special functionality.
- `OrderedNode` overwrites some methods of class `Node`, which are relevant for the extraction. Moreover an `OrderedNode` contains a `SequenceGraph`, since only there the order of their children matters.

Note that all matching `OrderedNode` (i.e. same sequence number, artifact and position in the tree-structure) have the same instance of `SequenceGraph`. Also the extraction ensures that only one of these matching nodes is *unique* (also for `UnorderedNode`). Moreover the artifacts always point to the *unique* node as the containing node. Furthermore the references between the artifacts are resolved correctly during the intersection.

### 6.1.3 Composition

Figure 6.3 depicts the class `Composer`, which implements the composition of a new product respectively the recomposition of a known product variant. It includes the selection of required associations and calculating the corresponding warnings, the merging of the tree-structure, the resolving of references and the deriving of possible orders of nodes where the order matters. As can be seen in Figure 6.3 the `Composer` works with a `Database` where all the relevant informations are stored. The most important method in the interface of class `Composer` is `compose(String,FeatureSet)` which

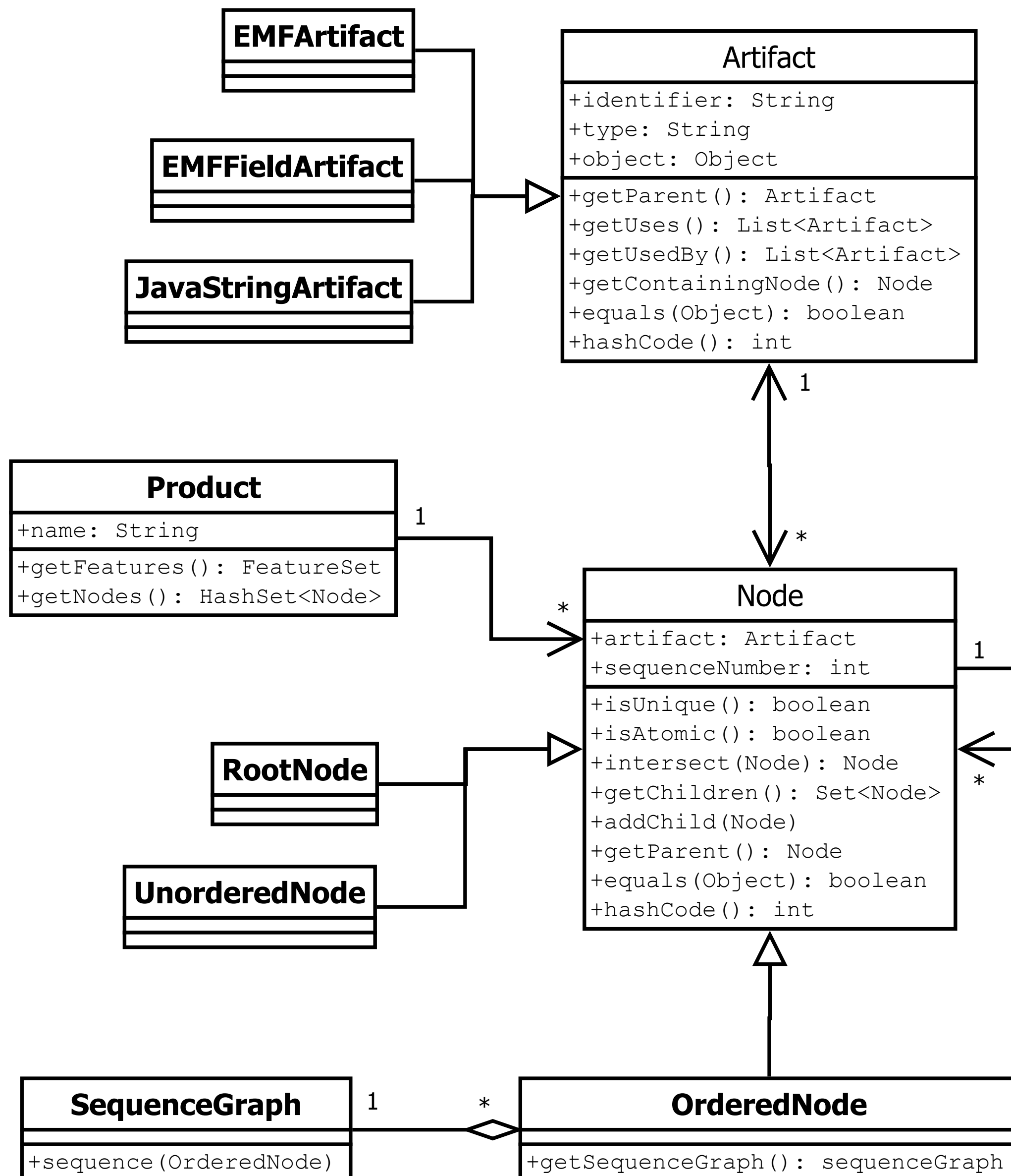



Figure 6.2: UML Class Diagram for Product, Artifact and Node

returns a product with the given string as name and containing the implementation artifacts for the features in the second argument. Furthermore the class also contains a method `compose(Collection<Node>)` which mergers the trees together and returns a list of the rejoined root nodes. This allows to use the composition ability even for nodes which are not associated to any modules.

The configuration of the composition (as described in Chapter 4) is also accessible through this interface. There are methods that set boolean flags

for activating respectively deactivating some behaviors. And also there are different integer values that can be set to fine tune the behavior of the `Composer`.

Furthermore the `Composer` offers an interface for registering different kinds of listeners, so that using programs get the corresponding events while composition.

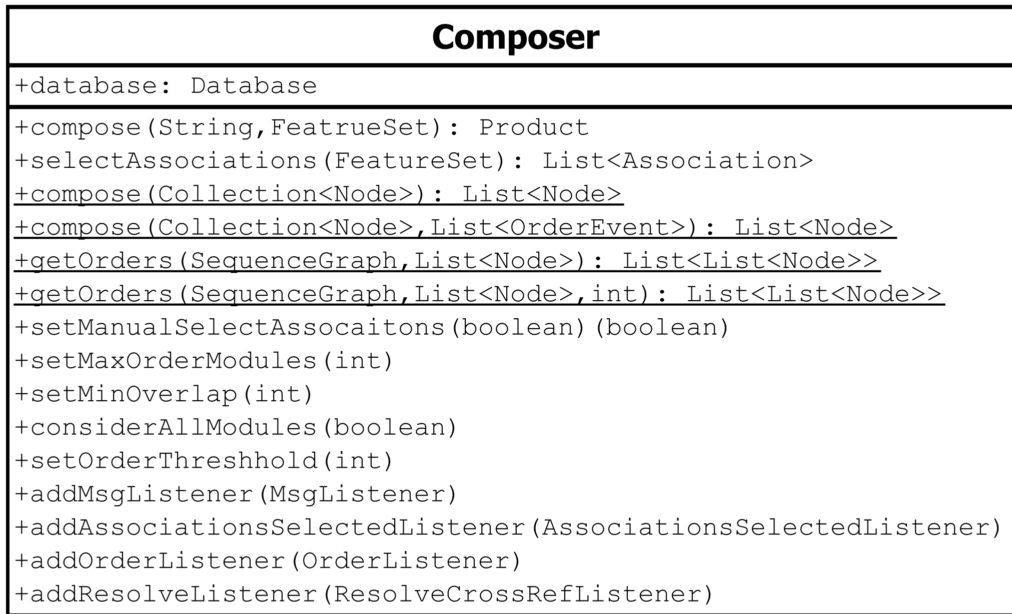



Figure 6.3: UML Class Diagram for Composer

### 6.1.4 Ordering

As discussed before, the valid orders of artifacts (e.g. statements) are stored in the `SequenceGraph`. Yet the `SequenceGraph` is implemented differently from the one illustrated before. We used a partial order relation to describe the order problem, since it is easier to show of the problems with this structure. However in the implementation we found it more suitable to go a different route. Figure 6.4 depicts the sequence graph as it was implemented. Due to spacing issues the statements have been replace according to the legend at the top right corner. The circles are the graph nodes and only serve to provide the structure of the graph. Nodes, with their containing artifacts, are in the transitions between the graph nodes.

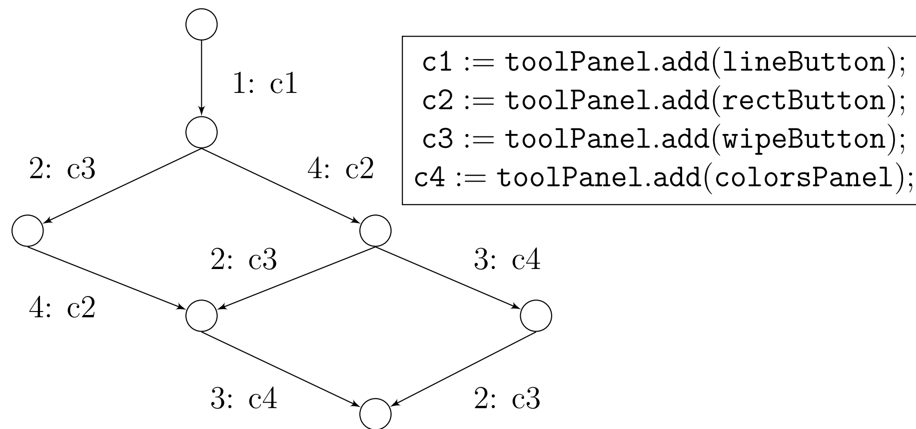



Figure 6.4: Sequence Graph as Implemented

Not only is it easier to align artifacts in this structure, but also the possible orders can easily be found by simply traversing the graph. As can be seen in Figure 6.4, every path through the graph represents a possible order of statements. Therefore the composition can generate all the possible orders by traversing the `SequenceGraph` and remember the orders along the way. However not every node that will be passed in the `SequenceGraph` also has to be in the composed product. For example when composing $\texttt{P}_4$, we do not want to include the statement that provides the color selection (`toolPanel.add(colorsPanel);`). Therefore the algorithm has to skip this node and continue with the next graph node after this skipped transition.

### Alternate Implementation

In an earlier stage of the implementation the extraction did not provide a sequence graph yet. Therefore the artifact order was calculated a little different. There the composition had to look up the different orders that existed in the input product variants.

First the algorithm makes a mapping of which nodes are before other nodes. It distinguishes two mappings there, the first one are nodes that are sometimes before another node and the other ones a nodes that are always before another node, which could indicate a dependency between these nodes. Based on these mappings the algorithm combines the orders of nodes from the input products to the possible orders which contain all

artifacts that occurred in the input sequences somewhere.

```
1 List<List<Node>> combineOrders(List<List<Node>> orders){
2   Map<Node, List<Node>> before = findBefores(orders);
3   Map<Node, List<Node>> alwaysBefore = findAlwaysBefores(
        orders);
4
5   for(0 <= i < orders.size()){
6     List<String> combined = new ArrayList<String>();
7     for(0 <= j < orders.size()){
8       List<Node> order = orders.get((i+j)%orders.size());
9       for(Node n : order){
10        insertOrder(combined, n, before, alwaysBefore);
11      }
12    }
13    if(!combinedOrders.contains(combined)){
14      combinedOrders.add(combined);
15    }
16  }
17
18  return combinedOrders;
19 }
```

Figure 6.5: Alternative Implementation to generate Orders

Figure 6.5 illustrates the algorithm used to find valid orders that contain all artifacts, out of the input orders. The method `findBefores` generates the mapping of nodes that are before a specific other node in at least one product. Further the method `findAlwaysBefores` generates the more precise mapping of nodes that occur before an other node in all input products, both nodes are in. The method `insertOrder` combines these orders based on these mappings, ensuring that the orders according to the `alwaysBefore` mapping are adhered to. Further the mapping `before` is also used to find possible orders. Yet this mapping can have some contradictions and therefore multiple orders might be valid, respectively it can be found that some artifacts do not depend on each other at all and therefore can be put together in any order.

## 6.2 Tool

In this section we will outline the major implementation parts of the tools logic.

### 6.2.1 Project

In Figure 6.6 we can see the most important interface parts of the `CompositionProject` which is the core of the tool support. All the essential parts are started in an instance of this class. Furthermore Figure 6.6 shows the six different worker-threads the `CompositionProject` manages. They are responsible for carrying out the more time consuming parts of our workflow, that would otherwise block the user interface of the tool. Moreover these threads can run concurrently. In particular the thread classes are:

- `ProductParsingThread` is responsible for parsing the artifacts for the different products. Products can consist of different kinds of artifacts, as long as there is a representation registered in the tool which supports theses artifacts.
- `SerializationThread` saves the products, the database and other valuable project data. Therefore the products do only have to be parsed once and are stored in the used data structure afterwards. Also the database does only have to be generated once.
- `DeSerializationThread` reversed the process of the `SerializationThread`. So it loads the products, database and other information again at program start or as soon as they are actually needed. Products are only loaded when they are viewed, therefore they do not use up so much main memory.
- `ExtractionThread` inserts new products into the database. Every time a new product variant is added to the input products this thread

updates the database and initiates an event when the database has changed. Moreover if an input product is deleted it has to regenerate the database with the remaining products, since a delete function in the database is not accounted for.

- `CompositionThread` is triggered when a composition is executed form the software engineer. It generates a `Composer` object with the current database and starts the composition with the desired feature combination as parameter. Furthermore it manages the correct desired configuration of the `Composer` object and ensures the registration of all required listeners.

- `ProductExportThread` is responsible for storing the products artifacts in their initial form, if supported, in a *src* folder. Furthermore it stores the products features as a list in a *features.txt* file, to conform to the tools convention and can easily be parsed again.

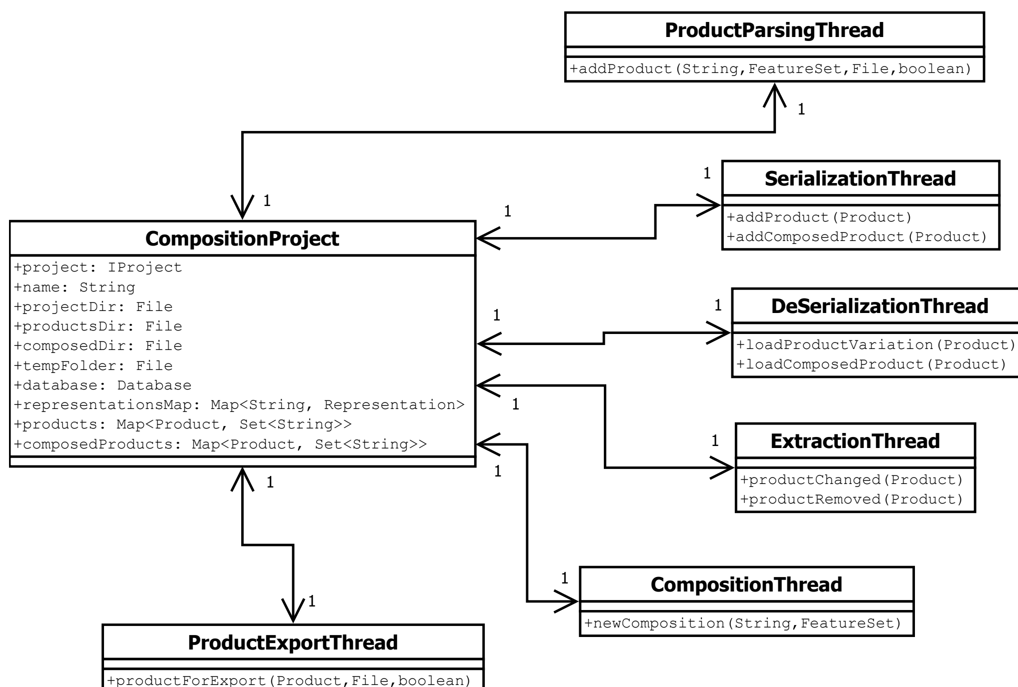



Figure 6.6: UML Class Diagram for `CompositionProject` and its Workload Threads

### 6.2.2 Representation

Figure 6.7 depicts the implementation of representations in the tool. Representations allow programmers to extend the tool with different languages and formats for input artifacts.

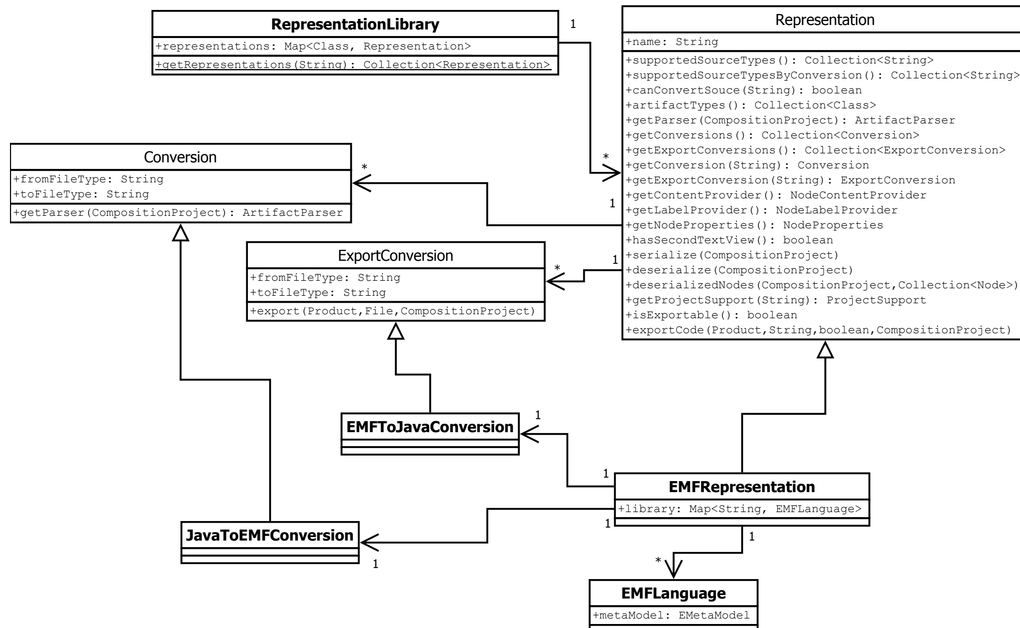



Figure 6.7: UML Class Diagram for Artifact-Representations

In the top left corner of Figure 6.7 the class `RepresentationLibrary` is shown, where all representations have to be registered to, by inserting them into the map `representations` with their used `Artifact` class as a key. Note that a `Representation` can have more than one different artifacts, but an `Artifact` class can only be used for one `Representation`.

The abstract class `Representation` provides interfaces for parsing, printing (export), serializing, de-serializing artifacts of this representation, along with many other useful functionality. In our case we implemented the concrete class `EMFRepresentation` for supporting EMF artifacts. EMF can again represent all kinds of different languages, models, and other implementation artifacts. This means the tool can support basically all languages EMF supports. As also can be seen in Figure 6.7 there is a class `EMFLanguage` which holds a meta model for an EMF based language the

tool can support. This meta model provides structural information of the artifacts, along with the information which parts are important for defining an identifier for a certain artifact.

Moreover the representations can support conversions, which allow an implementer to further extend the capabilities of a representation. The abstract class `Conversion` allows to provide a separate parser for different input sources, that are not available in the used representation natively. By implementing an `Conversion` one can parse these sources and convert them into the intended representation. The inverse of an `Conversion` can be provided by an `ExportConversion`, which will print the artifacts back into their native form. For this thesis we have implemented one of both conversion types. The `JavaToEMFConversion` parses Java code into the `EMFRepresentation` used by the tool. Complementary we are able to print Java code again using the `EMFToJavaConversion`. For both these conversions we use JaMoPP [6].

## 6.3 Parsers

This section explains the implementation of parsers within the tool. In Figure 6.8 we can see the abstract class `ArtifactParser` that provides the interface for parsing arbitrary types of artifacts, and the main two parsers used in the tool and during the evaluation of this work. The individual parsers are provided by the desired and compatible representation.

### 6.3.1 EMF-Parser

This is the main parser the tool uses at the moment, since the respective representation is the only one that provides printing capabilities to generate source files again. The `EMFParser` utilizes the EMF resource handling to parse artifacts. A resource in our case is a source file on the system [5]. The artifacts are represented as `EObject`, which are contained in the generated `EMFArtifact` that are added into the data structure.

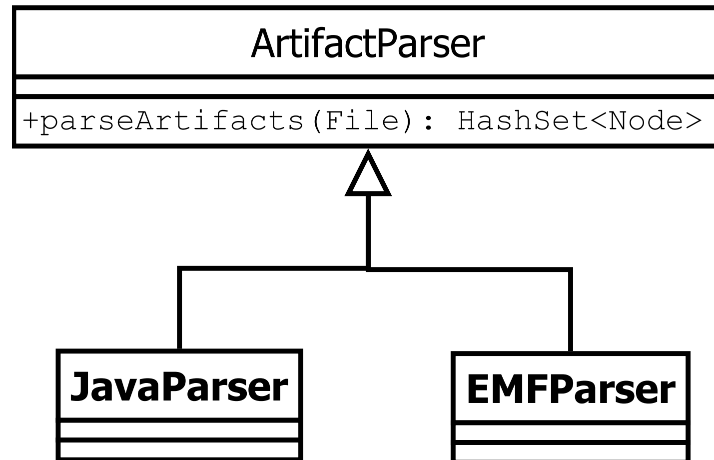



Figure 6.8: UML Class Diagram for Artifact-Parsers

As discussed before we use JaMoPP [6] to convert Java code into the EMF representation. The granularity of the code artifacts is very fine in this case and even goes beyond statement level. So statements are also broken into their bits and pieces down to just references of filed and variables in some expressions.

Furthermore it is not clear how to specify an identifier for all of the different artifacts, possible in EMF. Therefore we need to add a meta model for every `EMFLanguage` we want to add to the tool. According to this meta model we include the type of the `EObject` (e.g. Class) and the attributes it contains along with their attribute values (e.g. name=Main). Therefore we can derive simple identifiers for the artifacts.

However this is still not enough for all kinds of artifacts, therefore we came up with three annotation rules to extend the meta model with to be able to generate a descriptive identifier for every `EMFArtifact`.

- *id:* This rule can annotate any reference from one `EObject` to another, containing or cross-reference. It indicates objects that need to be included in the identifier of an artifact. E.g. Java class methods have their parameters as child artifacts, but we need them in the identifier of the method artifact, therefore we need to add an id-rule to the parameters reference.

- *atomic:* Sometimes we may want to individually control the level of granularity the artifacts are broken up to. E.g. in Java we want to stop at the statement level, therefore we add an atomic-rule to the expression statement in the meta model and indicate that all the child artifacts should be considered in the identifier of the atomic artifact. An atomic artifact also will set the atomic flag of int containing node, so the extraction and composition will know to treat them in a special way (see Chapter 4).

- *ignore:* Not all the attributes of an `EObject` are important, so we want to be able to ignore some of them and not include them to the identifier of the artifact. E.g. in Java comments are attributes of an `EObject`, but these do not change the implementation of the artifacts, therefore we want to exclude them form the identifier.

These rules are all structures the same. They are contained in an `EAnnotation` object, named *rules* withing the type or reference. The rules for this special type of `EObject` are entries within these annotations with the name of the rule as key and as value is the string true if the rule should apply here. Therefore rules can be enabled and disabled easily by changing their value. An example for an atomic rule can be seen in Figure 6.9, for the case of Java expression statements.

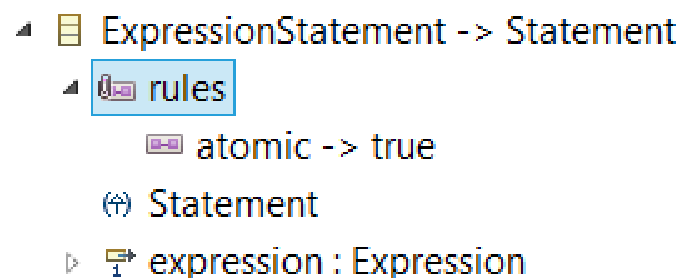



Figure 6.9: Atomic Rule for Java Expression Statements

### 6.3.2 Java-Parser

The `JavaParser` uses the Java Compiler API [7] to parse Java source code into an abstract syntax tree and wrap it into our used data structure for artifacts. This is the parser used for the evaluation of this thesis (see Chapter 7). The artifacts created by this parser are of the type `JavaStringArtifact` that represents code as a string and go down to the statement level in granularity. Since we do not handle the references in the evaluation and leave them unresolved anyway, the `JavaParser` does not even produce any references, that would arise from the code, for simplicity.

## 6.4 Printer

In this section we highlight the printing of artifacts, back into the source representation, or even into a different compatible representation. The printing is only implemented for the `EMFRepresentation` at the moment and is accessible through the *export* interface of the `Representation` class. This will be executed if a product is exported, respectively moved into an independent project, in the tool support. For exporting the `EMFRepresentation` the tool uses the EMF resource handling, like in the parser, but in the different way. Here the tool needs to create new resources and fill them with the corresponding `EObject` instances in the product. Afterwards these resources can be saved using their provided (from EMF) API. As discussed before the tool can convert the `EMFRepresentation` of Java code into plain Java code again, using JaMoPP [6]. Therefore it is also possible to print Java code and get a products files as usually intended.

# Chapter 7

# Evaluation

This chapter evaluates the implemented approach using 4 case studies. The focus of the evaluation will be the composition, since it is the main contribution of this thesis. We will begin with introducing the case studies, followed bu the description of our evaluation scheme and the calculated metrics. At the end we will conclude the chapter with the analysis of the result.

## 7.1 Case Studies

The evaluation was performed using the 4 case studies shown in Table 7.1. All these case studies are implemented in Java and in the following subsection we will explain them in more detail.

| **Case-Study** | **#F** | **#P** | **LoC** | **#Art** |
|---|---|---|---|---|
| Draw | 5 | 12 | 287 - 473 | 491 |
| VOD | 11 | 32 | 4.7K - 5.2K | 5.5K+ |
| ArgoUML | 11 | 256 | 264K - 344K | 192K+ |
| ModelAnalyzer | 13 | 5 | 35K - 59K | 94K+ |

#F: Number of Features, #P: Number of Products, LoC: Range of Lines of Code, #Art: Number of Distinct Artifacts

Table 7.1: Case Studies Overview

The products of the first 3 case studies where derived from SPLs. Therefore they represent a quite ideal case for our approach. Nevertheless they are useful for the evaluation of the approach, since the products have no

evolutionary changes and did not diverge from each other, just as we would expect products to be if they were implemented using our approach from the beginning.

The last case study *ModelAnalyzer* represents the worst case for our approach. Its product variants have been developed by different engineers independently over the course of many years. Therefore the variants diverged a lot from each other. for example bug fixes have been applied to some products, but not all of them. This makes it extremely difficult for the extraction to achieve a good result, which also strongly affects the composition of new products.

### 7.1.1 Draw

The *Draw* case study is an SPL of simple drawing applications. Some of its member products where used as examples in previous chapters. Its feature model, which allows for 12 different products, supporting up to 5 features, is depicted in Figure 7.1.

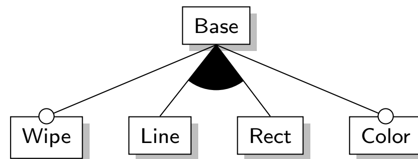



Figure 7.1: Feature-Model for Draw

### 7.1.2 VOD

*VOD* is an SPL for video-on-demand streaming applications. It supports 11 features of which 6 appear in every variant. The feature model is shown in Figure 7.2. It consists of 32 member products.

### 7.1.3 ArgoUML

With 256 possible product variants (see Figure 7.3), *ArgoUML* is the largest of our case studies. It has 11 features, of which 3 appear in every product.

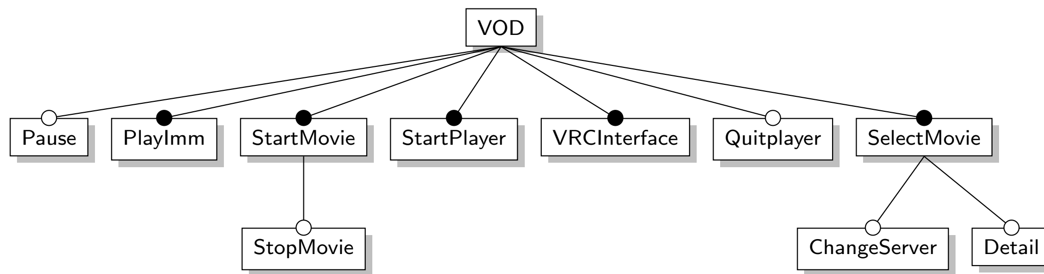



Figure 7.2: Feature-Model for VOD

*ArgoUML* is an open source UML modeling tool that was refactored into an SPL [8].

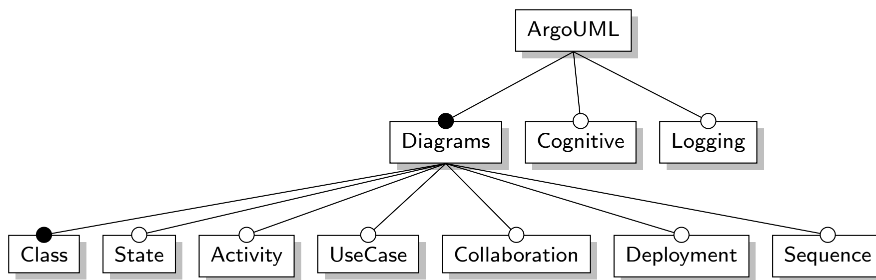



Figure 7.3: Feature-Model for ArgoUML

### 7.1.4 Model Analyzer

The last case study *ModelAnalyzer* is a consistency checking and repair technology. There is no feature model for this case study, because it is not an SPL. Rather it consist of 5 product variants that where developed through copying and adapting by different engineers pursuing their own separate goals.

We where able to identify 13 features, through interviews with the developers. There where some difficulties because some developers partially copies feature implementations and never completed them, because they did not use them anyway. Also different developers used different names for features. Therefore common a naming had to be established first. Please note that no prior cleanup of the variants took place.

## 7.2 Evaluation Scheme

For the evaluation of the approach, it was automatically executed with different numbers of input products using the case studies mentioned above.

As an initial run the evaluation uses all the existing product variants in a product portfolio to create the database, recomposes all of them and computes the metrics to show that the extracted information is sufficient to fully re-engineer all input products. Subsequently the number of product variants used as input for the creation of the database is decreased from 100% to 0% in steps by removing products randomly. Next we generate these products that were not among the input products using the composition procedure and compare them to the respective original product of the portfolio (i.e. its oracle product). This allows us to draw a conclusion on the quality of newly generated products that were not used as input. The quality of the composed products depends not only on the number of input product variants, but also on the features they implement. Therefore for every decreasing step we perform 10 runs for the current number of input products, where we pick the input products randomly. We compute the composition metrics for the recomposed products for every selection of input products and then calculate the average values.

All of our here used case studies consist of Java source code. For performing this evaluation we used the Java parser (see Chapter 6). All placeholder nodes where included in the composition, to fulfill the structural dependencies of the data structure (i.e. include a parent node when at least one of its child nodes is included). Moreover all use references to not included artifacts where left unresolved. The order of the artifacts was considered correct, if the order in the oracle product was included among the possible orders to choose from. This means the software engineer would have gotten the correct order as an option to select, which was always the case in our case studies.

## 7.3 Evaluation Metrics

### 7.3.1 Composition Runtime

We measured the runtime of each composition performed, not included parsing of the input products and extraction of the database (i.e. the creation of a composer object and starting the composition with a set of features as a parameter). Therefore this metric tells us exactly how long the entire composition procedure takes. For each number of input products we computed the average runtime for composing the products, as depicted in Figure 7.4. As we can see there the runtime is mostly affected by the size of the product since there are more artifacts to be restructured. Furthermore the number of features to be composed and the size of the database influence the runtime of the composition.

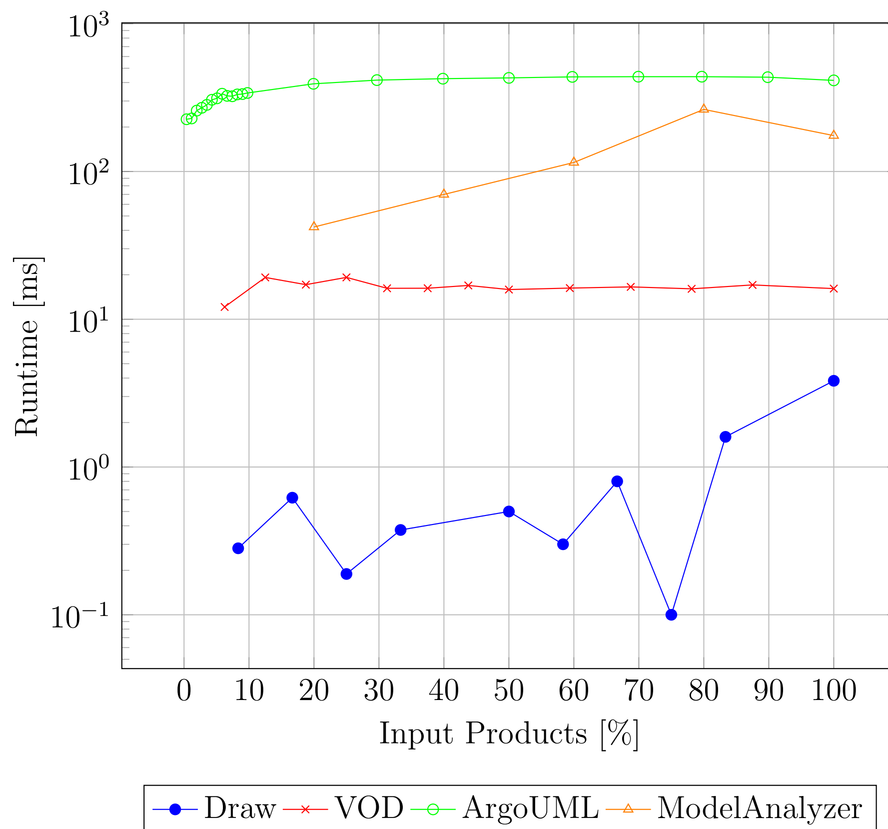



Figure 7.4: Runtime Overview

For more detail Table 7.2 shows the complete runtime the composition

took for the initial run, where all product variants have been re-composed, for each case study. These values again do not contain any other parts of the workflow like parsing or extraction and also exclude the time the evaluation took for comparing the products. In Table 7.2 the number of product variants and their implementation size are the most influencing the runtime.

| **Case-Study** | **Draw** | **VOD** | **ArgoUML** | **ModelAnalyzer** |
|---|---|---|---|---|
| **Runtime** | 46ms | 516ms | 1min45.9sec | 875ms |

Table 7.2: Composition Runtime for Initial Run

The evaluation was performed on the following system: Intel Core i7-4770 @3.40GHz Haswell, 16GB of Memory, 64Bit environment.

### 7.3.2 Correctness

The correctness expresses the quality of the composed product. It reveals if there are surplus artifacts and missing artifacts that where known to the composition (i.e. contained in the database). Missing artifacts that are not contained in the database, therefore have not been in any input product, do not affect this metric.

This metric it computed as the relation of the number of artifacts that the original product p and the composed product p′ have in common to the number of artifacts that both products in their union that are also known to the database *DB*. We use *A.artifacts* to denote a set of artifacts contained in the respective product or database *A*.

$$Correctness\ [\%]\text{: } \frac{commonCode}{allCode - missingCodeDB} \times 100$$
$$\text{where } commonCode = |p.artifacts \cap p'.artifacts| \text{ and}$$
$$allCode = |p.artifacts \cup p'.artifacts| \text{ and}$$
$$missingCodeDB = |p.artifacts \setminus DB.artifacts|$$

Figure 7.5 shows the correctness with respect to the number of used input products. We can observe that the correctness rises quickly with the number

of input products, as we would expect since the database gets more precise the more input products are used to calculate it. However we can also see that the composition already achieves a high correctness even with just a fraction of the products (from 20% upwards) as input. The exception to this is the *ModelAnalyzer* case study where the correctness drops if we use more products as input. This is due to the fact that the product variant only share a small portion of their implementation artifacts, which leads to many surplus artifacts, that could not be separated from the few actually needed artifacts gained by the new input product. Furthermore the limited number of product variant for this case study makes it difficult to extract more precise information. Nevertheless we still where able to achieve a correctness between 40-60%.

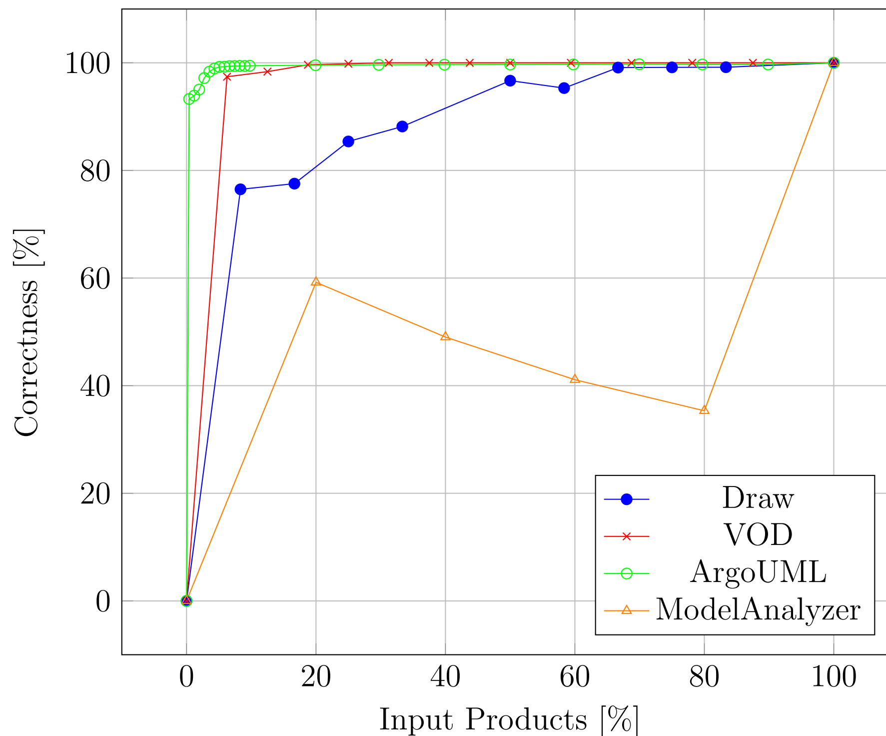



Figure 7.5: Correctness Overview

### 7.3.3 Completeness

Completeness is the percentage of artifacts from the oracle product p also found in the corresponding composed product p′.

$$Completeness\ [\%]\text{: } \frac{commonCode}{|p.artifacts|} \times 100$$

Complementary to the correctness, the completeness reveals whether there were artifacts missing in the composed product, even if these artifacts did not occur in the input products. Meaning the composition worked optimal if both these measures, the correctness and the completeness, are 100%.

In Figure 7.6 we can see that the completeness is rising to 100% even faster than the correctness. The cases in which the completeness is already at its maximum but the correctness is not, imply the composition put extra artifacts in the product, because the extraction could not trace all artifacts to a single module. For our first three case studies the results in completeness are very similar to the correctness and reach a near optimum already at around 20%. Again results achieved for the *ModelAnalyzer* case study are below the other ones, due to its strongly diverged product variants. Yet the completeness between 60-70% is still promising.

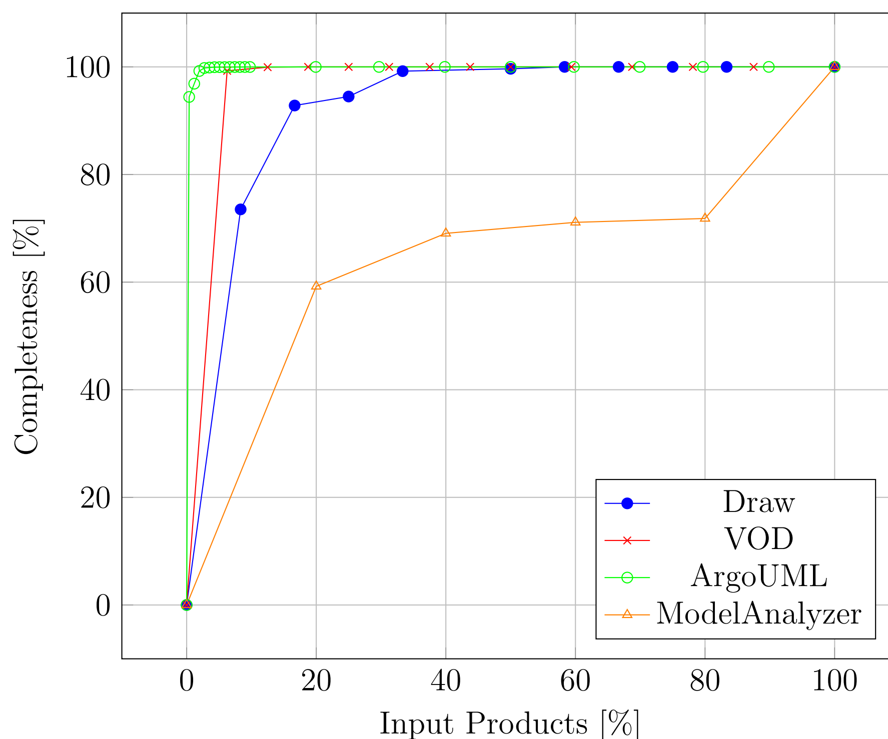



Figure 7.6: Completeness Overview

### 7.3.4 Warnings

The warnings generated during the composition are important for the software engineer to complete the composed product variants. Therefore we measured the number of warnings generated for each composition, shown in Figure 7.7 and Figure 7.8.

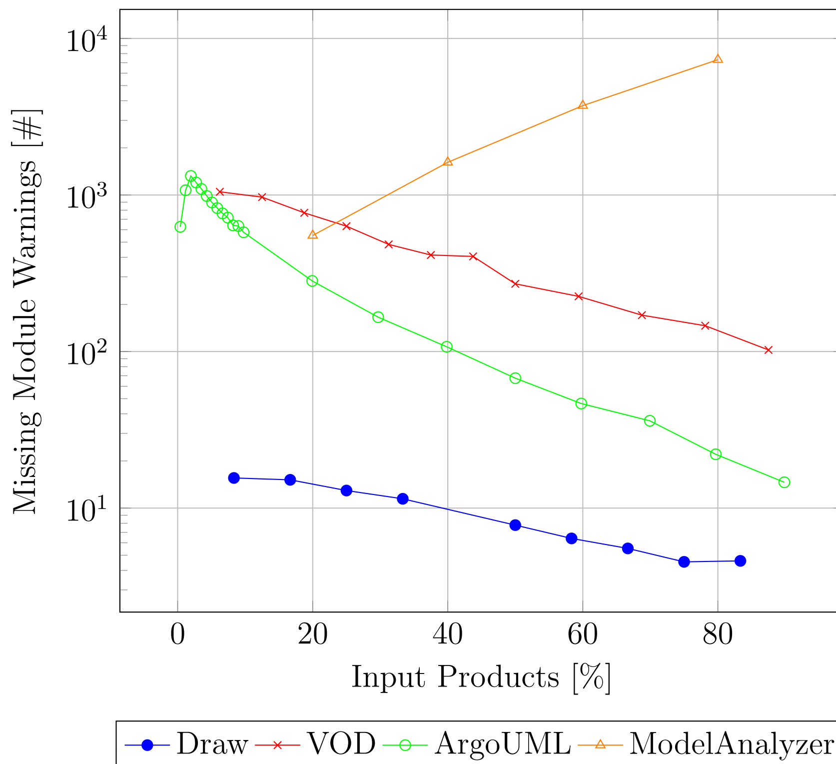



Figure 7.7: Missing Modules Overview

As we can see, after a short initialization, the number of warnings goes down exponentially with the number of input products, which correlates to the reading for correctness and completeness, since we should have less warnings the closer we are to the perfect composition. In the case of the *ModelAnalyzer* the number of warnings also correlate to the correctness, therefore we get more warnings with more input products since the variants are too far diverged in this case.

Furthermore we measured the warnings received during composition in respect of the order of the modules in the warnings. In Figure 7.9 we can see the warnings in average for each composition performed during the

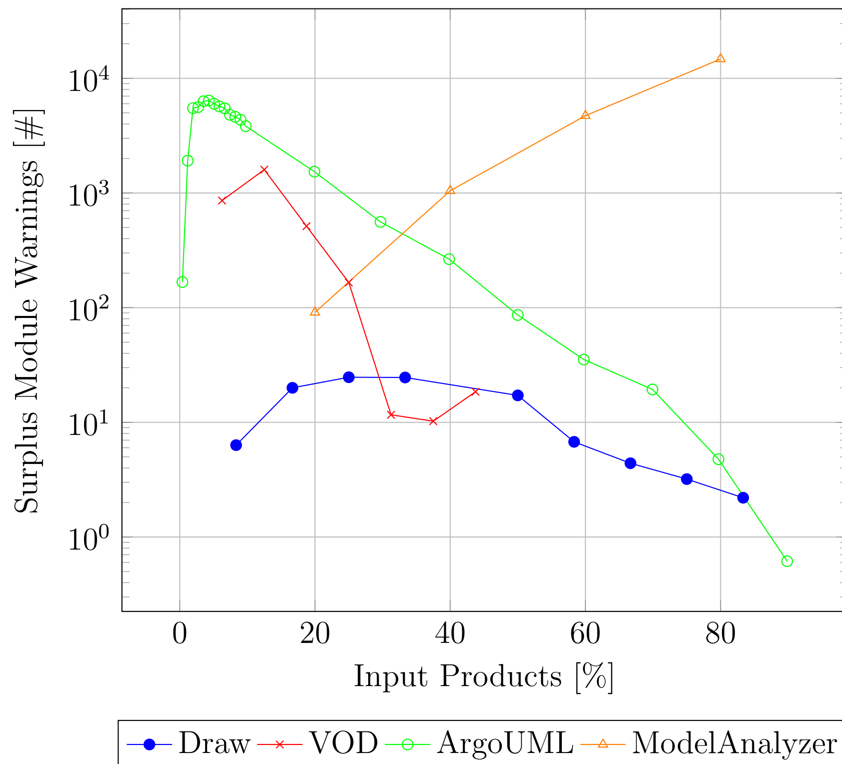



Figure 7.8: Surplus Modules Overview

evaluation, grouped by the order of their modules.

We can see that the behavior of the module warnings per order is very similar for all case studies. The possible orders of modules depend on the number of features in the product portfolio. So in the center of the graphs, there are the most warnings in the mid range of the possible orders of modules. Base module warnings did occur less often since the variants usually have the most features in common and also the features can be distinguished quite nice. For the *ModelAnalyzer* there are more base module warnings, since a lot of features are only implemented by single variants and moreover it is not possible to distinguish all the features, due to the low number of product variants. We can assume that the higher the order of modules, the less likelier their are any implementation artifacts associated with them. Therefore we can consider most of the higher order warnings as false positive warnings. That is the reason the tool allows to set a threshold for the modules, to filter out these less important warnings.

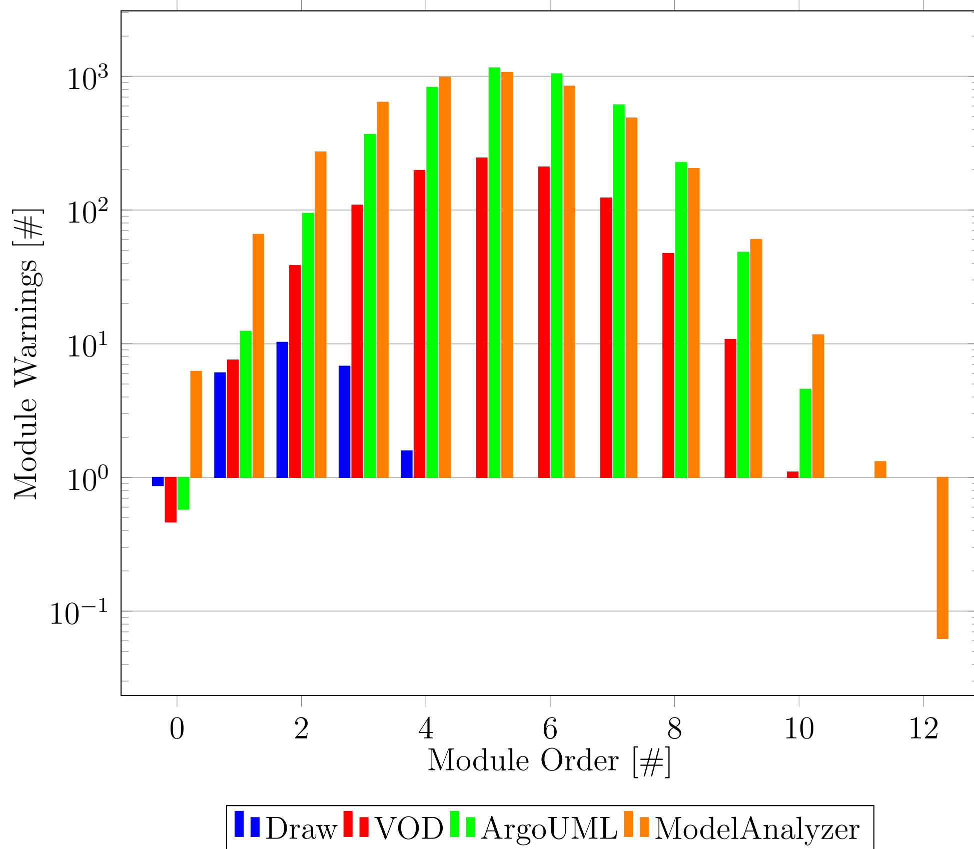



Figure 7.9: Warnings per Order

## 7.4 Analysis

In summary the results of the evaluation look very promising. The runtime for the composition is somewhere in the millisecond range and even for the fairly large products of *ArgoUML* the composition of a product took less than a second. Further the quality of the composed products looks very promising as well. The correctness and completeness get higher the more input products are provided, yet already achieve great results with just a fraction of the product portfolio as input. Only the *ModelAnalyzer* case study did not achieve such great, albeit still promising, results due to the strong diverged input products in combination with the low number of products available. What was learned from these results is also reflected in the warnings the composition generates. Therefore the better the quality of the composed products was, the less warnings have been issued, which

is of course the way it is intended. It makes sense to introduce a threshold for the maximum order of modules in the warnings to reduce them. This is valid since most higher order modules do not contain any implementation artifacts anyway.

# Chapter 8

# Threats to Validity

The extraction algorithm used in the presented approach, that lies behind the tool support, assumes that the product variants still have major portions of their implementation in common. However, if the approach should be used to extract trace information from products that where implemented more independent from each other, these products will not share the same code. This problem will also arise if evolutionary changes to the product variant (e.g. bug fixes) are not propagated through the entire product portfolio. In all cases where the product variants diverged too much during their development or maintenance the core assumption of the extraction algorithm will no longer hold. Nevertheless if the approach is used form the very beginning of developing a product portfolio these problems will not arise, since the variants will not diverge from each other. In this case the approach will find almost ideal conditions and its benefits can be fully leveraged. Moreover there are ways to better cope with this threat (e.g. code-clone detection techniques) to also recognize similarities in diverged products.

# Chapter 9

# Related Work

Xue et al. use diffing algorithms to identify the common and variable parts of product variants, which are subsequently partitioned using Formal Concept Analysis [23]. To these partitions, Information Retrieval algorithms are applied to identify the code units specific to a feature. In contrast with the proposed work, they do not explicitly distinguish the code units that stem from single features from those of feature interactions. However, we will explore how to leverage advanced diffing techniques employed in this work for detecting a wider spectrum of software artifact changes.

Rubin et al. propose a framework for managing product variants that are the result of clone-and-own practices [21]. They outline a series of operators and how they were applied in three industrial case studies. These operators serve to provide a more formal footing to describe the set of processes and activities that were carried out to manage the software variants in the different scenarios encountered in the case studies.

Koschke et al. aim to reconstruct the module view of product variants and establish a mapping of code entities to architecture entities, with the goal of consolidating software product variants into software product lines by inferring the software product line architecture [13]. For this they adapt the reflection method by applying it incrementally to a set of variants taking advantage of commonalities in their code, for which they use clone detection

and function similarity measures.

In [20] Rubin et al. survey feature location techniques for mapping features to their implementing software artifacts. The extraction process in our work can also be categorized as a feature location technique, only that we also consider additional problems like feature interactions instead of just single features and also the order of artifacts instead of just their presence or absence. Another feature location survey exists by Dit et al. [11]. Other traceability and information mining algorithms are presented by Ali et al. in [1], by Capobianco et al. in [9] or by Kagdi et al. in [19]. Indeed Ghezzi et al. argue that there could be additional information relevant for extraction and composition [18].

When making changes or bug fixes to features using our approach similar advantages can be leveraged as presented by Beek et al. in [22] when it comes to testing, like only having to test certain product variants and not all.

Chen et al. [10] present a way of displaying traceability links which could be used to visualize the traceability information extracted by our approach.

Nguyen et al. present JSync [17], a tool for managing clones in software systems. Techniques like these could be useful for us when recovering legacy product variants that have diverged significantly.

# Chapter 10

# Future Work

In our future work we want to extend the approach with a few technological additions. For example we want to reduce the number of possible order by detecting and eliminating orders that are semantically the same. Furthermore we want to refine the orders by removing statement sequences that do not make much sense (like assigning the same variable twice without reading it in between).

The composition could be enhanced by providing some rules, maybe in form of a meta model, to be able to make more intelligent decisions. E.g. in Java a method gets included because on of its statements traces to a selected feature. To be syntactically correct also the methods returns type and parameter have to be added, which may trace to a not selected association and therefore would not be included in the composed product. One could define rules to always include the return type and the parameters with their method.

Also the rules added to the EMF meta models could benefit from some refinement. So that they no longer can just we true or false, but rather point to elements that do or do not belong into the identifier. E.g. for Java methods we want their parameters in the identifier, but not the parameters names, since they do not belong to the methods signature. With the current implementation it is not possible to define this.

Lastly we want to extend the tool with the ability to be able to edit the database. Therefore software engineers would be able to manually refine the trace information, to enable the composition to provide better results. Moreover it should be possible to make changes to the code in the database, to maintain the contained features. This would enable the user to recompose all the products which implement the changed feature and not manually have to fix each product individually.

# Chapter 11

# Conclusion

We presented an composition algorithm along with a tool supporting software engineers in the compositions of products base on selecting features. Moreover the tool includes an extraction algorithm (implemented by [4]) to provide the necessary trace information to be able to compose code based on the desired features. The composition does not only consider traced code lines and their position, but also dependencies between implementation artifacts and their order. Further the concept is highly generic and can be used to compose any kind of artifacts, as long as the necessary parser and printer are provided.

During the evaluation we saw that the quality of the composed products is very high with already a small number of input products and that all the input products always could be reconstructed perfectly.

A overview of the implemented tool support was given, which covered all the major parts needed to utilize the tool. The approach lying behind this tool was also discussed in this thesis to clarify the operating principles used in the tool support.

# List of Figures


2.1 Product $P_3$ . . . 11
2.2 Source Code Snippets for the initial Drawing Applications . 15
2.3 Source Code Snippets for Product 4 . . . 16
2.4 Source Code Snippets for Product 5 . . . 16

3.1 Overview . . . 18
3.2 Associations extracted from our Drawing Applications . . . . 22

4.1 Data structure for Product $P_1$ [4] . . . 24
4.2 Sequence graph derived from Products $P_1$, $P_2$, $P_3$ . . . 26
4.3 Intersecting modules of $P_1$ and $P_2$ . . . 28
4.4 Intersecting code of $P_1$ and $P_2$ . . . 29
4.5 Sequence graph derived from Products $P_1$, $P_2$, $P_3$ . . . 31
4.6 Composition Steps . . . 33
4.7 Desired modules for composing $P_4$ . . . 33
4.8 Selected Associations for composing $P_4$ . . . 35
4.9 Modules of the selected associations . . . 35
4.10 Calculating *Missing* & *Surplus Modules* . . . 36
4.11 Merging Trees for composing $P_4$ . . . 37
4.12 Resolved references in product $P_4$ . . . 39
4.13 Possible orders of Buttons in $P_4$ . . . 40

5.1 Create new Project . . . 42
5.2 Define Name and Location . . . 42
5.3 Select input Product Variants . . . 43
5.4 Choose a Representation for Java files . . . 44
5.5 Parsed Product Variants . . . 44
5.6 Database View . . . 45
5.7 Compose Product $P_4$ . . . 46
5.8 Missing Modules . . . 46
5.9 Surplus Modules . . . 47
5.10 Resolve Use References . . . 48
5.11 Choose Order of Statements . . . 49
5.12 Composed Product $P_4$ . . . 49
5.13 Java Project for Product $P_4$ . . . 51
5.14 Locate Feature WIPE in the Code . . . 52
5.15 Add a Product Variant . . . 53

5.16 Features implemented in a Product Variant . . . . . . . . . . 54
5.17 Implementation Artifacts of a Product Variant . . . . . . . . 54
5.18 Configuring the Tool . . . . . . . . . . . . . . . . . . . . . . 55
5.19 Manual select Associations . . . . . . . . . . . . . . . . . . . 56

6.1 UML Class Diagram for Database, Association, Module and Feature . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.2 UML Class Diagram for Product, Artifact and Node . . . . 60
6.3 UML Class Diagram for Composer . . . . . . . . . . . . . . 61
6.4 Sequence Graph as Implemented . . . . . . . . . . . . . . . . 62
6.5 Alternative Implementation to generate Orders . . . . . . . 63
6.6 UML Class Diagram for `CompositionProject` and its Workload Threads . . . . . . . . . . . . . . . . . . . . . . . . . 65
6.7 UML Class Diagram for Artifact-Representations . . . . . . 66
6.8 UML Class Diagram for Artifact-Parsers . . . . . . . . . . . 68
6.9 Atomic Rule for Java Expression Statements . . . . . . . . . 69

7.1 Feature-Model for Draw . . . . . . . . . . . . . . . . . . . . . 72
7.2 Feature-Model for VOD . . . . . . . . . . . . . . . . . . . . . 73
7.3 Feature-Model for ArgoUML . . . . . . . . . . . . . . . . . . 73
7.4 Runtime Overview . . . . . . . . . . . . . . . . . . . . . . . . 75
7.5 Correctness Overview . . . . . . . . . . . . . . . . . . . . . . 77
7.6 Completeness Overview . . . . . . . . . . . . . . . . . . . . . 78
7.7 Missing Modules Overview . . . . . . . . . . . . . . . . . . . 79
7.8 Surplus Modules Overview . . . . . . . . . . . . . . . . . . . 80
7.9 Warnings per Order . . . . . . . . . . . . . . . . . . . . . . . 81

# List of Tables


2.1 Initial Drawing Application Product Variants . . . . . . . . . 10
2.2 New Drawing Product Variants . . . . . . . . . . . . . . . . . 12

7.1 Case Studies Overview . . . . . . . . . . . . . . . . . . . . . . 71
7.2 Composition Runtime for Initial Run . . . . . . . . . . . . . . 76

# Bibliography

[1] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol. Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. *IEEE Trans. Software Eng.*, 39(5):725–741, 2013.

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[3] I. Sommerville. Software engineering. *Pearson Education, Inc.*, 9th edition, 2011.

[4] L. Linsbauer. Reverse Engineering Variability from Product Variants. *In Master's Thesis, Johannes Kepler University Linz.*, September, 2013.

[5] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. EMF: Eclipse Modeling Framework. *Addison-Wesley Professional*, 2nd edition, December 16, 2008.

[6] JaMoPP (2013). JaMoPP. http://www.jamopp.org/ (accessed 2013).

[7] JCAPI (2013). Source code analysis using java 6 apis. http://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html. (accessed 2013).

[8] ArgoUML (2013). Argouml-spl project. http://argouml-spl.tigris.org/ (accessed 2013).

[9] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella. Improving ir-based traceability recovery via noun-based indexing of software artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013.

[10] X. Chen, J. G. Hosking, and J. Grundy. Visualizing traceability links between source code and documentation. In M. Erwig, G. Stapleton, and G. Costagliola, editors, *VL/HCC*, pages 119–126. IEEE, 2012.

[11] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[12] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, 2013. Best Paper Award.

[13] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17(4):331–366, 2009.

[14] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Recovering traceability between features and code in product variants. In *accepted for publication at the Sevententh International Software Product Line Conference (SPLC)*, 2013.

[15] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proc. of 28th int. conf. on Software engineering*, ICSE '06, pages 112–121, New York, NY, USA, 2006. ACM.

[16] T. Mende, F. Beckwermert, R. Koschke, and G. Meier. Supporting the grow-and-prune model in software product lines evolution using clone detection. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 163–172, 2008.

[17] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Trans. Software Eng.*, 38(5):1008–1026, 2012.

[18] G. Ghezzi and H. C. Gall. A framework for semi-automated software evolution analysis composition. *Autom. Softw. Eng.*, 20(3):463–496, 2013.

[19] H. H. Kagdi, M. Gethers, and D. Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18(5):933–969, 2013.

[20] J. Rubin and M. Chechik. A survey of feature location techniques. *Domain Engineering: Product Lines, Conceptual Models, and Languages. Springer, To appear.*

[21] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: a framework and experience. In T. Kishi, S. Jarzabek, and S. Gnesi, editors, *SPLC*, pages 101–110. ACM, 2013.

[22] M. H. ter Beek, H. Muccini, and P. Pelliccione. Guaranteeing correct evolution of software product lines. *ERCIM News*, 2012(88), 2012.

[23] Y. Xue, Z. Xing, and S. Jarzabek. Feature location in a collection of product variants. In *WCRE*, pages 145–154. IEEE Computer Society, 2012.

# Lebenslauf

## Persönliche Daten

| | |
|---|---|
| Name | Stefan Fischer |
| Geburtsdatum | 12.Dezember.1986 |
| Staatsbürgerschaft | Österreich |
| Führerschein | B |

## Ausbildung

| | |
|---|---|
| 2012 – jetzt | Master Studium Software Engineering an der Johannes Kepler Universität Linz |
| 2007 – 2012 | Bachelor Studium Informatik an der Johannes Kepler Universität Linz |
| 2006 – 2007 | Wehrpflicht geleistet in Hörsching, Wels und Ried im Innkreis |
| 2001 – 2006 | HTL für Industrielle Elektronik in Braunau |

# Berufserfahrung

| | |
|---|---|
| 2013 – jetzt | Studentischer Mitarbeiter am Institut für Software Systems Engineering an der Johannes Kepler Universität Linz |
| 2008 | 2 Monate Praktikum bei der SPS Industrie Elektrik in Braunau am Inn (Anlagen Automatisierung) |
| 2007 | 6 Wochen Praktikum bei der SPS Industrie Elektrik in Braunau am Inn (Anlagen Automatisierung) |
| 2005 | 4 Wochen Praktikum bei Kowe (Zerspanungstechnik) |
| 2004 | 3 Wochen Praktikum am Gemeindeamt Mining |
| 2002 | 4 Wochen Praktikum bei der IWK Maschinenbau in Altheim |

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, Februar 2014

Stefan Fischer