Technisch-Naturwissenschaftliche
Fakultät

Extracting Traceability Information from Products' Feature Sets and Code for Reverse Engineering Software Product Lines

BACHELORARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Lukas Linsbauer, 0956251

Angefertigt am:
Institute for Systems Engineering and Automation

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

Mitwirkung:
Dr. Roberto Lopez Herrejon M. Sc.

Linz, Juli 2012

Abstract

Quite often companies develop a range of similar software products, tailored to individual customers, by reusing code and other artifacts from related projects that share some functionality instead of writing them from scratch. Inadvertently, they do create the potential capability of reverse engineering a Software Product Line. However, tapping into the benefits offered by this development paradigm poses several challenges. One of them is tracing the features to the code that implements them across the different product variants. In this paper, we present an algorithm to address this problem and evaluate it at the level of class methods and fields. We applied our approach to three case studies of different sizes and problem domains. Out of the more than 22,000 elements at this granularity level only 0.7% could not be traced due to code being shared by disjunctive features.

Contents

Chapter 1

Introduction

This first chapter provides an introduction to *Software Product Lines (SPLs)* in general as the very foundation of this work, followed by its motiviation and the goal we are aiming at and what this thesis contributes to reaching it.

1.1 Software Product Line (SPL)

Product lines have existed for a very long time in traditional manufactoring. A typical example for a product line would be the manufactoring process of a car of a certain type. All cars of a certain model are similar, but most of them are not identical. One instance of this type of car may have a different color, the other a different engine or an extra feature like air conditioning. Each of these are features of a car that can be composed in various ways. Some features are optional and represent the variable part of the car while others may be mandatory and are the same for every instance of this type of car. Every type of car is then composed of a set of assets that represent certain features instead of developing each car from scratch depending on a customer's needs.

Now this concept is also applied in the discipline of software engineering in the form of strategic, planned reuse. A family of related software products is built using a set of assets, each implementing certain features. These assets are designed and developed for reuse. This way the products can be built to address certain market segments or types of customers without building every software product from scratch. Clements and Northrop define an SPL as follows:

Definition 1. *A Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [Clements and Northrop, 2002].*

According to van d. Linden et al. [van d. Linden et al., 2007] software product line engineering comprises two life-cycles: *domain-engineering* and *application engineering*. Domain engineering results in the common assets that together form the product line's

platform. It also ensures that the platform has the variability that is needed to support the desired scope of products. Application engineering develops the products in the product line. It results in the delivered products. These two life-cycles consist of sub-processes that interact with each other. The domain engineering sub-processes result in common assets that are used in their application engineering counterparts to create products. In return the application engineering sub-processes generate feedback that is used in domain engineering to improve the common assets.

According to Clements and Northrop [Clements and Northrop, 2002] on the other hand there are three essential product line activities: *management*, *core asset development* and *product development*, where core asset development corresponds to domain engineering and product development corresponds to application engineering [Northrop, 2008].

A product line strategy in software engineering can lead to dramatic improvements in cost, quality, productivity and time-to-market [Clements and Northrop, 2002, van d. Linden et al., 2007]. However, applying the concept of product lines in software engineering efficiently and being able to handle the complexity of large product lines at all requires proper techniques and tool support.

In addition, there exist a lot of legacy software products that may be similar but were not designed as a product line from the start. In such cases it may be worth it to refactor these products into a software product line in order to be able to benefit from the advantages. And this is where our work starts.

1.2 Goal

Companies usually develop similar software products without designing them as SPLs. However, these software products will most likely have artifacts in common. For example, almost every application needs some kind of login or authentication. As already mentioned, software product lines offer various benefits, like increased reusability, fewer defects and reduced time to market among others [Clements and Northrop, 2002, van d. Linden et al., 2007]. To reverse engineer an SPL from product variants and take advantage of these benefits several challenges have to be addressed. First, a common feature model has to be derived from product variants [Haslinger et al., 2011], and it is also crucial to know where in the code each feature is actually implemented.

In this thesis, we present an algorithm to trace features of product variants to the source code that implements them at the level of class methods and fields. We describe the algorithm based on feature algebra as presented in [Liu et al., 2006], holding on to the assumption of a unique trace for every code (i.e. one piece of code traces to exactly one module) while allowing the removal of code.

The algorithm is evaluated on three case studies. Of more than 22K code elements totalled by our case studies at the granularity of class methods and fields, more than 99% could be traced to features such that the original products could be reconstructed using the extracted traceability information. The code pieces that could not be traced either

did not have a unique trace because they belonged to multiple disjunctive features or, in very few cases, there were mistakes in the source code of some product variants such that there was code left over from other variants that did not actually belong there anymore.

Chapter 2

Background and Example

The driving goal of our work is to provide support to help derive an SPL from programs that were not originally conceived to be part of a common product line, but they may share a history. For that we need to trace the programs' features in their source code in order to be able to compose different variants of programs. We start off with a set of programs. For each program we initially know the features that it implements and its source code. By comparing these programs with each other we derive associations between parts of the source code and features. In our approach we assume that products which have common features also have common code, and that this common code implements exactly these common features. In terms of code granularity we focus on classes, methods and fields. We perform a basic experiment at this level of granularity to see if the approach works.

2.1 Example: Draw Product Line (DPL)

The example we use to illustrate our approach consists of four product variants. We will refer to the SPL formed with these four variants as the *Draw Product Line (DPL)*. DPL variants are simple drawing programs with different capabilities - depending on the selected features of the product - such as drawing lines and rectangles, wipe the drawing area clean, or select a color to draw with. Their feature combinations and code are shown in this chapter along with the necessary background to understand the underlying theoretical model of our work.

2.2 Feature Model

A *Feature Model (FM)* describes the features of a SPL and their relations to each other. It is a tree structure with the nodes being features. The root node of a feature model is always included in all products. A feature can only be part of a product if its parent feature is also part of it. A feature can be *mandatory* (denoted with filled circles at the child end of an edge) or *optional* (denoted with empty circle at the child end of an edge). A mandatory feature is part of a product whenever its parent feature is. An optional feature

can but need not be part of a product if its parent is. Features can be grouped into an *inclusive-or* relation (filled arc) where one or more features of the group can be selected or an *exclusive-or* relation (empty arc) where exactly one feature must be selected. In addition features can depend on each other across the whole tree in the so called *cross-tree constraints*. Typically these are *requires* relations where selecting a feature `A` also requires the selection of another feature `B` (denoted with doted single-arrow lines), and *excludes* relations where selecting a feature `A` prohibits the selection of another feature `B` (denoted with doted double-arrow lines). Feature models without cross-tree constraints are called *basic feature models* [Haslinger et al., 2011, Kang et al., 1990].

The full feature model to the DPL is given in Figure 2.1. At least one of the features `LINE` and `RECT` has to be selected. Features `WIPE` and `COLOR` are optional. In case of the feature `COLOR` being selected at least one of the colors has to be selected as well.

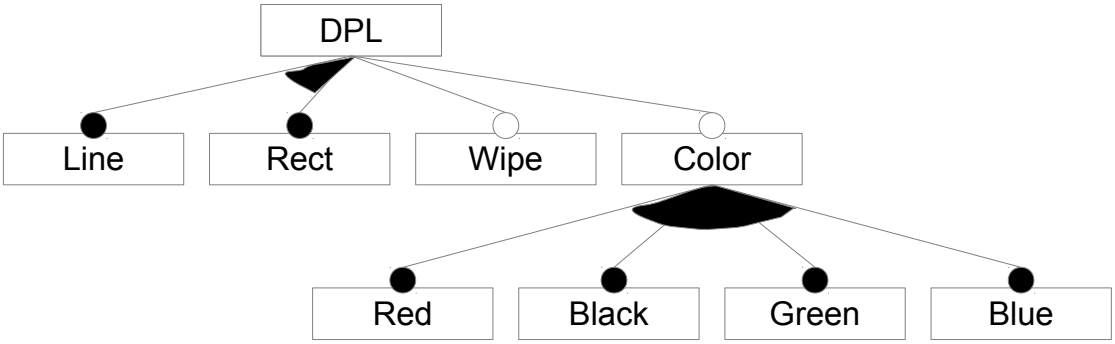

Figure 2.1: DPL Feature Model

Every product of this SPL has the feature DPL as it is the root of the feature model. For simplicity the root feature was left out in the example products, otherwise the module expressions would have become large and the examples complex. The feature DPL would be associated with code that is present for every product within this SPL. We also omit the features for the different colors and for drawing a rectangle for simplicity.

2.3 Feature List

A *feature list* FL is a set containing all the features in a feature model. In our example, we assume a feature list $\texttt{FL} = \{\texttt{LINE}, \texttt{WIPE}, \texttt{COLOR}\}$. Feature `LINE` is mandatory and features `WIPE` and `COLOR` are optional. A *feature set* of a product is a 2-tuple $(sel, \overline{sel})$ where sel is the set of features that are selected in the product and $\overline{sel}$ is the set of features that are not selected. The products' feature sets can also be described in form of a feature set table as shown in Table 2.1, there is a row for each product and a column for every feature in the SPL. If a product provides a feature there is a mark in the corresponding field. In other words, each row represents the feature set for the corresponding product [Haslinger et al., 2011].

Products	Wipe	Line	Color	Feature Sets
Product 1	✓	✓		({W, L}, {C})
Product 2		✓		({L}, {W, C})
Product 3		✓	✓	({L, C}, {W})
Product 4	✓	✓	✓	({W, L, C}, {})

Table 2.1: Feature Set Table

2.4 Source Code

The code snippets for the example products we will use are shown in Code Listing 1. For each of the four products the classes `Line` and `Canvas` are shown. $\texttt{Product}_1$ is from Line 1 to Line 13 and $\texttt{Product}_2$ from Line 15 to Line 26. The class `Line` looks the same for both. But in the class `Canvas` there is no method `wipe` for $\texttt{Product}_2$ as the feature `WIPE` is not part of it. $\texttt{Product}_3$ is from Line 28 to 40. The constructor of class `Line` at Line 32 now also contains a parameter for the color. The old constructor was removed. And the class `Canvas` contains a method for setting the color at Line 38. $\texttt{Product}_4$ is from Line 42 to Line 55 and is the same as $\texttt{Product}_3$ with the only difference that in class `Canvas` at Line 53 the method `wipe` is present again. This code gives us a number of unique code pieces to work with. They are listed in Figure 2.2.

c_1	`Point Line.startPoint`
c_2	`Point Line.endPoint`
c_3	`void Line.paint(Graphics g)`
c_4	`Line.Line(Point start)`
c_5	`void Line.setEnd(Point end)`
c_6	`Line.Line(Color color, Point start)`
c_7	`List < Line > Canvas.lines`
c_8	`void Canvas.wipe()`
c_9	`void Canvas.setColor(String colorString)`

Figure 2.2: Unique Code Pieces

2.5 Feature Algebra

The products can also be described via feature algebra as presented in the work of *Feature Oriented Software Development (FOSD)* [Liu et al., 2006]. We use this algebraic notation to describe our traceability mining algorithm more precisely. The idea is that a *product* can be composed by adding *features*. Features are written in uppercase letters. For example $\texttt{Product}_1$ is composed by extending feature `LINE` with feature `WIPE`, written as `WIPE(LINE)`. This is called a *feature expression*.

Each *feature* consists of *modules*, which represent its implementation. The *base module* [Liu et al., 2006] of a feature contains the code that is always present in a product that has this feature, independent of any other features that may or may not be present. A base module is denoted in lowercase letters. For example, the base module of feature `LINE`

```
1 /* Product 1 (WIPE, LINE) */
2 class Line {
3    Point startPoint, endPoint;
4    void paint(Graphics g) {...}
5    Line(Point start) {...}
6    void setEnd(Point end) {...}
7    ...
8 }
9 class Canvas {
10   List<Line> lines = new LinkedList<Line>();
11   void wipe() {...}
12   ...
13 }
14
15 /* Product 2 (LINE) */
16 class Line {
17   Point startPoint, endPoint;
18   void paint(Graphics g) {...}
19   Line(Point start) {...}
20   void setEnd(Point end) {...}
21   ...
22 }
23 class Canvas {
24   List<Line> lines = new LinkedList<Line>();
25   ...
26 }
27
28 /* Product 3 (COLOR, LINE) */
29 class Line {
30   Point startPoint, endPoint;
31   void paint(Graphics g) {...}
32   Line(Color color, Point start) {...}
33   void setEnd(Point end) {...}
34   ...
35 }
36 class Canvas {
37   List<Line> lines = new LinkedList<Line>();
38   void setColor(String colorString) {...}
39   ...
40 }
41
42 /* Product 4 (COLOR, WIPE, LINE) */
43 class Line {
44   Point startPoint, endPoint;
45   void paint(Graphics g) {...}
46   Line(Color color, Point start) {...}
47   void setEnd(Point end) {...}
48   ...
49 }
50 class Canvas {
51   List<Line> lines = new LinkedList<Line>();
52   void setColor(String colorString) {...}
53   void wipe() {...}
54   ...
55 }
```

Code Listing 1: DPL product snippets

is denoted as `line`. It contains the code of class *Line* that is always there if the feature `LINE` is selected. This means the constructor is not part of the base module, because it changes depending on other selected features. All products have the feature `LINE`, but not all have the same constructor in class *Line*.

The feature `WIPE` in $\texttt{Product}_1$ also consists of another module $\delta\texttt{line}/\delta\texttt{WIPE}$ which is called a *derivative module* [Liu et al., 2006]. It contains the changes the feature `WIPE` makes to the module `line`. In case of $\texttt{Product}_1$ there is no source code associated with that module and there are no changes. A change can be the addition of code as well as the alteration or removal of code. This interpretation is different to the one presented in [Liu et al., 2006] in that we allow the removal of code as will be explained in Section 3.3. Derivative modules basically model the interaction of features, how features influence each other. For example, only the combination of a number of specific features makes a certain piece of code necessary (or respectively unnecessary). There are also higher order derivatives like $\delta^2\texttt{color}/\delta\texttt{LINE}\delta\texttt{WIPE} \bullet$ in $\texttt{Product}_4$. It represents the changes that features `LINE` and `WIPE` make to module `color`. So the principle is the same, there are just more features involved. This example models the interaction of three features.

There are *two operations* on modules that allow us to compose them [Liu et al., 2006]. The first operation is + which is a binary operation that unifies the code of two base modules. The code of two base modules is disjoint. The second operation is $\bullet$ which either composes two derivative modules into a composite derivative module or weaves the changes of a derivative module into a base module yielding a so called *woven base module*.

With these two operations the relationship between a *feature expression* and the corresponding *module expression* that implements it can be defined as shown in Figure 2.3 for the four products of Table 2.1. For example, $\texttt{Product}_3$ is composed by applying feature `LINE` to feature `COLOR` as can be seen in its feature expression. The product is implemented by the respective base modules `line` and `color` which contain the code that is always present for these features. In addition the changes that feature `LINE` makes to the base module `color` are woven into module `color` in the form of the derivative $\delta\texttt{color}/\delta\texttt{LINE}$.

$$[\ \texttt{Product}_1\ ] = [\ \texttt{WIPE(LINE)}\ ] = \texttt{wipe} + \delta\texttt{line}/\delta\texttt{WIPE} \bullet \texttt{line}$$

$$[\ \texttt{Product}_2\ ] = [\ \texttt{LINE}\ ] = \texttt{line}$$

$$[\ \texttt{Product}_3\ ] = [\ \texttt{LINE(COLOR)}\ ] = \texttt{line} + \delta\texttt{color}/\delta\texttt{LINE} \bullet \texttt{color}$$

$$[\ \texttt{Product}_4\ ] = [\ \texttt{WIPE(LINE(COLOR))}\ ] = \texttt{wipe} + \delta\texttt{line}/\delta\texttt{WIPE} \bullet \texttt{line} + \delta^2\texttt{color}/\delta\texttt{LINE}\delta\texttt{WIPE} \bullet \delta\texttt{color}/\delta\texttt{LINE} \bullet \delta\texttt{color}/\delta\texttt{WIPE} \bullet \texttt{color}$$

Figure 2.3: Products in Feature Algebra

As can be seen the module expressions grow very fast with the number of features. A

product with n features has a feature expression consisting of n features and a module expression consisting of $2^n - 1$ modules. Notice that modules are separated by operations $+$ or $\bullet$.

Now we have our examples set up. We know for each of our example products their features, their module expressions and their source code. The ideal solution to the stated problem would be to know for each piece of source code to which module it belongs and the pieces of source code each module consists of. In practice however, it will not be possible to isolate every module. There may be modules for which their source code could not be distinguished, for example because two modules never exist without each other.

Chapter 3

Traceability Mining Algorithm

In this section we present an algorithm for tracing products' features in their source code. The algorithm expects as input a number of products with their corresponding feature sets and code. For example, $\texttt{Product}_1$ with features `LINE` and `WIPE`. This product consists of only two features as its feature expression `WIPE(LINE)` shows. The corresponding module expression however consists of three modules: the base modules `line` and `wipe` and the derivative module δ`line`/δ`WIPE`. Each of these modules consists of its own code (if any). The algorithm then computes what code belongs to which modules.

3.1 Basic Insight

The general idea behind this algorithm is the observation that products that have features in common will also have code in common and vice versa. A *product* is defined as a tuple where the first element is its feature set, accessed via `product.f`, and the second element is the set of code it contains, accessed via `product.c`:

$$\texttt{product} = (\texttt{featureset}, \texttt{codeset})$$

Note that it is not sufficient to just look at a product's features as the above example shows, because the interactions between features, which are exactly the derivatives, would be left out. So instead of looking at products' features in their feature expressions we look at the modules in their module expressions.

The modules that two input products have in common are then associated with the code these same two products have in common. Assume $\texttt{Product}_1$ and $\texttt{Product}_2$ as input. $\texttt{Product}_2$ consists of only one module, which is the base module `line`. The code these two products have in common is therefore associated with the base module `line`. The remaining code in $\texttt{Product}_1$ is left for the base module `wipe` and the derivative δ`line`/δ`WIPE`.

The basic concept behind this is to treat products as *associations* between modules and code. We define an association to be a tuple where the first element is the set of

modules and the second element is the set of code:

$$\texttt{association} = (\texttt{moduleset}, \texttt{codeset})$$

We access an association's module set via `association.m` and its codeset via `association.c`. For $\texttt{Product}_1$ and $\texttt{Product}_2$ these associations initially are:

$\texttt{association}_1 = \texttt{a}_1 = (\ \{\texttt{line}, \texttt{wipe}, \delta\texttt{line}/\delta\texttt{WIPE}\}\ ,\ \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7, \texttt{c}_8\}\ )$

$\texttt{association}_2 = \texttt{a}_2 = (\ \{\texttt{line}\}\ ,\ \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7\}\ )$

Associations are then intersected by respectively intersecting their module sets and their code sets (see Figure 3.1). By doing so, it is possible to obtain new associations that are added to the list or alter existing associations. For example, after intersecting these two associations the set of associations is:

$\texttt{association}'_1 = \texttt{a}'_1 = (\texttt{a}_1.\texttt{m} \setminus \texttt{a}_2.\texttt{m}, \texttt{a}_1.\texttt{c} \setminus \texttt{a}_2.\texttt{c}) = (\ \{\texttt{wipe}, \delta\texttt{line}/\delta\texttt{WIPE}\}\ ,\ \{\texttt{c}_8\}\ )$

$\texttt{association}'_2 = \texttt{a}'_2 = (\texttt{a}_2.\texttt{m} \setminus \texttt{a}_1.\texttt{m}, \texttt{a}_2.\texttt{c} \setminus \texttt{a}_1.\texttt{c}) = (\ \{\}\ ,\ \{\}\ )$

$\texttt{association}_3 = \texttt{a}_3 = (\texttt{a}_1.\texttt{m} \cap \texttt{a}_2.\texttt{m}, \texttt{a}_1.\texttt{c} \cap \texttt{a}_2.\texttt{c}) = (\ \{\texttt{line}\}\ ,\ \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7\}\ )$

The existing associations are altered by removing the elements in the intersection from their modules and code. The intersection is added as new $\texttt{association}_3$. For example, code $\texttt{c}_8$ which corresponds to method `wipe` in class `Canvas` (see Figure 2.2) is now associated with modules `wipe` and $\delta\texttt{line}/\delta\texttt{WIPE}$ in association $\texttt{a}'_1$ and code $\texttt{c}_1$ which corresponds to field `startPoint` in class `Line` (see Figure 2.2) is associated with the module `line` in association $\texttt{a}_3$ as displayed in Figure 3.1. This intersection process is repeated as new product sets are integrated. At the end, every feature and every piece of code appear exactly once. This means that a piece of code can only belong to a single module. Therefore our algorithm rests on the following assumption:

Unique Trace Assumption. *A piece of code is unique to a module, that is, it does not trace to multiple modules.*

3.2 From Features to Modules

The algorithm starts with just the features of each product, so it needs to calculate the modules. A product with a set of x features has a module expression with $2^x - 1$ modules. The modules are obtained by building the powerset of the set of features without the empty set: $\texttt{moduleset} = \mathcal{P}(\texttt{featureset.sel})\backslash\{\emptyset\}$. For $\texttt{Product}_1$:

$\texttt{moduleset} = \mathcal{P}(\{\texttt{LINE}, \texttt{WIPE}\})\backslash\{\emptyset\} = \{\{\texttt{LINE}\}, \{\texttt{WIPE}\}, \{\texttt{LINE}, \texttt{WIPE}\}\}$

Sets with exactly one feature represent the base modules and sets with more than

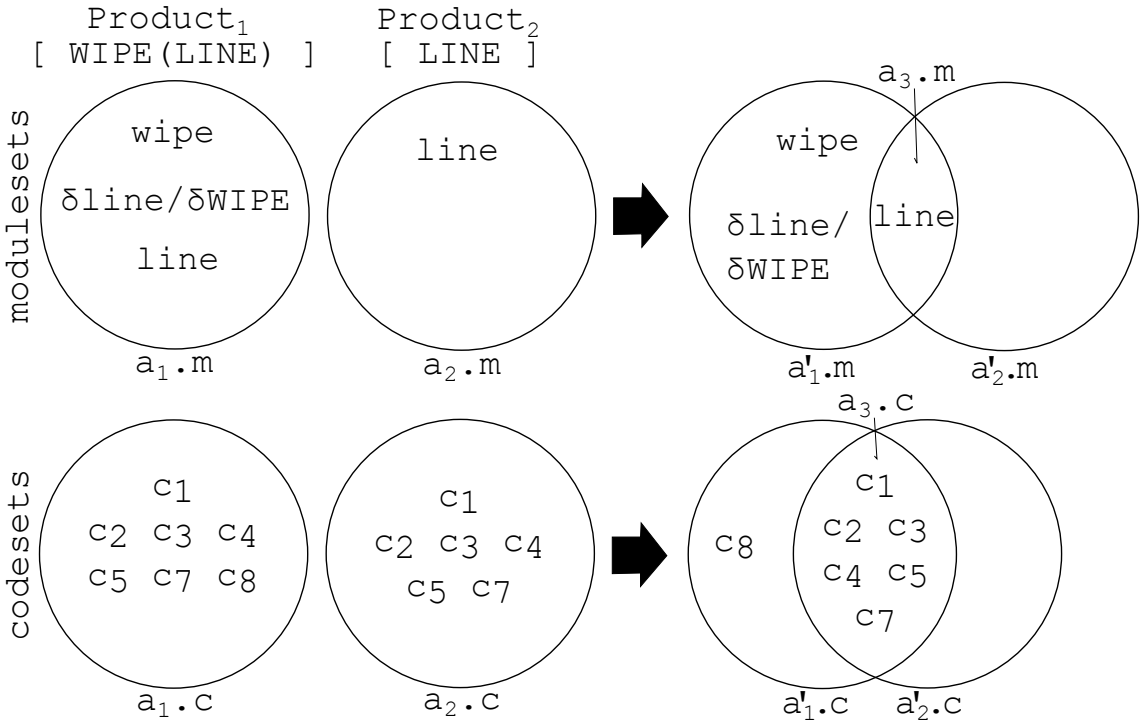

Figure 3.1: Intersection of Product 1 and Product 2

one feature (e.g. $\{\texttt{LINE}, \texttt{WIPE}\}$) represent the derivatives. There is no order in a set, so $\delta\texttt{line}/\delta\texttt{WIPE}$ has to be equal to $\delta\texttt{wipe}/\delta\texttt{LINE}$ in order for this to be legitimate. For our algorithm it does not matter in what order the features are added, it is only important to know whether two features interact or not, so we can easily assume that is the case.

$\{\texttt{LINE}, \texttt{WIPE}\} = \delta\texttt{line}/\delta\texttt{WIPE} = \delta\texttt{wipe}/\delta\texttt{LINE}$

3.3 Dealing with Code Removal

An assumption for this approach so far is that code can only be added. Modules are not allowed to remove code. So we have to find another way to model such effect. Assume we have a base module `line` which adds some pieces of code $\{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7\}$ to a program. Then we add the feature `COLOR`, and therefore the modules `color` and $\delta\texttt{line}/\delta\texttt{COLOR}$, to the program. Module `color` adds some code just as module $\delta\texttt{line}/\delta\texttt{COLOR}$ does. But $\delta\texttt{line}/\delta\texttt{COLOR}$ also removes code $\texttt{c}_4$ (the old constructor) from the program. At this point the question arises whether code $\texttt{c}_4$ was part of the base module `line` to begin with. If it is not always present when the module `line` is present, then it is obviously not a part of it. But where else would it belong?

This is actually exactly what happens with the examples $\texttt{Product}_2$ and $\texttt{Product}_3$. Their intersection is shown in Figure 3.2. One can see, that code $\texttt{c}_4$ has no corresponding modules. The intersection of the modules at that point is empty.

Indeed, in a situation like this the code could be left over without any modules to

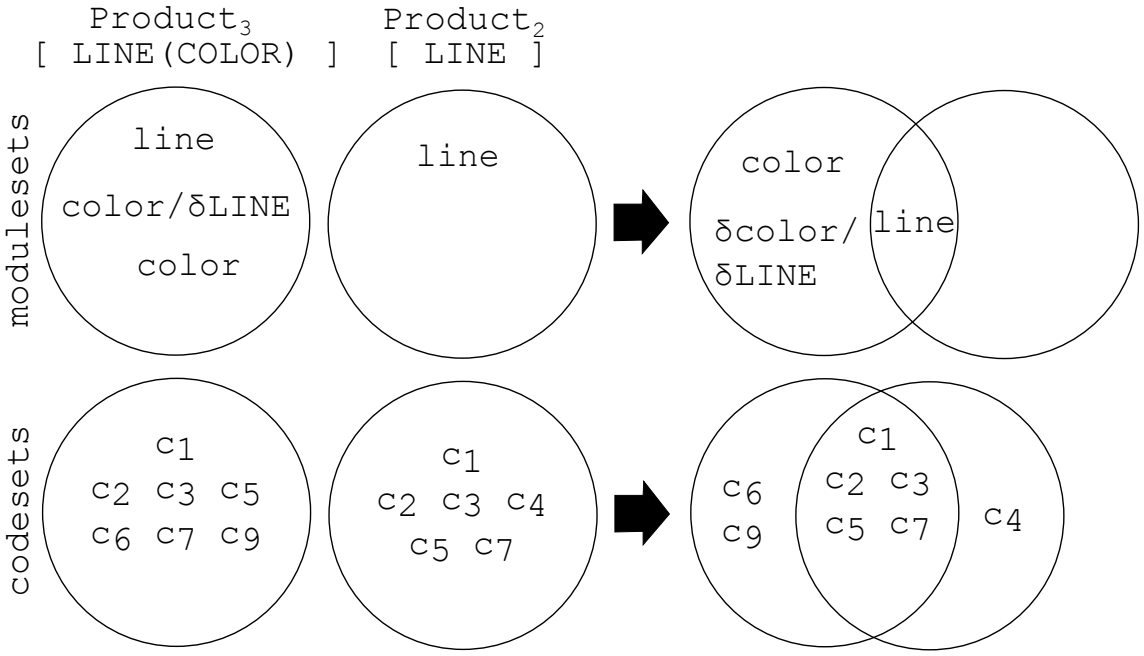

Figure 3.2: Intersection of Product 3 and Product 2

associate it with. It also should be noted, that this problem only applies to derivatives, as base modules could not remove code from a module other than itself, which would make no sense.

3.3.1 Shared and Unique Code for Derivatives

The first idea was to distinguish between shared and unique code for derivatives, but that approach left us with some problems.

If a derivative of two features δ`line`/δ`COLOR` removes code c_4 from module `line`, wouldn't it be the same as to say that said code has never been part of module `line` to begin with, and instead the derivatives of line with any other feature but `COLOR` add this same code? δ`line`/δ`COLOR` does not remove code c_4, but δ`line`/δ`WIPE`, δ`line`/δ`RECT`, ... all add this code. This would mean, that the relation between modules and code is no longer a 1:n relation but an n:n relation. One piece of code can now belong to several different modules. We call these code pieces "shared". Other code that is only part of one module is called "unique". But does this really solve our problem? Take a look at our $\texttt{Product}_4$ with features {`LINE`, `WIPE`, `COLOR`}. This product contains the derivative δ`line`/δ`WIPE` as well as the derivative δ`line`/δ`COLOR`. The code c_4 would be part of δ`line`/δ`WIPE` and should therefore be present in the product, but it is not. Because truly it depends on the presence (or absence) of δ`line`/δ`COLOR` whether c_4 is present or not. The "removal" aspect of δ`line`/δ`COLOR` binds stronger than the "addition" of δ`line`/δ`WIPE`. And this is a behaviour that we do not really want. So this approach was discarded as well. Nonetheless, such an approach might be suitable for getting rid of our unique trace

assumption as part of our future work.

3.3.2 Negative Features

Instead of having a derivative remove a piece of code $\texttt{c}_4$ we would rather have another derivative add this same piece of code. But it does not fit into any of our derivatives we have so far. So we introduce *negative features*. For every feature F there is now also its negation $\neg$F. This leaves us with many new derivatives to work with. We can now associate $\texttt{c}_4$ with the derivative δline/$\delta\neg$COLOR. So instead of having δline/δCOLOR remove $\texttt{c}_4$ we are now having δline/$\delta\neg$COLOR add $\texttt{c}_4$. We have yet to define what exactly this means though and how we want to interpret such modules containing negative features. For example, δline/$\delta\neg$COLOR could be interpreted as the derivatives of line and anything that is not color, but that is not what we want. Such an interpretation would just be an abbreviation for a list of other derivatives. It could also be interpreted as a synonym for the one derivative of line and all features except color. But this is also not what we want. We want δline/$\delta\neg$COLOR to be its own module, with its own unique code, that does not have anything to do with other modules. And that is exactly how we interpret and use these modules.

With negative features, the feature and module expressions of our products now look different (see Figure 3.3). Each feature expression now contains every feature in the feature list exactly once, either positive or negative. Therefore each product consists of $2^N - 1$ modules now, where N is the number of features in the whole feature list (not just the features that are implemented by the product as before). Modules that contain only negative features/modules can be discarded. Negative features/modules only make sense as a derivative with at least one positive feature/module.

$$[\ \texttt{Product}_1\ ] = [\ \neg\texttt{COLOR}(\texttt{WIPE}(\texttt{LINE}))\ ] = (\neg\texttt{color}) + \delta\texttt{wipe}/\delta\neg\texttt{COLOR} \bullet \texttt{wipe} + \delta^2\texttt{line}/\delta\texttt{WIPE}\delta\neg\texttt{COLOR} \bullet \delta\texttt{line}/\delta\neg\texttt{COLOR} \bullet \delta\texttt{line}/\delta\texttt{WIPE} \bullet \texttt{line}$$

$$[\ \texttt{Product}_2\ ] = [\ \neg\texttt{WIPE}(\neg\texttt{COLOR}(\texttt{LINE}))\ ] = (\neg\texttt{wipe} + \delta\neg\texttt{color}/\delta\neg\texttt{WIPE} \bullet \neg\texttt{color}) + \delta^2\texttt{line}/\delta\neg\texttt{WIPE}\delta\neg\texttt{COLOR} \bullet \delta\texttt{line}/\delta\neg\texttt{COLOR} \bullet \delta\texttt{line}/\delta\neg\texttt{WIPE} \bullet \texttt{line}$$

$$[\ \texttt{Product}_3\ ] = [\ \neg\texttt{WIPE}(\texttt{LINE}(\texttt{COLOR}))\ ] = (\neg\texttt{wipe}) + \delta\texttt{line}/\delta\neg\texttt{WIPE} \bullet \texttt{line} + \delta^2\texttt{color}/\delta\texttt{LINE}\delta\neg\texttt{WIPE} \bullet \delta\texttt{color}/\delta\texttt{LINE} \bullet \delta\texttt{color}/\delta\neg\texttt{WIPE} \bullet \texttt{color}$$

$$[\ \texttt{Product}_4\ ] = [\ \texttt{WIPE}(\texttt{LINE}(\texttt{COLOR}))\ ] = \texttt{wipe} + \delta\texttt{line}/\delta\texttt{WIPE} \bullet \texttt{line} + \delta^2\texttt{color}/\delta\texttt{LINE}\delta\texttt{WIPE} \bullet \delta\texttt{color}/\delta\texttt{LINE} \bullet \delta\texttt{color}/\delta\texttt{WIPE} \bullet \texttt{color}$$

Figure 3.3: Products in Feature Algebra with negative Features

For example, taking a closer look at $\texttt{Product}_3$ in Figure 3.3. The feature expression now contains one more feature, namely $\neg$WIPE. It is now explicit that the feature WIPE is not implemented by this product. This also reflects in the corresponding module expression.

The first module $\neg$`wipe` in parenthesis can be omitted as it is negative and does not interact with any positive features or modules. The second module δ`line`/$\delta\neg$`WIPE` however makes sense, as the implementation of the positive feature `LINE` can be influenced by feature `WIPE` not being present. In this case the *base module* `line` contains code that is always present if feature `LINE` is present. In addition, the module δ`line`/$\delta\neg$`WIPE` adds code that is specific to the implementation of feature `LINE` if feature `WIPE` is not present.

3.4 Algorithm Pseudo-Code

The following helper functions are used in the algorithm:

- *NOT(featureset)*: Negates all features contained in the set. For example: $NOT(\{\texttt{LINE}, \texttt{WIPE}\}) = \{\neg\texttt{LINE}, \neg\texttt{WIPE}\}$.
- *POW(featureset)*: Generates the powerset of the set (without the empty set and without modules consisting only of negative features/modules). This basically generates the modules for a set of features. For example: $\texttt{POW}(\{\texttt{LINE}, \neg\texttt{WIPE}\}) = \{\{\texttt{LINE}\}, \{\texttt{LINE}, \neg\texttt{WIPE}\}\}$.

The first part of the algorithm prepares all the input products for processing by converting them into initial associations. Each of these associations obtains its code from the corresponding product. The modules are calculated as the powerset of the union of the features of the product and the negated version of the features not contained in the product. The second part of the algorithm does the actual processing. One initial association after the other is processed and new associations are added to the final list of associations to be returned. The algorithm is shown in Algorithm 1. Some optimizations like not adding empty associations or merging associations that contain modules but no code are not shown to keep it simple.

Assume $\texttt{Product}_1$ and $\texttt{Product}_2$ as input for this algorithm. From line 5 to 14 the initial associations and data structures are prepared as follows:

$$\texttt{association}_1 = \\ (\ \{\texttt{line}, \texttt{wipe}, \delta\texttt{line}/\delta\texttt{WIPE}, \delta\texttt{line}/\delta\neg\texttt{COLOR}, \delta\texttt{wipe}/\delta\neg\texttt{COLOR}, \delta^2\texttt{line}/\delta\texttt{WIPE}\delta\neg\texttt{COLOR}\}\ , \\ \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7, \texttt{c}_8\}\ )$$

$$\texttt{association}_2 = \\ (\ \{\texttt{line}, \delta\texttt{line}/\delta\neg\texttt{WIPE}, \delta\texttt{line}/\delta\neg\texttt{COLOR}, \delta^2\texttt{line}/\delta\neg\texttt{WIPE}\delta\neg\texttt{COLOR}\}\ , \\ \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7\}\ )$$

$$\texttt{init\_assocs} = \{\texttt{association}_1, \texttt{association}_2\}$$

$$\texttt{associations} = \{\}$$

We start with an empty set `associations` to be returned by the algorithm and add associations as we iterate over `init_assocs` at line 16. Each initial association is intersected with every association in the `associations` set. We start with $\texttt{association}_1$. As `associations` is still empty there are no associations to intersect it with, the loop at line 19 is not entered. The `remainder`, which is equal to $\texttt{association}_1$, is added to the set as it is at lines 33 and 34.

$$\texttt{remainder} = \texttt{result}_1 = \texttt{association}_1{}^{1}$$

$$\texttt{associations} = \{\texttt{result}_1\}$$

The next association to be processed is $\texttt{association}_2$. It is intersected with $\texttt{result}_1$ from line 20 to 31, as it is now in the `associations` set resulting in the following new associations:

$$\texttt{result}_1 = (\ \{\texttt{wipe}, \delta\texttt{line}/\delta\texttt{WIPE}, \delta\texttt{wipe}/\delta\neg\texttt{COLOR}, \delta^2\texttt{line}/\delta\texttt{WIPE}\delta\neg\texttt{COLOR}\}\ ,\ \{\texttt{c}_8\}\ )$$

$$\texttt{remainder} = \texttt{result}_2 = (\ \{\delta\texttt{line}/\delta\neg\texttt{WIPE}, \delta^2\texttt{line}/\delta\neg\texttt{WIPE}\delta\neg\texttt{COLOR}\}\ ,\ \{\}\ )$$

$$\texttt{intersection} = \texttt{result}_3 = (\ \{\texttt{line}, \delta\texttt{line}/\delta\neg\texttt{COLOR}\}\ ,\ \{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4, \texttt{c}_5, \texttt{c}_7\}\ )$$

$\texttt{result}_1$ is altered (at lines 29 and 30) and `remainder` and `intersection` are added as new results (at lines 33 and 34).

$$\texttt{associations} = \{\texttt{result}_1, \texttt{result}_2, \texttt{result}_3\}$$

As there are no more initial associations the algorithm is done and the `associations` set is returned as result. It contains all the associations that could be extracted. At this point every module and every piece of code appear exactly in one association, no matter in how many of the input products they appeared. Modules appearing together in one association could not be separated from each other. This means one can now look up which code pieces implement which modules by looking at the association that contains the module of interest and the associated code. However, if the association contains more than that one module then it may also contain additional code implementing other modules. As these modules could not be separated, their code could not be separated either.

[1] $\texttt{result}_x$ and $\texttt{association}_x$ are auxiliary variables to explain the algorithm, they do not appear in the code.

Algorithm 1 Traceability Mining Algorithm

```
1  Input: A List of Products (products),
2         A List of all Features (FL)
3  Output: A List of Associations (associations)
4
5  {convert products into initial associations}
6  init_assocs := {}
7  for p in products begin
8     association := (
9        POW(p.f.sel ∪ NOT(FL \ p.f.sel)),
10       p.c
11    )
12    init_assocs := init_assocs ∪ {association}
13 end
14 associations := {}
15 {iteratively process initial associations}
16 for a in init_assocs begin
17    remainder := (a.m, a.c)
18    new_assocs := {}
19    for a2 in associations begin
20       intersection := (
21          remainder.m ∩ a2.m,
22          remainder.c ∩ a2.c
23       )
24       remainder := (
25          remainder.m \ a2.m,
26          remainder.c \ a2.c
27       )
28       {alter existing association}
29       a2.m := a2.m \ intersection.m
30       a2.c := a2.c \ intersection.c
31       new_assocs := new_assocs ∪ {intersection}
32    end
33    associations := associations ∪ new_assocs ∪ {remainder}
34 end
35 return associations
```

Chapter 4

Experimental Setting

4.1 Workflow

An overview of the implemented system is shown in Figure 4.1. We now describe each of the steps it consists of.

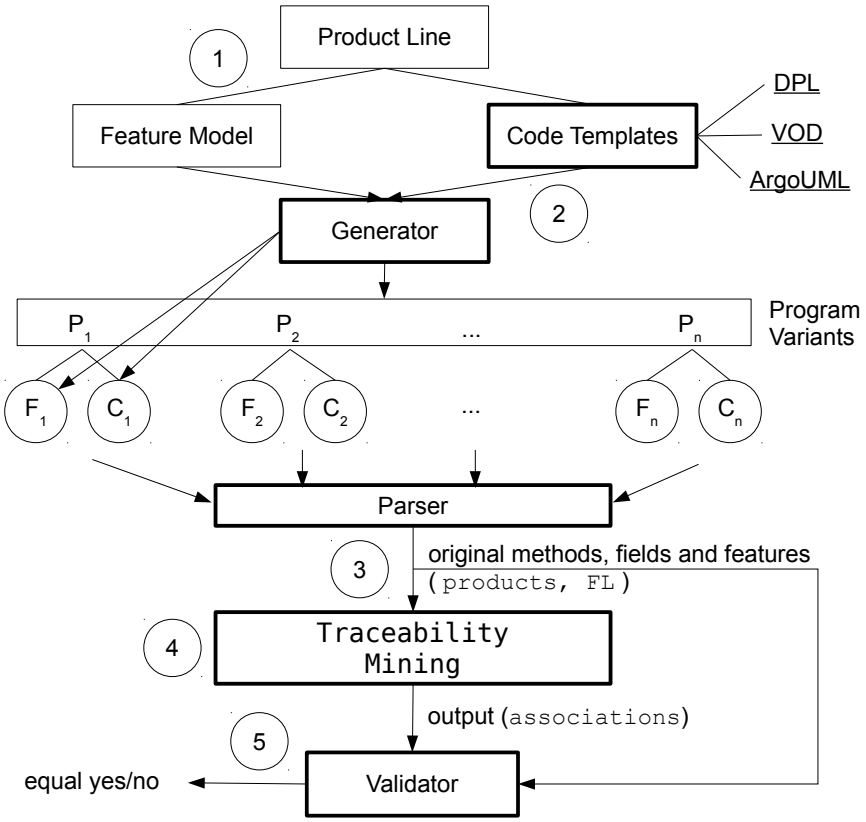

Figure 4.1: System Overview

4.1.1 Step 1: Product Line and Feature Model.

It should be noted that to the best of our knowledge there are only very few publicly available software repositories from which evolved program variants could be mined. Thus, in order to emulate more product variants to evaluate our approach, for some of our case studies we took existing product lines and generated products according to their feature models. The generated products are then used as input for our algorithm to see how well it performs.

4.1.2 Step 2: Templates and Generator.

This part of the system sets up the testing environment. We need it to generate products (the Java source files) from a product line so we can use them as input. For that purpose we have the whole source code of a product line in the form of templates where each piece of code is guarded with certain features. The templates are parsed by the code generator using the Apache Velocity template engine [ASF] or the JavaPP [Kropf], depending on the product line. Code Listing 4.2 shows an excerpt from the Velocity template file for the class `Canvas`. Line 2 is only included in the product if feature `LINE` is selected.

```
1 #if ($LINE)
2 protected List<Line> lines = new LinkedList<Line>();
3 #end
```

Figure 4.2: Template snippet of class `Line`

Figure 4.3 shows how code that corresponds to a first order derivative module is guarded in the template, namely with a conjunction of features (AND).

```
1 #if ($F1 && $F2)
2 ...
3 #end
```

Figure 4.3: Template snippet for code that belongs to derivative δ`F1`/δ`F2`

Code that is guarded as shown in Figure 4.4 with a disjunction of features (OR) violates the unique trace assumption, as it traces to each of the features.

```
1 #if ($F1 || $F2)
2 ...
3 #end
```

Figure 4.4: Template snippet for code that violates the unique trace assumption

4.1.3 Step 3: Parser.

For every given product used as input, the parser reads the features from a text file and extracts all the code elements from the Java source files using the Java Compiler API

[Richard]. The extracted code is on the granularity of class methods and fields.

4.1.4 Step 4: Traceability Mining.

This is the core of our approach. The extracted code and features are fed into the algorithm as well as into the validator for later verification. The algorithm computes associations between modules and source code (fields and methods) as output.

4.1.5 Step 5: Validator.

The validator receives the original products as well as the output from the algorithm as input. With the traceability information provided by the algorithm the validator reconstructs products with the same features as the original products. The reconstruction is done for each input product separately by taking the features it implements and generating the code elements as the union of the code elements of the output associations corresponding to these features:

$$P' = Reconstruct(P.f, associations)$$

where P' is the reconstructed product and P is the corresponding original product. As final step, each reconstructed products code elements are compared to the original product's code elements.

4.2 Implementation

The complete framework is implemented in Java using Eclipse as IDE. In this section the implementation of the framework is shown using UML diagrams.

4.2.1 Data Structures

The *source code* of program variants is represented by the abstract class `Code` of which several different concrete implementations exist, for example one for methods and one for fields (Figure 4.5). Every instance has a `clazz` it belongs to, additionally fields have a `type` and a `name` whereas a method is uniquely identified with its `signature`. The next step would be to also represent statements within methods. They could be described with the signature of the method they belong to and its position within the method (because for statements the order matters). Generally it does not matter what information is stored about code pieces and how it is stored as long as it is possible to uniquely identify a piece of code and code pieces that are meant to be different can be distinguished. We do this by means of the methods `hashCode` and `equals`. This is important if we later want to use more advanced clone detection techniques where for example two methods should be considered equal even though they have a different signature. In such cases only a

new subclass of `Code` has to be added that takes that into account while the rest of the framework is not affected at all.

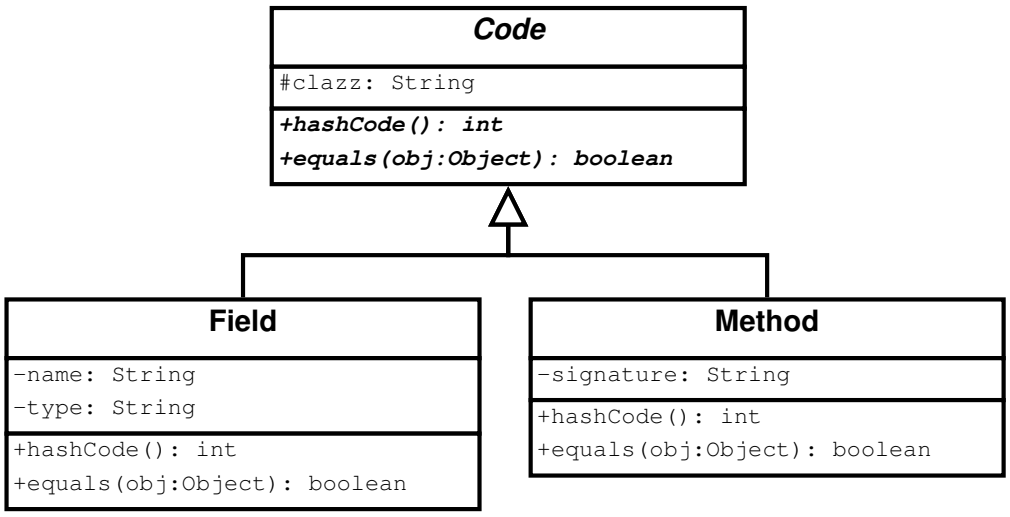

Figure 4.5: UML Class Diagram for Code

The next thing that has to be represented are the *features* of program variants. They are a simple class `Feature` with two fields for `name` and `description` (Figure 4.6). A subclass `NegFeature` of `Feature` exists that represents a negative feature.

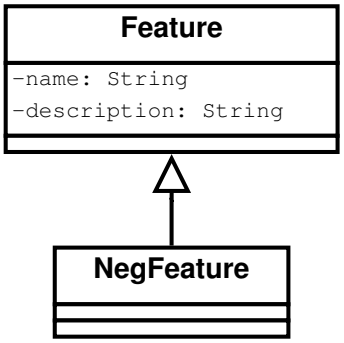

Figure 4.6: UML Class Diagram for Feature

Since we never work with just one feature or one piece of code but with *sets* on which we have to perform different actions we have our own `Set` data structure that helps us do that (Figure 4.7). It is a generic data structure that extends the class `HashSet` from the Java standard library [Oracle, a,b]. A `Set` can do the standard set operations unification, intersection and difference with another set. In addition it can build the powerset of a set which we need for our algorithm to generate modules out of features. A module is then simply represented as a set of features, and a module set therefore is a set of sets of features.

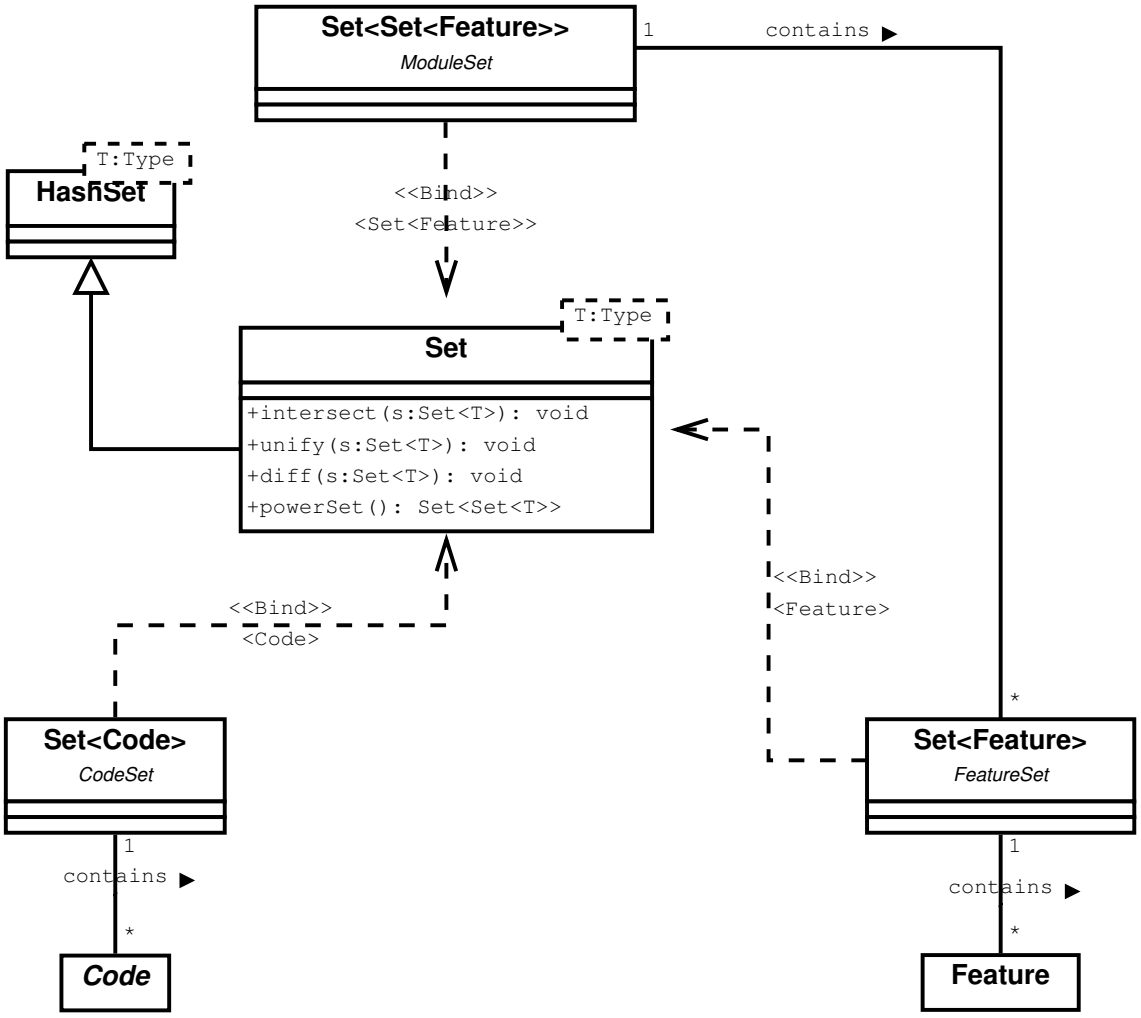

Figure 4.7: UML Class Diagram for Set

4.2.2 Framework

The framework is built in such a way that different algorithms can be implemented and used without having to change the whole system. Therefore an abstract class `Algorithm` exists that is the superclass of every implemented algorithm (Figure 4.8). Every algorithm has to provide the method `evaluate` that returns a list of associations. The class `Association` contains a code set and a module set. The algorithm presented in the previous chapter is implemented in the class `MainAlgorithm`. It expects as input the product variants, the complete list of features and the complete list of code pieces.

What is left is to put all this together. That is done in the class `ProductLine` (Figure 4.9). The method `parseFromFile` reads a previously serialized product line and returns it. With the method `parseFromDir` a folder containing a number of products can be parsed into a product line.

The file structure of a product line on the file system for the parser to be able to read it must be as follows:

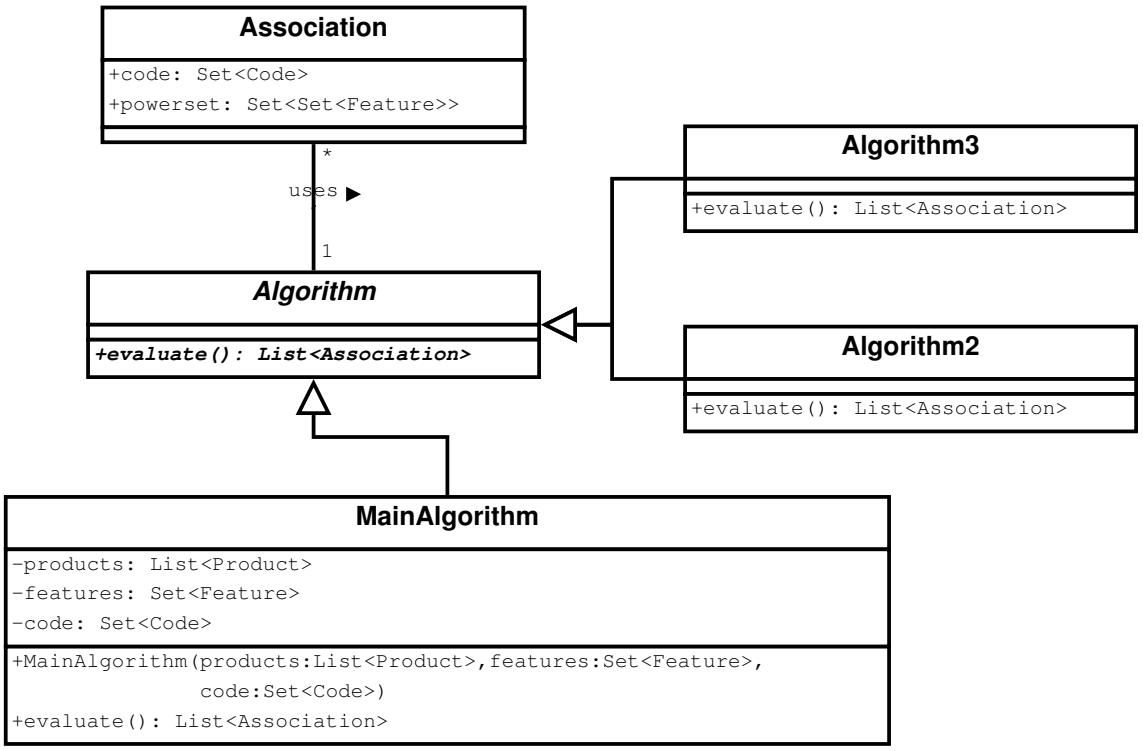

Figure 4.8: UML Class Diagram for Algorithm

- `ProductLine/`
  - `Product1/`
    - `src`
    - `features.txt`
  - `Product2/`
    - `src/`
    - `features.txt`
  - ...

The folder of the product line has to be specified. Every folder inside that folder is parsed by the method `parseProductFromDir` in class `Product` and therefore has to be a valid product. Every product folder must contain the file `features.txt` listing the features of the product and their description separated by ";", one per line (see Figure 4.10). And the folder `src` must contain the Java source files including the package structure. This folder is parsed using the class `Parser` that makes use of the Java Compiler API [Richard] and returns a code set.

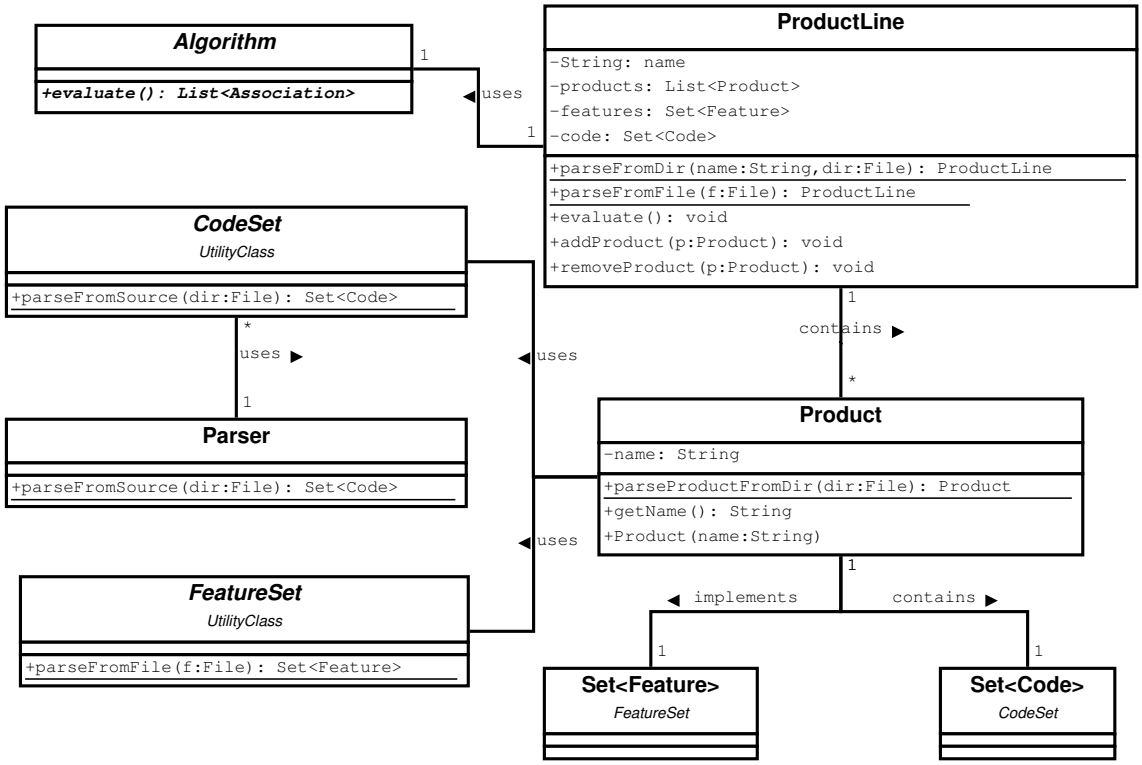

Figure 4.9: UML Class Diagram for ProductLine

```
1 LINE;draw line
2 COLOR;select color
3 DPL;draw product line
```

Figure 4.10: Example for file features.txt from a DPL product

Chapter 5

Evaluation

An overview of the case studies used for evaluation is shown in Table 5.1. It is very difficult to get hands on real world software variants that fit our scenario. Therefore for our first two case studies we used existing SPLs from which we generated a number of variants and treated them as if they were not derived from a product line. This is sufficient to show the correctness of our approach. The third case study however, was taken as is from [MobileMedia-Lancaster]. In the following we present our evaluation criteria and the results obtained in our three case studies.

	VOD	**ArgoUML**	**MM**
Mandatory Features	6	3	6 [1]
Optional Features	5	8	8 [1]
Possible Products	32	256	7 [1]
Lines of Code	5.3K+	340K+	5K+
Classes (*.java Files)	42	1915	50
Fields	392	4452	223
Methods	249	16676	422
Unique Code Pieces	642	21128	645
Associations (with Code)	5	26	22
Correctness [%]	100	99.4	99.6
Performance [sec]	0.9	45	1.3
Distinguishability	63.8	7.8	3060.8

[1] Estimated values.

Table 5.1: Data about Case Studies

5.1 Evaluation Criteria

Based on the experimental setting, we identified three criteria for assessing our algorithm.

Definition 2. *Correctness is the average percentage of code overlap between each original input product and its corresponding reconstructed product (see Section 4 Step 5) using the extracted traceability information.*

$$Correctness = \frac{1}{n} * \sum_{i=1}^{n} \frac{|P_i.c \cap P_i'.c|}{|P_i.c \cup P_i'.c|}$$

where n is the number of products used as input, P_i is an original input product and P_i' is the corresponding reconstructed product.

If all original products are reconstructed in this manner and the comparison shows that they are equal then the extracted traceability information must be correct, at least for the given products. That would mean a 100% value for this metric.

Our algorithm may only err in incorrectly assigning a code element to a module. Consider now that there are two modules and let us assume that our algorithm incorrectly assigns a code element to $\texttt{module}_1$ although it belongs to $\texttt{module}_2$. For any product that includes both modules, this error would remain undetected because together they exhibit the right code elements. However, for any product that contains one of the modules only, the product would either be missing a code element or have an extra code element. Hence, the need to assess correctness by reconstructing and comparing all products used by the algorithm.

Definition 3. *Performance is the execution time of the traceability mining algorithm, not including the experimental setup such as the generation of products or the parsing of the original source code.*

The execution times were measured on an Intel® Core™ i5 Sandy Bridge with 8 GB of memory.

Definition 4. *Distinguishability is the average cardinality of all module sets whose respective associations contain code and at least one module.*

$$Distinguishability = \frac{1}{n} * \sum_{i=1}^{n} |association_i.m|$$

where n is the number of associations that contain code and at least one module and $association_i$ is such an association.

The optimal value for this metric is 1, meaning every association containing code has exactly one module.

The measure is important because our approach can only distinguish modules if one of them appears in at least one product in which the other doesn't. Consider, for example, mandatory features that all products must have. As they always appear together and never without each other, the corresponding modules and code cannot be distinguished.

5.2 Case Study: Video On Demand (VOD)

The *Video On Demand (VOD)* product line consists of simple video streaming applications. The feature model is given in Figure 5.1.

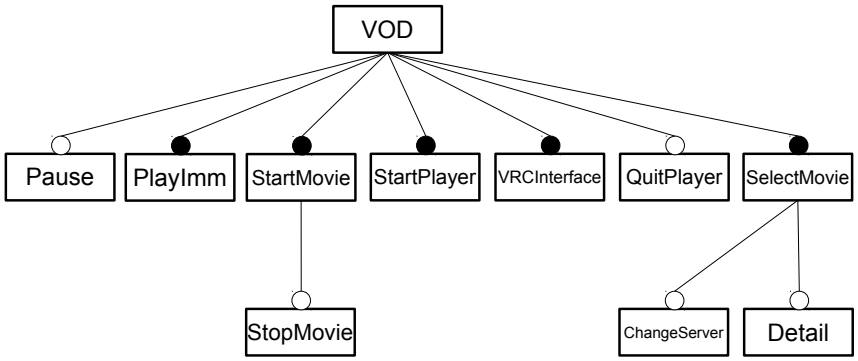

Figure 5.1: Feature Model for VOD

We generated all possible products and used them for evaluation. The achieved correctness was at 100% with a performance of 0.9 seconds. The distinguishability was at 63.8. The lower bound for the number of modules possible in an association due to mandatory features in this case study is at $2^6 - 1 = 63$.

5.3 Case Study: ArgoUML-SPL

The *ArgoUML-SPL* is the SPL for the UML Modelling Tool ArgoUML [Couto et al., 2011, ArgoUML-SPL]. The feature model is given in Figure 5.2.

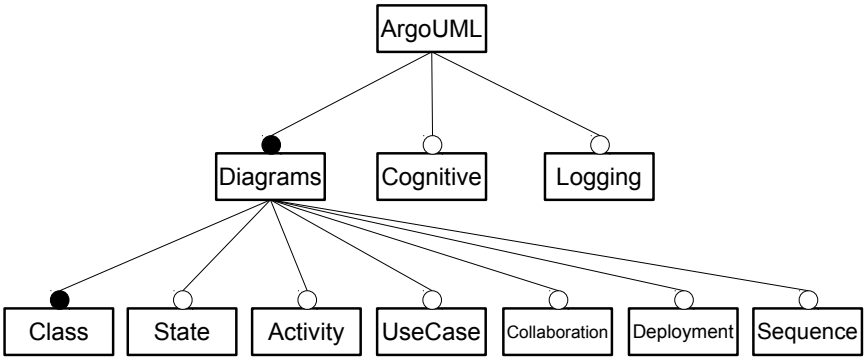

Figure 5.2: Feature Model for ArgoUML

Again all possible products were generated and used for the evaluation. The achieved correctness was at 99.4%. Only 150 code pieces out of 21128 could not be associated with the correct modules, in fact, they were not associated with any module at all, since for these code pieces multiple traces existed, which violates the unique trace assumption (see Section 5.5 Analysis). The distinguishability was at 7.8 modules per association containing code with the lower bound for the number of modules in an association at $2^3 - 1 = 7$.

5.4 Case Study: MobileMedia (MM)

The third case study we evaluated has 7 product variants obtained from a system called *MobileMedia (MM)*. In contrast to the previous case studies, each variant corresponds to an evolutionary step of the system development [MobileMedia-Lancaster]. The features for each product were assigned manually by inferring them from the corresponding paper [Figueiredo et al., 2008]. The number of features ranges from 6 for the smallest to 14 for the largest product.

With all 7 product variants as input the achieved correctness was at 99.6%. Only one piece of code could not be assigned to a module. Taking a closer look at this piece revealed that it was accidentally left over in one of the products where it was not needed anymore. It was correctly assigned after removing it from this product. So in a way our algorithm pointed us at a mistake in one of the original products which we then corrected.

The distinguishability for this case study was 3060.8. This is due to the very small subset of the possible products for this number of features used as input. Most of the modules are higher order derivatives that don't exist in the form of code anyways (see Section 5.5 Analysis). Removing all modules with an order higher than 1 (no interactions between more than 2 features) led to a distinguishability of 7.14 while having no influence on the correctness.

5.5 Analysis

In our experiment, we found that some pieces of code could not be associated with any module. The reason is that, generally speaking, such pieces of code appear in disjunctive features. For example, product $\texttt{P}_1$ uses piece of code `c` in feature `A`, product $\texttt{P}_2$ uses the same piece of code in feature `B`, while `c` is annotated with a condition to include it if feature `A` OR feature `B` is implemented. Even though the two products do not share any common feature they *do* share a common piece of code. Thus code c violates the assumption of a unique trace for every piece of code, because it is added by multiple disjunctive features and therefore would have to be traced to multiple modules.

Another reason for code not being associated with modules can be mistakes in the input products, for example when code was not removed from products where it didn't belong to anymore. In a way, our algorithm points out those mistakes.

In addition to the correct modules, pieces of code were often also associated with a large number of higher order derivatives that, if they existed (in the form of code), would be implemented in this code, which our system cannot know. So basically our algorithm indicates that such code can belong to any (or several) of these modules. In order to avoid that, one could set a threshold for the maximum order that derivatives may have. For example one could assume that in a certain product line, no more than 4 features interact (depending on the coupling). Therefore, derivatives with an order higher than 3 could be omitted. This may drastically improve both the performance as well as the

distinguishability and have almost no negative impact on the correctness if the threshold is chosen wisely. It also should be noted, that most of these derivative modules seemed to stem from the undistinguishable mandatory features. Most of the other derivatives could be separated and filtered out (there was no code associated with them). A possibility to avoid that, would be to represent all the mandatory features with one single representative feature, as they can't be separated anyway.

In Figure 5.3 the number of modules with respect to their number of interacting features (which is the order of the derivatives) are shown for VOD and ArgoUML. VOD contains more higher order derivatives than ArgoUML, partially because it has more mandatory features that cannot be distinguished. The highest order of derivatives that would have been possible for both is 10, as both happen to have 11 features. Figure 5.4 shows the number of extracted associations after each additional product that is considered. Only roughly the first 15% of the products provide new associations (this depends on the selected features of the products and in what order they are processed). Also the distinguishability does not improve anymore at a certain point as can be seen in Figure 5.5, but it takes a little longer than for the associations. Additional products only yield associations that do not contain code and therefore do not influence the distinguishability measure and also do not increase the total number of associations because they are grouped into one association. However, in some cases this may still be useful information to know that a module does not contain any code. Also should be noted, that the final value for the distinguishability and the final number of associations are reached pretty soon whereas the runtime increases linearly over the number of products as shown in Figure 5.6.

All this shows us, that it would by far not have been necessary to generate all possible product variants and all higher order derivatives, which would have dramatically decreased the runtime and wouldn't be possible in a real world scenario anyway.

MobileMedia is not included in the plots because our information about it is not complete as we did not generate the variants ourselves.

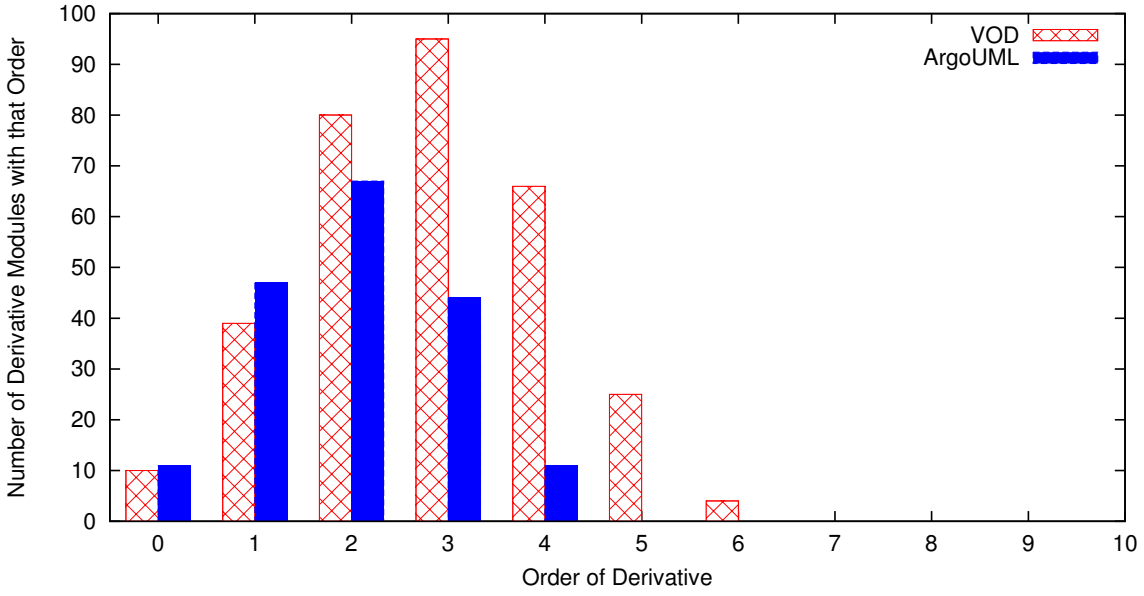

Figure 5.3: Number of Modules per order of Derivative for ArgoUML and VOD

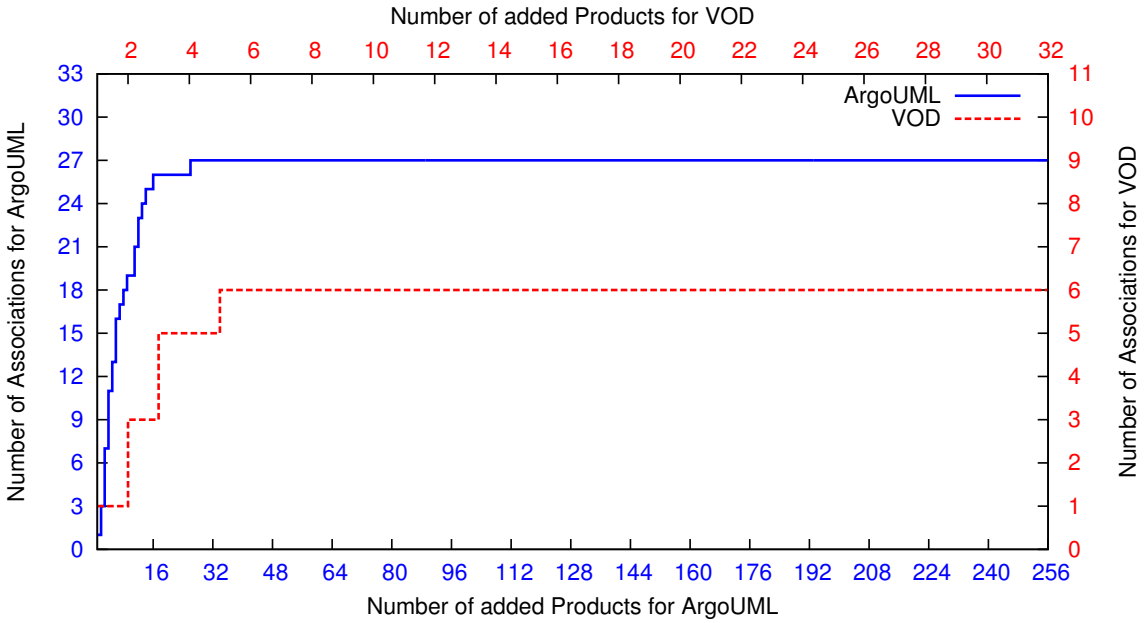

Figure 5.4: Number of Associations after each added Product for ArgoUML and VOD

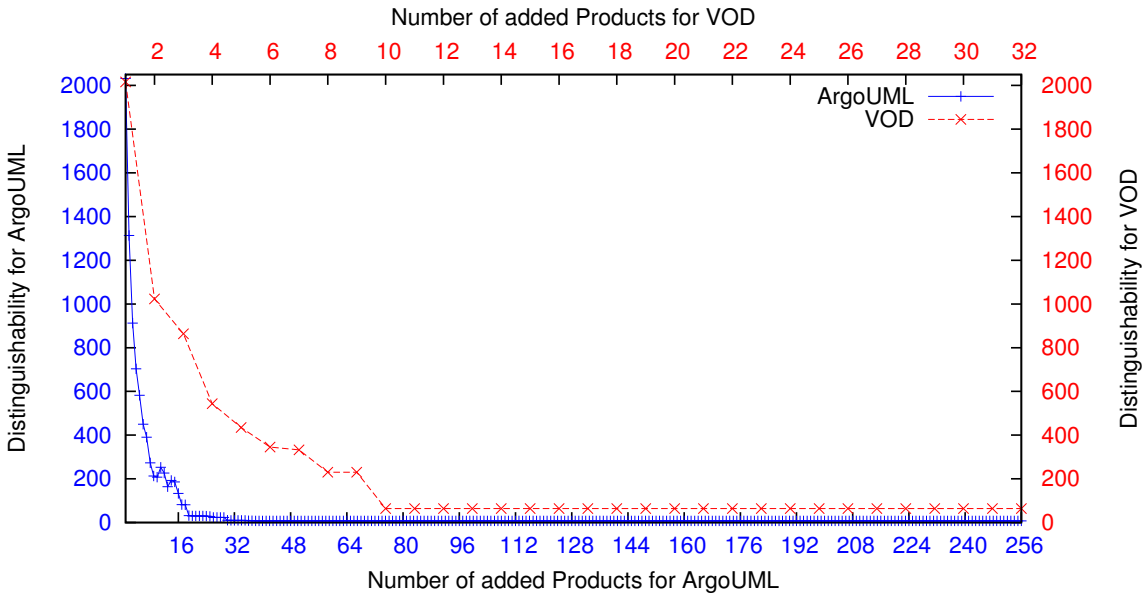

Figure 5.5: Distinguishability after each added Product for ArgoUML and VOD

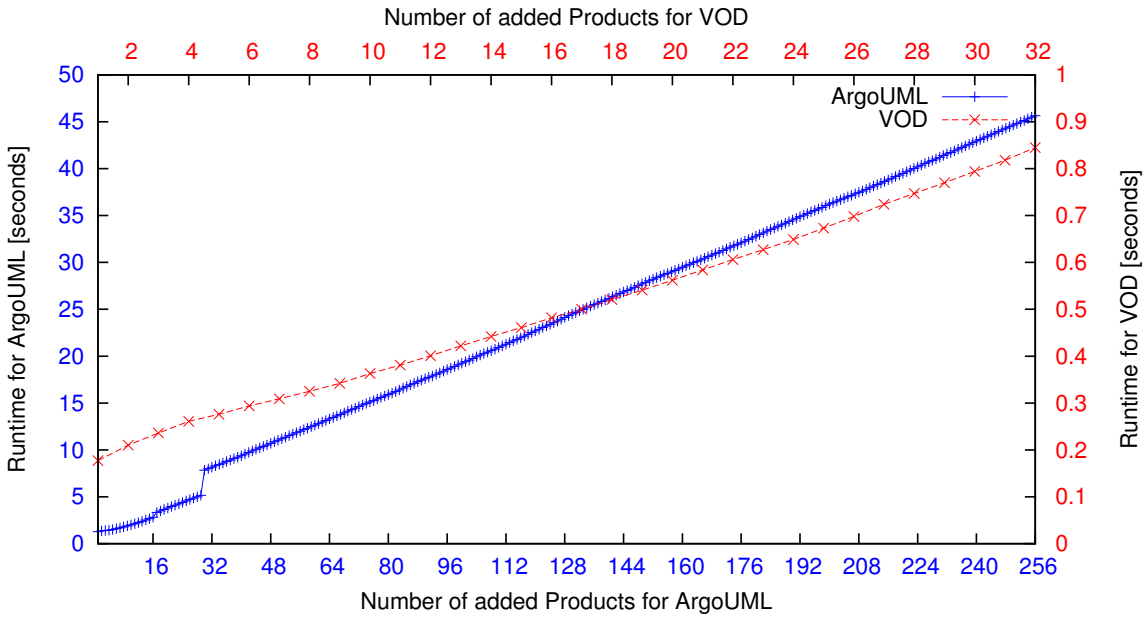

Figure 5.6: Runtime after each added Product for ArgoUML and VOD

Chapter 6

Related Work

Ziadi et al. present a partially automated approach to identify features from source code of product variants in order to help migrate software product variants into a product line. With the same goal in mind, our approach aims to extract traceability information from product variants about which the features are already known [Ziadi et al., 2012].

In the work of Kaestner et al. they examine the impact of the optional feature problem of product lines and survey different solutions and their trade offs [Kästner et al., 2009].

The work of Siegmund et al. aims to predict non-functional properties of SPL's products by generating and measuring a small set of products and approximating each feature's non-functional properties [Siegmund et al., 2011]. Our approach works similarly, only that we aim to extract each feature's source code instead of non-functional properties.

Liebig et al. analyse software projects with regards to disciplined and undisciplined preprocessor annotations and how undisciplined ones can be made disciplined [Liebig et al., 2011].

Schulze et al. analyse code clones in feature-oriented programming and whether it makes a difference if an SPL has been developed from scratch or refactored from legacy code, as we aim to do [Schulze et al., 2010]. Also they discuss the removal of such code clones. In [Schulze et al., 2011] they investigate the relationship between code clones and preprocessor annotations in form of a large case study.

Ribeiro et al. provide data on preprocessor usage to show to what extent feature dependencies occur in practice and compare Virtual Separation of Concerns and emergent interfaces in terms of maintenance effort [Ribeiro et al., 2011]. Their work relates to ours in that we also use preprocessors or template engines on annotated source code and we then model such feature dependencies with derivative modules.

Neves et al. discover and analyse product line evolution scenarios by mining SVN histories to then describe product line safe evolution templates [Neves et al., 2011].

The work of Rubin et al. aims at generating a product line out of related products [Rubin and Chechik, 2012]. Their focus is on identifying the common and variable parts of a software system using a model representation in order to combine them into a product line.

The work by Duszynski et al. analyses the source code of product variants based on text lines in order to visualize commonalities and variabilities. The goal is to assess the reuse potential in these software systems [Duszynski et al., 2011].

Koschke et al. aim to reconstruct the module view of a system using the reflexion method and then map implementation components to the modules using clone detection techniques [Koschke et al., 2009]. At this point our work utilises very simple mechanisms to compare source code. We argue, that our system can also benefit from using more sophisticated methods, like higher clone detection levels.

Chapter 7

Conclusions and Future Work

We introduced an algorithm to extract traceability information from feature to code in product variants, on the level of class fields and methods, with the purpose of helping their refactoring into software product lines. We evaluated our approach with three case studies of different sizes and complexity. More than 99% of the code pieces were correctly assigned in a matter of seconds, even with large products like the ones from the ArgoUML-SPL. The remaining code pieces that could not be assigned to modules violated our unique trace assumption. These violations were duly identified in our case studies.

As part of our future work we plan to:

- relax or eliminate the unique trace assumption to allow more accurate tracing and handle non-assigned code pieces, for example by looking at call dependencies between methods and field accesses (dependencies between code pieces in general),
- enhance the algorithm with static and dynamic analysis of programs. Instead of - or in addition to - collecting the code pieces from source code they could be collected by analyzing a program during runtime (for example by using the *Java Debug Interface (JDI)* [Oracle, c]) and only include code pieces that were actually executed,
- extend the algorithm to work on granularity below method level. This would open the possibility of leveraging advanced clone detection methods, and
- perform a more detailed evaluation of our algorithm with more case studies.

List of Figures

List of Tables

Bibliography

ArgoUML-SPL. *ArgoUML-SPL Project*. URL `http://argouml-spl.tigris.org/`. (accessed July 15, 2012).

ASF. *Apache Velocity Project*. URL `http://velocity.apache.org/`. (accessed July 15, 2012).

P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting software product lines: A case study using conditional compilation. In T. Mens, Y. Kanellopoulos, and A. Winter, editors, *CSMR*, pages 191–200. IEEE Computer Society, 2011. ISBN 978-0-7695-4343-7.

E. Denney and U. P. Schultz, editors. *Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011, Portland, Oregon, USA, October 22-24, 2011*, 2011. ACM. ISBN 978-1-4503-0689-8.

S. Duszynski, J. Knodel, and M. Becker. Analyzing the source code of multiple software variants for reuse potential. In Pinzger et al. [2011], pages 303–307. ISBN 978-1-4577-1948-6.

E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. C. Ferrari, S. S. Khan, F. C. Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 261–270. ACM, 2008. ISBN 978-1-60558-079-1.

E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. Reverse engineering feature models from programs' feature sets. In Pinzger et al. [2011], pages 308–312. ISBN 978-1-4577-1948-6.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. S. Batory, and G. Saake. On the impact of the optional feature problem: analysis and case studies. In D. Muthig and J. D. McGregor, editors, *SPLC*, volume 446 of *ACM International Conference Proceeding Series*, pages 181–190. ACM, 2009.

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.

J. Kropf. *JavaPP Project*. URL `http://www.slashdev.ca/javapp/`. (accessed July 15, 2012).

J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In P. Borba and S. Chiba, editors, *AOSD*, pages 191–202. ACM, 2011. ISBN 978-1-4503-0605-8.

J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 112–121, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134303. URL `http://doi.acm.org/10.1145/1134285.1134303`.

MobileMedia-Lancaster. *MobileMedia*. URL `http://sourceforge.net/projects/mobilemedia/`. (accessed July 15, 2012).

L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulesza, and P. Borba. Investigating the safe evolution of software product lines. In Denney and Schultz [2011], pages 33–42. ISBN 978-1-4503-0689-8.

L. Northrop. *Software Product Lines Essentials*, 2008. URL `http://www.sei.cmu.edu/library/assets/spl-essentials.pdf`. (accessed July 22, 2012).

Oracle. *Java Technology*, a. URL `http://www.oracle.com/us/technologies/java/overview/index.html`. (accessed July 20, 2012).

Oracle. *Java Standard Edition API Specification*, b. URL `http://docs.oracle.com/javase/6/docs/api/`. (accessed July 20, 2012).

Oracle. *Java Platform Debugger Architecture*, c. URL `http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html`. (accessed July 15, 2012).

M. Pinzger, D. Poshyvanyk, and J. Buckley, editors. *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, 2011. IEEE Computer Society. ISBN 978-1-4577-1948-6.

M. Ribeiro, F. Queiroz, P. Borba, T. Tolêdo, C. Brabrand, and S. Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In Denney and Schultz [2011], pages 23–32. ISBN 978-1-4503-0689-8.

S. Richard. *Source Code Analysis Using Java 6 APIs*. URL `http://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html`. (accessed July 15, 2012).

J. Rubin and M. Chechik. Combining related products into product lines. In J. de Lara and A. Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2012. ISBN 978-3-642-28871-5.

S. Schulze, S. Apel, and C. Kästner. Code clones in feature-oriented software product lines. In E. Visser and J. Järvi, editors, *GPCE*, pages 103–112. ACM, 2010. ISBN 978-1-4503-0154-1.

S. Schulze, E. Jürgens, and J. Feigenspan. Analyzing the effect of preprocessor annotations on code clones. In *SCAM*, pages 115–124. IEEE, 2011. ISBN 978-1-4577-0932-6.

N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, editors, *SPLC*, pages 160–169. IEEE, 2011. ISBN 978-1-4577-1029-2.

F. J. van d. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane. Feature identification from the source code of product variants. In T. Mens, A. Cleve, and R. Ferenc, editors, *CSMR*, pages 417–422. IEEE, 2012. ISBN 978-1-4673-0984-4.