University of Magdeburg

School of Computer Science

Master's Thesis

Automating the Synchronization of Software Variants

Author:

Tristan Pfofe

December 21, 2015

Advisors:

Prof. Gunter Saake

Dipl.-Inform. Wolfram Fenske

University of Magdeburg · Technical & Business Information Systems

Dr.-Ing. Thomas Thüm

TU Braunschweig · Software Engineering and Automotive Informatics

Pfofe, Tristan:
Automating the Synchronization of Software Variants
Master's Thesis, University of Magdeburg, 2015.

Abstract

To overcome the increasing number of customer requirements, companies develop related software variants for individual customers. In industrial software development, it is common practice to apply a practice called clone-and-own, where a software engineer develops a new variant by copying and adapting existing variants. For this purpose, software engineers often map each variant to a branch of a version control system. If the number of variants raises, the number of branches and the effort to synchronize changes between variants will increase, because an extension or bug fix of one variant has to be transferred to other variants. Thus, clone-and-own has less up-front investments but the maintenance effort rapidly grows with an increasing number of variants. However, introducing a software product-line causes high initial costs and implementation effort which make the development of few variants unprofitable. Furthermore, if variants are developed with clone-and-own and a sufficient number of variants is reached, then the transition to a product line will cause high migration effort.

To bridge the gap between the development of single systems and product lines, we present *VariantSync*, a light-weight and language-agnostic concept to enhance variant development with clone-and-own. *VariantSync* has the goal to make variant development more efficient by automating the synchronization of variants. In addition, *VariantSync* accumulates domain knowledge to support the transition to a product line. We show that *VariantSync* is applicable by prototypically implementing this concept as an *Eclipse* plug-in. Moreover, we use *VariantSync* to simulate variant development over a distinct period of time and share main results.

Acknowledgements

I would like to thank Thomas Thüm for his comprehensive support and for many interesting and productive discussions. He always took the time to discuss my questions and gave me many advices that substantially improved my writing and the result of this thesis. I also thank Wolfram Fenske who carefully read my drafts and gave me many helpful comments that significantly improved this thesis. With Thomas and Wolfram, I had two very dedicated advisors.
Finally, I thank my family and my girlfriend for their support, especially at the final stage of this thesis.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Software is an essential part and a constant companion in everyone's life. Due to the exploding amount and variety of powerful hardware - from smart-phones, wearable computers and tablets via notebooks and computer workstations through to high available data centers - software is not only used in industrial and business areas, but also in a wide field of applications of daily life and everyday's activities. Software appears in all imaginable fields, e.g., intelligent systems in vehicles, systems trading at stock exchanges and systems to communicate all over the world. There is almost no modern product that gets along without software [vdLSR07]. Today and, highly probably, in the future, software conditions our existence [KD11]. Or, in words of *C++* designer Bjarne Stroustrup: "Our civilization runs on software." [Str12]
So, the need for software as well as the dependence on software is growing rapidly [KD11]. More than 80% of germany's exports depend on modern information technologies and software evolves to a dominating factor for products and services [BKPS04]. Software development has the challenging task to satisfy the society's and industrial needs for software. Software developers have to deliver code that is correct and efficient [Str12] because software becomes the key asset for modern and competitive products [vdLSR07]. Thereby, software development both addresses different software in different fields of application and similar requirements in similar fields of application, such as variants of flight control systems by the National Aeronautics and Space Administration (NASA) [Käs10], variants of the Linux kernel [SLB$^+$10] or variants of Hewlett-Packard printer firmware [ABKS13]. To develop similar software products, also known as *variants* or *program family*, in an efficient way, the reuse of existing code is an essential task to improve the software development process by developing faster, with lower costs and with less errors. Depending on the number of required variants, there are two ways to create and maintain program families. Using clone-and-own approaches is an easy solution to create a low number of variants, whereas for a high number of variants, software systems targeting a similar requirement space are often developed as a product line.
Product lines are the solution to fulfill today's requirements of software systems and

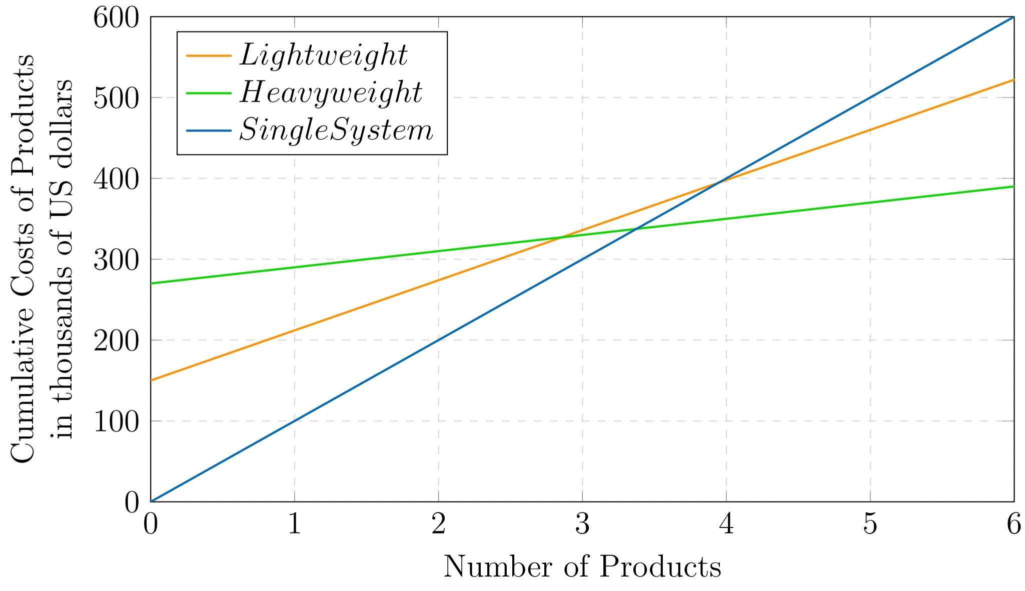


Figure 1.1: Comparison of Launching Strategies for Product-Line Development and Single-System Development [MNJP02]

software intensive products regarding comprehensive functionality and flexibility with low costs [BKPS04]. Besides reduced development and maintenance costs [vdLSR07], a product line also promises tailor-made products to individual customers, improved software quality by using standardized parts to construct different products, and time to market benefits by assembling existing, corresponding parts to quickly produce required products [ABKS13]. The idea of a product line is managing a family of software systems in a reuse-based way [AM14]. In more detail, software product-line engineering approaches support variant development sharing a common base of code artifacts [PBvdL10]. Furthermore, software variants are characterized in terms of *features*. Different variants share several common features and differ among other features [TKES11]. Several product-line engineering approaches generate variants based on a selection of features. Hence, product-line engineering approaches need a strategic and predictive planning of possible and required variants.

Dependent on initial and future product costs, there are different ways to initiate a product line. In comparison with the effort of single-system development, Figure 1.1 shows possible strategies to create a product line. Using heavyweight strategies, a product line will build from scratch before variants can be generated. Thus, product line's initial product costs are significantly higher than initial development costs of a single product in single-system development [MNJP02]. Besides, lightweight strategies are transition strategies which analyze commonalities and differences of existing variants to migrate them to a product line. These strategies demand less up-front investments, but they take longer to reduce cumulative costs [MNJP02].

Due to the fact that the development of a product line requires a high effort, building a product line is only efficient to generate and manage a sufficient number of variants. A product line only pays off in the long term when multiple tailored variants are developed [ABKS13]. So, to create a low number of variants, the single development of each variant is the more efficient way. For this purpose, clone-and-own is a common tool-driven approach to develop variants. Applying clone-and-own means to copy code fragments from existing variants and adapt them as needed for the development of a new variant. Clone-and-own approaches require minimal initial costs to develop a new variant from scratch. As further benefits, clone-and-own does not need preplanning and developed variants are independent from each other [KG08]. Moreover, it can be realized with well-known developer-tools, for example with the use of branching functionality of version control systems. However, clone-and-own approaches are often considered harmful to code quality [DRB$^+$13, LLMZ06]. Code clones lead to code smell in software systems and reduce software quality and maintenance [Fow00]. For the reason that variants are independent, changes need to be tracked and propagated to appropriate variants which is an error-prone process. For example, if an error appears in one copied code fragment, the error has to be fixed multiple times in each pasted code fragment. So, mid- and long-term maintenance effort increases heavily with each further variant. Despite these facts and due to their simple and cheap usage, clone-and-own approaches are still popular in many industrial organizations [DRB$^+$13]. Nevertheless, in the short term, clone-and-own is an efficient way to develop a fixed number of variants.

Motivation

Companies have three reasons to use clone-and-own instead of introducing a product line. First, the number of required variants is not always known at the beginning of the development. Introducing a product line would be a risky task that could not pay off if the number of variants remains small. Second, most companies have no knowledge and experiences in product-line development. So, introducing product-line methodology by teaching developers would cause additional costs. Third, single-system development has a comprehensive and well-engineered tool support, like version control systems and integrated development environments. However, tool support for product-line development is not as mature as tool support for single-system development.
Depending on the number of variants, single-system engineering provides appropriate ways to develop variants. The development of few variants can be supported by several approaches: (1) To apply clone-and-own development, version control systems can be used to create several variants in branches by representing each variant as a branch. From a common starting point, separate development paths diverge, so that there are several latest configurations in the repository, called variants [Bre04]. Using this approach, error-proneness for duplicated code is still given because changes on one branch need manually synchronized with other branches. (2) Other approaches detect similar code fragments in different variants and try to synchronize these fragments [Luo12]. (3) And yet other approaches combine version control systems and code clone detection approaches [Sch15]. All these approaches have in common that they can neither effectively maintain several software variants if variants are added in the future (1) nor support the migration to a product line (2, 3).

As opposed to single-system engineering approaches, product lines efficiently maintain variants if the number of variants is not fixed and variants are added in the future. Unfortunately, product lines only pay off for a sufficient number of variants. The transition to a product line is a time-consuming task that causes high effort [PBvdL10]. If the number of variants is undetermined and only one variant is occasionally added, then the transition is more expensive than only adding one further variant.
In summary, there is a gap between the development of a fixed and typically small number of variants using single-system engineering approaches and a sufficient and increasing number of variants using product-line engineering approaches.

Goal of this Thesis

The goal of this thesis is to enhance variant development with clone-and-own to develop and maintain a small number of variants which could increase over time. In particular, we want to make the development of few variants more efficient, faster, error-reduced and, hence, with lower costs. A further goal is to reduce the effort for the migration to a product line using a light-weight migration strategy.
For these purposes, we develop, present and discuss *VariantSync*, a strategy to support variant development with clone-and-own by efficiently synchronizing variants and accumulating domain knowledge. *VariantSync* detects changes that occur during variant development, tags these changes to *features* and automatically computes valid synchronization targets. We synchronize changes between variants from two different points of view. *Target-focused synchronization* focuses on change synchronization with a single variant, where a variant is synchronized with all relevant changes that occur on other variants. In contrast, *source-focused synchronization* propagates changes of one variant into appropriate target variants. *VariantSync* extends a prior concept that was developed to support variant development by synchronizing fine-granular changes between variants [Luo12].
In summary, our contribution is as follows. We provide:

- a light-weight and language-agnostic concept to enhance variant development with clone-and-own and support the migration to a product line,
- a prototypical tool to reduce the gap between tool support for single-system engineering and product-line engineering, and
- an evaluation on a simulated case study based on the development history of the *DesktopSearcher* product line.

Structure of this Thesis

In Chapter 2, we introduce the background on product lines, variant development with clone-and-own using version control systems, and merging.
In Chapter 3, we develop the *VariantSync* concept to automate the synchronization of variants. We first distinguish the development of few and many variants by comparing

clone-and-own approaches and product-line approaches. Then, we present a concept to automate the computation of synchronization targets and introduce an automated variant synchronization from different points of view.

In Chapter 4, we show that *VariantSync* is applicable. We first discuss version control systems and integrated development environments as possible platforms to perform the synchronization of variants. Then, we prototypical implement *VariantSync* as an *Eclipse* plug-in.

In Chapter 5, we evaluate our implementation of *VariantSync* with a simulated case study using the *DesktopSearcher* product-line. We cover the development of five variants over a distinct number of development steps and evaluate the degree of automated synchronization as well as code to *feature* mapping.

In Chapter 6 and Chapter 7, we present related work, summarize this thesis and point out our contributions. We further list suggestions for future work in Chapter 8.

2. Background

The subject of this thesis is to bridge the gap between single-system engineering and product-line engineering to develop and maintain a small number of variants which could increase over time. In this chapter, we give a brief introduction into the main topics and techniques of this thesis. For this purpose, we introduce main concepts of product-lines in Section 2.1. In Section 2.2, we explain clone-and-own strategies to support the development of product families, point out how to develop variants using these strategies and present version control systems to apply clone-and-own. Finally, we introduce code merging strategies and techniques in Section 2.3.

Running Example

To explain variant development and product-line technologies in scope of this thesis, we introduce the *DesktopSearcher* product line as a running example for this chapter. *DesktopSearcher* is a student project[1] that was developed as part of the product-line lecture at University of Magdeburg. This product line has 22 features and allows the generation of 462 different variants [Luo12]. Among others, typical functionality provided by this product line is to search specific contents inside *HTML*, *TXT* or *LATEX* files located in single or multiple directories. Furthermore, variants of *DesktopSearcher* can either run on *Windows* or *Linux* operation systems and either provide a graphical or command-line user interface.

2.1 Software Product-Line

Software product-line engineering describes the development of multiple similar software systems in a domain sharing a common code base [PBvdL10]. These similar systems share certain commonalities and represent a product family. A product line

[1]developed by Reimar Schröter, Sebastian Bress, Alexander Grebhahn, Sönke Holthusen

generates similar software products which are tailored to the specific needs of different customers and use cases. For this purpose, product lines introduce the concept of mass customization to software products which is exemplary known from automobile industry [ABKS13, Kru06]. As opposed to the development of a single one-size-fits-all solution that covers all customer needs in a mass market, a product line provides tailor-made solutions for different customers [Käs10]. The idea is to develop related software products in a coordinated manner, so that commonalities between software products are developed only once. The reuse of these software artifacts prevents the individual development of each product for each customer from scratch. A software product that is derived from a product line is called *variant*.
Traditional software engineering processes do not provide solutions to introduce the concept of mass customization to software products. Instead of collecting requirements for one target system and designing and implementing this system in consecutive phases or agile cycles, software engineers have to analyze a variety of similar desired systems. For this purpose, product-line engineering sets its focus on a well-defined and well-scoped domain [ABKS13, Käs10].
A product line is defined as follows:

> A software product-line is a set of software-intensive systems sharing a common, managed set of features that satisfy 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 [CN01].

In this context, a feature describes commonalities and differences between software variants. A feature is defined as:

> A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [KCH$^+$90].

Features are used to characterize the domain of a product line. In Section 2.1.1, we explain the term domain. Moreover, features cannot be combined in an arbitrary fashion to describe all characteristics of a software system. To describe valid combinations of features, we introduce the concept of feature models in Section 2.1.2. In Section 2.1.3, we explain different approaches to implement variability in a product line. Furthermore, we describe a two-phase approach to handle variability, systematically reuse implementation artifacts and generate tailor-made software variants in Section 2.1.4.

2.1.1 Domain Knowledge

As opposed to single-system development, the development of multiple variants that are similar, but not identical, requires a new look on software systems. Already in 1990, Prieto-Diaz pointed out that the systematic discovery and exploitation of commonalities across related software systems is a fundamental technical requirement for achieving successful software reuse [PD90]. As one solution, domain analysis can be

```
class Parser {
    [...]

    public boolean indexHTMLFiles(String filename, IndexWriter writer) {
        String content = HTML.getFileContents(filename);
        [...]
        HTMLParser parser = new HTMLParser(new StringReader(content));
        [...]
    }

    public void indexTXTFiles(String filename, IndexWriter writer) {
        String content = PlainTXT.getFileContents(filename);
        [...]
        TXTParser parser = new TXTParser(new StringReader(content));
        [...]
    }

    [...]
}
```

Listing 2.1: *DesktopSearcher* with a Parser for HTML and TXT files

applied to meet this requirement [KCH+90]. Domain analysis describes requirements and behavior of software inside a domain. Thereby, a domain is an area of knowledge which includes concepts understood by practitioners in that area and which includes knowledge to describe the building of software systems in that area [CE00, ABKS13]. Domain knowledge consists of a conceptual model that contains concepts and relations between these concepts [SAA+00].
Reuse is defined as using previously acquired concepts or objects in a new situation [PD89]. To reuse software during development process, objects of reusability (reusable artifacts) support developer to reuse concepts or objects while creating new software. In this context, reusable artifacts are any information which a developer needs to create software [Fre83]. Domain knowledge comprises reusable artifacts [Cyb96] and is essential to support software reuse.

Implicit Domain Knowledge

In software development, code fragments have an implicit meaning for the developer. If a developer writes code fragments, each piece of code has an importance and belongs to a distinct concern or requirement, so that the developer already relies on domain knowledge when writing code fragments. While creating a new variant, the developer has implicit domain knowledge and he can use this knowledge to prevent duplicated effort on the same concern or requirement.
For example, one variant of *DesktopSearcher* implements a parser for two different file types: *HTML* and *TXT*. Assume, a developer has to create a new variant implementing parsers for *HTML* and *LATEX*. In Listing 2.1, class *Parser* in variant one contains code to parse *HTML* and *TXT* files. The developer starts developing variant two. Thereby,

```
class Parser {
    [...]

    public boolean indexHTMLFiles(String filename, IndexWriter writer) {
        String content = HTML.getFileContents(filename);
        [...]
        HTMLParser parser = new HTMLParser(new StringReader(content));
        [...]
    }

    public void indexLATEXFiles(String filename, IndexWriter writer) {
        String content = LATEX.getFileContents(filename);
        [...]
        LATEXParser parser = new LATEXParser(new StringReader(content));
        [...]
    }

    [...]
}
```

Listing 2.2: *DesktopSearcher* with a Parser for HTML and LATEX files

class *Parser* in variant two has to provide parsers for *HTML* and *LATEX* files. Instead of implementing the whole class from scratch, the developer profits from his domain knowledge and copies selected code from class *Parser* of variant one for the needs of class *Parser* in variant two to implement a parser for *HTML* files. Hence, the developer only has to implement a parser for *LATEX* files in Listing 2.2.
Variant development with clone-and-own is an unsystematic approach that principally does not support reuse. Nevertheless, developers can use implicit domain knowledge to systematically reuse software artifacts while creating a new variant.

Explicit Domain Knowledge

In contrast to developers, tools do not have implicit domain knowledge. Without further help, they are not able to acquire domain knowledge the same way developers acquire information. This is a problem for the development of variants with tool support, for instance to automate the synchronization of variants. Tools are able to check syntax and semantic of code files, but they cannot retrieve domain knowledge without additional information. Annotating or modularizing code belonging to a certain domain characteristic makes domain knowledge explicit in variant code [ABKS13]. For example, tools do not have domain knowledge about the method *indexHTMLFiles* in Listing 2.3. However, in Listing 2.4, the preprocessor annotation *#if HTML* explicitly maps domain knowledge to a code fragment by indicating that *indexHTMLFiles* belongs to the domain characteristic *HTML*.

```
class Parser {
    [...]

    public boolean indexHTMLFiles(String filename, IndexWriter writer) {
        String content = HTML.getFileContents(filename);
        [...]
        HTMLParser parser = new HTMLParser(new StringReader(content));
        [...]
    }

    [...]
}
```

Listing 2.3: Class without Domain Knowledge

```
class Parser {
    [...]

    #if HTML
    public boolean indexHTMLFiles(String filename, IndexWriter writer) {
        String content = HTML.getFileContents(filename);
        [...]
        HTMLParser parser = new HTMLParser(new StringReader(content));
        [...]
    }
    #endif

    [...]
}
```

Listing 2.4: Class with Domain Knowledge as an Annotation

2.1.2 Feature Modeling

Features represent domain abstractions [ABKS13] and distinguish products of a product line. Different combinations of features lead to different products. However, not all features can be arbitrarily combined [Thü15]. For example, a product line for the *Linux* kernel provides support for x86 and x64 processor architectures, but it is not allowed to choose both processor architecture options for the same product.
A feature model describes the features and their relationships and documents the variability of a domain [ABKS13]. It describes valid combinations of features of a product line [KCH+90] and is represented as a hierarchically arranged set of features [Bat05b]. Furthermore, a feature model may also define dependencies between features with cross-tree constraints. Moreover, feature diagrams are graphical representations of feature models [KCH+90]. A feature diagram visualizes the features of a feature model in a tree hierarchy. Each node of this tree is labeled with a feature name. The diagram includes mutual relations between features and defines that each feature may have child features which are mandatory, optional or belong to an *or*-group or an *alternative*-group. [Thü15, ABKS13].

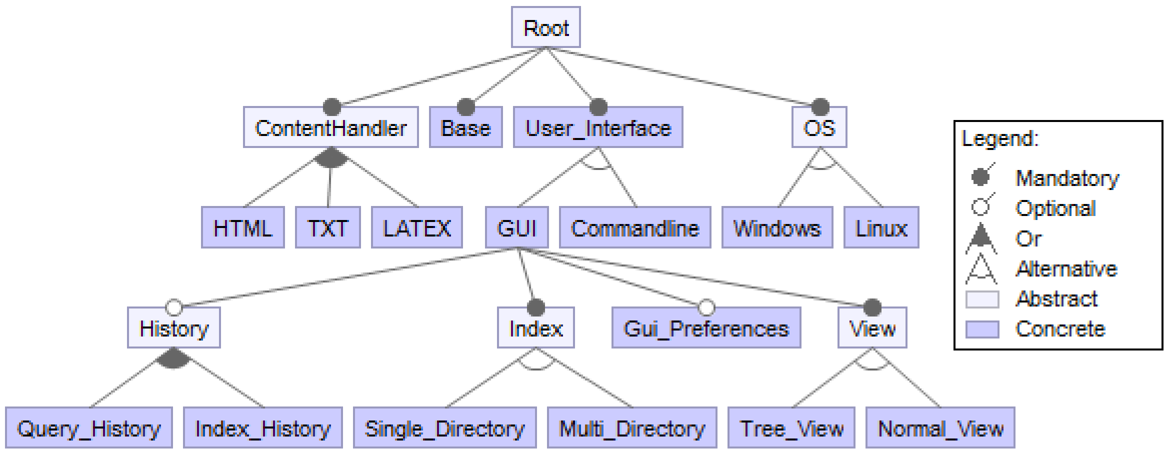


Figure 2.1: Feature Diagram of *DesktopSearcher* Product Line

Figure 2.1 shows an example feature diagram representing a feature model for the *DesktopSearcher* product line. For example, a *DesktopSearcher* variant can parse different file types to search for specific text. The features *HTML*, *TXT* and *LATEX* implement a parser for the different file types. Furthermore, feature *User_Interface* determines whether the *DesktopSearcher* variant either has a graphical or command-line user interface modeling features *GUI* and *Commandline* as an *alternative*-group. The graphical user interface shows the *Index* for either *Single_Directories* or *Multi_Directories* as well as either lists files as a *Tree_View* or a *Normal_View*. Moreover, feature *GUI* has mandatory as well as optional features. For example, features *History* and *GUI_Preferences* are optional because it is possible to search files without indexing files, saving search requests in a *Query_History* or configuring the graphical user interface.

Finally, a *DesktopSearcher* variant can either run on *Windows* or *Linux* operating systems.
Besides, propositional formulas are a further representation of feature models. Each feature is mapped to a boolean variable and the propositional formula is only true, if the selection of features is valid [Thü15]. Feature diagrams can be automatically transformed into propositional formulas [Bat05b].
Unfortunately, features often do not work isolated of other features. On the one hand, they typically interact in a positive and intended way to exchange information, to reuse functionality of other features, to accomplish a task in collaboration or to refine the behavior of other features [ABKS13]. However, on the other hand, critical and inadvertent feature interactions can cause erroneous behavior and result in undesired system states [ABKS13]. Assume, a feature works well in a given system. If this feature is combined with other features, it could exhibit inadvertent behavior [ABKS13]. A feature interaction between two or more features is defined as:

> [..] an emergent behavior that cannot be easily deduced from the behaviors associated with the individual features involved [ABKS13].

Hence, the feature-interaction problem defines that it is a difficult task to predict mutual interactions when independently developed features are combined [ABKS13].
Finally, feature interactions can be described as a boolean combination of features written as a logic expression, called feature expression [ALHM+11]. Figure 2.2 shows a two-way interaction of features f_1 and f_2. The feature expression $f_1 \wedge f_2$ describes the interaction between both features. This code has dependencies to both feature f_1 and feature f_2.

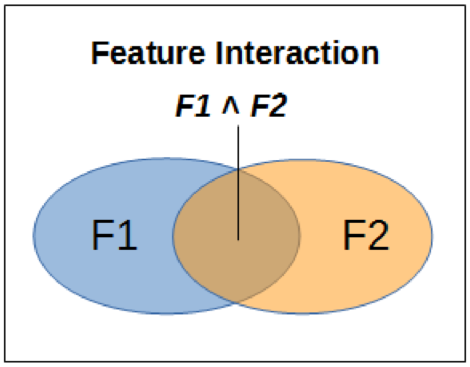


Figure 2.2: Visualization of a Two-Way Feature Interaction

2.1.3 Variability Implementation Techniques

On the one side, feature models are one possibility to model variability on the domain level. On the other side, variability is implemented in the code base on the application level. Besides, variability is defined as:

> the ability of a software system or artifact to be efficiently extended, changed, customized or configured for the use in a particular context [SvGB05].

Variability implementation mechanisms can be divided in composition-based and annotation-based mechanisms. The difference between both mechanisms is how they realize the separation of concerns to implement features [ABKS13].
Composition-based mechanisms provide a physically separation of concerns [ABKS13]. All code fragments and other kinds of artifacts, which belong to a feature, are separated in a cohesive unit. To realize modularized implementations, programming language concepts, like aspect-oriented programming or feature-oriented programming, provide separation techniques for crosscutting concerns [ABKS13]. Composition-based mechanisms usually allow coarse-grained extensions. They typically define points that can be extended by features to introduce new classes, methods or fields, or to extend methods by using method refinement or method overriding techniques [ABKS13].
For example, using frameworks is a possible implementation technique realizing composition-based mechanisms. Frameworks are a set of abstract and concrete classes which can be extended on variation points to adapt them for special needs [ABKS13]. In Listing 2.5 on the facing page, a *DesktopSearcher* can be applied on *Windows* as well as *Linux* operation systems with different path separators in file paths. Each operation system is represented by a feature. We implement variability in class *ContentHandler* using the black-box framework approach. The *ContentHandler* retrieves functionality to determine the path separator using a plug-in, where the *ContentHandler* does not know the implementation of the plug-in. From the *ContentHandler's* point of view, the plug-in is a black box. Decoupled by the *PathPlugin* interface, the concrete implementation to return the path separator in *Windows* or *Linux* operation systems is realized with a *WindowsPlugin* or a *LinuxPlugin*. Dependent on which plug-in implementation is set in the constructor of the *ContentHandler*, the *ContentHandler* initializes the path separator.
As opposed to composition-based mechanisms, annotation-based mechanisms virtually separate concerns. All features are implemented in the same code base and annotations map code fragments to features. One possibility to implement annotation-based variability is to use a preprocessor. A preprocessor modifies code before compilation. Depending on the feature configuration, it removes annotated code from the single code base to create a custom product. Annotation-based mechanisms allow extensions at arbitrary level of granularity [ABKS13].
To implement the path separator depending on the operating system by applying an annotation-based mechanism, we use the *antenna* preprocessor. *Antenna* integrates into code editors for *Java* and preprocessor directives are directly written as comments in the editor [ABKS13]. The functionality to set system-specific path separators is wrapped with conditional-compilation directives *#if feature_name* and *#endif*. Before compilation starts, the *antenna* preprocessor comments out code fragments, whose annotated features are not selected in the feature configuration. In the feature configuration of the *DesktopSearcher* variant, the *Windows* operating system is chosen. So, *antenna* has commented out the if-branches that are annotated with feature *Linux*.

```
interface PathPlugin {
    String getPathSeparator();
}

class WindowsPlugin implements PathPlugin {
    public String getPathSeparator() {
        return "\\";
    }
}

class ContentHandler {
    private PathPlugin plugin;
    private String pathSeparator;

    public ContentHandler(PathPlugin plugin) {
        this.plugin = plugin;
    }

    public void initPathSeparator() {
        pathSeparator = plugin.getPathSeparator();
    }

    [...]
}
```

Listing 2.5: Black-Box Framework Example

As we see in Listing 2.5 and Listing 2.6, frameworks only allow coarse-grained extensions inserting methods, whereas preprocessors allow fine-granular extensions changing the single line in method *initPathSeparator* of class *ContentHandler*. In the scope of this thesis, we need to handle coarse-grained as well as fine granular changes. To support change synchronization between variants, we need to detect changes on variants. For example, if a developer adds or removes files or functions inside files, then he performs coarse-grained changes. In contrast, developers often perform fine-granular changes by adapting single lines inside a file during development.

2.1.4 Domain and Application Engineering

In Section 2.1.1 and Section 2.1.2, we introduced how to describe variability of software systems on the domain level. Furthermore, we explained how to implement variability on the application level in Section 2.1.3. One goal of this thesis is to support the transition of multiple variants to a product line. For this purpose, we explain how a product line manages variability and provides tailored variants in this section.
A product line must fulfill requirements of multiple customers in a domain including current customers and potential future customers [Käs10]. Moreover, a product line needs to generate tailored variants. Feature-oriented software development fulfills these requirements. Feature-oriented software development consists of a *domain engineering* process and an *application engineering* process. The *domain engineering* process builds

```
class ContentHandler {

    private String pathSeparator;

    public void initPathSeparator() {
        // #if Windows
        pathSeparator = "\\";
        // #endif

        // #if Linux
        // @ pathSeparator = "/";
        // #endif
    }

    public String getPathSeparator() {
        return pathSeparator;
    }

    [...]
}
```

Listing 2.6: Preprocessor Example

the product-line architecture consisting of reusable core assets and features. First, developers analyze the domain and its requirements. Then, they describe possible variants in features determining commonalities and differences between the variants. Finally, developers implement the product line with the aim that variants can be generated from common and variable artifacts. The *domain engineering* process is performed in iterations to first set up the product line and further extend the product line, while the *application engineering* process is performed for every variant separately. *Application engineering* focuses on the derivation of new and tailored variants to satisfy the needs of particular customers. Furthermore, *application engineering* has the goal to reuse artifacts from *domain engineering* whenever it is possible [ABKS13, Käs10].
Figure 2.3 on the next page gives an overview about the *domain* and *application engineering* process as well as the *problem* and *solution space*. The *problem space* contains the processes of *domain* and *requirement analysis* to describe requirements and behavior of a software system with domain-specific abstractions, for example features. In contrast, the *solution space* includes implementation-oriented abstractions, for example code artifacts. Between features in the problem space and artifacts in the solution space exists a mapping [TKES11]. Form and complexity of this mapping reaches from simple mappings based on naming conventions to comprehensive machine-processable rules encoded in generators [CE00, Käs10, ABKS13].
Regarding the *problem* and *solution space*, the development of feature-oriented product-lines has four main tasks [ABKS13]:

Domain analysis determines the scope of the domain. It declares the covered products and relevant features. The analysis is documented in a feature model. In the

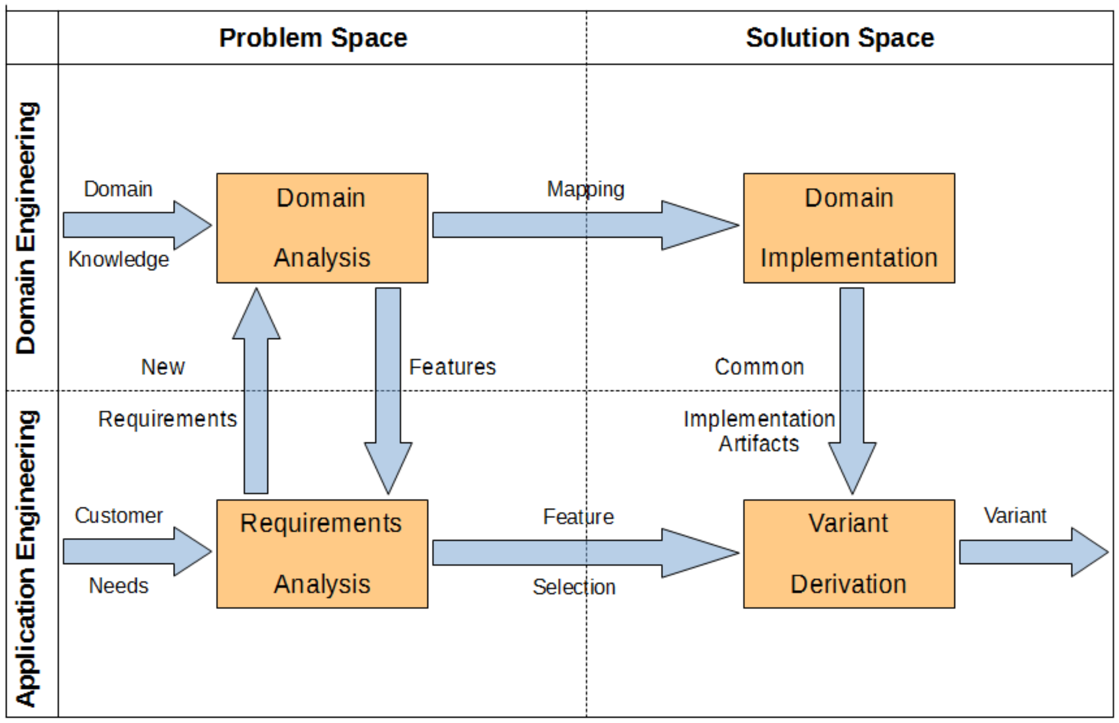


Figure 2.3: Overview of the Domain and Application Engineering Process [ABKS13]

scope of this thesis, we use a feature model as a variability model to describe the domain of several variants.

Requirement analysis examines specific customer needs. The requirements are mapped to feature selections. In case that any features exist to fulfill a certain requirement, the requirement will be analyzed in the domain analysis task. In the scope of this thesis, we use features and feature expressions to describe the requirements perspective of variants.

Domain implementation creates reusable artifacts that are mapped to features from the domain analysis. In the scope of this thesis, we concentrate on code artifacts.

Product derivation combines reusable artifacts based on the results of the requirement analysis. In the scope of this thesis, product derivation is based on feature configurations.

One goal of feature-oriented software development is full automation to derive a variant based on a feature configuration. To enable the generation of variants, variability implementation techniques can be used, as we introduced in Section 2.1.3. One target of our work is to support the transition of variants to a product line, where each variant was developed using concepts of single-system development. Therefore, we describe the *problem space* for all variants and establish a mapping between features and code artifacts of variants.

2.2 Clone-and-Own

In simple words, clone-and-own means to copy a similar and successful variant and adapt it as needed [DB07]. Hence, variant development can be applied using a clone-and-own approach [DB07]. In this section, we discuss traditional clone-and-own, introduce version control systems as a technical platform to support developers applying clone-and-own strategies and present a possible process for developing variants in branches of a version control system.

2.2.1 Traditional Clone-and-Own

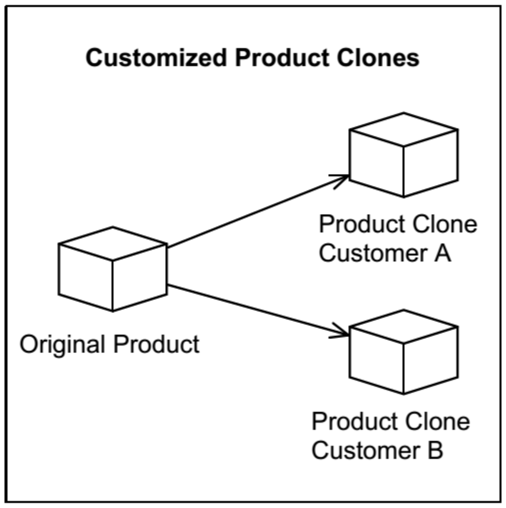


Figure 2.4: Customized Product Clones [Kla14]

Clone-and-own approaches duplicate software artifacts, like small parts of source code, components of a product or whole software systems [DB07]. The aim is to meet requirements by copying existing software artifacts and modifying them [DRB+13]. A common field of application is the development of product families [RKBC12]. In this context, clone-and-own is a fast strategy to develop new variants. To implement a new variant, software developers often fork an existing product and modify it to meet the requirements of the new variant [EEM10]. Figure 2.4 shows this customization scenario. First, the original product is cloned into two product clones. Then, both clones can be modified, so that one product clone fulfills the requirements of customer *A* and the other product clone fulfills the requirements of customer *B*. Both clones have separate development lines that are independent of the original product. This process represents the traditional clone-and-own strategy [Kla14].
Beside the simple application of cloning, code clones are known for their disadvantages in the field of software engineering. Cloning leads to redundancies, lack of traceability, lack of control and inconsistencies [RKBC12]. If not carefully managed, code clones can unintentionally diverge in the long term [LWN07]. Common practical problems are the difficult change propagation between clones, the difficult integration of cloned artifacts and, thus, the increased maintenance effort [DRB+13]. With a growing number

of products, the maintenance effort to keep track and propagate changes between each cloned product increases [RKBC12]. To summarize, using code cloning in the long term is considered harmful to the code quality.
In contrast, advantages of traditional clone-and-own are low adaption costs, independence from other developers and the easy and fast reuse of software artifacts to meet new requirements [AJB+14, RKBC12]. Furthermore, the development of a cloned product starts on existing and already tested code. Moreover, Dubinsky et al. [DRB+13] point out three further reasons which explain why code cloning is still popular in industrial practice:

Efficiency: Saving time and resources, cloning is a simple mechanism to start the development from already implemented and verified artifacts. It is easy and fast to perform and provides independence because developers can make any change to their clones [KG08].

Short-Term Thinking: Organizations primarily focus on producing individual products and disregard any reuse management.

Lack of Governance: In most organizations, knowledge about reuse is not maintained. Information about cloned assets only exist in people's mind.

2.2.2 Usage of Version Control Systems

Version control systems provide functionality that supports code cloning, like forking an existing variant or managing variants in separate branches. As a main task, version control systems assist developers to track the development of software systems. They provide techniques and tools to manage large and complex software systems [Men02].
In software engineering, version control systems are best known for managing different revisions of software systems. They facilitate collaborative development by tracking changes in development artifacts, for instance source code. A revision gets an identifier and a comment that explains the changes [ABKS13]. It describes ordered variations over time. However, a variant describes variations that exist parallel [ABKS13]. A version comprises revisions and variants [ABKS13].
Developers commonly use version control systems to track revisions, implement bugfixes in independent branches and merge branches back into the main development line. As a typical workflow, software developers can derive revisions from another revision or a base program, check out revisions from the repository, change them and check them in again [ALB+11]. The workflow to implement and manage revisions is called variation over time [ABKS13]. Using this development approach, a software system typically has a linear development path, whereas each revision of the software is derived from the previous revision. There are variants that exist in different revisions. In this context, a release is a selected revision that has a release name and is deployed to customers [ABKS13].
Beside the development on the main development path, version control systems provide the concept of *branching* and the concept of *forking*. A branch is an independent

development path that is separated from the main development line. Using the ability to branch, software developers can create independent revisions because changes on one branch do not affect files in other branches [ABKS13]. However, a fork is a local copy of a repository. Working on a fork does not affect the original repository. After finishing changes on a fork, a developer can send a pull request to get the code accepted.
Using the concept of parallel variant development in branches, version control systems are able to support separate development paths which are diverged from a common starting point [Bre04]. Instead of managing one latest revision of the software, there are several latest revisions. These concurrent revisions represent different variants of the product, where each variant has an own revision history. Thus, version control systems are external tools that can be used to manage variability and that enable the development of tailored and customer-specific products.

2.2.3 Variants in Branches

One possibility to manage several variants is to develop each variant in a separate branch. This strategy is called *branch-per-product* development [ABKS13]. A main development branch tracks code that is used by multiple variants. When a customer requests individual product requirements, the developer uses the code of the main development branch as a baseline and implements customer-specific features. For each customer, a customer branch is created where the development of a specific variant happens. If it is necessary, changes like bug fixes or other development results can be merged from the main development branch into customer branches. The other direction is also allowed, changes in customer branches can be merged into the main development line [ABKS13].
Figure 2.5 shows the concept of *branch-per-product* development. The *branch-per-product* strategy creates a branch for each required variant. If a new variant is required, then an existing branch will be forked to create a new branch representing a new variant. This branch can be adapted as needed to implement the new variant. For example, the main development line tracks the code base that is shared by *variant 1* and *variant 2*. For each customer product, a new branch is created. This branch contains all code from the main development line and it represents a new variant. *Variant 1* needs a bug fix which is implemented in a separate branch. The result of the bug fix is merged into *variant 1* and into the main development line. Then, *variant 2* is created and it already contains the bug fix of *variant 1*. At a later time in development, a software developer changes *variant 1* by extending a feature which is also implemented in *variant 2*. To keep all development paths consistent, the changed code has to be merged into the main development line and *variant 2*. The presented process in Figure 2.5 is a simplified example. In practice, software development typically requires a high amount of branching and merging operations [Ber90]. The more variants are developed in branches, the more merge operations are necessary to propagate changes between branches.

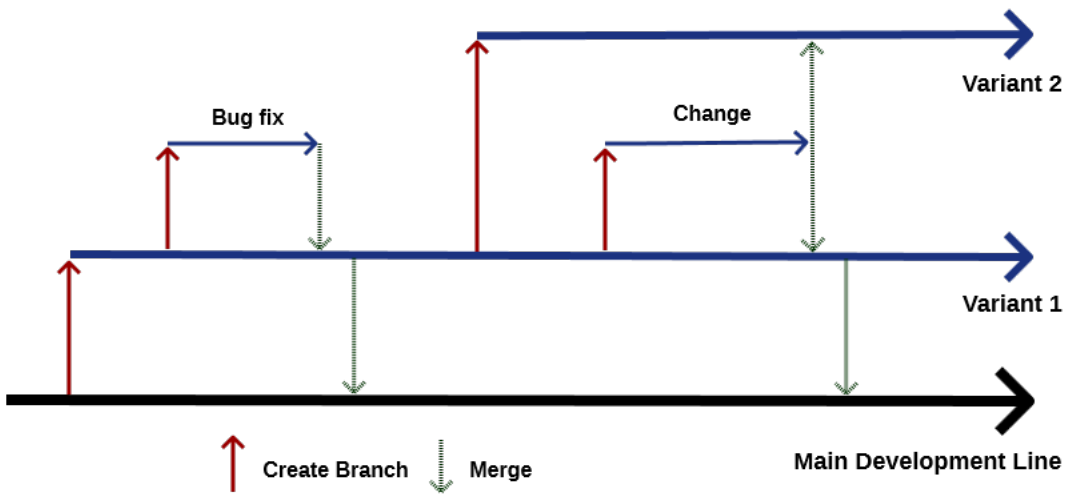


Figure 2.5: Branch-per-Product Development [ABKS13]

2.3 Merge Techniques

Merging is one of the core tasks of optimistic version control systems. It transfers several versions of a software artifact into a shared version [Men02]. To be more precise, merging encompasses the process to integrate changes from one branch into another branch. During the merge process, parallel modifications on different versions of a software artifact cause conflicts which need to be resolved [Men02]. The degree of merge automation depends on the merge technique [Men02].

2.3.1 Two-Way vs. Three-Way Merging

Merge techniques can be distinguished between two-way and three-way merge techniques. A two-way merge technique merges two versions of a software artifact. It only compares the differences between both versions. As a drawback, two-way merge techniques cannot determine whether differences are caused by insertions, modifications or removals in one of both versions or by parallel modifications in both versions [Men02]. As a result, the merge cannot be automated and causes a merge conflict. In contrast, a three-way merge technique merges two versions in comparison with their common ancestor. The common ancestor is that version from which both versions that need to be merged originated [Men02]. Regarding the common ancestor, the three-way merge is able to resolve merge conflicts that cannot be resolved by a two-way merge. For example, Figure 2.6 shows that there is a difference between the left and right version of *JavaElement*. Without further information, it is unclear whether the field *name* was added in the left version or removed in the right version. Regarding the common ancestor, three-way comparison determines that field *name* already existed in the common ancestor. Hence, it was removed in the right version. The merge can be automatized by removing the field *name* in the left version, too.

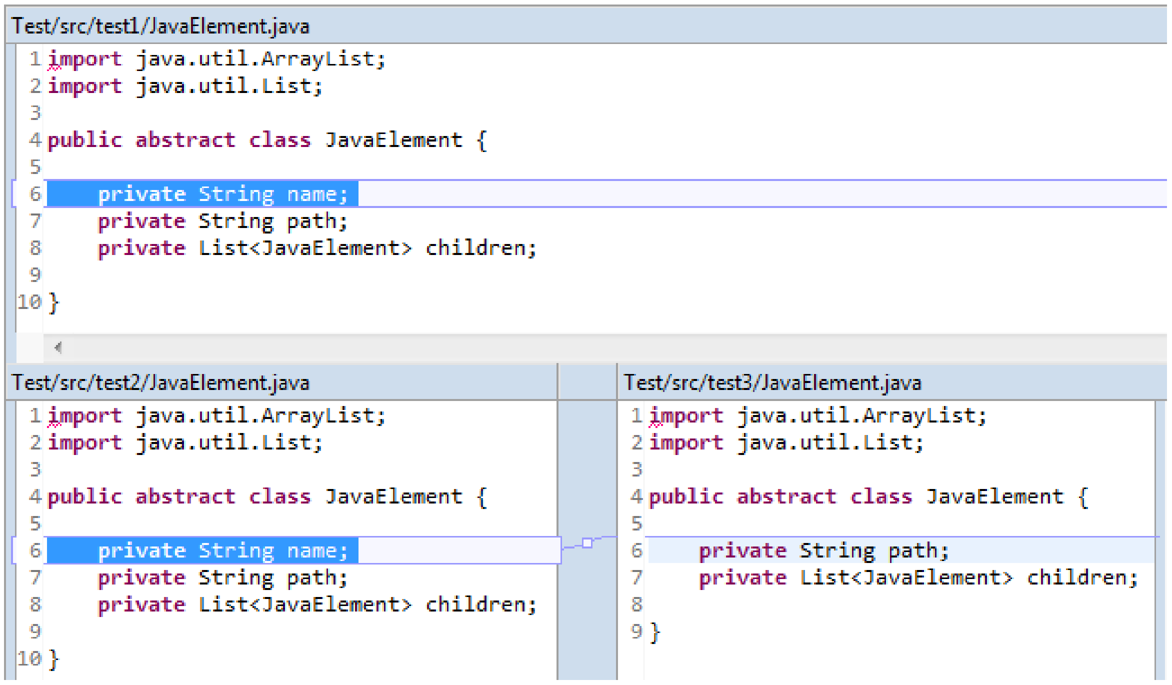
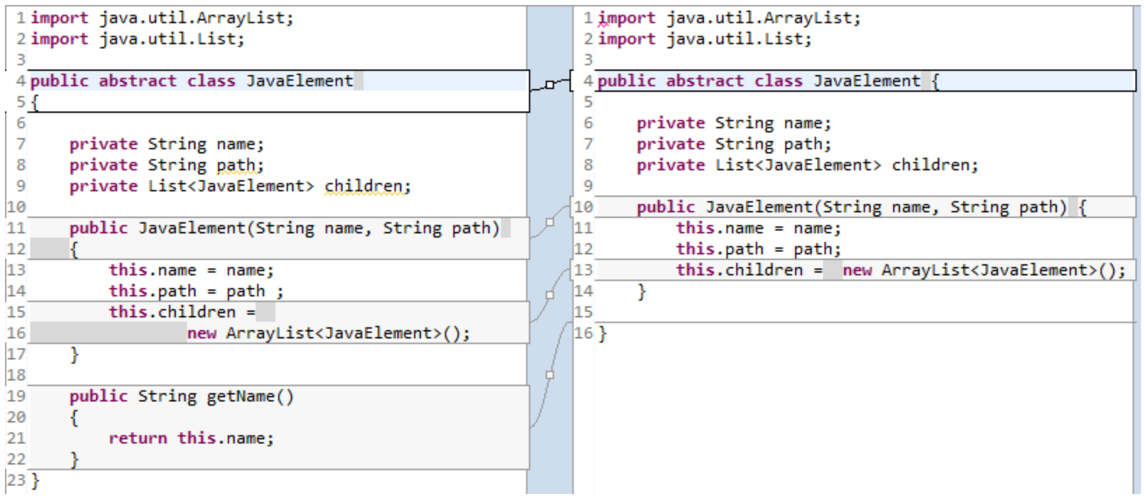

Test/src/test1/JavaElement.java
```
import java.util.ArrayList;
import java.util.List;

public abstract class JavaElement {

    private String name;
    private String path;
    private List<JavaElement> children;

}
```
Test/src/test2/JavaElement.java
```
import java.util.ArrayList;
import java.util.List;

public abstract class JavaElement {

    private String name;
    private String path;
    private List<JavaElement> children;

}
```
Test/src/test3/JavaElement.java
```
import java.util.ArrayList;
import java.util.List;

public abstract class JavaElement {

    private String path;
    private List<JavaElement> children;

}
```

Figure 2.6: Three-Way Comparison

2.3.2 Unstructured vs. Structured Merging

A further distinction between merge techniques is based on the representation of software artifacts. In the scope of this thesis, we need to merge changes between variants. In this context, we want to achieve a high degree of merge automation. Beside a two-way or three-way merging, the degree of merge automation depends on how the merge technique regards software artifacts. On the one side, unstructured text-based merge techniques regard code as a text file. On the other side, structured merge techniques exploit language-specific knowledge. Most commercial merge tools are based on unstructured merge techniques. These merge techniques commonly use a line-based merge of text files that compare text files to detect whether common text lines have been modified, inserted, deleted or moved [Men02]. Line-based merging is an efficient, scalable and accurate merge technique which is able to merge about 90% of changed files automatically without merge conflicts [Men02]. However, unstructured merge techniques have too coarse granularity, so that two parallel modifications to the same line cannot be handled [Men02]. For example, two revisions A and B change code in the same file on the same line. Then, line-based merge techniques cannot merge the changes and identify a merge conflict. As a further limitation in order to automate the merge, unstructured merge techniques only compare text and do not analyze the syntax and semantic of text fragments [Men02].
As opposed to unstructured merging, structured merging is the more powerful alternative that provides a higher degree of merge automation. As the name implies, it analyzes the structure of files regarding the syntax and semantic. Compared to unstructured merging, a syntactic merge is not affected by unimportant conflicts like code



```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public abstract class JavaElement
5  {
6
7      private String name;
8      private String path;
9      private List<JavaElement> children;
10
11     public JavaElement(String name, String path)
12     {
13         this.name = name;
14         this.path = path ;
15         this.children =
16                 new ArrayList<JavaElement>();
17     }
18
19     public String getName()
20     {
21         return this.name;
22     }
23 }
```

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public abstract class JavaElement {
5
6      private String name;
7      private String path;
8      private List<JavaElement> children;
9
10     public JavaElement(String name, String path) {
11         this.name = name;
12         this.path = path;
13         this.children =  new ArrayList<JavaElement>();
14     }
15
16 }
```

Figure 2.7: Code-Formatting Problem of Line-Based Merging

comments that are modified by different developers or line breaks that are inserted to make the code more readable [Men02]. For example, we assume that two similar files contain the same syntactic code and only differ in line breaks and tabs. Then, method *getName()* is added at the end of file one. Figure 2.7 shows how a line-based merge technique failed to automatically merge the insertion of method *getName()* from left to right version caused by a changed code formatting. Line breaks are removed at several positions in the right version. As a consequence, a line-based merge reports a difference between left and right version at each changed position. In the left version, *getName()* was inserted at line 19. Due to the changed code formatting, *getName()* needs to be inserted at line 15. A line-based merge is not able to resolve this conflict. In contrast, a syntactic merge ignores conflicts caused by a changed code formatting and is able to merge *getName()*. However, syntactical merging has several drawbacks. Syntactical merges are slower than line-based merges. Moreover, they are NP-complete [ZJ94]. As a further drawback, syntactical merging causes a merge conflict if the merged code has a syntactic error.

Beside syntactic merging, semantic merging solves conflicts that cannot be detected by syntactic merge techniques. While syntactic merge techniques implement tree-based merge approaches like an abstract syntax tree, semantic merge techniques implement graph-based merge approaches. These approaches are able to detect conflicts caused by the relationship between function definition and function invocation. The graph representation simplifies the detection of incompatibilities between a definition and its invocation because it maintains an explicit link between them [Men02].

3. Synchronization of Software Variants

In this chapter, we discuss the development of variants and introduce a concept to support the conventional development of software variants by efficiently synchronizing a small number of variants that could increase over time. In Section 3.1, we compare the development of variants with a clone-and-own approach and product-line implementation techniques. Furthermore, we discuss their benefits and drawbacks and we illustrate the problem to develop few variants when it is not clear how many variants are added in the future. To support the development of variants, we introduce the *VariantSync* concept consisting of a code to feature mapping concept (Section 3.2) and two strategies to synchronize variants from different points of view (Section 3.3). Finally, we present a batch synchronization in Section 3.4.
VariantSync is based on a prior concept to synchronize fine-granular changes between variants [Luo12]. This concept detects changes on variants and provides a semi-automated computation of synchronization targets. We extend this concept with a concept for an automated computation of synchronization targets, a concept for an automated synchronization from different points of view and a concept to support the migration to a product line.

3.1 Variant Development

The development of variants can be performed with different approaches depending on the number of required variants. On the one hand, the development of few variants typically requires a light-weight approach with minimal effort and a fast implementation result. On the other hand, the development of many variants requires a heavy-weight approach with strategic planning to minimize implementation and maintenance effort [MNJP02].

3.1.1 Development of Few Variants

Developing few variants, each variant is developed separately. For this purpose, clone-and-own is a common tool-driven approach to support the development. As we introduced in Section 2.2.2, version control systems provide functionality to apply clone-and-own using the *branch-per-product* strategy. Using version control systems has the advantage that developers are familiar with the technology and its application. Today's single-system engineering typically uses version control systems to track development of software systems. Thus, under the condition that the number of variants will not increase, developers are able to use an established and well-known technique. Furthermore, branching can be used uniformly for code and non-code artifacts, changes can be performed at arbitrary granularity and changes concerning crosscutting features do not require any preplanning [ABKS13].
Nevertheless, there are some significant drawbacks. Diverging of variants quickly increases because variants consist of disconnected and separated code clones [YGM06]. Single-system engineering approaches, like clone-and-own approaches, are appropriate to develop a small number of variants. However, the number of variants typically increases over time. If variants are added, software evolution becomes more complicated. Thus, the more variants are added using clone-and-own, the more mid- and long-term maintenance costs increase. For example, developers often create a new branch for each variant, even if two variants only differ in small parts. Due to the fact that branches are copies of the code base in the main development line, they provide no structured reuse or modularity [ABKS13].
Moreover, variants are changed by applying bug fixes or implementing new requirements during development. Propagating these changes between variants cause several synchronization issues. First, determining which changes need to be propagated to other variants is a difficult task. As we introduced in Section 2.1.1, tools have no implicit domain knowledge about the variants. Due to the fact that they only manage source code without further knowledge about commonalities between the variants, it is difficult to compute valid synchronization targets that need to be synchronized with a given change. Second, merging the changes requires manual effort. For example, version control systems provide merge support, but for the purpose of synchronizing disconnected variants, merging tools are not as powerful as needed. The merge operation only copies changes from one branch into another branch. The developer manually needs to extract the change, determine propagation targets, apply the merge using a merge tool and resolve possible merge conflicts. In this context, branches easily diverge more than intended because it is easy to lose track of branches or to forget some merges [ABKS13]. So, using tools from single-system engineering like version control systems, change propagation is error-prone and causes high merge effort. To avoid these drawbacks, the effort to coordinate variant development needs to be decreased.

3.1.2 Development of Many Variants

In contrast to clone-and-own approaches, the development of many variants requires a preplanned and coordinated process to avoid unsustainable maintenance costs. For this purpose, product-line engineering provides a coordinated variant development that supports the systematic and planned reuse of software artifacts by explicitly managing commonalities and variabilities in a domain.

As the benefits, product lines provide both product quality aspects like a higher reliability and process-oriented aspects like a reduction of development costs as well as maintenance costs. Using modular product-line implementation techniques like components or frameworks, common software artifacts are developed and tested once, so that variants profit of reusing high-quality artifacts [PBvdL10]. Product lines further reduce costs of variant development. Instead of developing variants from scratch, reusable artifacts can be combined to speed up development. Moreover, product-line engineering provides a coordinated variant development and variant management that reduces maintenance costs [vdLSR07].

However, focusing on mid- and long-term results, product-line engineering requires certain up-front investments [vdLSR07]. The development needs certain time to enable variant creation or generation and to benefit from the up-front investments. Product-line engineering requires several development steps before a variant will be created. Among others, the domain needs to be scoped and analyzed, customer requirements need to be analyzed, variability needs to be modeled and reusable software artifacts need to be implemented and mapped to variability modeling. All these steps require investments without a direct reward in the form of variants. So, product-line engineering is only profitable from a certain number of variants onwards.

As a further drawback, tool support for product-line engineering is not as well-engineered as tool support for single system engineering. A variety of tools can be used to build a product line, but there is no single tool that fulfills the needs of all product lines. However, many tools supporting single-system engineering are also applicable in a product-line context. The problem is that product-line engineering has particular needs and risks, which complicate tool support. First, as opposed to single-system engineering, software artifacts are long lived and more complex because they can be reused across a number of variants. In addition, product-line tools must support variability among multiple variants. For example, relationships are established between artifacts and variants, artifacts and the product-line architecture and between two and more artifacts [BCD+00]. So, product-line tools have the challenging task to support concurrently creation, maintenance and usage of product-line artifacts in a coordinated way. Second, tool support for product-line engineering requires not only the usage of a set of tools, it also requires the interoperability of these tools to automate the production of variants. For example, product-line engineering covers many development phases, e.g., activities of domain and application engineering, and tools should be able to work with output from other tools as well as provide input for other tools [BCD+00]. A lack of interoperability causes inconsistencies and gaps between product-line activities. Third, product-line tools need to represent variability to be flexible for changing customer needs. For example, a product line should be able to generate custom-tailored variants and product-line tools need to provide mechanisms to ensure this variant generation based on reusable artifacts. The lack of tool support for variability leads to difficulties representing variability in the architecture of a product line [BCD+00].

3.1.3 Problem Statement

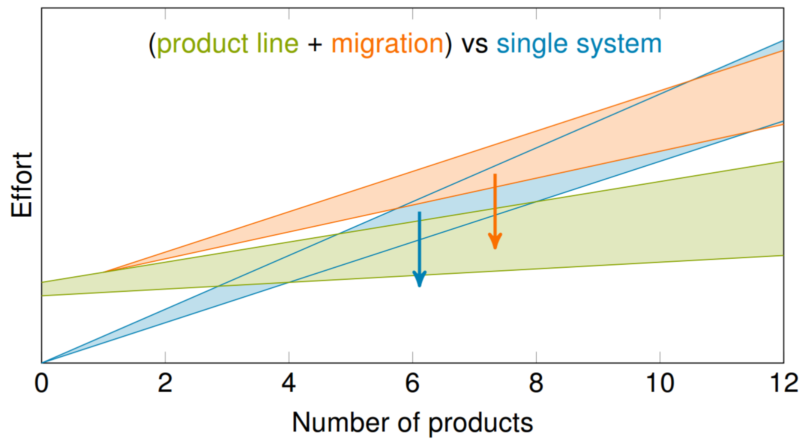


Figure 3.1: Effort of Variant Development

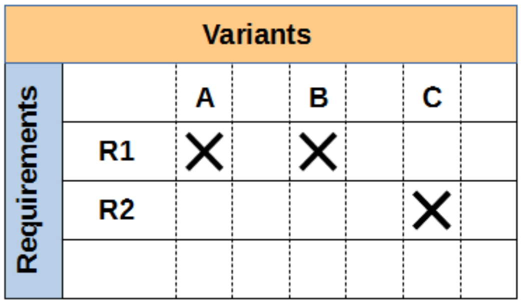
As we described in Section 3.1.2, product lines provide several advantages developing a sufficient and increasing number of variants, but they require certain initial costs and risks [vdLSR07]. Regarding up-front investments and the lack of tool-support, using a product-line approach to develop few variants is not preferable. Besides, single-system engineering approaches are designed to develop a fixed number of variants, but they cause problems when variants are added in the future. Figure 3.1 compares the costs of product-line engineering and single-system engineering. A product line is preferable if the product line is able to generate a sufficient number of products. In many publications [JKB08, vdLSR07, Kru06, MNJP02, WL99], this break-even point is approximately located after three variants. The break-even point cannot precisely located because it depends on the reuse potential between products, the chosen product-line transition strategy as well as several organization and market influences, such as customer base, expertise or kinds of products [PBvdL10].
As opposed to product lines, single-system engineering approaches are designed to develop a fixed number of variants. If variants are added in future, then effort and costs to both maintain existing variants and develop new variants will rise, as it is shown in Figure 3.1. If a sufficient number of variants is reached, the cumulative costs of a product-line engineering approach is cheaper than single-system engineering approaches [AM14, MNJP02] and a product line is more appropriate to manage variants in a cost-effective manner. To this end, variants need to be transformed to a product line using an extractive approach where variants are used as a baseline for bootstrapping a new product line [VBP12]. Before a product line can be initiated, the commonalities and differences of variants need to be analyzed. The transition of already existing variants to a product line is a difficult and time-consuming task which requires high initial investments and causes high effort [PBvdL10, vdLSR07, VBP12]. Furthermore, the transition is not profitable if the number of variants is not fixed and if new variants are only added infrequently. For example, a company develops a product. Three months after the release, the customer requires a similar product and the company develops a

further variant of this product. Six months later, another customer requires the same product with additional features and the company develops a third variant. Following this process, the company has n variants and always adds one new variant from time to time, so that the company has to maintain $n + 1$ variants. While the transition to a product line causes much more effort than only adding a further variant in the short term, effort and costs to maintain existing and develop new variants will raise with each further variant in the long term. The problem is that it is not clear if a further variant will be added in future. So, there is no optimal point in time to perform the transition to a product line. Thus, a company is always afraid to perform the transition, which is admittedly expensive in the short term but reasonable in the long-term.
Summarizing, there is a gap between single-system engineering and product-line engineering to efficiently develop a small number of similar variants which could increase over time. To reduce the gap, we provide a light-weight and language-agnostic concept, called *VariantSync*. On the one hand, *VariantSync* supports the maintenance of a small number of variants which could increase over time by automating the synchronization of variants. On the other hand, *VariantSync* supports the light-weight transition to a product line by accumulating domain knowledge.

3.2 Automating the Identification of Synchronization Targets

The efficient and highly automated synchronization of variants is a challenging task. The basic idea of synchronization is described as follows: First, changes that occur during development of a variant need to be detected and collected. Second, changes need to be synchronized with appropriate variants. We use domain knowledge to detect appropriate variants that need to be synchronized with a change that occurred in another variant.

3.2.1 Making Domain Knowledge Explicit


Variants
Requirements

	A	B	C
R1	X	X	
R2			X

Figure 3.2: Targets of Change Propagation

In Section 2.1.1, we introduced domain knowledge to support software reuse and introduced the meaning of domain knowledge for developers and tools. A developer acquires

implicit domain knowledge while implementing a variant, because each written code fragment has a meaning and belongs to a concern or requirement. As opposed to this, tools are not able to acquire domain knowledge the same way developers do. Our *VariantSync* strategy has the aim to make the synchronization of variants more efficient by propagating changes into appropriate variants. Figure 3.2 exemplarily shows change propagation into appropriate variants. Variant A and variant B implement requirement *R1*, whereas variant C implements requirement *R2*. Inside variant A, code is changed that implements requirement *R1*. Propagating this change, variant B is a valid propagation target because variant B also contains code that implements requirement *R1*. Due to the fact that variant C does not implement requirement *R1*, variant C is not affected by this change and does not need to be synchronized with this change. Hence, using domain knowledge to represent variability enables us to compute valid propagation targets of changes that occurred inside a variant.
To compute valid propagation targets, we simply need a list of concerns of each variant. This list describes the variability of a variant and enables tools to compute valid propagation targets. In Figure 3.2, we used requirement descriptions to express variability in a domain. However, requirement descriptions are simplified and coarse-grained variability descriptions. To profit of existing and comprehensive variability modeling concepts, we adapt variability models of product lines. In contrast to requirement descriptions, variability models provide comprehensive descriptions of the variability in a domain. Examples of variability models are orthogonal variability models [PBvdL10] or decision models [SRG11]. For the *VariantSync* strategy, we adopt feature modeling because it is currently the most popular form of variability models [ABKS13]. Thus, we introduce a feature model that describes variability for all variants. Beside its advantages, such as documenting features and their relationships [ABKS13], a feature model needs to be manually created by a developer. Based on the feature model, each variant is characterized by a set of features. To achieve this, feature models provide the ability to derive feature configurations. A feature configuration is a valid selection of features from the feature model. Each variant is characterized by a feature configuration. That means a variant implements all features specified by the feature configuration. However, at this point does not exist a mapping between code fragments and features.
To summarize, the challenge is to map code fragments to features. For this purpose, we present and discuss concepts of feature tagging.

3.2.2 Tagging Code to Features

To bridge the gap between code fragments on implementation level and features on domain level, product-line approaches provide a wide range of methods that use frameworks, components, inheritance, aspects, generative programming or preprocessor directives [ABKS13]. However, on the one hand, these methods require a high effort to transform the existing code of the variants into the chosen coding technique. On the other hand, these techniques differ in granularity.
Composition-based techniques usually extend code with coarse granularity. Some of these techniques provide extension points that can be extended by features, like plug-in

architectures in frameworks and components. Other composition-based techniques like aspect-oriented programming are able to virtually extend methods. Using composition-based techniques, it is only possible to add classes, methods, or fields and to extend whole methods with wrappers, like method refinements or method overriding [ALB+07]. This is a significant drawback for already existing variants because these variants were not designed for a composition-based development. Moreover, a developer typically not only adds whole classes or methods but also adds new statements into existing methods, changes method signatures, or deletes lines of code during implementation of a variant [Käs10].
Beside composition-based techniques, annotation-based techniques are able to mark code fragments at arbitrary levels of granularity. On the granularity of single code lines, they set annotations at that code line that should be extended. For example, preprocessor directives can wrap a code line to indicate the feature this line is mapped to. Furthermore, annotations can be nested to express feature dependencies. Nevertheless, the comprehensive use and nesting of annotations easily pollutes the code and leads to obfuscated code. In addition, annotation-based product-line techniques require a complete code to feature mapping because non-mapped code belongs to all features [Käs10].
In summary, composition-based product-line techniques are not appropriate to map code fragments to features for a small number of variants that were not designed for feature support, because these techniques change the code inside variants, which causes high customization effort. Annotation-based techniques admittedly provide a code mapping on an appropriate granularity, but code obfuscation and high effort to annotate the whole code of all variants are still given. To overcome the high mapping effort, we introduce a lightweight tagging concept. Thus, our concept has to fulfill five requirements:

1. Apply tagging on a fine granularity level to even map single lines of code to features.
2. Avoid code pollutions.
3. Mapping should not heavily interfere with existing coding paradigms to keep the mapping effort low.
4. Handle the feature interaction problem as well as code that belongs to multiple features.
5. Obtain code to feature mapping of changed code fragments while synchronizing changes into propagation targets.

To meet the first and second requirement, we use concepts of annotation-based techniques and adapt them to our needs. We obtain a fine granularity by marking code fragments with information about its feature on the code level. Instead of using annotations, we store the tagged information as metadata. So, we do not adapt code and fulfill the second requirement by avoiding code obfuscation.

To meet the third requirement, we partly map variant code to features. In the long term, automated synchronization will synchronize code that was changed inside one variant to other variants. Therefore, we only map code that was changed during the development process. As an advantage, the amount of mapped code grows with ongoing development process.

To meet the fourth requirement, we need to handle code that belongs to multiple features as well as the feature-interaction problem. For this purpose, we map code fragments to feature expressions instead of features. The feature expression describes the feature interaction of multiple code to feature mappings. Evaluating the feature expression against feature configurations of variants identifies valid propagation targets. For example, code fragment A belongs to feature f_1 and feature f_2. In this case, we create a new feature expression $f_1 \wedge f_2$ and map code fragment A to this expression. Now, all variants that implement feature f_1 and f_2 and include feature expression $f_1 \wedge f_2$ in their feature configuration are valid propagation targets. To automatically reason about whether a feature expression is included by a feature configuration, the feature expression and feature configuration are represented as propositional formulas. This has the advantage that automated reasoning techniques are able to determine if the propositional formula of the feature expression satisfies the propositional formula of the feature configuration. For example, feature expression $f_1 \wedge f_2$ evaluates to true against feature configuration $f_1 \wedge f_2 \wedge \neg f_3 \wedge f_4$.

Lastly, to meet the fifth requirement, the presented tagging concept can be used as an incremental feature location technique because we aggregate mapping knowledge that describes which code fragment is referenced to which feature(s). This knowledge enables us to establish a feature to code traceability which is an essential requirement to migrate variants to a product line. Changed code is mapped to a feature. While synchronizing changed code into a propagation target, we recover existing code to feature mappings and the propagation target integrates mapped code. Thus, the more code is mapped the stronger the migration to a product line is supported.

In the following, we propose and discuss two concrete tagging strategies that realize this tagging concept.

Manual Tagging

Manual tagging has the aim that the developer keeps control over the tagging. Before and after working on code, the developer has the ability to manually tag code fragments to feature expressions. He decides whether to tag code after each implementation step or to tag code at the end of a coding session. Therefore, the process of tagging code fragments is decoupled from the process of changing code fragments.

As an advantage, the developer has complete control about the tagging. After implementing a feature, he can evaluate his work by tagging changed code to feature expressions. If the developer tagged a code fragment to the wrong feature expression, he can easily correct the tagging. Later in synchronization, only code will be synchronized that was tagged by the developer. As a further advantage, the granularity of

tagging is very fine. The developer is able to tag each command or, in case that it is necessary, each single word inside a line of code to a different feature expression.
Nevertheless, the developer has to perform a lot of manual work to mark each code fragment that belongs to a feature expression. As a drawback, there is no mechanism that supports the developer by automating the tagging. If many positions inside a code file change, the developer has high effort to tag each change. Moreover, the error rate of tagging depends on the fact how precise the developer works. For example, the developer could forget to tag certain code fragments.

Tagging with Feature-Expression Contexts

Tagging with feature-expression contexts has the aim to automate tagging and to keep manual tagging effort low. For this purpose, we automate code tagging using a context in which the developer can work on the code. Before the developer starts working, he chooses the feature expression on which he wants to work on and activates a working context. In the following, all code changes during implementation are mapped to the activated feature expression. If the developer wants to work on code that belongs to another feature expression, he only needs to change the active context. For example, the developer implements a class containing a function of feature f_1 and a function of feature f_2. First, the developer activates the context of feature f_1. Then, he implements the first function. In the following, the developer stops the context of feature f_1, activates the context of feature f_2 and implements the next function. Afterwards, he stops the context of feature f_2. As a result, the first function is automatically tagged to feature f_1, whereas the second function is tagged to feature f_2.
Using a context has the advantage that the developer has less effort to tag code changes to feature expressions. Except activating the desired context, his development workflow is not different from clone-and-own.
However, automated tagging has the drawback that once performed, tagging cannot be changed afterwards. For example, the developer forgets to switch the context and works first on code that belongs to feature expression A and then on code that belongs to feature expression B. Now, code changes that belong to feature expression B are mapped to feature expression A. As opposed to manual tagging, the developer has no possibility to correct the mapping. He needs to reactivate context A and manually undo each implementation step that was mistakenly mapped to feature expression A. As a further drawback, the developer has effort to frequently change the active context if the developer works on small code fragments belonging to different feature expressions. For example, the developer works on the class *Calculator*. This class contains methods to *add*, *subtract*, *multiply* and *divide* integer values. Each method belongs to a different feature (*Addition, Subtraction, Multiplication, Division*). The developer activates the context for feature *Addition* and starts working on method *add*. Then, he needs to change one line of code in method *subtract*. Therefore, he needs to switch the context. To continue with method *multiply*, the developer needs to switch the context again. Finally, to implement method *division*, the developer again switches the context. This process is complicated and could lead to a wrong code to feature tagging if the developer once forgets to switch contexts.

Combining Manual Tagging and Tagging with Feature-Expression Contexts

In the previous sections, we discussed advantages and disadvantages of manual tagging and tagging with feature-expression contexts which both traces features to code fragments. Applying manual tagging, the developer has complete control about the code that he wants to tag. For example, manual tagging enables developers to tag single words to different feature expressions. However, a developer has high effort to manually tag each code fragment that belongs to a feature expression. To minimize tagging effort, tagging with feature-expression contexts automates tagging but once performed tagging cannot simply changed, for example to correct an erroneous tagging. If we combine both tagging strategies, we join their advantages and minimize their disadvantages. Tagging with feature-expression contexts automates the process of code to feature tagging and manual tagging provides manual control about the tagging. For example, if an error occurs during tagging with feature-expression contexts, then the developer can simply correct the tagging manually.

3.3 Automating the Synchronization

Using manual tagging or tagging with feature-expression contexts, we make domain knowledge explicit by tagging changed code fragments to feature expressions. Now, we need to synchronize code that is changed inside a variant with compatible target variants. Nevertheless, synchronization has several challenges:

1. Compute valid synchronization targets.
2. Decrease synchronization effort.
3. Expand the code to feature mapping.
4. Decouple implementation from synchronization.

As the first synchronization challenge, we compute synchronization targets for automatic and manual synchronization. We gave a short overview how to support the computation of valid synchronization targets in Section 3.2. A variant is a compatible synchronization target if the variant implements the same feature expression that the changed code fragment is tagged to. Now, we take a deeper look at possible synchronization targets and classify variants concerning their suitability to perform an automated synchronization. As we show in Figure 3.3, there are three different kinds of synchronization targets.
The first class are *variants without merge conflicts*. These variants are synchronization targets that do not cause merge conflicts. It is undefined whether these variants implement the changed feature expression. A change can be lexically merged to these variants but a synchronization could be wrong from a requirements perspective. *Variants without merge conflicts* that do not implement the changed feature expression are invalid synchronization targets and need to be ignored. For example, the change in

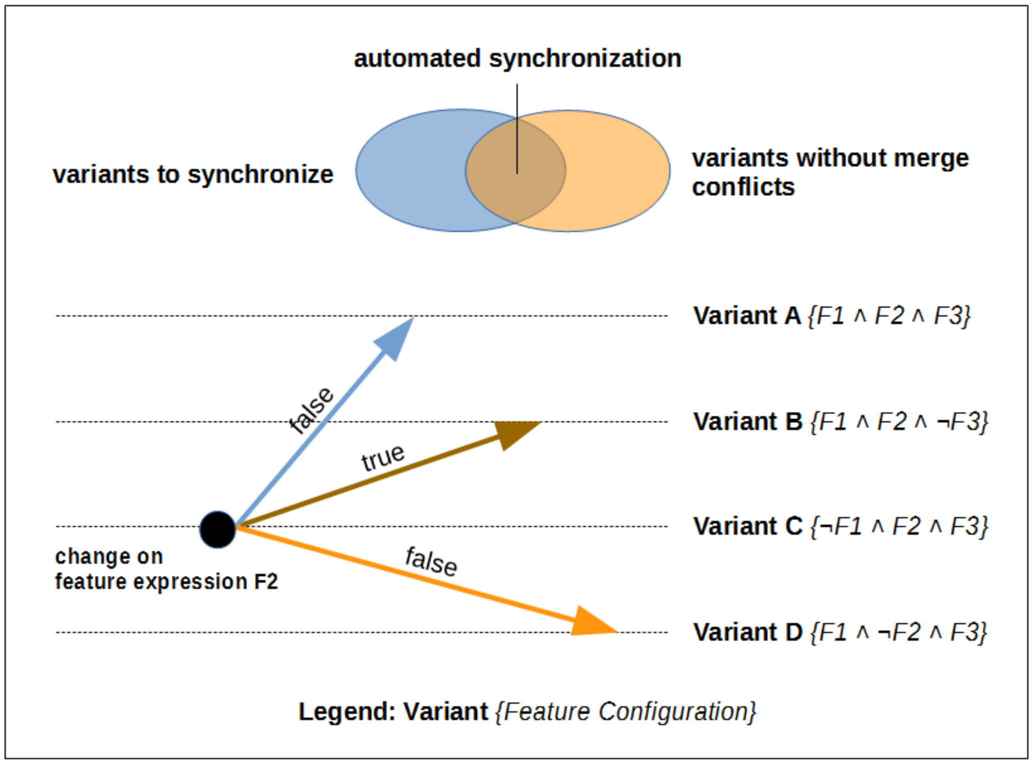

automated synchronization
variants to synchronize
variants without merge conflicts
Variant A {F1 ∧ F2 ∧ F3}
Variant B {F1 ∧ F2 ∧ ¬F3}
Variant C {¬F1 ∧ F2 ∧ F3}
Variant D {F1 ∧ ¬F2 ∧ F3}
false
true
false
change on feature expression F2
Legend: Variant {Feature Configuration}

Figure 3.3: Synchronizing Code Changes

feature expression *F2* could be lexically merged into variant *D*, but a merge is not necessary because variant *D* does not implement feature expression *F2*.
The second class defines *variants to synchronize*. These variants implement the changed feature expression, but a merge could cause merge conflicts. For example, variant *A* is a variant to synchronize and implements feature expression *F2*. We assume that feature expression *F3* affects the implementation of feature expression *F2*. So, the code change of feature expression *F2* would cause a merge conflict if it is synchronized with variant *A*. *Variants to synchronize* causing a merge conflict can only be synchronized by applying a manual merge.
Finally, the third class is the intersection of *variants to synchronize* and *variants without merge conflicts*. These variants implement the changed feature and allow an automated synchronization without merge conflicts. For example, the change in feature expression *F2* can be lexically merged into variant *B* and the synchronization is correct from a requirements perspective, because variant *B* implements feature expression *F2*.
Without *VariantSync*, only *variants without merge conflicts* can be automatically identified. For example, version control systems are able to check whether changes can be merged into branches. *Variants to synchronize* can only manually identified if the developer has implicit domain knowledge about the variants. To summarize, *VariantSync* automates the identification of *variants to synchronize* as well as *variants without merge conflicts*.
As the second synchronization challenge, we decrease the synchronization effort by

achieving a high degree of automated synchronization. A frequent synchronization using *VariantSync* could increase the merge frequency which leads to a reduced number of merge conflicts. Especially merge conflicts occur if many code changes are synchronized in a wrong order, e.g., when they are not synchronized in the order that they occurred. To be more precise, certain merge conflicts occur if two parallel changes can only be merged in a certain order [Men02]. Merging these changes in an inverse order causes inconsistencies [Men02]. In the following, we name this kind of conflicts *ordering conflicts*. For example, variant A is changed. This change needs to be synchronized into variant B. The developer does not synchronize the change and continues his work on variant A. Then, he changes again code in variant A and this code depends on the first change. Now, there are two changes that need to be synchronized into variant B. To avoid an ordering conflict, the first change in variant A needs to be synchronized into variant B first, then the second change in variant A needs to be synchronized into variant B. Synchronizing changes, *VariantSync* keeps track of the order in which changes occur. Furthermore, synchronization effort decreases with falling number of merge conflicts because these conflicts require a manual conflict resolution. On the one hand, the success of the merge depends on the kind of synchronization targets. As we show in Figure 3.3, the degree of automation depends on the amount of *variants to synchronize* that cause a merge conflict. On the other hand, the success of the merge depends on the merge techniques (Two-Way/Three-Way Merge, Unstructured/Structured Merge). Choosing merge techniques is a design decision on the implementation level. *VariantSync* can be applied independently of distinct merge techniques.

As the third synchronization challenge, we expand the code to feature mapping by applying change propagation, as we introduced in Section 3.2.2. If a change is merged into a propagation target, then the merged code is tagged to the same feature expression that the change is tagged to. Thus, code is automatically tagged to feature expressions and the degree of tagged code in synchronized variants automatically increases.

As the fourth synchronization challenge, we decouple implementation from synchronization. To be more precise, we separate the process of changing code fragments from change propagation to compatible variants. In this way, it is possible that one developer works on a variant and another developer, who is responsible for the target variant, propagates the changes. A decoupled change propagation allows to synchronize variants from different points of view. Hence, we present two synchronization strategies in Section 3.3.1 and Section 3.3.2.

3.3.1 Target-Focused Synchronization

The target-focused synchronization focuses on synchronizing changes to a single variant. This variant receives code fragments that were changed in other variants. We regard features of a single variant and compute whether changes occurred in compatible feature expressions that are implemented in different variants. The workflow using target-focused synchronization is as follows: A developer chooses a variant, possibly when initiating a new release. Then, *VariantSync* requests changes. These changes occurred in other variants and they are tagged to the same feature expression(s) that the

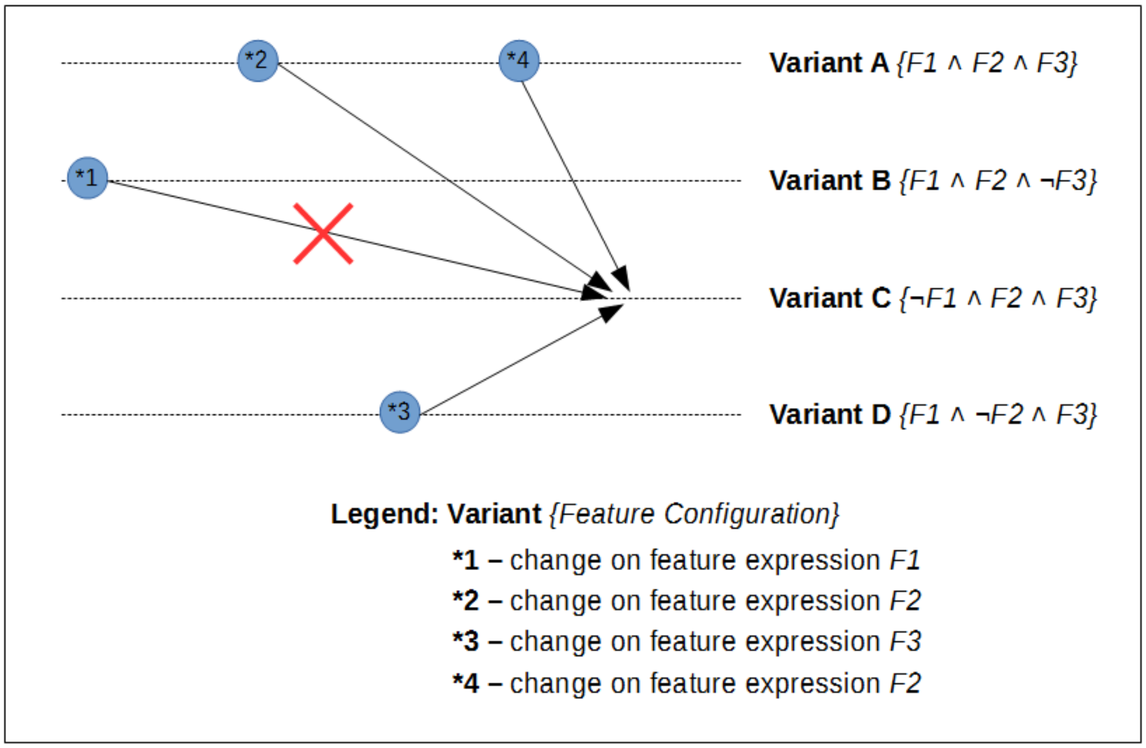

*1
*2
*3
*4
Variant A {F1 ∧ F2 ∧ F3}
Variant B {F1 ∧ F2 ∧ ¬F3}
Variant C {¬F1 ∧ F2 ∧ F3}
Variant D {F1 ∧ ¬F2 ∧ F3}
Legend: Variant {Feature Configuration}
*1 – change on feature expression F1
*2 – change on feature expression F2
*3 – change on feature expression F3
*4 – change on feature expression F2

Figure 3.4: Example of a Target-Focused Synchronization

selected variant implements. Changes are ordered by creation date. Finally, changes are merged into the chosen variant. For example, in Figure 3.4, variant C is the variant that needs to be synchronized with changes that occurred in variants A, B and D. For this reason, changes in compatible feature expressions of variants A, B and D are detected. Changes on feature expression $F2$ in variant A and feature expression $F3$ in variant D are compatible with variant C. Consequently, these changes are merged into variant C.

The target-focused synchronization has the advantage that the process to synchronize a variant is similar to the development of branches in version control systems. If one branch is merged into another branch, the version control system computes differences between both branches and merges the changes of one branch into another branch. Thus, developers are mostly familiar with the workflow to synchronize variants on a target-focused point of view. As a difference, the target-focused synchronization strategy provides changes aggregated to feature expressions of different variants.

Moreover, variants can be synchronized by experts on the respective variant. One developer can work on a variant and another developer, who is responsible for the target variant and potentially an expert on the target variant, can propagate the changes. Furthermore, the target-focused synchronization supports developers to add variants in the future. Only specifying a feature configuration, a new variant can be synchronized with changes the same way existing variants are synchronized. With each synchroniza-

tion, the new variant receives tagged code and the degree of code to feature mapping of the new variant increases. Moreover, the target-focused synchronization facilitates the integration with unaligned release cycles. Developing multiple variants using clone-and-own and branches of a version control system, variants may have have different release dates. Using target-focused synchronization, variants are developed separately and if one variant was not regularly developed and should be prepared as a release candidate, then this variant gets the relevant that changes that occurred in the meantime in other variants.
Nevertheless, the target-focused synchronization has the drawback that synchronization of a variant could cause high effort. If the variant was not synchronized for a long time, probably a huge amount of changes occurred in other variants. Merging these changes into the variant can be a time-consuming task. Moreover, the developer performing target-focused synchronization is probably not an expert of changes that need to be synchronized into the synchronization target. For example, developer one works on variant *D* of Figure 3.4. Then, developer two performs target-focused synchronization of variant *C*. Now, developer two synchronizes changes that occurred in variant D. The problem is that developer two is probably not an expert of changes that developer one performed. So, developer two may have problems during manual conflict resolution because he cannot estimate the significance of the changes. Furthermore, target-focused synchronization could lead to ordering conflicts because changes are not ordered in the same order that they occurred. They are admittedly ordered inside a feature expression of a variant, but changes are not ordered across variants. For example, a developer works in variant *A* and variant *B* on feature expression f_1. He first changes code of variant *A* and propagates this change to variant *B*. Then, the developer works on variant *B* and implements a function that depends on that code he synchronized from variant *A*. Now, another developer, who is not familiar with the development of variant *A*, synchronizes variant *C* using the target-focused synchronization. For feature expression f_1, the two changes in variant *A* and variant *B* are listed. The unfamiliar developer does not know that the change in variant *B* depends on code that was changed in variant *A* and only merges the change of variant *B* into variant *C*. As a result, the merged code contains an error because the code requires the former change in variant *A*. So, variant *C* has incorrect code. Additionally, ordering conflicts could also appear if a change inside one feature expression might depend on former changes in other feature expressions of the same variant. If this dependency is not modeled with constraints on features or feature expressions, the synchronization will cause a conflict. The problem of ordering conflicts needs to be resolved on the implementation level, for example by introducing a global timestamp that orders the change propagation across different variants.

3.3.2 Source-Focused Synchronization

The source-focused synchronization focuses on propagating changes that are tagged to the same feature expression in different variants. This strategy includes that changes inside one variant need to be actively propagated to other variants. The workflow using source-focused synchronization is as follows: A developer works on a variant and implements new functionality. Applying code to feature tagging, changed code fragments

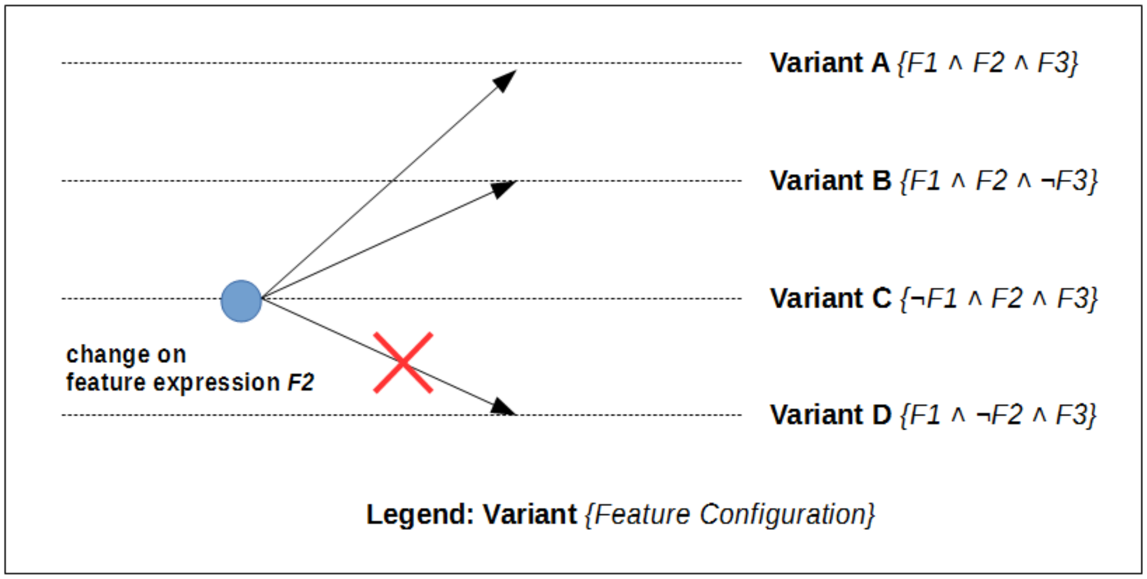

Variant A {F1 ∧ F2 ∧ F3}
Variant B {F1 ∧ F2 ∧ ¬F3}
Variant C {¬F1 ∧ F2 ∧ F3}
Variant D {F1 ∧ ¬F2 ∧ F3}
change on feature expression F2
Legend: Variant {Feature Configuration}

Figure 3.5: Example of a Source-Focused Synchronization

are mapped to feature expressions. Now, the developer chooses a feature expression of a variant and actively propagates these changes to compatible target variants. For example, in Figure 3.5, a change occurs in feature expression *F2* of an arbitrary variant. Applying source-focused synchronization, this change is actively merged into compatible target variants *A* and *B*, which both implement feature expression *F2*.
The source-focused synchronization has several advantages. If each change is regularly propagated to all compatible propagation targets, variants are synchronized. Regularly applying source-focused synchronization retains all variants on the same implementation level. Moreover, the amount of incorrect synchronizations that are caused by missing domain knowledge about the propagation target is reduced. One developer changes a variant and another developer, who is an expert on the propagation target, performs the change propagation and merges the changes into the target variant. Additionally, propagating all changes of a feature expression into all variants implementing that feature expression adjusts release cycles of variants. If each variant receives a change immediately after the change occurred, then all variants are synchronous concerning code changes. If a variant should be released, then this variant already contains all relevant changes that occurred in the meantime in other variants. This process is similar to the generation of variants in product lines. If generated variants are composed of reusable software artifacts, then the code of these artifacts is developed in the same release cycle. However, change propagation has several risks. Assume, a developer works on a variant and is an expert of this variant. If this developer tries to propagate changes into variants he is unfamiliar with, he could propagate changes into compatible target variants which do not need to be synchronized for distinct reasons. For example, variant *A* and variant *B* implement the same function with two different algorithms. The algorithm of variant *A* is imprecise but fast, whereas the algorithm of variant *B* computes pre-

cise results but is significantly slower. Thus, changes of the algorithm of variant A do not need to be synchronized into the algorithm of variant B, because both algorithms have completely different implementations. Furthermore, the consequent propagation of changes requires increased development effort. Additional to conventional development, the developer needs to either directly propagate a change after the change occurred or propagate a collection of changes after finishing the implementation applying source-centric synchronization in a *batch mode*, which we introduce in the following section. The longer changes are not propagated during development, the more the propagation effort increases. If changes are not directly propagated, then merge frequency will decrease. A decreased merge frequency, however, could lead to ordering conflicts during synchronization, as we explained in Section 3.3.

3.4 Synchronizing in Batch Mode

Using target-focused synchronization, the developer has to select changes in several sources to synchronize them into the target variant. Using source-focused synchronization, the developer has to propagate each change into selected and valid synchronization targets. However, synchronizing single changes causes effort for target-focused and source-focused synchronization. To reduce synchronization effort, we introduce a so-called *batch mode* that synchronizes a collection of changes. This batch mode automates the change propagation with source-focused synchronization as well as variant synchronization with target-focused synchronization.

3.5 Summary

In this chapter, we discussed benefits and drawbacks of variant development for few as well as many variants using clone-and-own or product-line techniques. We described the gap between single-system engineering and product-line engineering to develop a small number of variants which could increase over time. With *VariantSync*, we introduced into a light-weight and language-agnostic strategy to bridge this gap by making the development of few variants more efficient and reducing the effort for product-line migration. We presented a concept to automate the identification of synchronization targets by accumulating domain knowledge. For this purpose, we discussed different strategies to tag code to features. Moreover, we introduced into challenges to automate the synchronization between variants. We presented and discussed strategies to synchronize variants from different points of view. Finally, we introduced a concept to automate the synchronization of variants using the batch mode.

4. Implementation: Variant Synchronization with VariantSync

In Chapter 3, we presented *VariantSync* to support the development of variants by automating the synchronization. To establish an automated synchronization, we make domain knowledge explicit for tool support and synchronize changes in compatible target variants. In this chapter, we first discuss platforms on which *VariantSync* could be realized. Then, we describe our implementation of *VariantSync* on a synchronization platform to show that this concept is applicable. For this purpose, we implement an *Eclipse* plug-in for variant-based software development, called *VariantSync*.

4.1 Synchronization Platforms

In this section, we discuss platforms on which *VariantSync* could be realized. In many cases, variant development with clone-and-own uses tools from single-system engineering. Developers typically use an integrated development environment (IDE) to implement a software system, such as the *Eclipse IDE* or *Visual Studio*, and use a version control system to track their changes during development, such as *Git* or *Subversion*. Because developers are familiar with these tools, we discuss these tools as a possible *synchronization platform*. In order to implement *VariantSync* on a certain synchronization platform, the synchronization platform needs to support the automated identification of synchronization targets and the automated synchronization of changes. For this purpose, an appropriate synchronization platform should support the realization of following *VariantSync* tasks:

1. Apply a tagging strategy.
2. Detect changes on code fragments.
3. Synchronize changes using source-focused or target-focused synchronization.
4. Automatically synchronize changes in a batch mode.

To realize target-focused as well as source-focused synchronization, synchronization platforms need to be extended. In the following two subsections, we describe how to extend synchronization platforms and discuss benefits and drawbacks using a version control system or an integrated development environment as a synchronization platform. As part of the discussion, we refer how each synchronization platform realizes the previously specified functions.

4.1.1 Version Control System

As we introduced in Section 2.2.2 on page 19, version control systems track the development of software systems. In the following, we determine how version control systems support concepts of *VariantSync*. On the one hand, version control systems support the detection of changed code fragments using techniques to compute the differences between files as well as the synchronization using *merge techniques*. On the other hand, version control systems are not able to provide a code tagging without user-specific customizations. However, version control systems are not primary designed to be extensible regarding user-written extensions. As a consequence, extending a version control system to use it as a synchronization platform differs among concrete version control systems.
In the following, we exemplarily show possible extension concepts that are realized in concrete version control systems. For this purpose, we refer to *Git* [Cha15], *Mercurial* [Bab15] and *Subversion* [Fou15] as implementations of version control systems. Basically, *Git* consists of shell scripts and *Unix* commands that are summarized to a common library or a git executable. Extending *Git* requires to write own shell scripts and install them as a client-side or server-side hook [CS14]. A hook is a small program that is triggered by repository events [CSFP08]. Hooks perform distinct tasks and either run in advance of a repository event or after the completion of a repository event. Similarly to *Git*, *Subversion* can also be extended using repository hooks. Moreover, *Mercurial* also provides extensibility via hooks. As opposed to *Git* and *Subversion*, *Mercurial* additionally provides an API which supports the development of user-defined extensions, similarly to the plug-in concept of integrated development environments [O'S09].
As a reason to use version control systems as a synchronization platform, version control systems are widely used. Nearly each industrial or open-source software project uses a version control system to track changes or to collaborate on projects where several developers work on the same code. Furthermore, two core functions of each version control system are to compute the difference between two revisions and to merge one branch into another branch. Both functions are also tasks of *VariantSync*. Detecting

code changes between two revisions of a file can be achieved using techniques of the version control system to compute differences between files. Besides, synchronizing one change into a synchronization target can be supported using existing *merge techniques*. Nevertheless, there are some significant drawbacks extending version control systems to implement functionality of *VariantSync*. Version control systems do not provide all the necessary functions to perform *VariantSync*, such as applying code to feature tagging or visualizing tagging information. Unfortunately, extensibility depends on the concrete version control system. If the concrete system does not provide extension points, then this version control system is inappropriate to be extended for *VariantSync*. Furthermore, version control systems cannot perform code tagging. For this purpose, version control systems need to be extended. The extension needs to analyze the commits and map changed code fragments to features. To select which features need to be mapped to which code fragments, the extension needs a concept to communicate with the developer, for example committing specified files with mapping information or directly asking via command line or interacting via a graphical user interface (GUI). In addition, the developer does not get visual feedback about code tagging. The extension could admittedly provide a graphical user interface that visualizes tagging, like code highlighting in code editors of integrated development environments, but implementing a visualization requires a high effort. Instead, existing graphical user interfaces for version control systems could be extended. For example, *TortoiseHg*[1] is a graphical user interface for *Mercurial* which provides certain visualizations, such as a commit dialog that shows differences between commits or a view that shows the development history for different branches. *TortoiseHg* could be extended to visualize code to feature tagging. Nevertheless, adapting graphical user interfaces has two drawbacks. First, the graphical user interface is an external tool that is not integrated in an development environment. For each change, the developer needs to switch from the integrated development environment to the external tool to map code to features. Second, external tools does not have a concept to provide code tagging with feature-expression contexts. This concept needs additionally to be implemented. So, compared to the adaption of existing technologies in integrated development environments, the processes of mapping code to features and visualizing code tagging require high effort to either implement an extension from scratch or adapt existing graphical user interfaces.

4.1.2 Integrated Development Environment

An integrated development environment is a platform that supports software development processes by reducing development costs and increasing effectiveness regarding the development of application systems [New82]. Furthermore, an integrated development environment provides mechanisms to communicate with developers. Integrated development environments further ease the expression and understanding of designs and implementations [New82]. Regarding ongoing development of technologies supporting and facilitating development processes, integrated development environments need to be extensible concerning new data models, high-level language extensions and

[1] http://tortoisehg.bitbucket.org/, retrieved on 17.12.2015

system executability [New82]. Today, most modern and popular integrated development environments fulfill the requirement of extensibility. Modern integrated development environments are characterized by a plug-in architecture that enables developer to customize them to their needs by integrating plug-ins. For example, *Eclipse* and *Visual Studio* are the most popular integrated development environments, based on the fact how often integrated development environments are searched on *Google*[2]. Both are extensible via plug-ins. Furthermore, most integrated development environments support plug-in development providing tool support and an application programming interface, like the *Eclipse plug-in development environment* or *Visual Studio software development kit*. Consequently, integrated development environments can be extended to realize *VariantSync* as a plug-in.

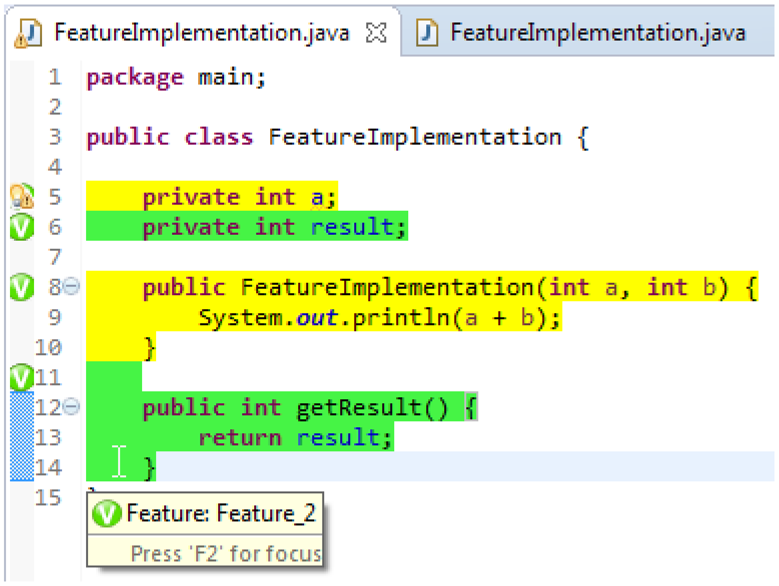
As an advantage, integrated development environments are widespread distributed in industrial as well as academic software development. Most popular and extensible development environments provide a kind of global market place that supports the distribution and installation of plug-ins. Hence, implementing *VariantSync* as an plug-in for an integrated development environment increases the probability to install *VariantSync* on a high number of target systems.

A further advantage is the presented extensibility concept. A plug-in can use comprehensive functionality provided by the integrated development environment or another plug-in. Most integrated development environments provide comprehensive code editors. Code editors facilitate development of source code providing functions like syntax highlighting or auto completion. Applying code to feature tagging in a code editor enables us to visualize feedback about tagged code using background highlighting and code annotations. For example, code mapped to a feature can be colored in a certain background color and an annotation at the left margin of the editor describes the mapped feature, as we show in Figure 4.1. All code fragments mapped to the same feature are colored in the same background color. Using background colors avoids code pollution and supports understanding of feature mapping [Käs10]. As a consequence, the developer retains the overview about code changes in different features, especially if many changes need to be synchronized. We use this concept of *code highlighting* in the implementation of *VariantSync*.

Moreover, the use of code editors supports tagging with feature-expression contexts. Before developing a feature, the developer only needs to activate the context. Then, each change in the code editor is directly and easily tagged to the active feature-expression context. As a result, tagging is performed during development.

Finally, code editors simplify the detection of code changes. Integrated development environments monitor code files that are managed in the given environment, for example a workspace in case of using *Eclipse*. File monitoring supports change detection notifying which code fragments are changed during development and need to be tagged. For example, a developer changes three functions of a code file and saves his work. To identify changed code, the difference between the two latest revisions of the changed file needs to be computed. Due to the fact that integrated development environments save

[2]http://pypl.github.io/IDE.html, retrieved on 14.12.2015



```
package main;

public class FeatureImplementation {

    private int a;
    private int result;

    public FeatureImplementation(int a, int b) {
        System.out.println(a + b);
    }

    public int getResult() {
        return result;
    }
}
```

Figure 4.1: Visualization of Code to Feature Tagging in *Eclipse*

the history of files, the two latest revisions of a file can easily retrieved and compared to identify changes. Admittedly, version control systems are also able to compute file changes, but they only compute changes after the files are committed. Indeed, integration development environments directly compute changes while the developer writes code. This has the advantage that the developer can directly tag the code. If the developer tags all changes after performing a commit, then he potentially could forget to which feature expression(s) the changes need to be tagged.
However, computing the difference between two files as well as merging changed code into a file do not belong to the core tasks of integrated development environments. Due to the facts that it is a challenging task to implement *diff & merge* functionality and that well-designed *diff & merge* algorithms already exist, it would be desirable to use existing implementations. As a solution, integrated development environments can be extended with plug-ins that provide *diff & merge* functionality, e.g., extending *Eclipse* with *Eclipse Compare Packages*.
Beside *diff & merge* functionality, the *VariantSync* workflow requires changes as compared to clone-and-own. A developer needs to be worked in to efficiently use the new plug-in. For example, implementing *Java applications* minimally requires an editor to create and change code files. *VariantSync* additionally needs to maintain a list of features for each variant, applying code tagging, starting target-focused or source-focused batch synchronization and resolving conflicts in case that the synchronization cannot be automatically performed.

4.1.3 Summary

Version control systems as well as integrated development environments are widely used and provide functions to support some of the tasks of *VariantSync*, but not all

of them. In particular, version control systems provide functionality to support change detection and change synchronization into given synchronization targets and integrated development environments provide functionality to support fine-granular tagging inside a context. *VariantSync* can be applied on both target platforms if they are extended to support missing tasks of *VariantSync*.
In contrast to version control systems, integrated development environments have the major benefit that tagging can be visualized with less effort, so that the developer can directly profit from visual feedback, like code highlighting.
Hence, we choose an integrated development environment as a synchronization platform to implement *VariantSync*, which is also in line with the prior prototype to support variant development by synchronizing fine-granular changes between variants [Luo12].

4.2 External Tools

In Section 4.1, we discussed version control systems and integrated development environment concerning their applicability to support *VariantSync*. Based on this discussion, we decided to use an integrated development environment as a synchronization platform. To realize *VariantSync* as an extension for an integrated development environment, we need several external tools. Obviously, we need an integrated development environment, which we introduce in Section 4.2.1. We further need to extend the integrated development environment with extensions to support tasks of *VariantSync*, as we described in Section 4.1.1 and Section 4.1.2. For this purpose, we use *FeatureIDE* (Section 4.2.2) to support feature modeling, and *java-diff-utils* (Section 4.2.3) to extend the integrated development environment with *diff & merge* functionality.

4.2.1 Eclipse IDE

We decide to develop *VariantSync* as an extension of an integrated development environment and use *Eclipse* as the technical synchronization platform. Our decision is based on three reasons.
First, *Eclipse* provides a comprehensive extensibility via plug-ins that enables developers to customize the environment with tailored functionality. *Eclipse* can customized to be a development environment supporting single software development of *Java applications* using *Java Development Tools*[3] (JDT plug-in) as well as custom development paradigms, like the development of product-lines. To fulfill tasks of *VariantSync* (Section 4.1), *Eclipse* provides code editors that support tagging and change detection, as we described in Section 4.1.2. Furthermore, extensions like *EMF Diff/Merge*[4] or other *diff & merge* packages support the synchronization of variants.
Second, using *Eclipse* enables us to adapt functionality of *FeatureIDE* to support feature modeling. For example, there are extensions that support modeling variability in a domain, such as the feature diagram editor of *FeatureIDE*[5]. We introduce *FeatureIDE*

[3]http://www.eclipse.org/jdt/, retrieved on 02.12.2015
[4]http://www.eclipse.org/diffmerge/, retrieved on 02.12.2015
[5]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/, retrieved on 02.12.2015

in the following section.
Third, *VariantSync* is independent of a programming language. For this purpose, the integrated development environment also needs to support software development independent of a specific programming language. The *Eclipse platform* provides software development independent of a programming language. To develop software with a specific programming language, the developer only needs to install an appropriate plug-in. For example, the *Eclipse platform* can be extended with the *JDT* plug-in to develop *Java applications*. In contrast, using the *C/C++ Development Tooling*[6] (CDT plug-in) enables the *Eclipse* platform to support the development of *C/C++* applications.
In the following two sections, we present *Eclipse* extensions to support feature modeling using *FeatureIDE* and *diff & merge* functionality using *java-diff-utils*.

4.2.2 FeatureIDE

> *FeatureIDE is an open-source framework for feature-oriented software development (FOSD) based on Eclipse. [TKB+14]*

FeatureIDE supports feature-oriented software development and provides tools to implement product lines in *Eclipse*. To give a short overview, *FeatureIDE* supports domain analysis by providing, among others, a graphical editor for feature modeling, a feature constraint editor and feature-model refactoring using the feature-model edit view. Requirement analysis is supported by a feature-configuration editor. Moreover, to support the domain implementation, *FeatureIDE* provides several product-line implementation techniques and languages like feature-oriented programming using *FeatureHouse* [TKB+14].
VariantSync uses and adapts several functions of *FeatureIDE* which consists of multiple *Eclipse features* that can be combined to create a tailored integrated development environment. In this context, an *Eclipse feature* is a bundle of one or more *Eclipse* plug-ins. Due to the reuse-friendly architecture of *FeatureIDE*, we can reuse functions of *FeatureIDE* by including these bundles into *VariantSync*. We use the plug-in *de.ovgu.featureide.fm.core* to manage and maintain features and feature configurations, including constraints on feature models. Furthermore, we use the graphical feature-model editor to enable the developer within the creation of feature models and we adapt the constraint editor to create and manage feature expressions. Both editors are located in the bundle *de.ovgu.featureide.fm.ui*. To summarize, *VariantSync* uses functions of *FeatureIDE* to model, manage and maintain features and feature expressions.

4.2.3 Diff Utilities

In Section 3.3 on page 34, we presented challenges to automate the synchronization of variants. The first challenge is to compute valid synchronization targets. As we presented in the previous section, we adapt functionality of *FeatureIDE* to determine which changes need to be synchronized into which variants. So, determining changes of files

[6]https://eclipse.org/cdt/, retrieved on 02.12.2015

and merging changes from one file into another file are core functions of *VariantSync*. Both functions need to compute the difference between two files or fragments of two files. Concerning a *diff & merge* implementation, we have three requirements:

- Compute the difference between two text files.
- Compute possible merge conflicts using a three-way merge technique.
- Perform a three-way merge.

```
Patch DiffUtils.diff(List<?> original, List<?> revised)
```

Listing 4.1: Computing the Delta of Two Text Fragments

```
List<?> DiffUtils.patch(List<?> original, Patch patch)
        throws PatchFailedException
```

Listing 4.2: Merging Text Fragements

Due to the fact that we want to synchronize software artifacts language-independent with *VariantSync*, we decide to use lexical *diff & merge* operations instead of syntactical *diff & merge* operations. To perform *diff & merge* operations on text files, *Eclipse* needs to be extended. We do not use *Eclipse Compare Packages* or *EMF Diff/Merge Packages*. In contrast, we decide to use *java-diff-utils*[7] which is a small open-source library providing a robust algorithm to determine the differences between plain text files [Mye86].
We use *java-diff-utils* to create a patch of two versions of a file to support the identification of merge conflicts between two files regarding a common ancestor, and to apply merging changes into a target file. In this context, a patch is the difference between two versions of the same file. We concretely use the function in Listing 4.1 to compute the difference between the original and revised sequence of text and the function in Listing 4.2 to perform a merge. To identify merge conflicts, we use an algorithm that checks whether the deltas of a three-way comparison are compatible. Deltas of two files are identified by computing the patch of these files. A delta describes a difference between two files. For this purpose, a delta describes the position of changed lines containing old lines and new lines of the difference. Applying the three-way merge technique, the algorithm compares two lists of deltas and determines whether deltas could be merged into the target file without causing a conflict.

4.3 Change Management

One of *VariantSync's* core tasks is to synchronize changes between variants. To achieve this, we need to manage changes in an appropriate manner. First, file changes during

[7]https://code.google.com/p/java-diff-utils/, retrieved on 02.12.2015

work sessions of a developer need to be detected. Second, these changes need to be persistently saved in a change history to decouple implementation from synchronization. For this purpose, we present a change management fulfilling both tasks in the following two sections.

4.3.1 Change Detection

To determine changes in variants, we use *Eclipse* functions. First, we register and implement the *IResourceChangeListener*. This listener notices changes on resources in the workspace. For example, the listener notices that a user saves a file in *Eclipse* or refreshes the *Eclipse* workspace. After a changed resource is determined, we evaluate the type of the change. If a resource was added or removed, we do not need to compute the difference between the previous and current version of the file. Instead, we save the content of the resource if the resource was added, and we save the name of the resource if the resource was removed. If the resource was changed, we need to determine all changes that occurred between the previous and the current save action.
Second, we retrieve the previous version of the changed file using the *IFile* interface. *Eclipse* saves past states of each file in the workspace in a list that is descending ordered by timestamps. So, we retrieve the history of a *IFile* object and get a list of past states. The first state is the version of the file that does not contain the change.
Third, we compute the difference between the previous version and the current version of the changed file using *java-diff-utils*. The *java-diff-utils* library returns the difference as a patch object. A patch describes the difference between original and revised text sequences. Listing 4.3 shows the method of *VariantSync* that implements the change computation after the *IResourceChangeListener* has notified a changed resource.

4.3.2 Change History

In order to decouple implementation from synchronization, we make change information explicit. For this purpose, we store code changes independent from the implementation of a variant in a change history. If a developer changes code in a variant and saves the changed file, then this code fragment will be tagged to a feature expression depending on the context. We persistently save the change information as a snapshot and add them to the change history, as we visualized in Figure 4.2. The change history is a first-in-first-out queue. A change entry in the change history consists of two snapshots of the changed file and a timestamp indicating when the change occurred. The first snapshot represents the version of the file before the change was performed. The second snapshot shows the file after the code was changed. Both snapshots are necessary for the posterior three-way merge because the first snapshot will be the common ancestor and the second snapshot will be the left version of the three-way merge, as we introduced in Section 2.3.1 on page 21. With the change history, we decouple the process of changing code fragments and propagating changes to other compatible variants.

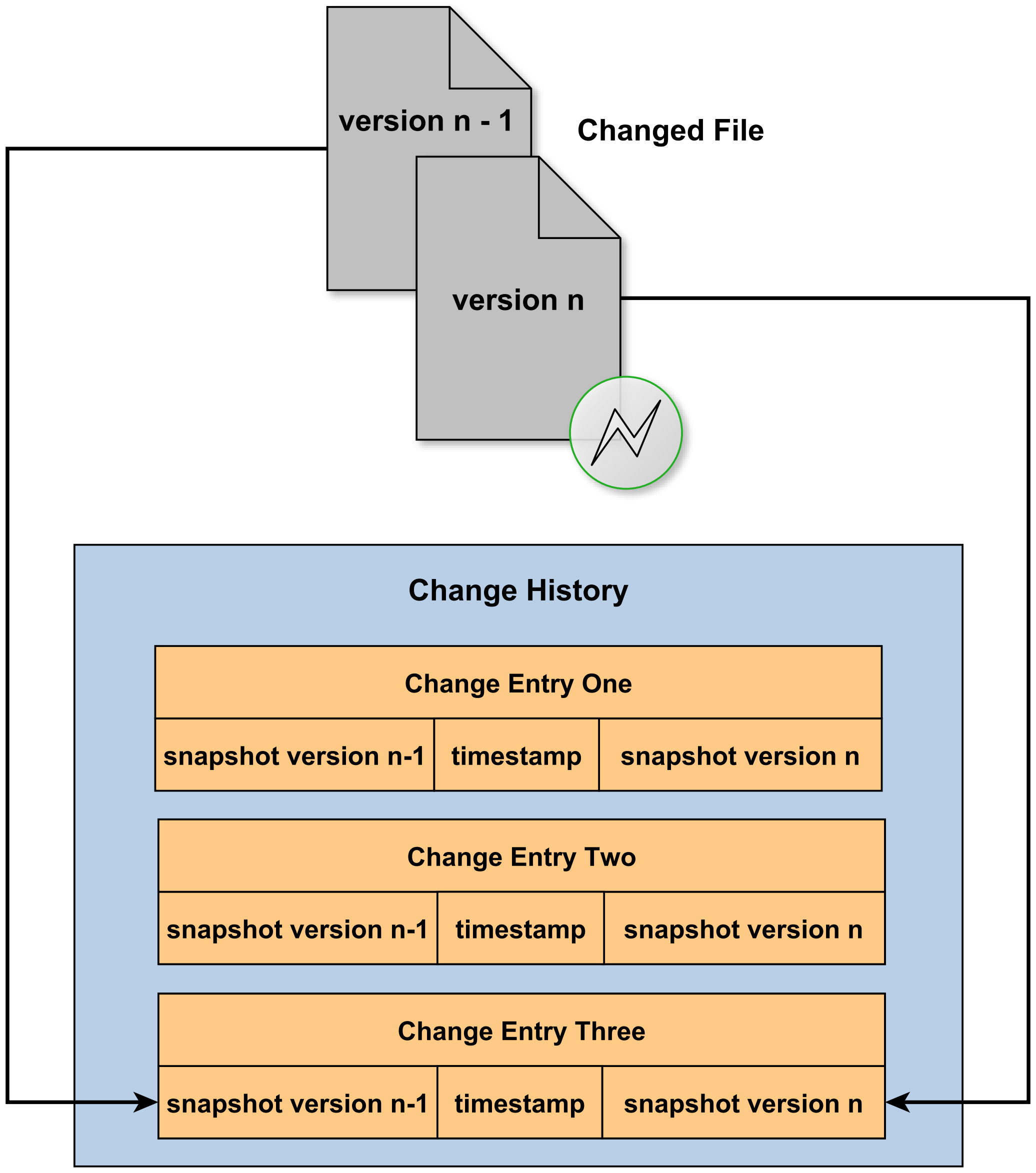


Figure 4.2: Change History

```java
/**
 * Computes the difference (delta) between the actual and ancient
 * version of the changed file.
 *
 * @param res
 *            changed file
 */
public void createPatch(IResource res) {
    IFile currentFile = (IFile) res;
    IFileState[] states = null;
    try {
        states = currentFile.getHistory(null);
    } catch (CoreException e) {
        LogOperations.logError(ERROR_FILE_STATES, e);
        return;
    }
    List<String> currentFilelines = Util.getFileLines(res);
    List<String> historyFilelines = null;
    try {
        historyFilelines = persistanceOperations.readFile(
                states[0].getContents(), states[0].getCharset());
    } catch (CoreException | FileOperationException e) {
        LogOperations.logError(ERROR_FILE_STATES, e);
    }

    Patch patch = externalDeltaOperations.computeDifference(
            historyFilelines, currentFilelines);
    [...]
}
```

Listing 4.3: Determining Changes of a File in *VariantSync*

4.4 Code to Feature Mapping

To automate the synchronization of variants, we need to compute valid synchronization targets that need to be synchronized with changes from a requirements perspective. In Section 3.2.1 on page 29, we pointed out that we need to make domain knowledge explicit for tool support to automate the identification of synchronization targets. For this purpose, we adopt functionality of *FeatureIDE* to create feature models, derive feature configurations and create feature expressions (Section 4.4.1). Then, we make domain knowledge explicit by applying the tagging with feature-expression contexts strategy (Section 4.4.2).

4.4.1 Feature Modeling

To model variability, the workspace needs to contain a project called *variantsyncFeatureInfo*. This project contains a feature model to describe variability for all variants. Furthermore, the project *variantSyncFeatureInfo* contains feature configurations derived from the feature model, where the variability of each variant is represented by

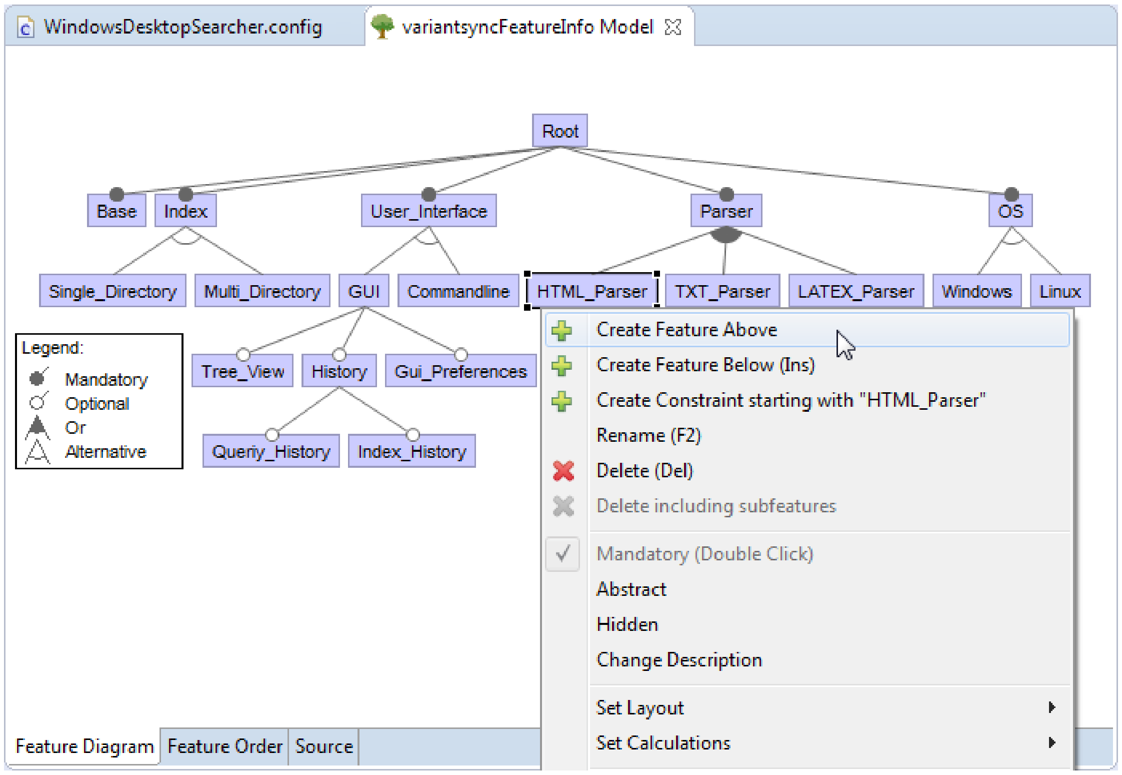
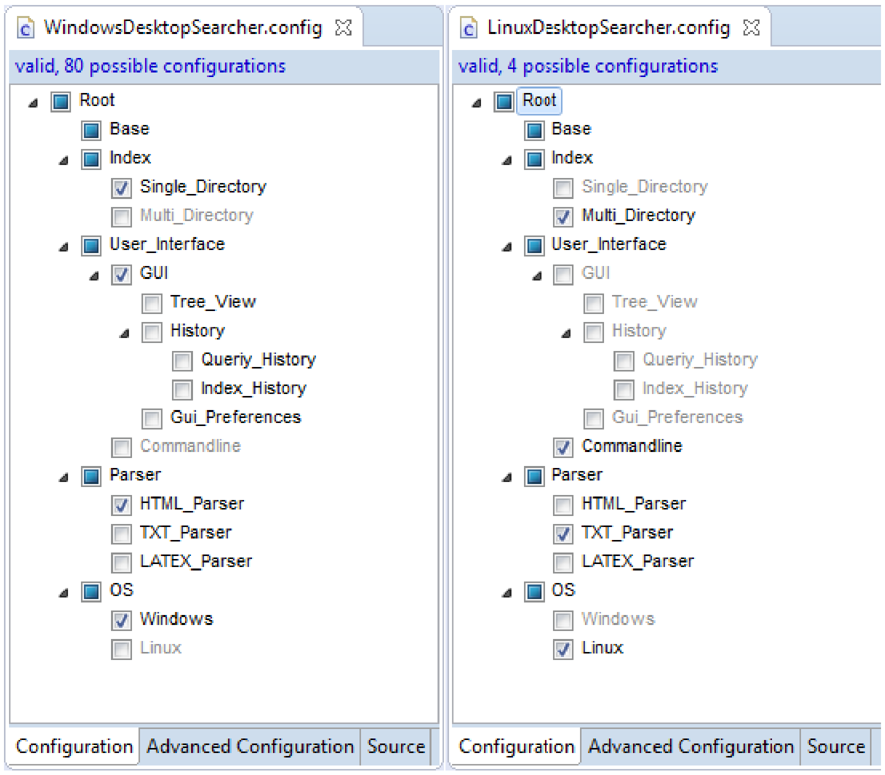


Figure 4.3: Feature Model Editor with the Feature Model of *DesktopSearcher*

a feature configuration. The feature configuration needs to have the same name than the variant it represents. For example, a variant called *WindowsDesktopSearcher* is described by a feature configuration with the name *WindowsDesktopSearcher.config*. This variant is an *Eclipse* project. To support the creation of feature models and the derivation of feature configurations in *VariantSync*, we use existing functionality of *FeatureIDE*. We reuse the graphical feature model editor as well as the feature configuration editor from the bundle *de.ovgu.featureide.fm.ui*. Figure 4.3 shows the feature model of the *DesktopSearcher* project that was created by the graphical feature model editor. The editor enables the developer to create or rename features as well as moving features via *drag & drop* [TKB+14]. Moreover, the feature configuration editor supports the developer to choose a feature selection based on the given feature model, as we show in Figure 4.4. Among others, the feature configuration editor indicates whether a chosen feature configuration is valid or invalid [TKB+14].
In general, it is not necessary that *VariantSync* uses a feature model for variability descriptions. *VariantSync* only needs a list of features that exist in a domain. For example, all features could be listed as optional features under the same root element. Furthermore, *VariantSync* only needs a list of features for each variant instead of feature configurations. However, we use feature models and feature configurations because they facilitate variant development, e.g., by detecting invalid feature relationships.
As we described in Section 3.2.2 on page 30, we use feature expressions to handle the feature interaction problem (Section 2.1.2 on page 12) as well as code that belong to



WindowsDesktopSearcher.config

valid, 80 possible configurations

- ■ Root
  - ■ Base
  - ■ Index
    - ☑ Single_Directory
    - ☐ Multi_Directory
  - ■ User_Interface
    - ☑ GUI
      - ☐ Tree_View
      - ☐ History
        - ☐ Queriy_History
        - ☐ Index_History
      - ☐ Gui_Preferences
    - ☐ Commandline
  - ■ Parser
    - ☑ HTML_Parser
    - ☐ TXT_Parser
    - ☐ LATEX_Parser
  - ■ OS
    - ☑ Windows
    - ☐ Linux

LinuxDesktopSearcher.config

valid, 4 possible configurations

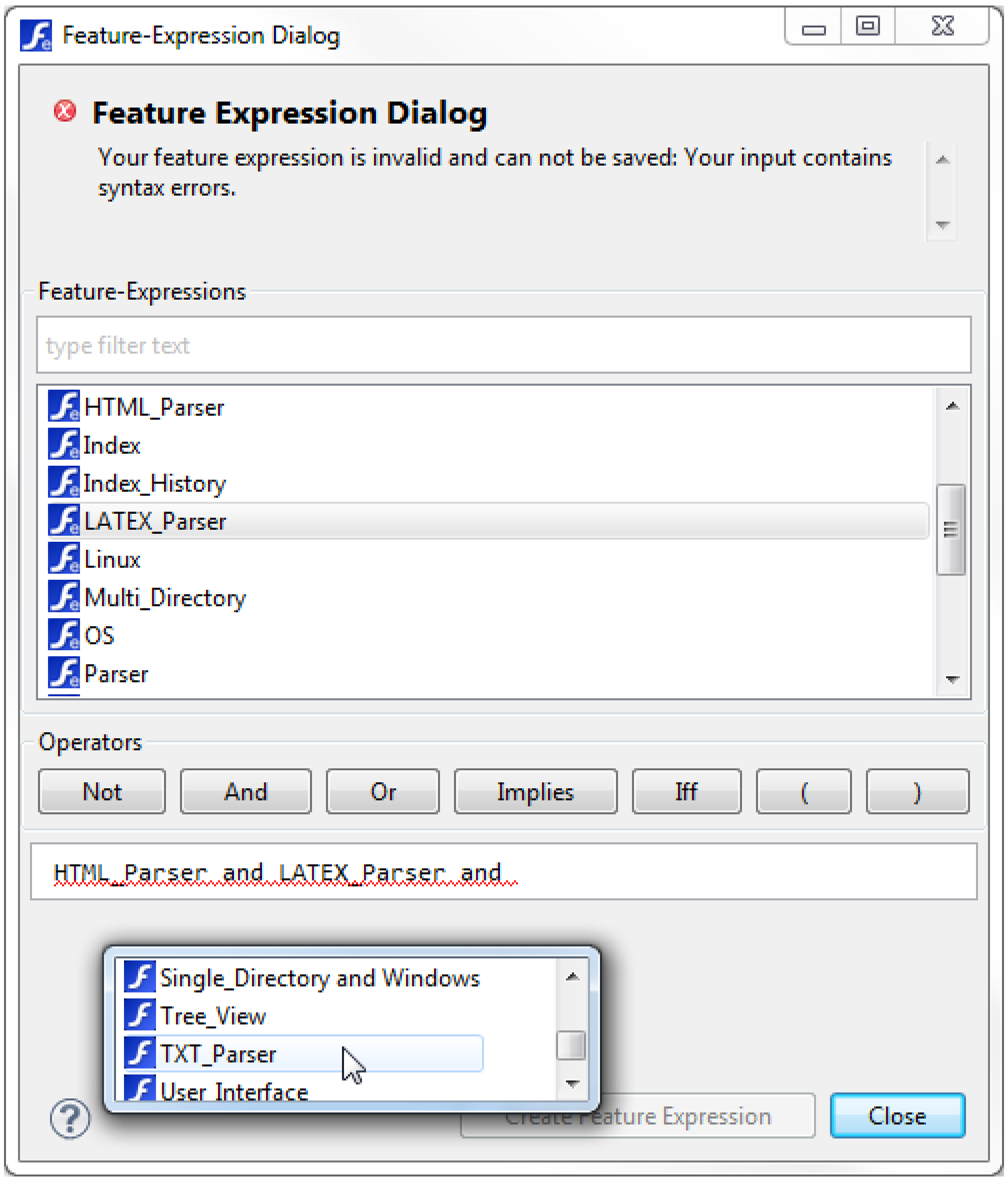
- ■ Root
  - ■ Base
  - ■ Index
    - ☐ Single_Directory
    - ☑ Multi_Directory
  - ■ User_Interface
    - ☐ GUI
      - ☐ Tree_View
      - ☐ History
        - ☐ Queriy_History
        - ☐ Index_History
      - ☐ Gui_Preferences
    - ☑ Commandline
  - ■ Parser
    - ☐ HTML_Parser
    - ☑ TXT_Parser
    - ☐ LATEX_Parser
  - ■ OS
    - ☐ Windows
    - ☑ Linux

Figure 4.4: Feature Configurations of Two Different Variants

multiple features. Using the feature model, we retrieve all features that are modeled in the domain. With the help of feature configurations, we know which features each variant implements. To create feature expressions derived from the feature model, we provide a graphical editor. The *editor for cross-tree constraints* of *FeatureIDE* enables the developer to create feature constraints for a given feature model by checking the syntax of the modeled constraint and providing a context assist during the creation of feature expressions [TKB+14]. We adapt the *editor for cross-tree constraints* to use it for the creation of feature expressions in *VariantSync*, as we show in Figure 4.5. Finally, we need to evaluate if a feature expression is included by a feature configuration. For this purpose, we represent both feature expressions and feature configurations as propositional formulas. We use a *boolean satisfiability problem solver* (SAT-solver) to determine whether the propositional formula of the feature configuration is satisfied by the propositional formula of the feature expression. To be more precise, we use the SAT-solver that is included in *FeatureIDE*, as we show in Listing 4.4.

4.4.2 Tagging Code to Features

Due to the fact that we decided to implement *VariantSync* as an *Eclipse* plug-in, we are able to implement manual tagging as well as tagging with feature-expression contexts. Furthermore, we are able to adapt the code editor of *Eclipse* to give the developer visual feedback about tagged code. In the scope of the prototypical implementation


Feature-Expression Dialog
Feature Expression Dialog
Feature-Expressions
type filter text
Operators
Not
And
Or
Implies
Iff
(
)
Create Feature Expression
Close

Your feature expression is invalid and can not be saved: Your input contains syntax errors.

HTML_Parser
Index
Index_History
LATEX_Parser
Linux
Multi_Directory
OS
Parser

HTML_Parser and LATEX_Parser and

Single_Directory and Windows
Tree_View
TXT_Parser
User Interface

Figure 4.5: Feature-Expression Dialog

```
/**
 * Determines whether the propositional formula of a feature
 * configuration is satisfied by the propositional formula
 * of a feature expression.
 *
 * @param fc
 *            maps feature names to literals
 * @param fe
 *            the feature expression to validate
 */
public boolean evaluateFeatureExpression(Map<String, Literal> fc,
    String fe) {
    final int TIMEOUT = 5000;
    Node[] formula = new Node[fc.size() + 1];
    formula[0] = new Literal(fe, true);
    i = 1;
    for (Literal literal : fc.values()) {
        formula[i++] = literal;
    }
    boolean isSatisfiable = false;
    try {
        final SatSolver solver = new SatSolver(new And(formula),
            TIMEOUT);
        isSatisfiable = satsolver.isSatisfiable();
    } catch (TimeOutException e) {
        [...]
    }
    return isSatisfiable;
}
```

Listing 4.4: SAT-Solver to Validate Feature Expressions against Feature Configurations

of *VariantSync*, we decided to implement the tagging with feature-expression contexts strategy to support the automated code to feature mapping. In the following, we first determine how we visualize tagged code in the *Eclipse* editor. Then, we describe how we implemented the tagging with feature-expression contexts strategy.

Visualizing Tagged Code

In Section 3.2.2 on page 30, we described our goal to avoid code obfuscation through tagging. Hence, we decided to avoid code annotations by storing code to feature mapping as metadata. For this purpose, we adopt the code visualization concept of *CIDE* (Colored IDE), an *Eclipse* plug-in supporting product-line development based on conditional compilation [Käs10]. *CIDE* has the goal to avoid obfuscated source code using *ifdef* preprocessor statements. To improve the understanding and to maintain code containing *ifdef* statements, *CIDE* visualizes features with colors. Each feature is assigned to a certain color. In the *Eclipse* editor, code that belongs to a feature will be colored in the assigned color. We use this concept to visualize the code to feature tagging in *Variant-Sync*. *Eclipse* provides direct access to the editor. Due to this possibility, we are able to visualize tagging information by highlighting code that belongs to a feature expression in a certain color. To realize code highlighting, we extend our plug-in with an extension point to create markers (*org.eclipse.ui.editors.markerAnnotationSpecification*). Figure 4.1 shows the visualization of tagged code. The class *FeatureImplementation.java* contains code that belongs to two different features. Code of *Feature_1* has a yellow background color, code tagged to *Feature_2* has a green background. A mouse-over event names the tagged feature expression as a tool tip for the underlying code. This representation of tagging information keeps the code clean and provides all information that an annotation would also provide. Unfortunately, product lines typically support several hundred features. So, domains describing all variants managed by *VariantSync* can also consist of a high number of features. It is not useful to map each feature to a different color because humans cannot distinguish a high number of similar colors without direct comparison. *CIDE* solves this problem by allowing a developer to assign the same color to more than one feature and indicating the feature of a colored code fragment by using tool tips [FKF+10]. We adapt this concept and introduce a color dialog with nine predefined colors, as we show in Figure 4.6. The developer can assign each feature expression to a predefined color. If a color is changed, then existing code highlightings in an open editor will be adapted.

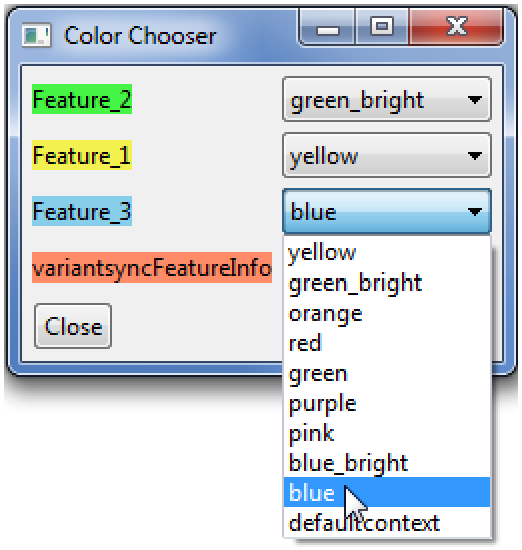

Feature_2 green_bright
Feature_1 yellow
Feature_3 blue
variantsyncFeatureInfo

Figure 4.6: Color Dialog of Feature Expressions

Implementation: Tagging Inside a Feature-Expression Context

To perform tagging inside a feature-expression context, the developer starts to develop under a context by selecting a feature expression, as we show in Figure 4.7. Then, each change in the *Eclipse* editor is automatically tagged to the chosen feature expression. If the developer switches or deactivates the context, or if a variant changes, tagged code will be saved persistently. In the following, we explain the process to automatically tag

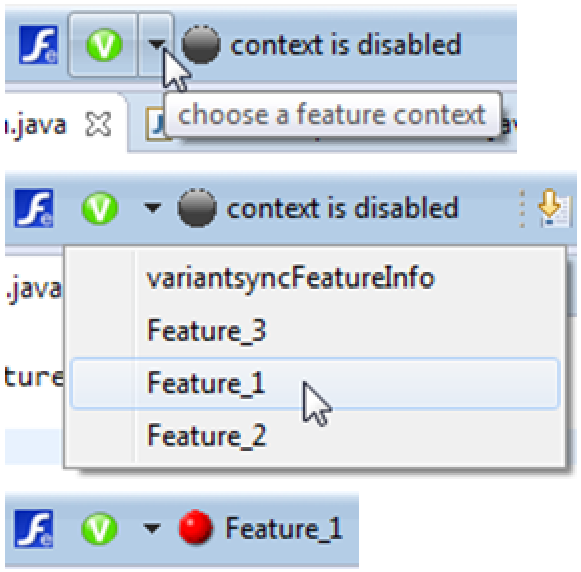


Figure 4.7: Activating a Feature-Expression Context

code changes to a feature expression in detail. We split the process into two steps. First, we describe the algorithm that activates a context. Then, we introduce the tagging algorithm. We further describe the interaction between the developer and *VariantSync*.

Figure 4.8 shows the process to activate a context. The developer wants to work on code inside a variant. To activate the context, the developer opens a list of available feature expressions. On the one hand, the list contains all features that are defined in the feature model of the domain of the variants. On the other hand, the list contains all user-defined feature expressions. Then, the developer chooses the feature expression he wants to work on. Now, the context is active.
While the context is active, all changes are tagged, as we described in Figure 4.9. The developer can work on one or several variants by adding, changing, or deleting code. If the developer saves a file, then *VariantSync* starts the process to tag the changed code to the feature expression of the context. The process consists of four steps:

1. Compute changes.
2. Tag changed code to a feature expression.
3. Save changed and tagged code.
4. Refresh the editor.

As the first step, we determine all changes that occurred between the previous and the current save action. As we described in Section 4.3.1, we retrieve the previous version of the changed file using *Eclipse functions*. Then, we compute the difference between

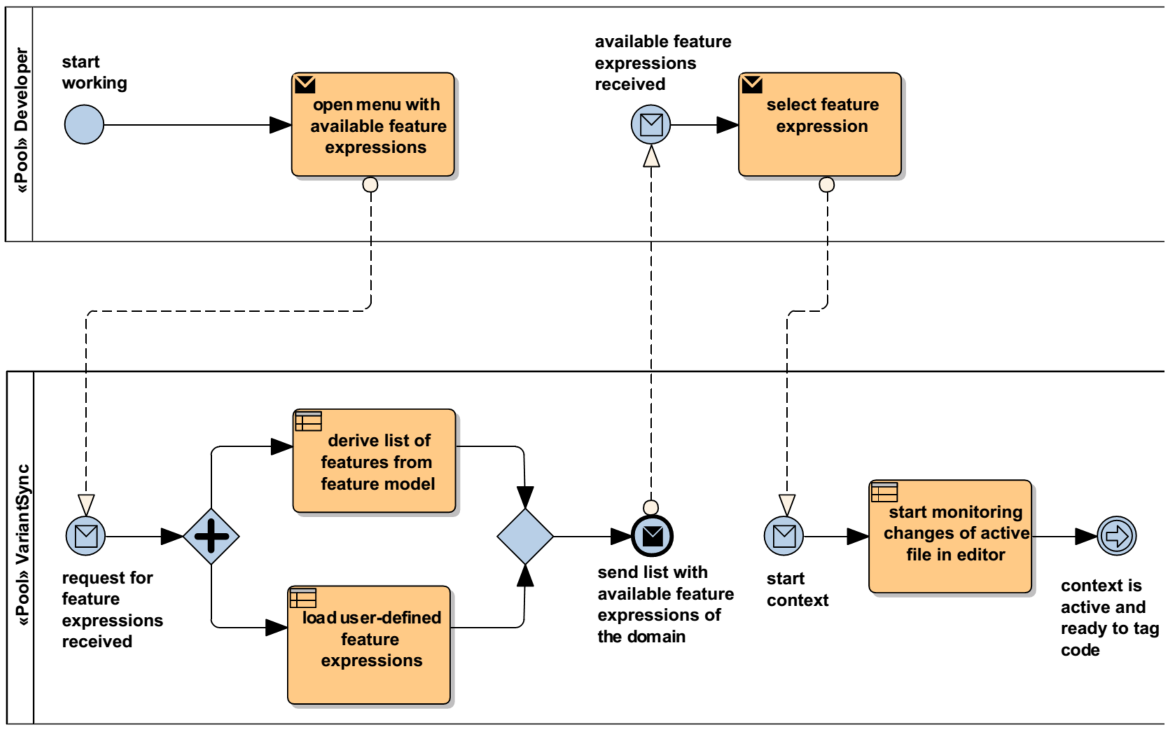

«Pool» Developer
start working
open menu with available feature expressions
available feature expressions received
select feature expression
«Pool» VariantSync
request for feature expressions received
derive list of features from feature model
load user-defined feature expressions
send list with available feature expressions of the domain
start context
start monitoring changes of active file in editor
context is active and ready to tag code

Figure 4.8: Process to Enable a Feature-Expression Context

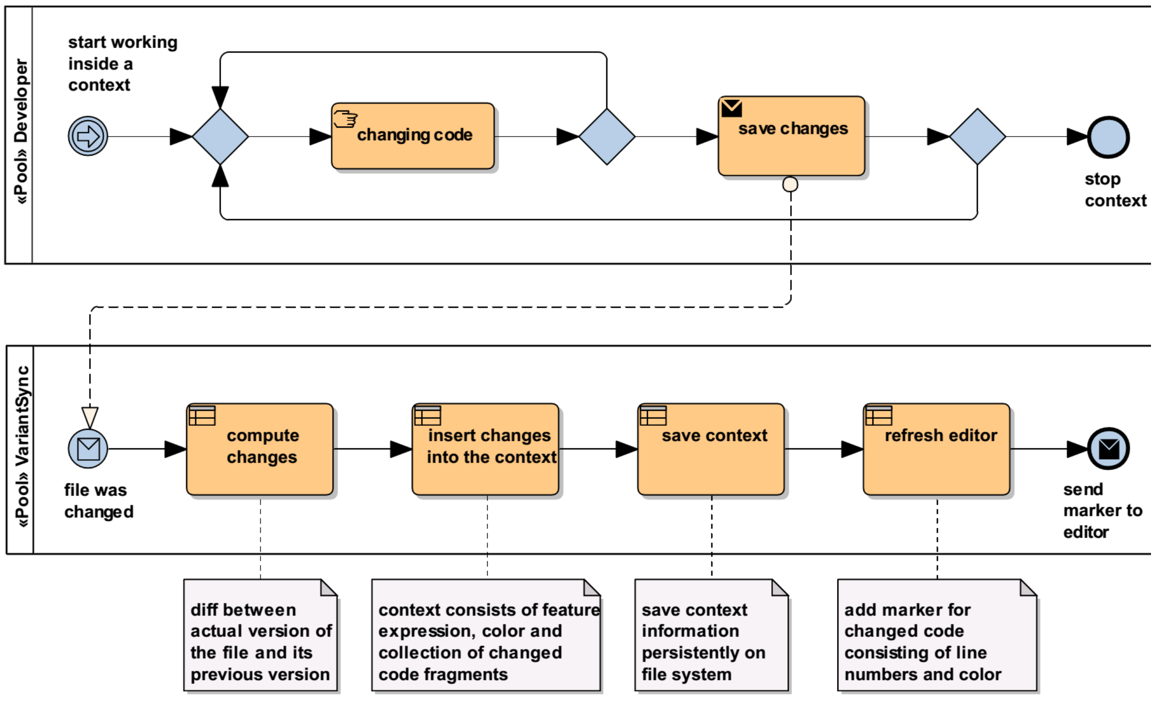

«Pool» Developer
start working inside a context
changing code
save changes
stop context
«Pool» VariantSync
file was changed
compute changes
insert changes into the context
save context
refresh editor
send marker to editor
diff between actual version of the file and its previous version
context consists of feature expression, color and collection of changed code fragments
save context information persistently on file system
add marker for changed code consisting of line numbers and color

Figure 4.9: Process to Automatically Tag Code to a Feature Expression Context

the previous version and the actual version of the changed file using *java-diff-utils*.
As the second step, we map the changed code to a feature expression by inserting changed code fragments into the data structure of the active context. A context consists of a feature expression, a color for code highlighting, a map to log synchronized changes and a list of variants that contain changes which are tagged to the feature expression of the context, as we show in Figure 4.10. If code of a variant is changed inside a context, then an object will be created. This object represents the changed variant and contains the changed code. In the data model, we describe variants in the same way that they exist in the workspace. So, a variant is represented by a collection of folders and files. This data model includes the change history that we introduced in Section 4.3.2.

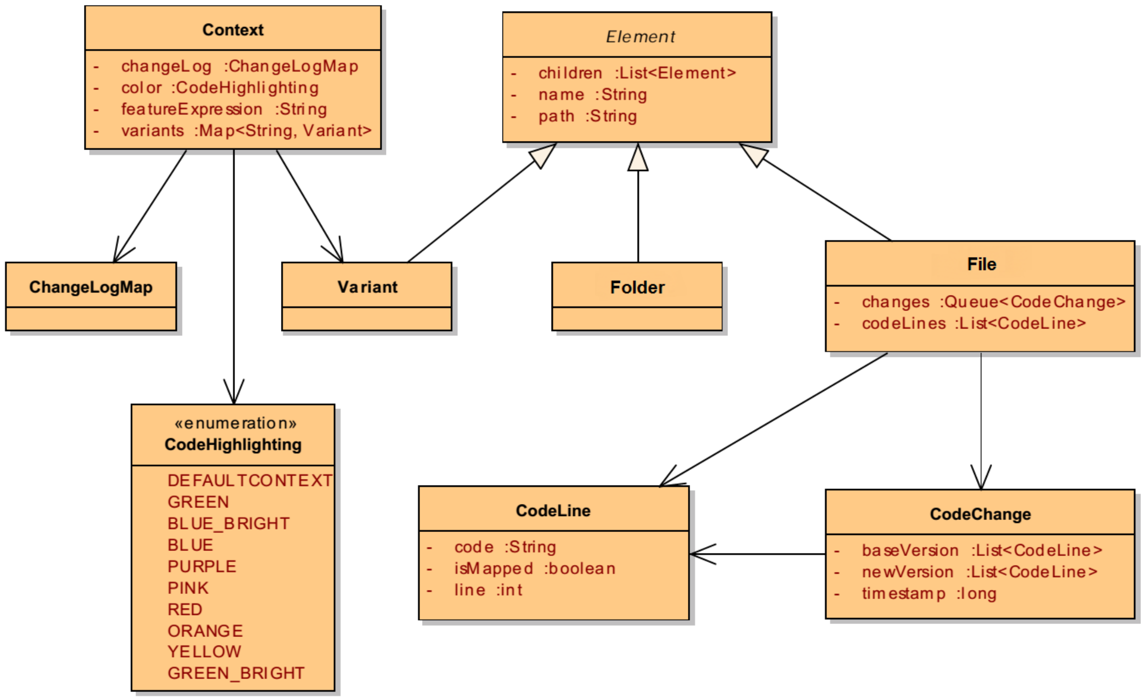


Figure 4.10: Data Model to Describe Feature Expression Contexts

As the third step, the presented data structure needs to be persistently saved. For this purpose, we save each context object in a *XML file* using *JAXB*[8]. *JAXB* is a framework that supports developers to transform *Java elements* to *XML representations* and vice versa. The *XML files* are saved in a sub folder of the *variantsyncFeatureInfo* project in the workspace. Listing 4.5 shows the content of a *XML file* that represents a feature-expression context.
As the fourth step, we refresh the editor by highlighting the changed and tagged code. For this purpose, we create a marker for each code change. A marker consists of a color, line numbers and the feature expression. Each marker is assigned to the changed file, highlights the tagged code and adds an annotation in the left bar of the editor by quoting the feature expression, as we show in Figure 4.1.

[8]http://www.oracle.com/technetwork/articles/javase/index-140168.html, retrieved on 06.12.2015

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Context>
 <changeLog/>
 <color>YELLOW</color>
 <featureExpression>HTML_Parser</featureExpression>
 <variants>
  <entry>
   <key>Variant_1</key>
    <value>
     <member xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
       xsi:type=javaPackage>
      <member xsi:type=javaClass>
       <changes>
        <change>
         <baseVersion code=public class HTML extends ContentHandler {
          line=31 mapped=false>
         [...]
         <newVersion code=public class HTML extends ContentHandler {
          line=31 mapped=false>
         [...]
        </change>
       <code>
        <codeline code=public void show(); line=13 mapped=true>
        [...]
       </code>
      <name>HTML.java</name>
      <path>Variant_1/src/html/HTML.java</path>
     <name>html</name>
     <path>Variant_1/src/html</path>
    </member>
    <name>Variant_1</name>
    <path>Z:/runtime-EclipseApplication/Variant_1</path>
    <variant>Variant_1</variant>
   [...]
```

Listing 4.5: Feature-Expression Context in XML Representation

4.5 Applying Automated Synchronization

In Section 4.4.1, we use concepts of feature modeling to model variability. Furthermore, we make domain knowledge explicit by applying the tagging with feature-expression contexts strategy. In this section, we automate the synchronization of changed code fragments. As the first step, we need to compute valid synchronization targets, as we described in Section 3.3 on page 34. To achieve this, we describe our implementation to detect synchronization targets in Section 4.5.1. Through the change history (Section 4.3.2), we are able to compute valid synchronization targets for changed code fragments from two different points of view. For this purpose, we implement two *Eclipse views* (Section 4.5.2 and Section 4.5.3). The target-focused synchronization view realizes target-focused synchronization, whereas the source-focused synchronization view implements source-focused synchronization. Both *Eclipse views* use our realization of automated and manual merging, which are presented in Section 4.5.4. Moreover, target-focused synchronization and source-focused synchronization can be automatically applied in a batch mode. We describe this batch synchronization in Section 4.5.5.

4.5.1 Computing Synchronization Targets

In Section 3.3 on page 34, we introduced three classes of synchronization targets: variants to synchronize, variants without merge conflicts and variants for automated synchronization. Hence, on coarse granularity, a valid synchronization target is a variant that is classified as a variant to synchronize. On the granularity of files, a file is a valid synchronization target if it meets the following requirements:

1. Synchronization targets have the same name and path as the changed file.
2. Synchronization targets are not located in the same variant as the changed file.
3. Synchronization targets are located in variants that implement the feature expression that the changed code is tagged to.
4. Synchronization targets have not already been synchronized with the given change.

Due to the data structure of a context (Figure 4.10), the first two requirements are easily met by iterating over the elements of a variant and comparing variant names and file names. To meet the third requirement, we evaluate the feature expression that the changed code is tagged to against feature configurations of target variants using a SAT-solver, as we described in Section 4.4.1. To fulfill the fourth requirement, we implement a change-log map. After propagating a change into a synchronization target, this change is logged in the change-log map. A change-log describes the change with its time-stamp and the synchronization target. The change-log map prevents a duplicated synchronization of a change into the same synchronization target. As we presented in Figure 4.10, the change-log map is part of the context data structure. Thus, the map is persistently saved together with change and tagging information.

4.5.2 Synchronizing with Target-Focused Synchronization

The target-focused synchronization view enables the developer to perform the target-focused synchronization strategy, which was presented in Section 3.3.1 on page 36. Using the target-focused synchronization view, a developer updates a variant. The target-focused synchronization view aggregates all code fragments that were changed in other variants. From a requirement's perspective, these changes can be synchronized into the chosen variant. Figure 4.11 shows the target-focused synchronization view while synchronizing a single change into a chosen variant. The combo box in ① lists all variants that are managed by *VariantSync*. The developer chooses the variant that he wants to update. Then, all feature expressions that contain changed code fragments of other variants are listed in ②. If the developer selects a feature expression, then

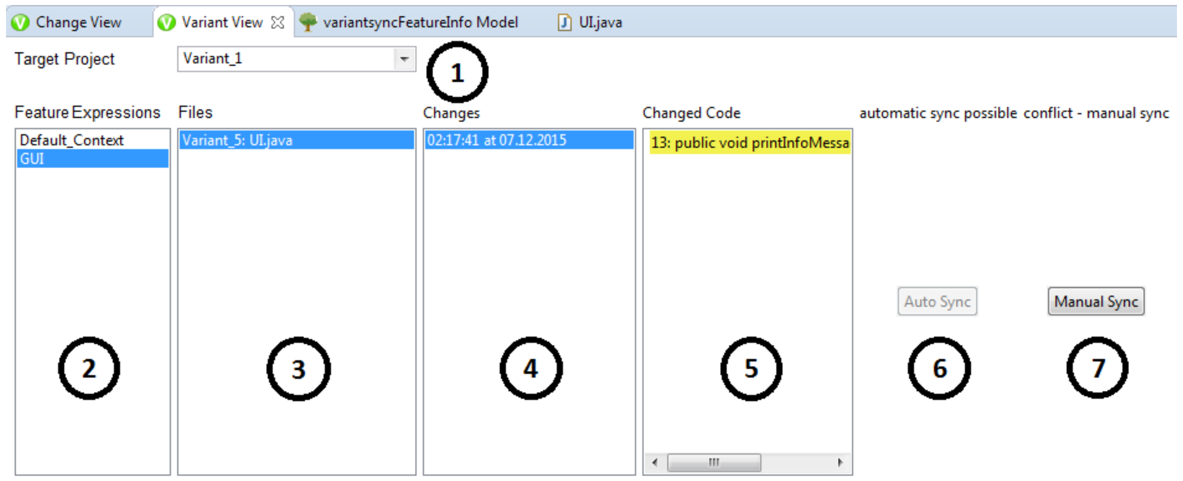


Figure 4.11: Synchronizing with the Target-Focused Synchronization View

all files containing changes mapped to the selected feature expression are shown in (3), except files of the variant that was selected in (1). After selecting a changed file in (3), the target-focused synchronization view displays all changes of the selected file in (4). Changes are ordered by timestamps. Now, the developer selects a change entry and the target-focused synchronization view shows the changed code fragment in (5) and determines whether the change can be automatically merged into the chosen variant. If the synchronization could be applied without a merge conflict, then the target-focused synchronization view automatically merges the change into the chosen variant in (6). Otherwise, the developer needs to merge the change manually in (7).

4.5.3 Synchronizing with Source-Focused Synchronization

The source-focused synchronization view enables the developer to perform the source-focused synchronization strategy, which was presented in Section 3.3.2 on page 38. Using the source-focused synchronization view, a developer actively propagates changed code fragments of a feature expression into valid synchronization targets. Figure 4.12 shows the source-focused synchronization view while synchronizing a single change. To perform change propagation, we decided to present the synchronization information from the highest to smallest change aggregation-level. Highest change aggregation-level is the feature expression that contains changed code fragments. In the combo box on top of the view in (1), the source-focused synchronization view lists all features that are modeled in the feature model of the *variantsyncFeatureInfo* project as well as all user-defined feature expressions that contain changes. The developer has to choose a feature expression. Then, the source-focused synchronization view lists all variants that contain changed code fragments tagged to the selected feature expression. The developer chooses one of the variants listed in (2). Afterwards, the source-focused synchronization view displays those classes of the selected variant containing changed code fragments tagged to the selected feature expression in (3). When the developer chooses

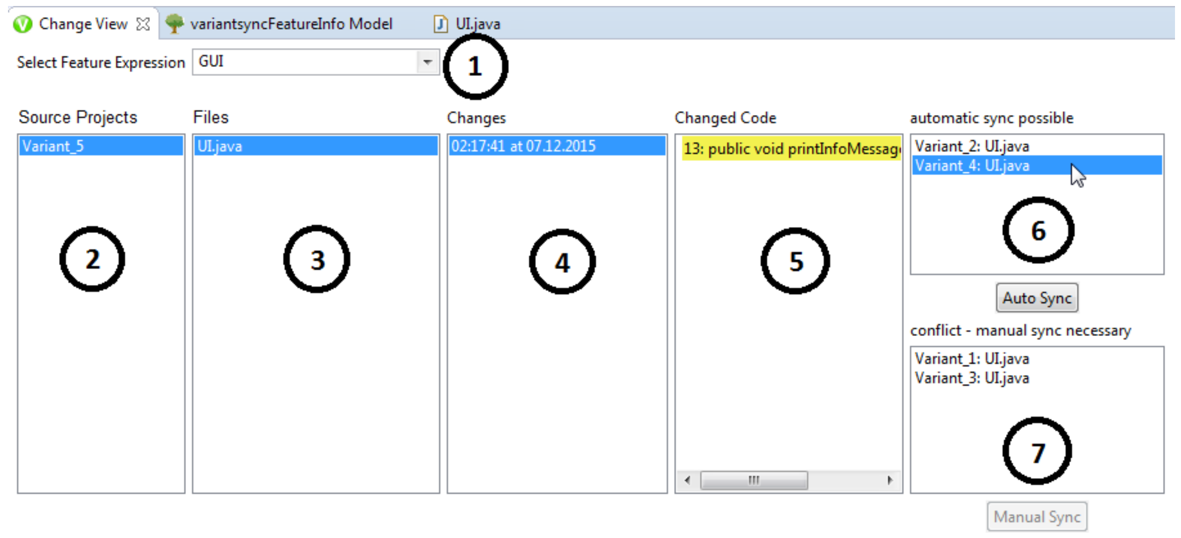


Figure 4.12: Synchronizing with the Source-Focused Synchronization View

a file, then all changes of the selected file are listed in (4), ordered by timestamps. After selecting a change, the changed code fragment is shown in (5) and the source-focused synchronization view computes valid synchronization targets for the automatic and manual synchronization. If changed code can be merged into a target file without causing a merge conflict, the synchronization will be automated in (6). However, if a merge conflict appears, the developer needs to perform a manual merge in (7). If a change is synchronized into all listed synchronization targets, then the change entry will be removed from the source-focused synchronization view.

4.5.4 Merging Code Fragments into Variants

In Section 4.5.2 and Section 4.5.3, we described the implementation of the target-focused and source-focused synchronization view. During the synchronization, we merge changed code fragments into valid synchronization targets. For this purpose, we use a lexical merge together with the three-way merge technique (Section 2.3 on page 21). If a changed code fragment can be merged into a target variant, then we will automatically perform the merge.

Automatic Merge

To detect merge conflicts and to perform the merge, we use *java-diff-utils*, as we presented in Section 4.2.3. We first perform the three-way comparison to prove whether changes of file one (changed version) could be merged into file two (version to synchronize). For this purpose, we adopted an algorithm from a prior prototype which also supports variant development [Luo12]. As we show in Listing 4.6, this algorithm uses *java-diff-utils* to compute the differences between common ancestor and changed version and between common ancestor and the version to synchronize. As a result, we receive two delta objects that describe the differences on a fine granularity in line positions. Then, the algorithm checks whether a conflict between both deltas exists. A conflict will be detected if text inside the same line differs in both deltas.

```
public boolean checkConflict(List<String> fOrigin, List<String> fLeft,
  List<String> fRight) {
  Patch patchAncestorWithLeftVersion = DiffUtils.diff(fOrigin,
    fLeft);
  Patch patchOriginWithRightVersion = DiffUtils.diff(fOrigin,
    fRight);
  List<Delta> deltasLeft = patchAncestorWithLeftVersion.getDeltas();
  List<Delta> deltasRight = patchOriginWithRightVersion.getDeltas();
  return checkConflict(deltasLeft, deltasRight);
}
```

Listing 4.6: Detecting Merge Conflicts using Three-Way Comparison

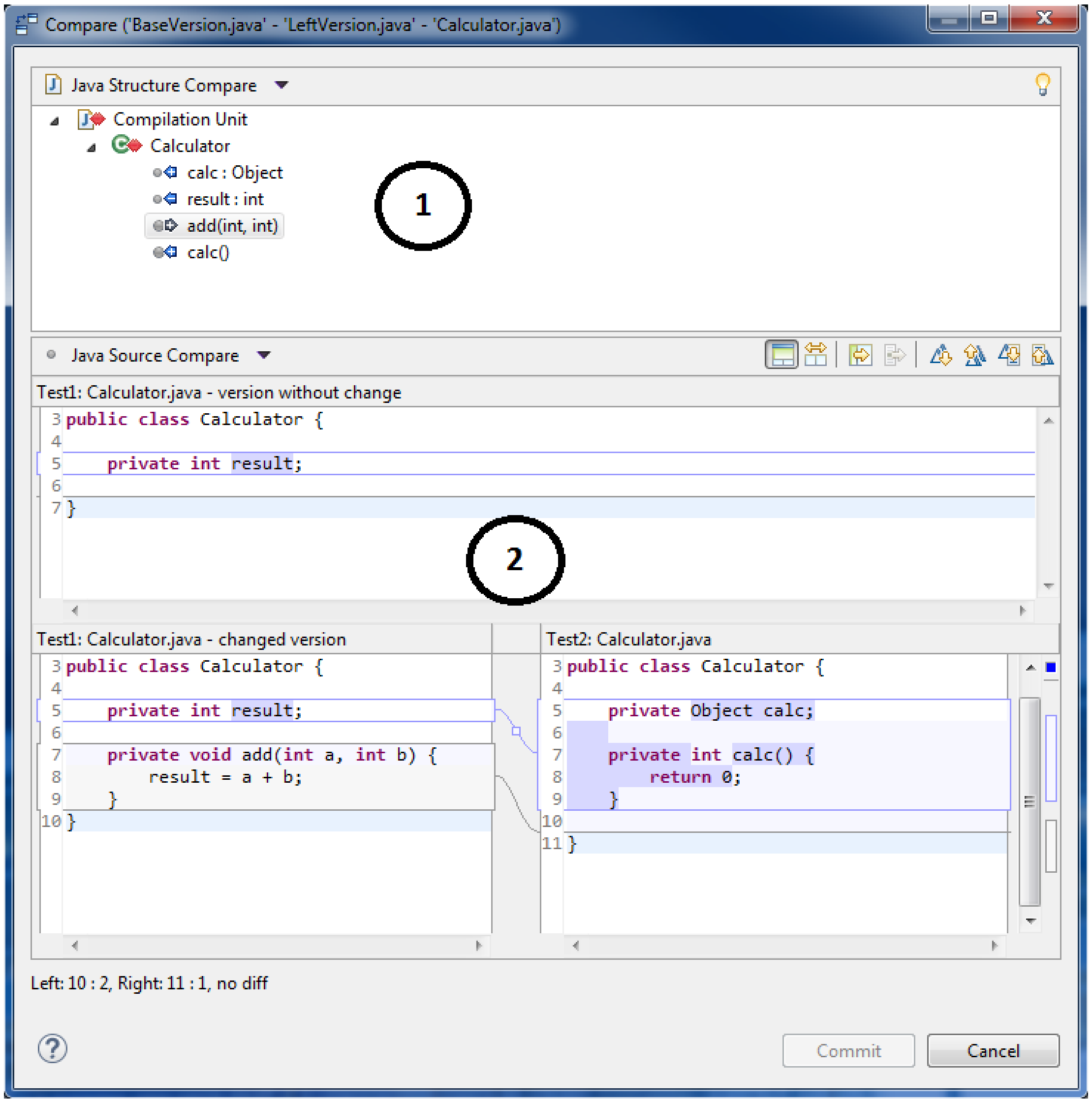
If differences between two files can be merged without a merge conflict, we perform an automated merge. For this purpose, we again use *java-diff-utils* and adopted a further algorithm from the prior prototype [Luo12]. We reuse the delta objects and create a patch object based on the deltas. Afterwards, we use *java-diff-utils* to merge the deltas into the original list. The result is a list containing each line of text of the merged class.

```
public List<String> performThreeWayMerge(List<String> fOrigin,
  List<String> fLeft, List<String> fRight) {
  [...]
  List<String> result = null;
  Patch patch = new Patch();
  Set<Delta> deltas = new HashSet<Delta>();
  deltas.addAll(deltasLeft);
  deltas.addAll(deltasRight);
  for (Delta d : deltas) {
    patch.addDelta(d);
  }
  try {
    result = (List<String>) DiffUtils.patch(origin, patch);
  } catch (PatchFailedException e) {
    [...]
  }
  return result;
}
```

Listing 4.7: Merging Two Files using Three-Way Comparison

Manual Merge

If the automatic merge process detects a merge conflict, then we will use the *Eclipse compare editor* to perform a manual merge. Figure 4.13 shows the *compare editor* comparing two versions of the file *Calculation.java*. The compare editor shows differences


Compare ('BaseVersion.java' - 'LeftVersion.java' - 'Calculator.java')
Java Structure Compare
Compilation Unit
Calculator
calc : Object
result : int
add(int, int)
calc()
1
Java Source Compare
Test1: Calculator.java - version without change
2
Test1: Calculator.java - changed version
Test2: Calculator.java
Commit
Cancel

```
public class Calculator {

    private int result;

}
```

```
public class Calculator {

    private int result;

    private void add(int a, int b) {
        result = a + b;
    }
}
```

```
public class Calculator {

    private Object calc;

    private int calc() {
        return 0;
    }

}
```

Left: 10 : 2, Right: 11 : 1, no diff

Figure 4.13: Manual Merge Dialog

between two files using a three-way comparison. If *Java files* are compared, then the editor will additionally parse the source code to compare the semantic structure of both files, as we show in (1). In this example, the structured comparison indicates four differences between left and right version, including the ancestor version for a three-way comparison. The right version contains the field *calc* of type *Object* and the method *calc()* that do not exist in the left version. Furthermore, the right version does not contain the field *result* that exists in the left version. However, the left version provides method *add(int, int)* for a merge into the right version.
The result of the source comparison is shown in (2) of Figure 4.13. *Calculator.java* of variant *Test2* differs in line five to ten from *Calculator.java* of variant *Test1*. Through the three-way comparison with the ancestor file (*Calculator.java* without changed code), we know that lines five to ten of the left version need to be merged into the right version. The editor provides a toolbar to navigate through the differences and to copy highlighted differences between left and right version. A difference is defined as a section that consists of one or more consecutive lines which differ between left and right version. Differences are highlighted in blue color, whereas a change is modified text within a line and highlighted in red color.

4.5.5 Synchronizing in Batch Mode

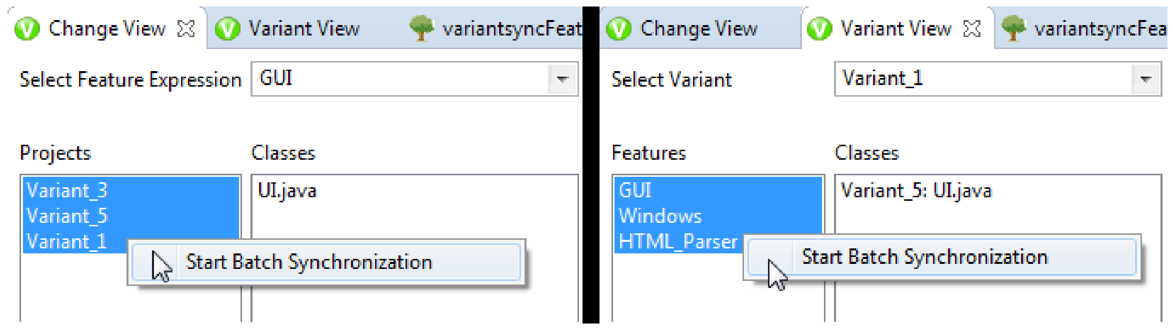


Figure 4.14: Start Batch Synchronization in Source-Focused (left) and Target-Focused Synchronization View (right)

As we presented in Section 4.5.2 and Section 4.5.3, the target-focused and source-focused synchronization view provide the synchronization of single changes into target variants. In both views, developers need to select feature expressions, variants, files, change entries and synchronization targets. As an advantage, the developer has full control about the change propagation. For example, if the developer does not want to merge a distinct change into a synchronization target, then he can skip this synchronization step. However, synchronizing single changes requires a lot of interaction between the developer and the view because the developer needs to interact with the view for each computation step. The batch mode automates the process of variant synchronization with the target-focused synchronization view and change propagation with the source-focused synchronization view.
As we show in Figure 4.14, the developer only needs to either select the variants or files

of variants whose changes he wants to propagate using the source-focused synchronization view. Besides, the developer needs to select feature expressions or files of feature expressions whose changes he wants to synchronize into the target variant using the target-focused synchronization view. Then, the batch mode collects all changes of selected elements and tries to automatically synchronize them. Using the source-focused synchronization view, changes will be propagated to all valid synchronization targets. Using the target-focused synchronization view, changes will be merged into the selected variant. The batch mode merges changes in the order that they occurred. Nevertheless, if a merge conflict occurs, the *compare editor* will open and the developer has to solve the conflict. Afterwards, the batch mode continues the automatic synchronization until a new merge conflict occurs or all changes are synchronized.

4.5.6 Increasing Code to Feature Mapping

To increase the code to feature mapping, we automatically tag code that was merged into a variant during the synchronization process. For this purpose, we first activate the feature-expression context of the change. Then, we perform the synchronization and merge code into the target variant. The feature-expression context notices that code of the target variant was changed, identifies the changed code fragment and tags this code. The tagging algorithm is similar to the algorithm presented in Figure 4.8 and Figure 4.9. The difference is that *VariantSync* takes the role of the developer. Instead of manual execution, *VariantSync* chooses the feature-expression, activates the context, performs changes on target files and saves the changes.

4.6 Limitations and Optimizations

In the previous sections, we presented the current state of the prototypical implementation of *VariantSync* as an *Eclipse* plug-in. In this section, we discuss open points which can improve *VariantSync*.
As a first optimization point, we could additionally introduce a syntactic merge to make the synchronization of variants more efficient. In this case, *VariantSync* would use a syntactic merge technique to perform the three-way merge on code files. As an advantage, the amount of merge conflicts would decrease (Section 2.3.2 on page 22). However, syntactic merging requires a special parser for each programming language. We would combine lexical and syntactic merge by merging non-code files with the lexical merge technique and merging code files with the syntactic merge technique, if the appropriate programming-language parser would be implemented.
As a second optimization point, we could enable a code to feature mapping that allows parallel modifications on the same file by different developers. Currently, tagging inside feature-expression contexts does not lock the files on which the developer works. We could either evaluate whether *Eclipse* locks files in a manner that parallel modifications are not possible or develop a strategy for parallel tagging.
As a third optimization point, we could evaluate if problems occur when feature-expression contexts are tracked by a version control system and distributed to different

developers. We assume that each developer can work on the same context in a different workspace if the *XML files* containing the context information are available. However, this use case is not yet tested.
As a fourth optimization point, we regard the user interface. On the one hand, we could enable the source-focused and target-focused synchronization view to be more configurable. Currently, both views have the limitation that a developer can neither select multiple variants in the target-focused synchronization view nor select multiple feature-expressions in the source-focused synchronization view. Especially for batch synchronization, this limitation prevents the batch synchronization of several feature expressions or variants. On the other hand, we could support the selection of a feature-expression context. Currently, if the developer wants to activate a context, then all feature expressions of the domain are listed in a drop-down menu. We could determine which variant is actually opened in the editor. Then, we would be able to provide a feature-expression selection that only contains feature-expressions implemented by the variant on which the developer is working. Finally, we could increase the readability of tagged code by adapting the color concept of *FeatureIDE*, which also uses colors to highlight code. We would replace our existing background colors with colors that are optimized to improve the readability of overlaying code.

4.7 Summary

In this chapter, we presented our implementation of *VariantSync*. We discussed synchronization platforms as well as our decision for integrated development environments as best suiting synchronization platforms for our purposes. Furthermore, we discussed our decisions to implement *VariantSync* as an *Eclipse* plug-in and to adopt and extend functionality of *FeatureIDE* by introducing domain knowledge into *VariantSync*. We presented and discussed details of our design decisions concerning change detection in variants, decoupled implementation from synchronization, code to feature mapping and automated variant synchronization. In particular, we presented our solutions to automatically compute valid synchronization targets, to actively propagate changes into variants, to synchronize changes into variants, to synchronize aggregated changes in a batch mode and to automatically increase the code to feature mapping during the synchronization. Finally, we presented possible improvements of *VariantSync* that were out of the scope of this thesis.

5. Evaluation

In this chapter, we evaluate the efficiency of variant development using *VariantSync*. Moreover, we illustrate how *VariantSync* supports the migration to a product line. In Section 5.1, we discuss our research questions which we use to evaluate *VariantSync* and explain our measurements. To answer our research questions, we apply *VariantSync* to develop five similar variants. For this purpose, we introduce the *DesktopSearcher* product line as a target system providing variants as well as their development history (Section 5.2). Furthermore, we describe how we use the *DesktopSearcher* product line to simulate variant development in Section 5.3. In Section 5.4, we present our measurements during variant development concerning the research questions. Finally, we discuss our results in Section 5.5 and summarize the evaluation in Section 5.6.

5.1 Research Questions

To evaluate *VariantSync*, we illustrate the applicability and implementation of *VariantSync's* core tasks. On the one hand, the evaluation shows how *VariantSync* makes the development of few variants more efficient compared to single-system engineering approaches, like clone-and-own. On the other hand, we evaluate how *VariantSync* supports the migration to a product line by establishing a code to feature mapping. As we introduced in Section 3.3 on page 34, *VariantSync's* core tasks are to decrease synchronization effort (1, 2), compute valid synchronization targets (3) and increase code to feature mapping (4). Referring to these tasks, we want to answer the following research questions:

- **RQ 1**: *How many changes are automatically merged into valid target variants?*
- **RQ 2**: *How many changes are grouped with the same context and synchronized in the batch mode?*

- **RQ 3**: *How many variants are valid synchronization targets for the change propagation of one context?*
- **RQ4**: *How much code is tagged to features during the synchronization?*

RQ 1 evaluates the potential for automation. Referring to Figure 3.3 on page 35, *VariantSync* automatically merges changes into variants that need to be synchronized and that can be synchronized without a merge conflict. A reduced amount of merge conflicts increases the degree of the automated merge and decreases the synchronization effort for developers. We evaluate how many changes can be automatically merged using a lexical merge implementation and how many changes need to be manually merged by a developer due to merge conflicts. Thus, the more conflicts occur, the lower is the potential for automation.
RQ 2 evaluates the potential for feature-expression contexts and the batch mode. Changes are collected by applying the tagging inside feature-expression contexts strategy during variant development. Without *VariantSync's* batch mode, developers need to separately propagate each change into target variants. We evaluate how much developers benefit using *VariantSync's* batch synchronization. For this purpose, we compare the number of consecutive changes that can be synchronized with the batch mode for multiple feature-expression contexts.
RQ 3 addresses the computation of valid synchronization targets. We evaluate in how many contexts changes of one context need to be propagated. Referring to Figure 3.3 on page 35, we evaluate the number of variants that need to be synchronized with a given change. In detail, we compare the number of computed synchronization targets with the number of synchronization targets that are manually determined and expected by a developer that is familiar with the domain. If the number of target variants is low, then the potential for variant synchronization will also be low.
RQ 4 illustrates the potential for supporting the migration to a product line. To establish a code to feature mapping, *VariantSync* implements the tagging with feature-expression contexts strategy. On the one hand, we evaluate the amount of tagged code during variant development with *VariantSync*. Besides tagging with feature-expression contexts, *VariantSync* also tags code by merging changes into variants. So, on the other hand, we evaluate whether *VariantSync* increases the degree of code to feature mapping during variant synchronization.

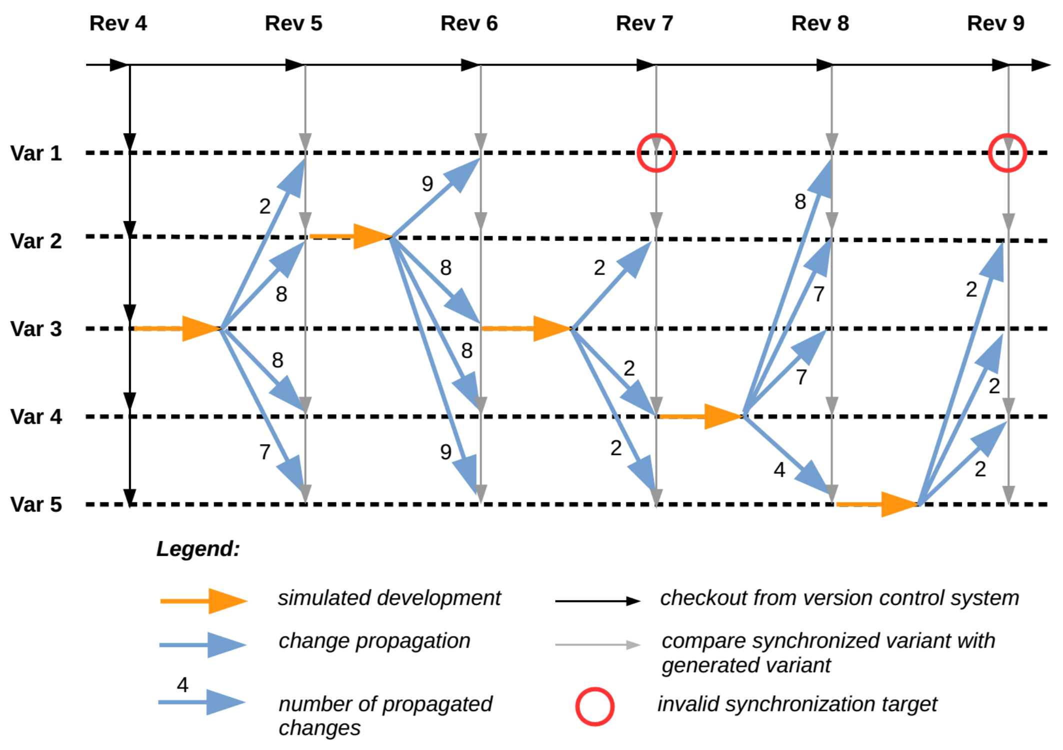
5.2 Target System

To perform the evaluation of *VariantSync*, we need to examine the development of similar variants using *VariantSync*. During the development process, we need to monitor information to answer our research questions, like determining the number of automated and manual merges, the number of contexts in which a change needs to be merged or the number of tagged lines per variant. However, monitoring the development of multiple variants is a time-consuming task that exceeds the scope of this thesis. Instead, we perform a simulated case study based on the development history of an existing

project. Thus, we simulate the development of similar variants over a distinct number of development steps.
For this purpose, we decide to simulate the development of variants that are generated by the *DesktopSearcher* product line. Introduced as a running example in Chapter 2, *DesktopSearcher* contains 22 features which allow us to generate 462 different variants. Figure 2.1 on page 12 shows the feature model at the end of the development process of *DesktopSearcher*. Furthermore, *DesktopSearcher* enables us to simulate the feature-oriented development of its variants due to two reasons. On the one hand, the development of *DesktopSearcher* was tracked with a version control system and provides a development history of 91 revisions. For a distinct revision, we generate five variants that differ in various features. Regarding differences between variants of two consecutive revisions, we are able to reproduce changes on variants to simulate the development. On the other hand, *DesktopSearcher* is modularized by applying the concept of feature-oriented programming using the AHEAD tool suite [Bat05a]. AHEAD defines features as building blocks of systems which encapsulate fragments of classes. Each feature is organized in a directory containing artifacts that belong to this feature. If a variant is generated, then the feature modules will be composed. That means corresponding files are recursively composed. This modularized architecture supports us to determine which change belongs to which feature.
So, we simulate variant development by implementing changes into one variant with an active feature-expression context and synchronizing these changes with variants.

5.3 Simulating Variant Development

Goal of the simulation is first to reproduce each change which is implemented for a certain revision in one variant and second to propagate these changes into valid synchronization targets. We define a *change* as a collection of consecutive change operations concerning one file. For example, the code snippet in Figure 5.4 shows two changes. First, Line 36 in Revision 4 is replaced with an empty line in Revision 5. Second, Line 45 in Revision 4 is replaced with six new lines in Revision 5. Furthermore, we define the following *change operations*: adding files or lines inside files, changing lines inside files, removing files or lines inside files. While reproducing changes, we apply the tagging with feature-expression contexts strategy. We simulate the application of *VariantSync* from the beginning of the variant development process. For this purpose, we simulate variant development between Revision 4 to 9. Furthermore, we simulate variant development between Revision 57 to 64 to evaluate the application of *VariantSync* when it is introduced while variants have already been developed over a period of time. We select Revisions 57 to 64 because these revisions contain changes on code files and they are nearly in the middle of the development process of *DesktopSearcher*. Moreover, we summarize the development between Revision 58 to 61 and perform the changes as single development step, because many changes do not affect code files.
Figure 5.1 visualizes how we simulate the variant development with *VariantSync*. We perform the following steps:



Rev 4
Rev 5
Rev 6
Rev 7
Rev 8
Rev 9
Var 1
Var 2
Var 3
Var 4
Var 5
2
8
8
7
9
8
8
9
2
2
2
8
7
7
4
2
2
2
Legend:
simulated development
change propagation
4
number of propagated changes
checkout from version control system
compare synchronized variant with generated variant
invalid synchronization target

Figure 5.1: Simulated Variant Development with the History of an Existing Product Line

1. Generate variants.

2. Reproduce the development of one variant between two revisions.

3. Propagate changes of step two into valid synchronization targets.

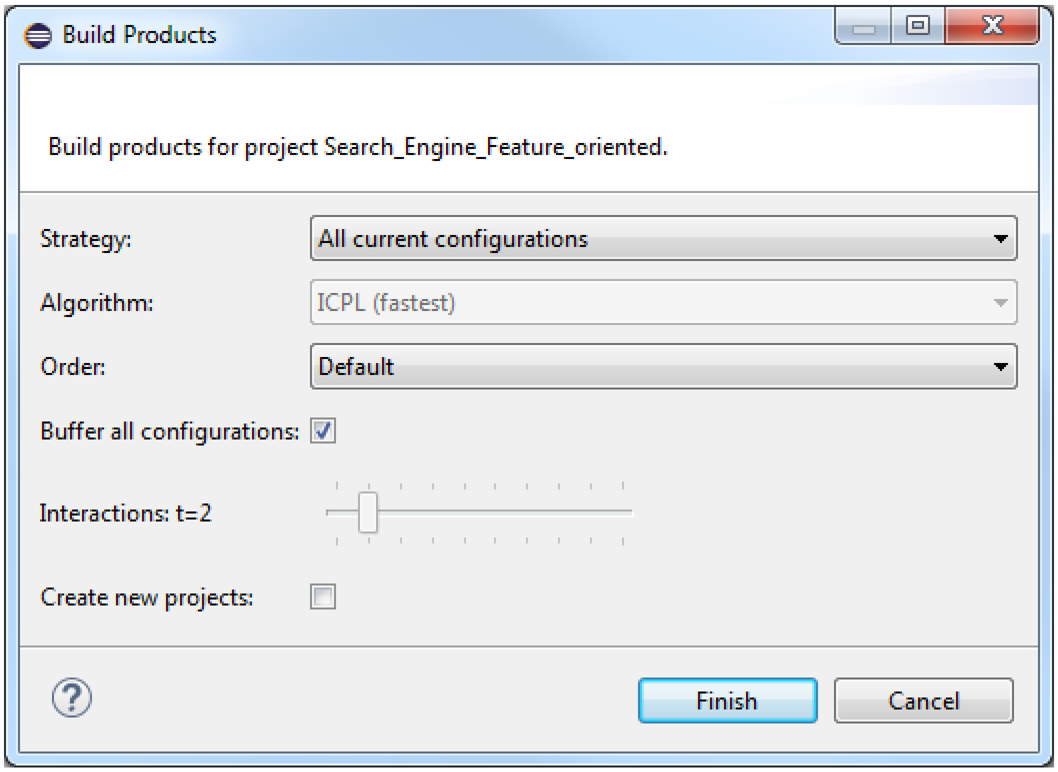
4. Compare the synchronization with the development history.

As the first step, we need to generate variants whose development we want to simulate. To achieve this, we first inspect revisions of the product line to detect the first revision containing code that enables us to generate variants. As a result, we choose revision four. Then, we determine feature configurations for five different variants, as we defined in Table 5.1. Finally, we use *FeatureIDE's* product generator to generate the variants, as we show in Figure 5.2.
As the second step, we reproduce the development of our variants between Revision 4 and 5. For this purpose, we compute the difference between both revisions using the *compare editor* of *Subclipse*.[1] The *compare editor* shows differences between feature modules. Between Revision 4 and 5, the compare editor in Figure 5.3 shows changed and added files that belong to the features *Base*, *GUI*, or *HTML_Parser*. Then, we

[1]Eclipse plug-in providing support for Subversion: http://subclipse.tigris.org/, retrieved on 11.12.2015

Feature	Var$_1$	Var$_2$	Var$_3$	Var$_4$	Var$_5$
Root	x	x	x	x	x
Base	x	x	x	x	x
Index	x	x	x	x	x
Single_Directory		x		x	x
Multi_Directory	x		x		
User_Interface	x	x	x	x	
GUI		x	x	x	x
View	x	x	x	x	x
Tree_View			x		x
Normal_View		x		x	
History		x	x	x	x
Query_History		x		x	x
Index_History				x	x
GUI_Preferences		x	x		x
Commandline	x				
HTML_Parser		x	x	x	
TXT_Parser	x	x			x
LATEX_Parser				x	x
OS	x	x	x	x	x
Windows			x	x	x
Linux	x				

Table 5.1: Feature Configurations of Generated Variants



Build products for project Search_Engine_Feature_oriented.

Strategy: All current configurations

Algorithm: ICPL (fastest)

Order: Default

Buffer all configurations: ✓

Interactions: t=2

Create new projects: ☐

Figure 5.2: Generating Variants of an AHEAD Product-Line with *FeatureIDE*

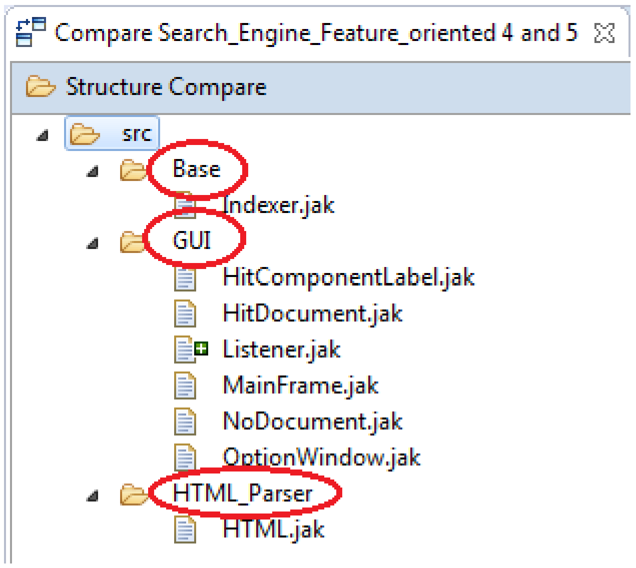
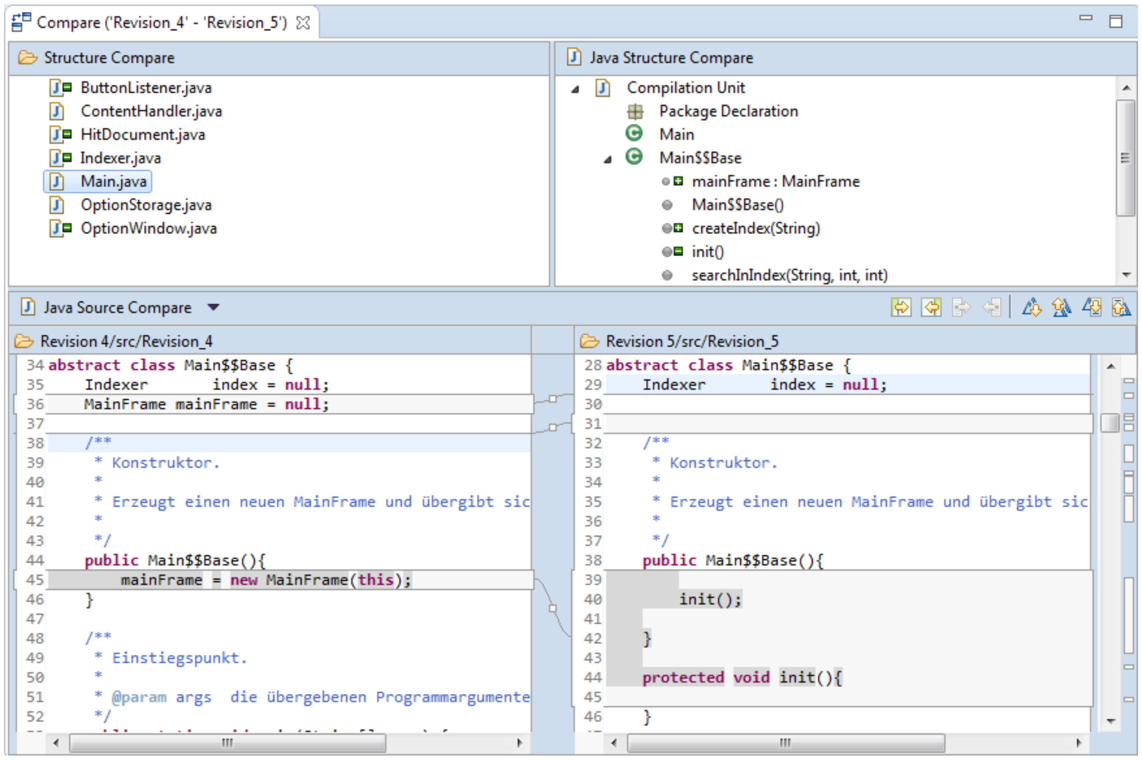


Figure 5.3: Differences between Revision 4 and 5 of *DesktopSearcher*



ButtonListener.java
ContentHandler.java
HitDocument.java
Indexer.java
Main.java
OptionStorage.java
OptionWindow.java

Compilation Unit
Package Declaration
Main
Main$$Base
mainFrame : MainFrame
Main$$Base()
createIndex(String)
init()
searchInIndex(String, int, int)

Revision 4/src/Revision_4

```
abstract class Main$$Base {
    Indexer        index = null;
    MainFrame mainFrame = null;

    /**
     * Konstruktor.
     *
     * Erzeugt einen neuen MainFrame und übergibt sic
     *
     */
    public Main$$Base(){
        mainFrame = new MainFrame(this);
    }

    /**
     * Einstiegspunkt.
     *
     * @param args  die übergebenen Programmargumente
     */
```

Revision 5/src/Revision_5

```
abstract class Main$$Base {
    Indexer        index = null;


    /**
     * Konstruktor.
     *
     * Erzeugt einen neuen MainFrame und übergibt sic
     *
     */
    public Main$$Base(){

        init();

    }

    protected void init(){

    }
```

Figure 5.4: Differences of *Variant_3* between Revision 4 and 5

determine the variant which we manually adapt to perform changes that occurred during development. We choose the variant which implements the most changed features according to Figure 5.3. As we see in Table 5.1, *Variant_2*, *Variant_3* and *Variant_4* implement all changed features. We randomly decide to choose *Variant_3*. As the next step, we compute changes that we need to manually implement. Therefore, we use the *Eclipse compare editor* to visualize the differences between *Variant_3* of Revision 4 and *Variant_3* of Revision 5, as we show in Figure 5.4. In this figure, the classes *ContentHandler*, *Main*, and *OptionStorage* are changed, whereas the classes *ButtonListener*, *HitDocument*, *Indexer*, and *OptionWindow* are added between Revision 4 and Revision 5. The *compare editor* lists lines that are added, removed or changed. Finally, we implement each change between Revision 4 and 5 until this variant has the same code than a variant that is generated from Revision 5 of *DesktopSearcher*. While implementing a change, we activate the feature-expression context where the change belongs to. In Table 5.2, we show change operations that we manually performed to simulate the development between revisions.

As the third step, we propagate changes of *Variant_3* into valid synchronization targets using source-focused synchronization in the batch mode. In case that a merge conflict occurs, we manually resolve the conflict with the *Eclipse compare editor*, as we described in Section 4.5.5 on page 66.

As the fourth step, we compare the result of the synchronization. We expect that synchronized variants are equal to variants that are generated in Revision 5. So, we

Revision/Change	Add Files or Lines	Remove Files or Lines	Change Lines
Rev. 4-5	4	1	3
Rev. 5-6	4	2	2
Rev. 6-7	1	1	0
Rev. 7-8	5	0	3
Rev. 8-9	0	2	0
Rev. 57-58	5	1	2
Rev. 58-61	1	0	3
Rev. 61-62	1	0	2
Rev. 62-63	3	0	3
Rev. 63-64	2	0	1

Table 5.2: Changes to Simulate Variant Development between Revision 4 to 9 and 57 to 64

checkout Revision 5 from the version control system and generate our five variants. Now, we compare synchronized variants with generated variants, again using the *Eclipse compare editor*. In case that differences occur, we repeat the steps two to four and adapt another variant in step two.

As the last step, we collect information about synchronization as well as tagging to answer our research questions. For this purpose, we count changes that we manually perform to adapt one variant. Moreover, we count how many changes *VariantSync* automatically synchronizes to compute the degree of automated merge (RQ 1). Besides, we count the number of changes of each context that are synchronized using the batch mode (RQ 2). Furthermore, we count the number of synchronization targets for a context (RQ 3). Finally, we identify the amount of code that was tagged during the simulated development and the amount of code that was automatically tagged during the synchronization (RQ 4). For this purpose, we access the *XML files* which contain context information, as we introduced in Section 4.4.2 on page 53. Each *XML file* represents one feature-expression context. The context consists of variants. Each variant contains tagged lines, as we visualized in Figure 5.5. To retrieve the number of tagged lines that belong to a variant, we implemented a small *Java application* that iterates through each *XML file* of *VariantSync* and collects the number of lines that are tagged to the same variant through different contexts. As the result, we get the number of tagged lines per variant.

If all synchronized variants contain the same code than generated variants, we successfully simulated the development of five variants between Revision 4 and Revision 5. To collect adequate information answering our research questions, we repeat the steps two to four to additionally simulate variant development between Revisions 6, 7, 8, and 9,

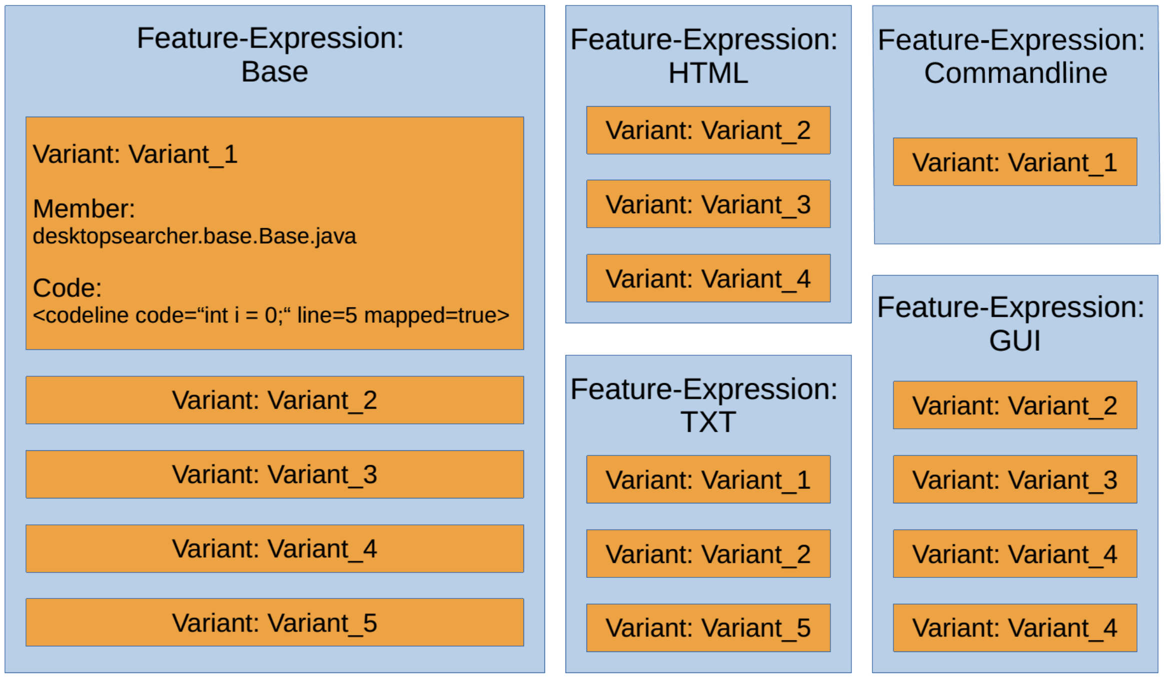


Figure 5.5: Accessing Tagged Lines inside Feature-Expression Contexts

as we show in Figure 5.1. Furthermore, we repeat the steps two to four to simulate variant development between Revision 57, 61, 62, 63, and 64.

5.4 Interpreting Results of Variant Development

In Section 5.3, we described our proceeding to collect information to evaluate variant development with *VariantSync*. In this section, we present our results to answer our research questions. On the one hand, we compare and interpret results using *VariantSync* from the beginning of variant development between Revision 4 to 9. On the other hand, we evaluate results introducing *VariantSync* during the development process between Revision 57 to 64.

How many changes are automatically merged into valid target variants?

While developing variants between Revision 4 to 9 and 57 to 64, 74% of all changes are automatically merged with *VariantSync*. We simulate the development between each revision and only count changes that are propagated during the synchronization. We do not count initial changes that we manually performed on variants.
The degree of merge automation depends on the amount of merge conflicts. The batch mode of *VariantSync* is likely to decrease the number of merge conflicts. On the one side, the batch mode avoids ordering merge conflicts by propagating changes ordered by their timestamps. On the other side, *VariantSync* merges fine-granular changes. For example, a *Java class* was changed at two different locations. First, the body of the

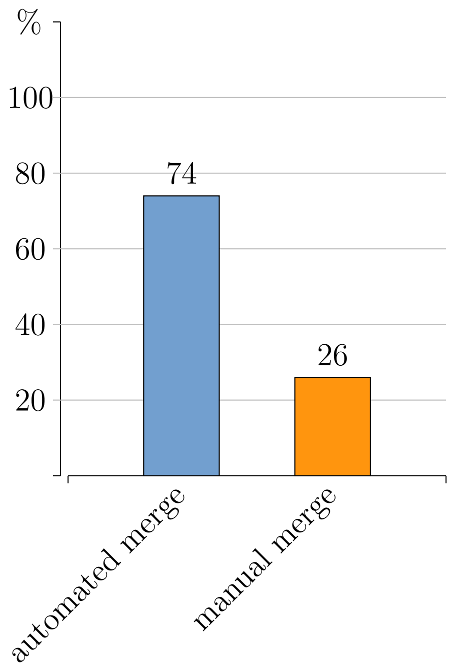


Figure 5.6: Automated vs. Manual Merge

constructor was changed which would cause a merge conflict during the synchronization with variant *A*. Second, *getter* and *setter* methods are added at the end of the file. Merging these methods into variant *A* does not cause a merge conflict. If both changes of the class would be merged as one coarse-granular step, then the merge needs to be manually resolved because the change of the *constructor* causes a merge conflict. Thus, the degree of automated merge would be 0%. Instead, *VariantSync* merges both changes independently from each other. So, the first step still needs to be manually resolved. However, the second change is automatically merged. Hence, the degree of automated merge is 50%.

How many changes are summarized inside a context and synchronized in the batch mode?

A context collects all changes across variants that are tagged to the feature-expression of the context. The advantage of the batch mode is the automatic change propagation into valid synchronization targets. Without the batch mode, the developer needs to manually propagate each change into target variants. Table 5.3 shows the number of changes that are aggregated in a feature-expression context and that are automatically merged in the batch mode. For example, the batch mode automatically synchronizes five changes of feature *GUI* between Revision 4 to 5. In average, three changes are summarized in a context for each revision. So, the batch mode decreases the synchronization effort for the developer.

Revision/Feature	Base	Index	User_Interface	GUI	View	HTML	TXT	OS	#Contexts
Rev. 4-5	2	-	-	5	-	1	-	-	3
Rev. 5-6	6	-	-	-	-	-	1	-	2
Rev. 6-7	-	-	-	2	-	-	-	-	2
Rev. 7-8	2	3	-	-	-	3	-	-	3
Rev. 8-9	-	-	-	2	-	-	-	-	1
Rev. 57-58	1	1	1	1	-	-	-	1	5
Rev. 58-61	3	-	-	3	1	-	-	-	3
Rev. 61-62	3	-	-	3	1	-	-	-	2
Rev. 62-63	2	2	-	2	-	-	-	-	3
Rev. 63-64	-	2	-	0	-	-	0	-	2

Table 5.3: Change Summary in the Batch Mode

How many variants are valid synchronization targets for the change propagation of one context?

Revision/Feature	Base	Index	User_Interface	GUI	View	HTML	TXT	OS
Rev. 4-5	4	-	-	3	-	2	-	-
Rev. 5-6	4	-	-	-	-	-	2	-
Rev. 6-7	-	-	-	3	-	-	-	-
Rev. 7-8	4	4	-	-	-	2	-	-
Rev. 8-9	-	-	-	3	-	-	-	-
Rev. 57-58	4	4	4	3	-	-	-	4
Rev. 58-61	-	-	-	3	3	-	-	-
Rev. 61-62	-	-	-	3	3	-	-	-
Rev. 62-63	4	4	-	3	-	-	-	-
Rev. 63-64	-	4	-	-	-	-	-	-

Table 5.4: Synchronization Targets for Change Propagation

VariantSync computes valid synchronization targets. In Table 5.4, we summarize the number of synchronization targets for the change propagation. A variant is a valid synchronization target if it implement the same feature expression that the change is tagged to. Furthermore, if a file was changed, a valid synchronization target needs to contain a file with the same name than the changed file. For example, *Variant_2* to *Variant_5* implement feature *GUI*. Propagating a change tagged to feature *GUI* requires a synchronization with four valid synchronization targets. Table 5.4 shows that *VariantSync* continuously computes correct synchronization target for changes in different revisions. Thus, *VariantSync* reduces the effort for developers to manually select synchronization targets.

How much code is tagged to features during the synchronization?

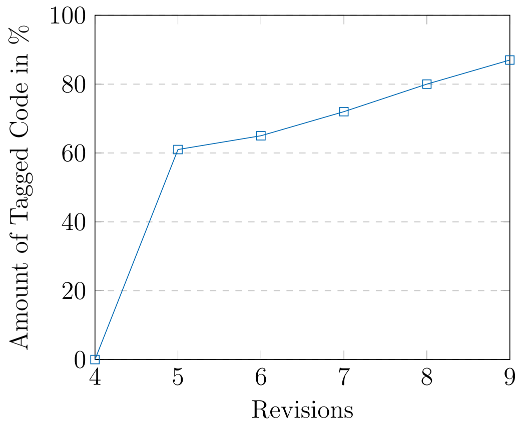


Figure 5.7: Tagged Code in Total (Rev. 4-9)

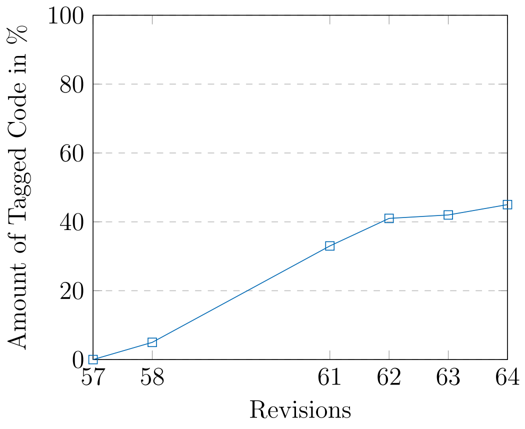


Figure 5.8: Tagged Code in Total (Rev. 57-64)

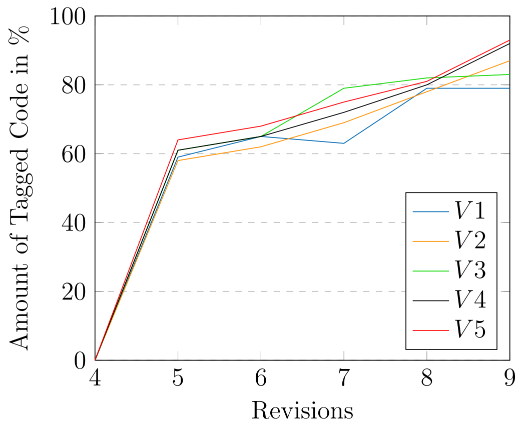


Figure 5.9: Tagged Code per Variant (Rev. 4-9)

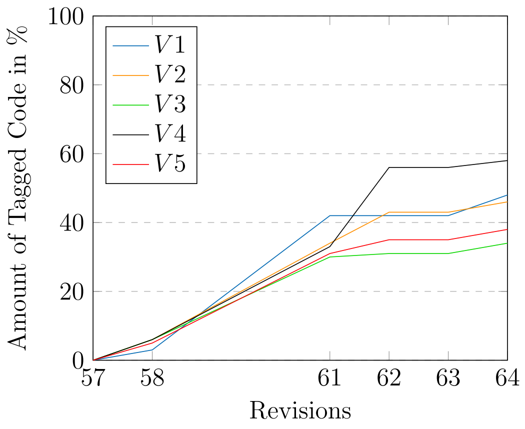


Figure 5.10: Tagged Code per Variant (Rev. 57-64)

To evaluate the degree of tagged code during variant development, we first compare the degree of tagged code when using *VariantSync* at the beginning of the variant development process and when introducing *VariantSync* while variants have already been developed over a period of time. Second, we compare how much code is tagged while developing with an active feature-expression context and how much code is automatically tagged during the synchronization.

In total, the degree of tagged code increases during variant development. Using *VariantSync* from the beginning of the variant development process, the degree of tagged code increases from 0% at Revision 4 to 89% at Revision 9, as we visualized in Figure 5.7. Between the first two revisions (4 to 5), the degree enormously grows. The reason is that typically many files are added at the begin of the development process and *VariantSync* directly tags the code of added files. In the following, the degree gradually grows from 61% to 89% between Revision 5 to 9. In contrast, the degree of tagged code only increases from 0% to 44% between Revision 57 and 64, as we see in Figure 5.7. Similar to Revision 5 to 9, the degree of tagged code also gradually grows, except between Revision 58 to 61. In this period, three files are added to *Variant_5* and propagated into other variants.
Moreover, the degree of tagged code is similar between the variants because most changes occurred on feature expressions *Base* and *GUI*. *Base* is implemented by all variants, *GUI* is implemented by variants two to five. Between Revision 4 to 6, the degree of tagged code only differs slightly, as we show in Figure 5.9. The reason is that 13 of 15 changes occur on the feature expressions *Base* and *GUI*. Therefore, changed code of *Base* is synchronized with all variants and changed code of *GUI* is synchronized with all variants except *Variant_1*. In the following development step, the amount of tagged code differs between *Variant_1* and *Variant_3* because only two changes on feature expression *GUI* occurred. *Variant_1* is not a valid synchronization target because it does not implement feature expression *GUI*. The degree of tagged code minimally decreases because two changes in the classes *OptionStorage* and *OptionWindow* needed to be manually implemented. Both changes replace existing code with less lines of code. So, the amount of tagged code decreases. However, *Variant_3* was manually adapted to simulate variant development between Revision 6 to 7. One of the changes causes a merge conflict during the change propagation into variants two, four and five. During manual conflict resolution, tagged code was replaced with changed code which only leads to a small increase of tagged code. Furthermore, the amount of tagged code between Revision 8 to 9 stagnates for *Variant_1* because only changes occurred that are tagged to the feature *GUI*. Due to the fact that *Variant_1* does not implement feature *GUI*, it is not a valid synchronization target and does not receive changed code. Between Revision 57 to 64, the development of the degree of tagged code per variant is similar to Revision 4 to 9, as we see in Figure 5.10. The variants have a similar but not identical degree of tagged code because they implement a different selection of features. The difference is the lower degree of tagged code in comparison with Revision 4 to 9.
Finally, Figure 5.11 shows the relation of automatically to manually tagged code. In average, three-quarter of tagged code was automatically tagged in our five variants. The reason is that we manually adapt one of five variants to simulate variant development. The following change propagations enable an automatic tagging. Most changes belong to features that are implemented by almost all variants. The features *Base, Index, User_Interface*, and *OS* are implemented by all variants and the features *GUI* and *View* are implemented by four of five variants. Thus, a change that occurred in feature *GUI* will be propagated into three valid target variants. After merging changed code into a target variant, this code will be tagged. In case of feature *GUI*, the degree

of automated to manual tagging is exactly 75%. Furthermore, the degree describes 80% automation when propagating changes of features *Base, Indexer, User_Interface*, or *OS*. However, rarely implemented features like *TXT* or *HTML* will be synchronized in less variants. So, the degree of automated tagging decreases. In case of feature *TXT*, the degree of automated tagging would be nearly 50% because feature *TXT* is implemented in two variants. In one variant, code is changed and tagged by manual tagging. Then, the change will be merged into the target variant. However, on an average, the degree of automated tagging is nearly 75% for our five variants and their feature configurations. So, the more variants exist, the higher is the potential for automation.
To summarize, the degree of tagged code grows gradually in most development steps. In some cases, the degree stagnates or minimally decreases between two development steps if tagged code is replaced with changed code or tagged code is removed during the synchronization. Using *VariantSync* at the beginning of the development process leads to a high degree of tagged code, whereas introducing *VariantSync* while variants have already been developed over a period of time leads to a lower amount of tagged code. Hence, tagging code during the synchronization strongly increases the degree of tagged code.

5.5 Discussion

We showed that *VariantSync* can reduce the synchronization effort for variant development. Furthermore, we illustrated that *VariantSync* accumulates domain knowledge to support the migration to a product line. In the following, we discuss the results of the evaluation. We start by reflecting the efficiency of variant development. Moreover, we discuss the amount of code tagging when using *VariantSync* from the beginning of the development process and when *VariantSync* is introduced while variants have already been developed over a period of time.

Efficiency of Synchronization

We define the efficiency of variant synchronization by evaluating our research questions one to three. Regarding research question one, *VariantSync* merges nearly 75% of all changes automatically, as we show in Figure 5.6. Referring to research question two, three changes are summarized in a context and synchronized in the batch mode in average while developing variants between two revisions. Concerning research question three, *VariantSync* correctly computes valid synchronization targets to propagate changes of a context. In regard of these facts, *VariantSync* makes variant development more efficient than a manual performed variant synchronization with tools from single-system engineering, for example using a version control system with the branch-per-product strategy (Section 2.2.3 on page 20).
Nevertheless, we can increase the degree of the automated merge by adopting or implementing another merge technique. Currently, *VariantSync* performs a lexical merge. The advantage of this merge technique is its widespread area of application. This merge technique can be applied on each text file. However, a lexical merge only compares single text lines without information about their syntax and semantic. Using syntactic and

semantic merge techniques, the degree of the automated merge would increase because typical lexical merge conflicts would be avoided, for example changed line positions or a changed code formatting. As a drawback, syntactic and semantic merge techniques cannot be applied on each kind of text files. For each type (e.g., *Java* files or *C++* files), a separated parser needs to be implemented.

Amount of Tagging

We classify code tagging into two categories. First, manual tagging describes code that is tagged because a developer (manually) changed code while working with an active feature-expression context. Second, automatic tagging comprises code that is tagged because changed code fragments are merged into synchronization targets. As we show in Figure 5.11, approximately 25% of the tagged code was manually and 75% was automatically tagged. So, we can conclude that the amount of manually tagged code rises during the synchronization. Automated tagging increases the degree of tagged code dependent on the number of valid synchronization targets. The more variants are synchronized with tagged code, the more code is absolutely tagged.
Manual tagging increases the amount of tagged code dependent on the kind of changes. If a developer adds a file, then all lines of this file are tagged. Adding additional lines inside a file also increases the amount of tagged code. However, the amount of tagged code decreases when removing tagged lines inside a file or removing files that contain tagged code.
In conclusion, the more files or lines inside files are added, the more code is manually tagged. During variant development, most files are initially created or added at the beginning of the development process. Hence, if *VariantSync* is used from the beginning, then the amount of manually tagged code significantly increases. Propagating tagged changes to target variants increases the amount of tagged code. As a result, 89% of all variant code was tagged while simulating the development of five variants for the first five revisions, as we see in Figure 5.9. Indeed, when *VariantSync* is introduced while variants have already been developed over a period of time, then more development steps are necessary to reach the same degree of tagged code. As we show in Figure 5.10, the degree of tagged code during the development between Revision 57 to 64 is half as the degree from the beginning of development in Figure 5.9. So, the earlier *VariantSync* is introduced in the development process, the more the amount of tagging rises.

5.6 Summary

In this chapter, we evaluated variant development with *VariantSync* on five similar variants. At first, we pointed out research questions that we wanted to answer with this evaluation. Then, we described the *DesktopSearcher* product line as the target system on which we performed the evaluation. Moreover, we introduced a concept to simulate variant development based on the *DesktopSearcher* product line. We illustrate the development history of *DesktopSearcher* and manually implemented changes on one selected variant between different revisions. Then, we propagated these changes to valid synchronization targets. Our measurements show that nearly 75% of all changes can be automatically synchronized. Furthermore, the code to feature tagging increases. Introducing *VariantSync* at the beginning of the development process leads to a degree of 89% tagged code. Finally, we discussed our measurements in regard to the research questions.

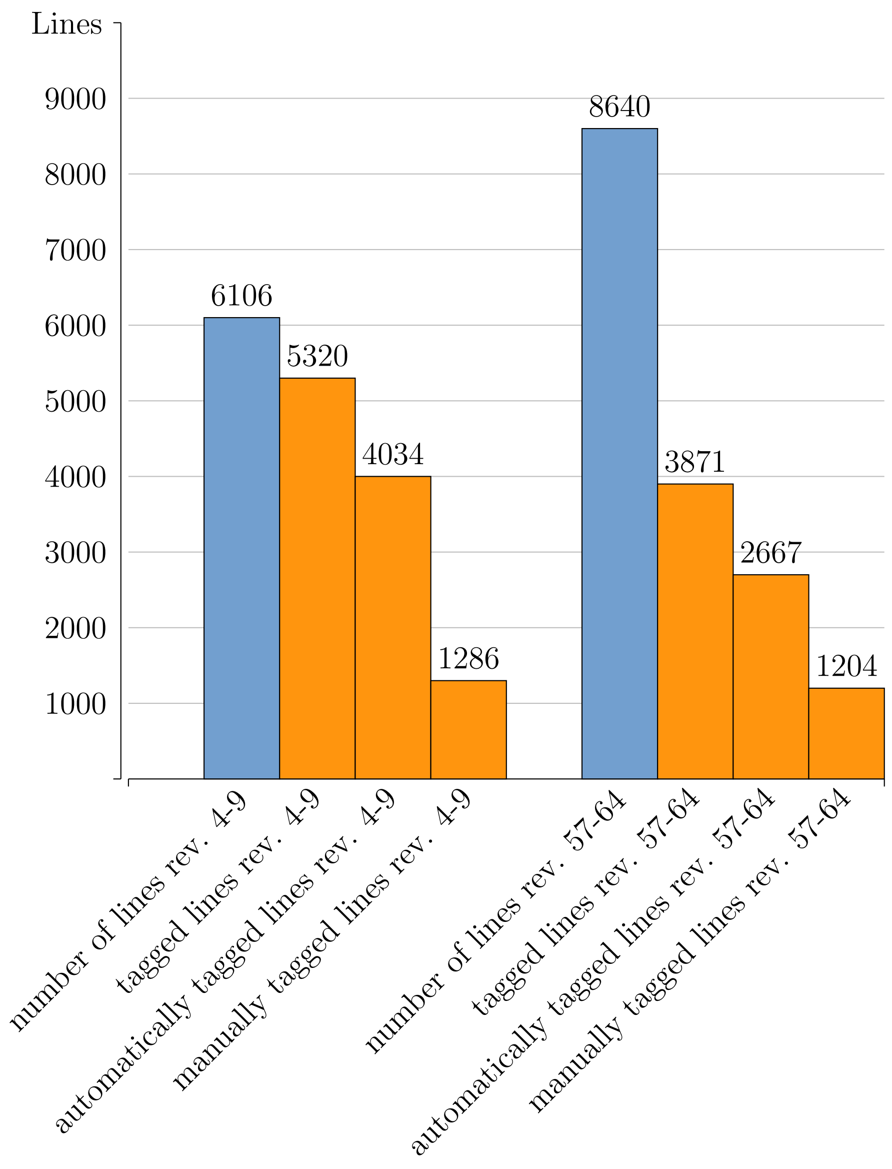


Figure 5.11: Automated Tagging vs. Manual Tagging

6. Related Work

Research on variant development is an active field that addresses works covering variant development with clone-and-own as well as variant development with a product line. In this chapter, we present several publications that are related to *VariantSync*. We first refer to a work that is the base of our *VariantSync* concept. Then, we present related work that focuses on clone-and-own development without domain knowledge in terms of features. In the following, we examine approaches that introduce the concept of feature modeling to automate parts of the variant development process. We further present a work that describes an approach to minimize the risks of product-line transitions. Finally, we refer to a publication that supports variant development over multiple product lines.

Our *VariantSync* concept is built upon a prior version of *VariantSync*. In 2012, Luo [Luo12] described a first version of *VariantSync* in his master's thesis. This approach logs changes inside variants on a fine granularity during variant development. After saving a file during development process, the approach computes the differences between the two latest version of the file with a lexical difference algorithm, where each difference is treated as a separate change. Then, the approach merges the changes with a lexical three-way merge into target variants. The computation of possible synchronization targets is semi-automated. A developer needs to create a feature model for all variants and derive a feature configuration to describe the variability of each variant. To propagate a change, the developer manually selects the feature that belongs to the change. If the feature configuration contains the selected feature, then the corresponding variant is a valid synchronization target. In the following, the developer can select these synchronization targets and manually start the synchronization. In case of a merge conflict, the conflict needs to be manually resolved. The approach is prototypically implemented as an *Eclipse* plug-in that provides a view for change propagation. In our thesis, we extend this first concept of *VariantSync*. We introduce tagging with feature-expression contexts, present a variant synchronization from two different point of views, automate the computation of valid synchronization targets and provide a batch mode to automate change propagation from a source-focused and target-focused point of view. We also implement *VariantSync* as an *Eclipse* plug-in that provides two views for source-focused and target-focused synchronization and implements the tagging with feature-expression contexts strategy.

With *VariantSync*, we presented a concept that makes domain knowledge explicit in terms of features to automate the synchronization of variants. In contrast, Schmorleiz [Sch15] presents an annotation-centric approach to similarity management that synchronizes code fragments between variants without using domain knowledge in terms of features. First, this approach analyzes the history of a software repository by traversing commits and detecting similarity evolutions, e.g., the approach detects whether code fragments of variants are always equal, converge to equal, diverge from equal, or are non-equal. Then, the approach annotates code fragments to express how they need to be further maintained. Goal is to perform change propagation to avoid unintentional divergence between variants. For this purpose, the annotation-centric approach synchronizes code fragments that have diverged over time but should be equal. Dependent on the kind of annotation, the approach automatically propagates annotated changes between code fragments or requires manual work to perform change propagation. In comparison to the annotation-centric approach, *VariantSync* also focuses on change propagation between variants, but on a different abstraction level. We map changes to feature expressions to automatically compute valid synchronization targets. Moreover, the annotation-centric approach uses version control systems as a *synchronization platform* and supports variant development with clone-and-own by synchronizing code changes between code clones on a low abstraction level. In contrast, *VariantSync* uses an integrated development environment as a *synchronization platform* and enables variant synchronization from different point of views. Furthermore, *VariantSync* establishes a code to feature mapping that reduces the gap to variant development with product lines. However, it is also possible to apply *VariantSync* with version control systems.

In Section 2.2.3 on page 20, we describe a possibility to develop variants in branches of version control systems. Moreover, we introduce version control systems as a possible *synchronization platform* for *VariantSync* in Section 4.1.1 on page 42. To support variant development in branches of version control systems, Rubin et al. [RKBC12] present a framework to facilitate the reuse of features between variants. The framework treats features as reuse units and detects inconsistencies in feature implementations. Furthermore, the framework detects relationships between features and supports developers to copy or delete features in branches. *VariantSync* also has the goal to avoid inconsistencies between variants, but it does not support the reuse of feature implementations. Instead, *VariantSync* synchronizes code changes between feature-expression contexts of different variants to prevent duplicate developments.

Similar to the goals of *VariantSync*, Fischer et al. [FLLHE15] introduce *ECCO* (Extraction and Composition for Clone-and-Own), an approach to enhance clone-and-own with systematic reuse to support variant development. *ECCO's* goal is to reduce the effort for variant development and maintenance, especially for adding new variants during development. For this purpose, *ECCO* automates the clone-and-own approach with extraction and composition. The extraction step traces associations between features and artifacts. Based on a set of selected features, the composition step composes variants from artifacts in selected traces. For example, if a developer selects a set of features, then *ECCO* will detect appropriate reusable software artifacts and will indicate which artifacts need to be manually adapted. *VariantSync* also has the goal to support the development of variants which could increase over time. In contrast to reusable artifacts of *ECCO*, *VariantSync* focuses on change synchronization between variants that can contain duplicated code fragments. *VariantSync* does not require an extraction process to trace features and artifacts. Instead, *VariantSync* is a light-weight solution that can be introduced at arbitrary points in the variant development process and only slightly changes the clone-and-own development process.

To reduce the risks of a product-line transition, Antkiewicz et al. [AJB+14] present the idea of a virtual platform that covers different clone-and-own strategies, from ad-hoc clone-and-own to product-line engineering with a fully-integrated platform. They introduce several governance level which guide the transition to a product line and provide incremental benefits at each level. Starting with ad-hoc clone-and-own on the first level, the virtual platform incrementally introduces domain knowledge and establishes a code to feature tracing in clone-and-own development. For example, the first governance level records provenance information about variants and cloned assets to coordinate the change propagation. The second level starts to describe functionality of variants in terms of features, whereas the third level introduces feature configurations to derive variants by selecting subsets of features. Furthermore, the fourth level uses a feature model to cover all variants and features. To create a new project, developers use an existing variant and fetch needed features from other variants when the feature model and its constraints allow the feature combination. The fifth and sixth level establish a configurable platform which enables developers to share assets among the platform and automatically derive new variants by configuring the platform. *VariantSync* can be classified into governance level four because it focuses on a variant synchronization which is oriented towards the clone-and-own strategy with a feature model. To compute valid synchronization targets, the *VariantSync* concept requires a list of concerns of each variant. For this purpose, the *VariantSync* prototype uses a feature model to cover all variants and implemented features. It further derives feature configurations that represent the functionality of each variant.

To support variant development over multiple product lines, Lettner et al. [LG15] present a feature feed approach that has the aim to improve developer awareness in software ecosystem evolution. A software ecosystem includes the development of multiple product lines in multiple organizations, where feature feeds propagate knowledge of developers about feature implementations. Similar to our implementation of *VariantSync*, the approach is based on feature modeling. The approach automatically tracks changes on features and publishes these information to subscribed developers. In this context, a developer can occupy different roles. For example, a domain engineer is responsible for changes on the feature model and an application engineer is responsible for changes in the feature implementation. During development, each role is informed about changes at different levels of granularity. Using *VariantSync*, developers also could occupy different roles. On the one side, a developer can be responsible to implement a variant. On the other side, a developer can be responsible to propagate changes and synchronize variants. For example, if a developer is an expert on one variant, then he can use the target-focused synchronization to update his variant with changes that occurred in other variants.

7. Conclusion

To overcome the increasing demand for tailored software systems, industrial software development often uses clone-and-own to build a new variant by copying and adapting an existing variant. Admittedly, this proceeding requires less up-front investments but the maintenance effort rapidly grows with an increasing number of variants. However, product lines have high up-front investments which make the development of few variants unprofitable. To reduce this gap, we presented *VariantSync*, a light-weight and language-agnostic concept to enhance variant development with clone-and-own and support the migration to a product line.

On the one hand, *VariantSync* makes variant development more efficient by automating the synchronization of variants. For this purpose, *VariantSync* detects and collects changes that occur during variant development and synchronizes these changes with appropriate variants. Our first contribution is to automate the identification of synchronization targets. For this purpose, we make domain knowledge explicit by mapping code to features. We introduce a feature-expression context that automates the code to feature tagging during variant development. If a software developer activates a feature-expression context, then *VariantSync* monitors all changes on variants and tags changed code fragments to a feature expression. Our second contribution is to perform a variant synchronization from different points of view. We decouple implementation from synchronization and provide a synchronization from a target-focused as well as source-focused point of view. The target-focused synchronization focuses on synchronizing changes with a single variant. One variant is synchronized with all relevant changes that occurred in other variants. In contrast, the source-focused synchronization focuses on actively propagating changes of one variant into other variants. A batch mode automates the synchronization from a target-focused or source-focused point of view.

On the other hand, *VariantSync* supports the light-weight transition to a product line. For this purpose, our third contribution is to accumulate domain knowledge. Using feature-expression contexts, *VariantSync* establishes a code to feature mapping. While synchronizing variants, we automatically increase the degree of tagged code. If a change

is propagated to valid synchronization targets, then the synchronized code will be tagged to the same feature expression that the change is tagged to.
Moreover, we showed that *VariantSync* is applicable. For this purpose, we discussed version control systems and integrated development environments as two possible *synchronization platforms*. Due to the fact that integrated development environments support the visualization of code tagging with less effort and to be in line with a prior prototype of *VariantSync* [Luo12], we implemented *VariantSync* as an *Eclipse* plug-in. To represent domain knowledge, we adapted functions of *FeatureIDE* to create and maintain feature models and feature configurations. The *VariantSync* plug-in realizes the application of feature-expression contexts and provides two *Eclipse* views to perform target-focused synchronization and source-focused synchronization. To ensure that the synchronization is language-agnostic, *VariantSync* uses a lexical merge technique to merge code fragments into variants.
Finally, we evaluated our implementation of *VariantSync* with a simulated case study by covering the development of five variants over a distinct number of development steps. To achieve this, we generated variants with the *DesktopSearcher* product line and used the development history of *DesktopSearcher* to simulate variant development with *VariantSync*. We found out that *VariantSync* automatically synchronizes 75% of all changes between variants and continuously increases the degree of code to feature tagging. While simulating variant development between the first five revisions of *DesktopSearcher*, *VariantSync* nearly tagged 89% of all variant code. Even if *VariantSync* is not used from the beginning and is only introduced during the development process, the amount of tagged code grows from 0% in Revision 57 to 44% in Revision 64.

8. Future Work

In this chapter, we describe several topics that can extend our contributions and evaluation. On the one hand, we refer to variant development and discuss how *VariantSync* could support developers in adding new variants. On the other hand, we propose suggestions to extend the code to feature tagging. Finally, we suggest how to extend our evaluation of *VariantSync*.

Adding New Variants

During variant development, it is a realistic scenario that a new variant needs to be initially created and maintained during the further development process. Currently, *VariantSync* supports the maintenance of new variants by synchronizing changes with these variants. *VariantSync* actually does not support the creation of new variants. Using clone-and-own, a new variant is typically built by first copying an existing variant and then adapting it for specific needs. If variants are developed with *VariantSync*, then variant code is tagged to feature expressions. Using this domain knowledge, *VariantSync* could support variant creation. First, a developer needs to specify a feature configuration for the new variant. Second, *VariantSync* could determine the variant with the nearest feature configuration and copies this variant inclusive the existing code to feature tagging. Third, *VariantSync* could either remove code that is tagged to feature expressions which are not implemented by the new variant or notify the developer that this code needs to be removed. Fourth, *VariantSync* could propose code of other variants that is tagged to the same feature expressions than the new variant to support the development of the new variant.

Combining Manual Tagging and Tagging with Feature-Expression Contexts

In Section 3.2.2 on page 30, we discussed manual tagging and tagging with feature-expression contexts to trace code fragments to features. We made the conclusion that the combination of both tagging strategies combines their advantages and minimizes their disadvantages. With manual tagging, the developer is able to decide which code he wants to tag. Nevertheless, the developer needs to manually tag each code fragment that belongs to a feature expression. To overcome this tagging effort, tagging with feature-expression contexts automates the tagging process. However, once performed, tagging can be hardly changed. Our prototypical implementation of *VariantSync* realizes the tagging with feature-expression contexts strategy. As a future work, we could show that the combination of both tagging strategies is applicable (or not). For this purpose, we could extend our *VariantSync* prototype to additionally provide a manual code to feature tagging. For example, we could extend *Eclipse's* code editor to tag selected code to a chosen feature expression.

Synchronizing Code to Feature Tagging

To increase the code to feature tagging, *VariantSync* actually tags code that was inserted into a variant during synchronization. As a consequence, the original code of this variant is replaced with the inserted code. Simultaneously, an existing tagging is also overwritten by the tagging of the inserted code. However, overwriting the tagging could lead to an erroneous tagging. For example, code fragment A is tagged to feature expression $F1$ and code fragment B is tagged to feature expression $F2$. If parts of code fragment A are overwritten with code fragment B during synchronization, then the tagging is also overwritten and $F1$ is replaced by $F2$, even if parts of code fragment A still need to be tagged to feature expression $F1$. As a future work, *VariantSync* could avoid erroneous tagging by synchronizing tagging information instead of overwriting them.

Extending the Migration to a Product Line

VariantSync supports the transition to a product line by establishing a code to feature mapping. In the next step, we could extend *VariantSync* to collaborate with existing tools for product-line migration. For example, we could evaluate whether *Variant-Sync* can be migrated to an implementation of delta-oriented programming, e.g., with *DeltaJ* [KHS+14], a prototypical implementation of delta-oriented programming for *Java*. Delta-oriented programming is an approach to implement a product line consisting of a core module and a set of delta modules. A core module represents a variant and delta modules define changes that are applied to the core module to implement further variants, e.g., by adding, changing or removing code or files [SBB+10]. *VariantSync* monitors changes on variants which could support the setup of delta modules.

Evaluating ***VariantSync*** with an Industrial Case Study

In this thesis, we evaluated *VariantSync* with a case study that simulated the development of variants which were generated by a product line. To evaluate *VariantSync's* capability, we could perform a long-term evaluation where *VariantSync* is used to support the variant development of an industrial software system. In this case study, we could evaluate how different developers collaborate when using *VariantSync*, especially while propagating changes into target variants with the source-focused synchronization or while synchronizing single variants with the target-focused synchronization. Furthermore, we could evaluate our research questions (Section 5.1 on page 69) in a long term application. At least, we could measure how regularly developers synchronize changes and prove whether a regularly change propagation decreases the amount of merge conflicts.

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, 2013. (cited on Page ix, 1, 2, 3, 8, 9, 10, 12, 13, 14, 16, 17, 19, 20, 21, 26, and 30)

[AJB⁺14] Michal Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stanciulescu, Andrzej Wasowski, and Ina Schaefer. Flexible product line engineering with a virtual platform. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 532–535. ACM, 2014. (cited on Page 19 and 89)

[ALB⁺07] Sven Apel, Christian Lengauer, Don Batory, Bernhard Möller, and Christian Kästner. An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706, University of Passau, 2007. (cited on Page 31)

[ALB⁺11] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 190–200. ACM, 2011. (cited on Page 19)

[ALHM⁺11] Mauricio Alférez, Roberto E. Lopez-Herrejon, Ana Moreira, Vasco Amaral, and Alexander Egyed. Supporting Consistency Checking between Features and Software Product Line Use Scenarios. In *Top Productivity through Software Reuse. 12th International Conference on Software Reuse*, pages 20–35. Springer Berlin Heidelberg, 2011. (cited on Page 13)

[AM14] RaFat Ahmad Al-MsieDeen. *Reverse Engineering Feature Models From Software Variants to Build Software Product Lines*. PhD thesis, University of Montpellier, France, 2014. (cited on Page 2 and 28)

[Bab15] Arne Babenhauserheide. Mercurial scm. https://www.mercurial-scm.org/, 2015. [Online; accessed 30-November-2015]. (cited on Page 42)

[Bat05a] Don Batory. A tutorial on feature oriented programming and the AHEAD tool suite. In *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, pages 3–35. Springer-Verlag Berlin, 2005. (cited on Page 71)

[Bat05b] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines*, pages 7–20. Springer-Verlag Berlin, 2005. (cited on Page 12 and 13)

[BCD$^+$00] Len Bass, Paul C. Clements, Patrick Donohoe, John McGregor, and Linda M. Northrop. Fourth Product Line Practice Workshop Report. Technical Report CMU/SEI-2000-TR-002, Software Engineering Institute, 2000. (cited on Page 27)

[Ber90] Brian Berliner. CVS II: Parallelizing Software Development. In *Proceedings of the Winter 1990 USENIX Conference*, pages 341–352. Prisma, Inc., 1990. (cited on Page 20)

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl, and Klaus Schmid. *Software Produktlinien: Methoden, Einführung und Praxis.* dpunkt.verlag, 2004. (cited on Page 1 and 2)

[Bre04] Tom Bret. Parallel Development Strategies for Software Configuration Management. *Methods and Tools*, 12(2):2–11, 2004. (cited on Page 3 and 20)

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley, 2000. (cited on Page 9 and 16)

[Cha15] Scott Chacon. Git. https://git-scm.com/, 2015. [Online; accessed 30-November-2015]. (cited on Page 42)

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc, 2001. (cited on Page 8)

[CS14] Scott Chacon and Ben Straub. *Pro Git.* Apress, 2014. (cited on Page 42)

[CSFP08] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion.* O'Reilly Media, 2008. (cited on Page 42)

[Cyb96] Jacob L. Cybulski. Introduction to Software Reuse. Technical Report Technical Report TR 96/4, The University of Melbourne, 1996. (cited on Page 9)

[DB07] Mark Dalgarno and Danilo Beuche. Variant Management. In *3rd British Computer Society Configuration Management Specialist Group Conference*, 2007. (cited on Page 18)

[DRB+13] Yael Dubinski, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 25–34. IEEE Computer Society, March 2013. (cited on Page 3, 18, and 19)

[EEM10] Neil A. Ernst, Steve M. Easterbrook, and John Mylopoulos. Code forking in open-source software: a requirements perspective. *CoRR*, abs/1004.2889, 2010. (cited on Page 18)

[FKF+10] Janet Feigenspan, Christian Kästner, Mathias Frisch, Raimund Dachselt, and Sven Apel. Visual Support for Understanding Product Lines. In *Proceedings of the International Conference on Program Comprehension.* IEEE CS, 2010. (cited on Page 56)

[FLLHE15] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proceedings of the 37th International Conference on Software Engineering*, pages 665–668. IEEE Press, 2015. (cited on Page 88)

[Fou15] The Apache Software Foundation. Apache subversion. https://subversion.apache.org/, 2015. [Online; accessed 30-November-2015]. (cited on Page 42)

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000. (cited on Page 3)

[Fre83] Peter Freeman. Reusable software engineering: Concepts and research directions. In *ITT Proceedings of the Workshop on Reusability in Programming*, pages 129–137, 1983. (cited on Page 9)

[JKB08] Mikolas Janota, Joseph Kiniry, and Goetz Botterweck. Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Technical Report Lero-TR-SPL-2008-02, The Irish Software Engineering Research Centre, 2008. (cited on Page 28)

[Käs10] Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* PhD thesis, University of Magdeburg, Germany, 2010. (cited on Page 1, 8, 15, 16, 31, 44, and 55)

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 8, 9, and 12)

[KD11] Rob Kitchin and Marting Dodge. *Code/Space: software and everyday life*. Massachussets Institute of Technology, 2011. (cited on Page 1)

[KG08] Cory J. Kapser and Michael W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Journal of Empirical Software Engineering*, 13(6):645–692, 2008. (cited on Page 3 and 19)

[KHS$^+$14] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. DeltaJ 1.5: delta-oriented programming for Java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 63–74. ACM, 2014. (cited on Page 94)

[Kla14] Benjamin Klatt. *Consolidation of Customized Product Copies into Software Product Lines*. PhD thesis, Karlsruher Institut für Technologie, Germany, 2014. (cited on Page ix and 18)

[Kru06] Charles W. Krueger. The emerging practice of software product line development. *Mil Tech Trends*, pages 1–3, 2006. (cited on Page 8 and 28)

[LG15] Daniela Lettner and Paul Grünbacher. Using Feature Feeds to Improve Developer Awareness in Software Ecosystem Evolution. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, page 11. ACM, 2015. (cited on Page 89)

[LLMZ06] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *Journal IEEE Transactions on Software Engineering*, 32(3):176–192, 2006. (cited on Page 3)

[Luo12] Lei Luo. Synchronize Software Variants with VariantSync. Master's thesis, University of Magdeburg, Germany, December 2012. (cited on Page 3, 4, 7, 25, 46, 63, 64, 87, and 92)

[LWN07] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, pages 18–22. IEEE Computer Society, 2007. (cited on Page 18)

[Men02] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002. (cited on Page 19, 21, 22, 23, and 36)

[MNJP02] John D. McGregor, Linda M. Northrop, Salah Jarrad, and Klaus Pohl. Initiating Software Product Lines. *IEEE Software*, 19(4):24–27, 2002. (cited on Page ix, 2, 25, and 28)

[Mye86] Eugene W. Myers. An O(ND) Difference Algorithm and its Variations. *Algorithmica*, 1(1):251–266, 1986. (cited on Page 48)

[New82] P. S. Newman. Towards An Integrated Development Environment. *IBM Systems Journal*, 21(1):81–107, 1982. (cited on Page 43 and 44)

[O'S09] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, 2009. (cited on Page 42)

[PBvdL10] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Publishing Company, 2010. (cited on Page 2, 4, 7, 27, 28, and 30)

[PD89] Ruben Prieto-Diaz. Classification of reusable modules. In *Software reusability: vol. 1, concepts and models*, pages 99–123. ACM, 1989. (cited on Page 9)

[PD90] Ruben Prieto-Diaz. Domain analysis: an introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990. (cited on Page 8)

[RKBC12] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference*, pages 156–160. ACM, 2012. (cited on Page 18, 19, and 88)

[SAA$^+$00] Guus T. Schreiber, Hans Akkermans, Anjo Anjewierden, Robert De Hoog, Nigel R. Shadbolt, Walter Van de Velde, and B. J. Wielinga. *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT Press, 2000. (cited on Page 9)

[SBB$^+$10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin Heidelberg, 2010. (cited on Page 94)

[Sch15] Thomas Schmorleiz. An Annotation-centric Approach to Similarity Management. Master's thesis, University of Koblenz and Landau, Germany, February 2015. (cited on Page 3 and 88)

[SLB$^+$10] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzystof Czarnecki. The Variability Model of The Linux Kernel. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMos)*, pages 45–51. University of Duisburg-Essen, January 2010. (cited on Page 1)

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 119–126. ACM Press, 2011. (cited on Page 30)

[Str12] Bjarne Stroustrup. Software Development for Infrastructure. *Computer*, 45(1):47–58, 2012. (cited on Page 1)

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research Articles. *Software - Practice Experience*, 35(8):705–754, 2005. (cited on Page 14)

[Thü15] Thomas Thüm. *Product-Line Specification and Verification with Feature-Oriented Contracts*. PhD thesis, University of Magdeburg, Germany, 2015. (cited on Page 12 and 13)

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79(5):70–85, 2014. (cited on Page 47, 52, and 53)

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–200. IEEE Computer Society, August 2011. (cited on Page 2 and 16)

[VBP12] Marco T. Valente, Virgilio Borges, and Leonardo Passos. A Semi-Automatic Approach for Extracting Software Product Lines. *IEEE Transactions On Software Engineering*, 38(4):737–754, 2012. (cited on Page 28)

[vdLSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer Publishing Company, 2007. (cited on Page 1, 2, 27, and 28)

[WL99] David M. Weiss and Chi T. R. Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc, 1999. (cited on Page 28)

[YGM06] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. Assessing merge potential of existing engine control systems into a product line. pages 61–67, New York, NY, USA, 2006. (cited on Page 26)

[ZJ94] Kaizhong Zhang and Tao Jiang. Some MAX SNP-hard Results concerning Unordered Labeled Trees. *Information Processing Letters*, 49(5):249–254, 1994. (cited on Page 23)

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

Magdeburg, den 21. Dezember 2015