Cost and Benefit of Embedded Feature Annotation: A Case Study

by

Wenbin Ji

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In software industry, organizations often need to develop a set of similar software-intensive systems in order to satisfy different requirements of customers. In the literature, it has been traditionally recommended that organizations adopt Product Line Engineering—an approach that uses a set of shared assets to derive the variants. However, in reality, organizations usually develop multiple variants using the clone-and-own approach, in which a new product is developed by cloning and modifying the assets of an existing product. Although the clone-and-own approach has several advantages, it can easily lead to inconsistencies and hardness to manage product portfolios.

In both the clone-and-own and the product line engineering context, the concept of feature can be used to characterize different variants. A feature is a function unit of a software product which provides a user-observable behavior and a unit of reuse. In the clone-and-own approach, there are two key challenges when doing cloning: reuse and consistency. For both of these activities, knowing the location of features is essential. In this thesis, we propose a lightweight approach for recording and maintaining feature models and mappings between features and software assets. We evaluated this approach in a case study, by applying it retroactively to an existing set of cloned projects in a way which simulated the actual development as if the approach had been used originally. Preliminary results showed that the extra cost of creating and maintaining a feature model and feature mapping information is negligible compared to the software development cost, and the benefit of it can justify the investment provided certain amount of reuse and consistency management is required.

Acknowledgements

First I would like to sincerely thank my supervisor Dr. Krzysztof Czarnecki for allowing me to join his amazing research group. His guidance, understanding, and patience allowed me to bring out the best in myself, improve my skills and rise up to challenges. I could not have done this without what he did for me.

I would also like to thank Dr. Thorsten Berger and Dr. Michal Antkiewicz for the countless discussions, continuous feedback, amazing advice and endless help throughout this study. Special thanks to Alexander Murashkin, who provided priceless information about the subject used in this study.

I would also like to thank Thomas Schmorleiz and Wei Wang for some earlier discussions about the initial ideas of this study.

All my love and countless thanks go to my beloved parents Yongqiu Ji and Mengping Dai, aunt Marilyn Dai, cousin Crystal Dai, grandpa Peiyu Dai and grandma Weixiang Yu. The world would not be meaningful without all of you in my life.

I would like to thank my best friends Jiewei Li, Chenxi Zhai, Cheng Wang, Xiaojun Xu, Chang Sun and Matthew Ma. I could not have done it without you.

Dedication

To my amazing parents whose unconditional love and support have always guided me to achieve my highest potential.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the software industry, organizations often need to develop a set of similar software-intensive systems in order to satisfy different requirements of customers within the same domain. For example, automotive companies often need to tailor their software systems of products according to conflicting requirements of different customers or different legal frameworks in different geographical regions. In the literature, it has been traditionally recommended that organizations adopt Product Line Engineering[33] for developing multiple variants. Product Line Engineering method provides a way of deriving multiple variants from a set of core assets in a systematic and predictable way.

However, in reality, organizations usually develop multiple variants using the *clone-and-own* approach[15], in which a new product is developed by cloning and modifying the assets of an existing product. Although the clone-and-own approach has several advantages, such as low adoption cost, it is not scalable since it can easily lead to inconsistencies and problem of control as the number of clones grows. In the literature, usage of cloning is considered a harmful approach[24], which is not recommended to be used in the long run. Yet the approach is still widely used in industry[15].

Different variants of products can be characterized by *features*. A *feature* is a functional unit of a software product which provides a user-observable behavior and a unit of reuse. A certain variant is characterized by the set of features it implements, and stakeholders compare different variants according to the different sets of features they implement. The concept of feature can be used in the traditional product line engineering, the clone-and-own approach and also developing a single product.

In the clone-and-own approach, there are two key challenges when doing cloning: reuse and consistency. In order to reuse a feature (e.g., cloning a feature from one variant to

another variant), a developer needs to know where the assets (code, documents etc.) of the feature are. To maintain consistency among variants, a developer can compare entire projects, assets, or individual features. In order to maintain consistencies among multiple copies of a feature, the location of the assets implementing the feature is also needed. So, for both of these challenges, knowing the location of features is required. Besides the two challenges, the feature location information is also essential for performing other kinds of tasks related to features, such as improving or re-factoring the assets of the feature. It also helps future transition into the traditional Product Line Engineering approach. Feature traceability information is useful throughout the development process. It can even be used to provide traceability between the code and the requirements.

In this thesis, we propose a lightweight approach for recording and maintaining a feature model[23] and feature traceability information (mappings to assets). The approach includes a lightweight annotation system that can be used to store the feature model and the feature traceability information within a source code repository, and guidelines for how to create and maintain the feature model and annotations during the development process. We evaluated this approach in a case study by applying it retroactively to an existing set of cloned projects in a way that simulated the actual development as if the approach had been used originally. After that we analyzed the cost and benefit of adopting the approach analytically and empirically. The set of cloned projects is a family of web-based tools for building and optimizing Clafer models. The results show that under the assumption that there is no feature location cost if developers add feature annotations while implementing a certain feature[1], the cost of creating and maintaining a feature model and feature location information with the proposed approach is negligible compared to the development cost. Also, under the assumption that recovering feature location information is much more costly[2] than recording the feature location information while the feature is being implemented, the benefit of the annotations can justify the investment if certain amount of reuse and consistency management is required. In fact, 18% of the feature recording and editing cost saved 90% of feature location cost needed for feature reuse tasks. The benefit of the feature annotations is even higher since they are also useful in the feature maintenance tasks, which constitute the majority of developers' work.

The rest of the thesis is organized as follows: Chapter 3 introduces the feature annotation approach we propose, in which feature annotations are created and maintained along the normal development process. Chapter 4 describes the design of our simulation case study. Chapter 5 describes the evolution patterns of the feature models, annotations

[1]This is because the developer has the feature location in mind while implementing a feature.

[2]This is because the original developers may forget the location of features, or the original developers may leave the organization.

and software assets. Chapter 6 describes analytical and empirical results of evaluating the feature annotation approach, which is based on analyzing the cost and benefit of adopting the approach. Chapter 7 discusses the results and some other findings, followed by the discussion of threats to validity in Chapter 8. Chapter 9 discusses the related work. Chapter 10 shows concluding remarks and future work.

Chapter 2

Background

2.1 Software Product Line

A *software product line*[33] (SPL) is a set of similar software-intensive systems that satisfies different requirements of customers within a certain domain, and is developed in a prescribed way from a set of core software assets. The core assets are also developed in a prescribed way, which is based on analyzing the commonalities and variabilities of the requirements within a certain domain. A software product line targets certain domains because systems within the same domain usually share many commonalities such as business needs, software architecture and implementation techniques. The economic benefit of adopting the software product line approach is from the large amount of reuse of core assets in different products of the product line.

Software product line engineering (SPLE) refers to a whole set of engineering methods and techniques used in developing software product lines. Activities in software product line engineering include domain analysis, core assets development, application system development (based on core assets), product line maintenance and management. The initial investment of developing a software product line is usually very high. As the number of products grows up, the benefit of adopting the software product line approach will gradually overweigh the initial investment.

In practice, the clone-and-own approach[15] is also used to develop a set of similar software systems. In the clone-and-own approach, a new product is developed by cloning and modifying the assets of an existing product. Advantages of this approach include that it has low adoption cost, and a developer of a certain product can develop the product

independently from developers of other products. However, it often yields substantial maintenance problems, such as inconsistency, redundancies and scalability problem.

2.2 Feature-oriented Approaches

The feature concept is widely used in developing software product lines. A feature is a unit of functionality that is relevant to some stakeholders, and also a unit of reuse in software product lines. The feature concept is used in *feature-oriented software development*[6], which is a programming paradigm for developing software product lines, and *feature-oriented domain analysis*[23], which is a domain engineering method that uses feature modelling to model commonalities and variabilities of requirements within a certain domain.

In the clone-and-own approach, the feature concept can also be used. Different variants developed by the clone-and-own approach can also be characterized by features. A variant is characterized by the set of features it implements. A feature provides a unit of reuse, which can be cloned or moved among variants. It also provides a unit of consistency management. Different copies of a feature in different variants are maintained consistent w.r.t certain kind of changes, such as bug fix.

2.3 Virtual Platform

In our earlier work, we propose an incremental and minimal invasive SPLE adoption strategy called *virtual platform*[5]. It covers a spectrum of strategies between the ad-hoc clone-and-own approach and the traditional SPLE approach with an integrated platform. The spectrum is divided into six governance levels. As the governance level increases, the cost of preparing reuse (by clone management or SPLE approaches) increases, while the benefit also increases given that the frequency of reuse increases.

Chapter 3

Feature Annotation Approach

In this study, we propose an approach of adding and maintaining a feature model and mapping between features and assets (annotations) along the normal development process. In order to store the feature model and annotations, we designed a lightweight annotation system. Besides that, we also propose some guidelines for developers to adopt the approach.

3.1 Proposed Feature Annotation System

In this study, we designed a lightweight annotation system, which is a set of conventions to maintain a feature model and annotate software assets (folders, files, and code/text fragments in files) with the features they belong to. This system is simple and lightweight. No specific tools are required to adopt this system, and it does not impact the normal development process (unlike the C pre-processor directives, which add an extra build step of pre-processing). It is also language independent.

In the annotation system, we use the Clafer modelling language[1, 7] to record the feature model in a separate file at the root folder of a project. The feature names in the feature model are called feature declarations. Any feature that is implemented and annotated in assets should have a corresponding feature declaration. Figure 3.1 shows an example of a feature model written in Clafer (on the left side), and how the features are referred to in feature annotations, which will be explained in the following paragraphs.

Besides the feature model, we use a set of conventions for annotating folders, files and fragments of files with the features they implement. In feature annotations, we need a way of uniquely referring to the features. Since the names of features are not required to

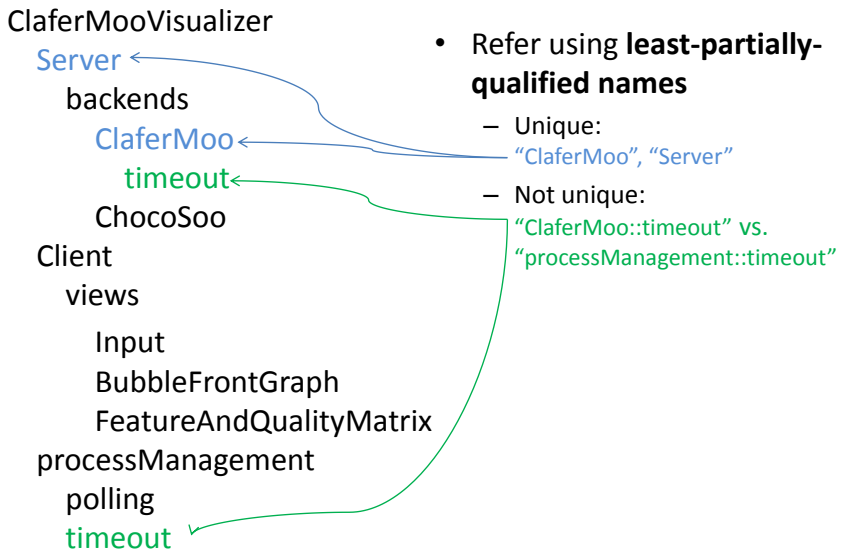

Figure 3.1: Example of feature model and LPQ feature name.

be unique, we use *least-partially-qualified name* (*LPQ name* for short) to refer to features. For a feature that has a unique name in the feature model, the *LPQ name* is the same as the feature name. If a name is not unique, then it must be qualified by some of its ancestor features to make the reference unique.

For example, in Figure 3.1, the features ClaferMoo and Server have unique names in the model, so their LPQ names are the same with their feature names. However, for the feature name timeout, there are two features named timeout. So, we use the names of their parent features to qualify them in order to make the reference unique. The LPQ names of the two features are, thus, ClaferMoo::timeout and processManagement::timeout. The term *least-partially-qualified name* means that we use the least possible number of ancestor features to qualify a certain feature name, compared with *fully-partially-qualified name*, which is the full path of a feature (e.g., ClaferMooVisualize::Server::backends::ClaferMoo::timeout and ClaferMooVisualizer::processManagement::timeout). The *fully-partially-qualified name* is much longer and brittle as compared to the *least-partially-qualified name* when the feature model evolves.

In the annotation system, we use separate files to annotate folders and files. In order to map a whole folder to a feature, a text file called .vp-folder is added in the folder. In the .vp-folder file, there is only one line, which includes the LPQ names of the features this folder maps to. For mapping files to features, a text file called .vp-files is added in a certain folder, which contains the mapping information of all the files in that folder. The .vp-files contains at least one mapping unit. Each mapping unit contains two lines, in which the first line is a list of files and the second line is a list of features. Each file in the first line maps to all the features in the second line.

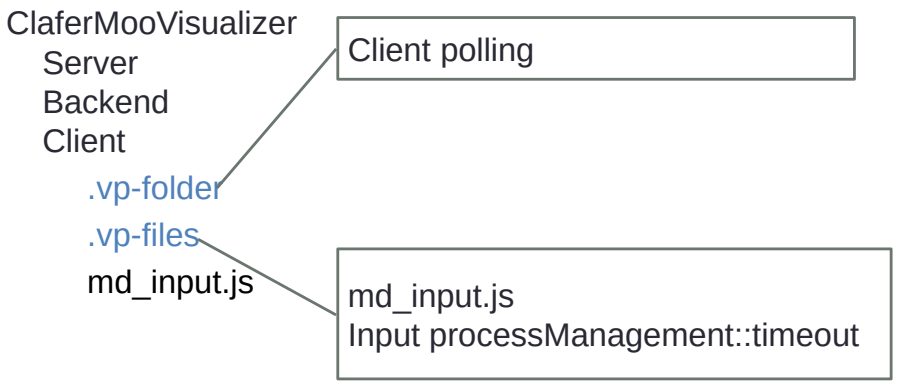

Figure 3.2: Examples of file and folder annotations.

Figure 3.2 shows examples of file and folder annotations. On the left side is a folder structure. Under folder Client, there is a folder annotation file .vp-folder, which maps the whole folder to two features (Client and polling). Also, there is a file annotation file .vp-files, which maps the JavaScript file md_input.js to features Input and timeout.

Besides annotating files and folders, the annotation system can also be used to annotate any arbitrary fragment of a text file (source code file or other text file) to feature(s). The annotation is embedded in the comments of the programming language (which is similar to

JavaDoc[2]), so it will not affect the syntax of the source language and the build process. Inside the comment, in order to mark the beginning of a fragment, developers should write down the list of features and surround the list of features with `&begin [` and `]` delimiters. To mark the end of a fragment, developers should surround the list of features with `&end [` and `]` delimiters.

To annotate a single line of a file, developers should use the appropriate in-line comment syntax of the used programming language to mark the line. Inside the comment, developers should surround the list of features with `&line [` and `]` delimiters. The exact syntax of the marker symbols in the delimiters is customizable depending on the programming languages used in the project in order to prevent syntax conflicts.

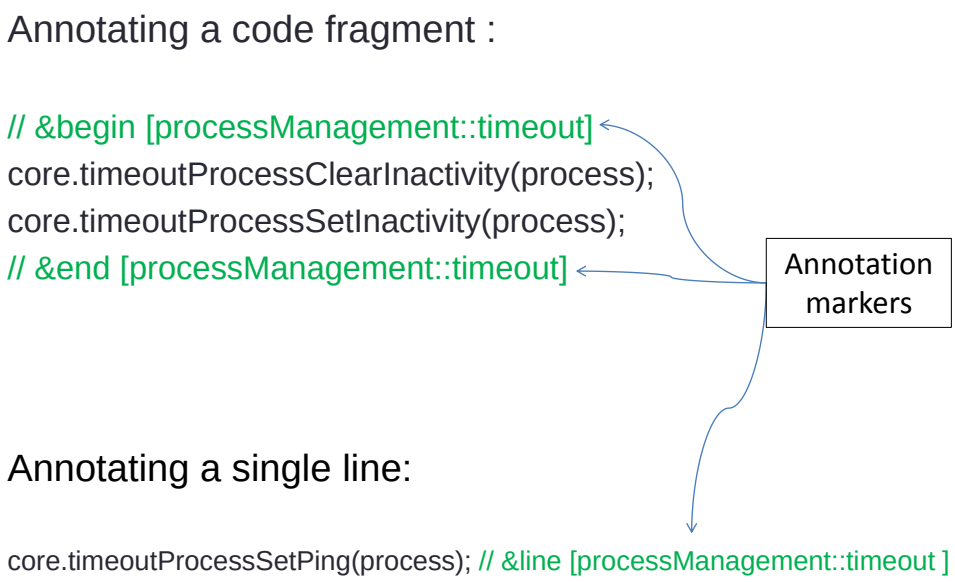

Figure 3.3: Examples of text fragment annotations.

Figure 3.3 shows examples for code fragment annotations. In the first example, a JavaScript code fragment is mapped to feature timeout by two annotation markers in the comments. In the second example, the single code line is mapped to feature timeout by an annotation marker in the inline comment.

In the annotation system, the fragment annotations are in comments, which are embedded inside source code files. This design makes the mappings between features and assets hard to break compared to external traceability records. For example, when some code is added inside a fragment, the annotations automatically shift with the code (see Figure 5.3). However, for external traceability records, such as a record that includes the start and end line numbers of a fragment, they are easy to break as the start and end line numbers are easy to change when a fragment is modified.

In the following parts of the thesis, we use the terminology *annotation marker* to refer to a certain fragment annotation in the comments (a single annotation "`&begin [ Feature List ]`", "`&end [ Feature List ]`" or "`&line [ Feature List ]`" is an annotation marker) or a line in .vp-folder or .vp-files files. It is used as a unit of estimating the cost of adding and maintaining annotations (see Section 6.2), which is similar with using a code line as a unit of estimating software development cost. The terminology *feature annotation* will be used interchangeably with *annotation marker* in the following text, since they refer to the same thing in the context of adopting our feature annotation approach. (In other contexts, feature annotation might have different meaning. For example, in C pre-preprocessor directives, an `#IFDEF` directive is a feature annotation, which can be used to control the inclusion of the annotated code.)

We also use the terminology *asset* to refer to a file, a folder or a fragment in source code files or other text files. An asset is a unit of the software, and it can be annotated by our annotation system.

The semantics of feature annotations is simply feature location. If a file, a folder or a fragment is mapped to some feature(s), this simply means that it belongs to that feature(s). More discussion about the semantics of feature annotations are in Section 7.1.

3.2 Developer Guidelines

In our proposed feature annotation approach, the developer should add and maintain feature declarations (in the feature model) and annotations during the development of software assets. This means that the feature model and annotations should always be consistent with the assets. For example, if a new feature is implemented in the code, a feature declaration of the feature should be immediately added to the feature model and the code of the feature should be immediately annotated.

In practice, the rule of keeping consistency is not required to be strictly enforced. For example, if a developer implements a feature separately in three consecutive commits (in

each of the commits only part of the feature is implemented), the developer is allowed to add the feature declaration and annotations after the three commits, instead of updating feature annotations in each commit. However, it should not be delayed for too long, otherwise the developer might forget where the location of the feature is and as a result extra feature location cost is needed to recover the feature location information. The principle is that the developer should record the feature traceability information as soon as possible, and at least before it is lost.

3.3 Traceability versus Variability

In our approach, the feature annotations are used to record feature traceability information. All assets that implement a certain feature should be annotated. This is different from the variability case, such as C-preprocessor directive annotation, in which only assets that need be switched on/off are annotated to features. Feature traceability and variability serve different purposes. Feature traceability is used to locate assets of features that are needed for performing feature-related development tasks (such as modifying a feature or reusing a feature), while variability aims at the automated derivation of individual variants, by defining variation points (where variants differ).

Figure 3.4 shows an example of the differences between traceability and variability. In the example, there is a feature called Feature3, and a C function f3() implements the feature. Function f3() needs to invoke a utility function util1() to realize its functionality. In the traceability case, all the two functions should be mapped to Feature3 via annotations, because for some use cases the developer needs to trace all the assets of the feature (e.g., when modifying the behavior of the feature, the developer might need to modify both the two functions). However, in the variability case, only f3() is switched on/off (by C-preprocessor directives) in order to enable/disable Feature3. Function util1() is not switched on/off because it is not required to do so, and the function might be needed by other parts of the software.

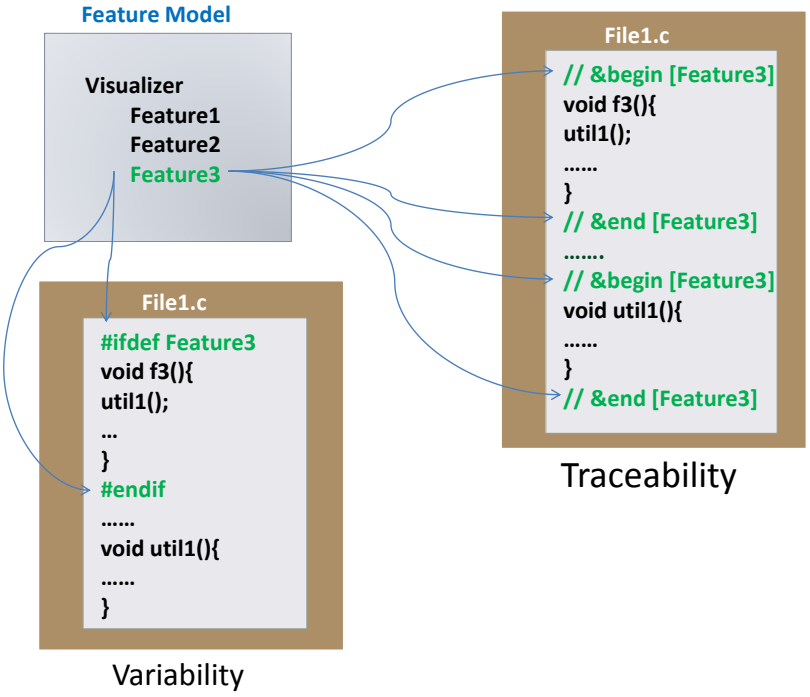

Figure 3.4: Differences between traceability and variability.

Chapter 4

Simulation Case Study

In this chapter, we describe the methodology used in this study, which is basically retroactively simulating the proposed feature annotation approach in a case study. We also describe some detailed information about the subject used in this study.

4.1 Research Questions and Assumptions

In order to investigate the cost and benefit of our proposed approach, we formulate four research questions:

RQ1: *What is the annotation recording and editing cost?*The cost of adopting our feature annotation approach arises from recording and evolving the feature model and annotations. The annotation recording cost arises from adding assets, and the annotation editing cost from evolving assets. We also investigate their ratio to understand the evolution cost in our case study. To measure these costs, we rely on a simple metric: number of annotation markers added/deleted/modified.

RQ2: *What percentage of annotation recordings and editings required additional feature location effort?*We assume that the effort of feature location is zero when the annotations are recorded immediately during developing the assets; however, sometimes the recording is delayed due to 1) annotation mistakes, 2) lower eagerness, or 3) incomplete location knowledge.

RQ3: *What percentage of the invested annotation recording and editing cost saved feature location cost during reuse cases?*We investigate the benefit-cost ratio of our feature annotation approach during reuse cases.

RQ4: *What percentage of feature location cost during reuse could be avoided?*We investigate, how many of the feature locations needed for reuse cases were covered by the recorded feature annotations.

In our study, we make the following assumptions:

1. The developer has the feature location information in mind while implementing a feature, so there is no feature location cost.
2. Adding a feature annotation is trivial and cheap.
3. The cost of recording and maintaining feature traceability information is amortized over multiple uses of such information in reuse or maintenance use cases, whereas, in the lazy strategy[1], the information must be recovered every time it is needed.

We make these assumptions since they are, in our opinion, reasonable to make given the design of our annotation approach. Confirming their validity in practice requires separate studies.

4.2 Subject Description

The subject we used in this study is a series of tools called Clafer Web Tools[3], which are based on the Clafer modelling language[1, 7]. Clafer Web Tools are three web-based tools: ClaferMooVisualizer (short for "Clafer Multi-objective Optimizer Visualizer"), ClaferConfigurator (short for "Clafer Configurator"), and ClaferIDE (short for "Clafer Integrated Development Environment"). ClaferMooVisualizer accepts a model in Clafer with optimization objectives, runs a multi-objective optimization, and visualizes the resulting set of optimal configurations. ClaferConfigurator also accepts a model in Clafer, runs the clafer instance generator, and presents the resulting configurations. ClaferIDE offers basic editing, compilation and instantiation services for a Clafer model over a web-based interface. All three tools are developed using the JavaScript language.

The three tools have many commonalities, and they are in the same domain. They have been developed using the clone-and-own approach[15], and later their common features are re-factored and moved into a central repository. All the tools use Git for configuration management.

[1]In the lazy strategy, feature traceability information is not recorded, but retroactively recovered when needed, by applying semi-automated feature location techniques or manually reading the code.

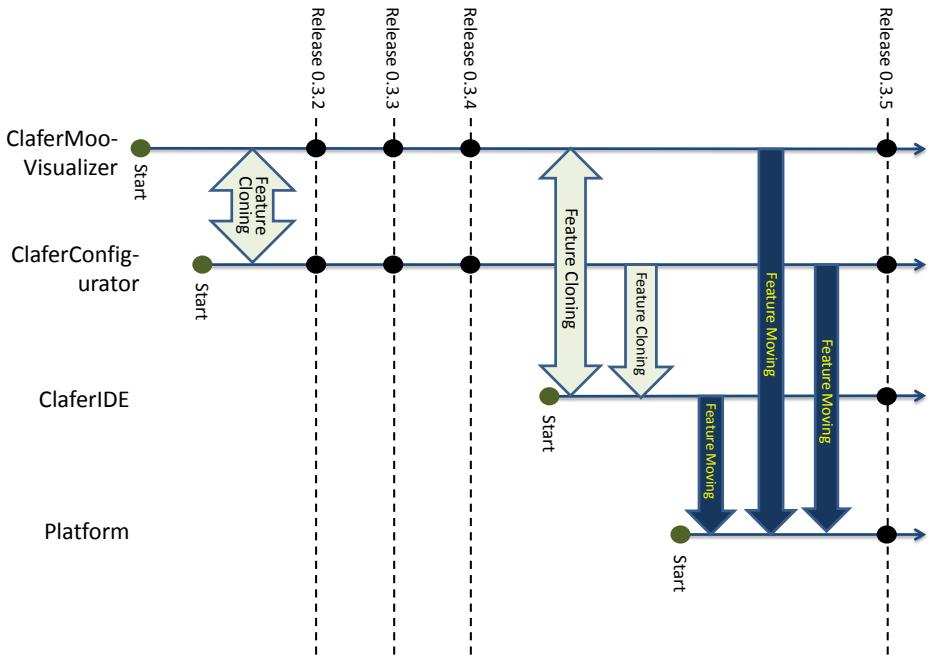

Figure 4.1: Clafer Web Tools development history.

Figure 4.1 shows the development history of Clafer Web Tools. Each horizontal line represents the development history of one of the tools. The three tools have their corresponding Git repositories, and the "Platform" is a separate Git repository which contains features shared by the three tools. The "Platform" repository is used as a Git sub-module by the repositories of all the other three tools, and can be regarded as a common library.

ClaferMooVisualizer was developed first. After ClaferMooVisualizer was developed for some time, ClaferConfigurator was developed by cloning lots of features from ClaferMoo-Visualizer. After the the development of ClaferConfigurator started, the two tools evolved and many features implemented in one of them were subsequently cloned to the other one. ClaferIDE was developed later in a similar way. Most of the features of ClaferIDE came from ClaferMooVisualizer and ClaferConfigurator, and a few features developed in ClaferIDE were also cloned back to Visualizer.

The Platform repository was created after the creation of ClaferIDE. After the Platform repository started, the common features of the three tools are re-factored and moved into the Platform repository, and the Platform repository was added as a sub-module to the

repositories of the other three projects.

The reason we choose this subject is because it is a series of similar products developed by the clone-and-own approach, and we can easily have access to all the resources of the projects, including original developers.

4.3 Simulation Study

In this study, we performed a simulation of the proposed feature annotation approach as if the approach had been used originally in the development of the subject projects. All the development history of the three tools and the Platform repository, before the 0.3.5 release, has been simulated (see Figure 4.1), which includes 779 commits in total. The development history happened before this study began.

4.3.1 Study Design

The simulation study was done in the following way:

1. Create a new Git branch (simulation branch) from the first commit of the subject project.
2. Add an initial feature model, which includes the root feature whose name is the same with the subject project.
3. Merge a few original commits of the subject project into the simulation branch.
4. After the merging, evolve the feature model and annotation, according to the commits that are merged into the simulation branch (e.g., if a new feature was added in the merged commits, then add a new feature declaration to the feature model and annotate the code that belongs to the feature). Besides adding the feature model and annotations, no other change to the original assets (code, documents) is made.
5. Go to Step 3. Repeat the merging and evolving process until all the commits (of development history under study) have been merged into the simulation branch.

Figure 4.2 shows the simulation process. In the figure, black dots represent commits in Git, and lines (both the horizontal lines and the curves) between commits represent their

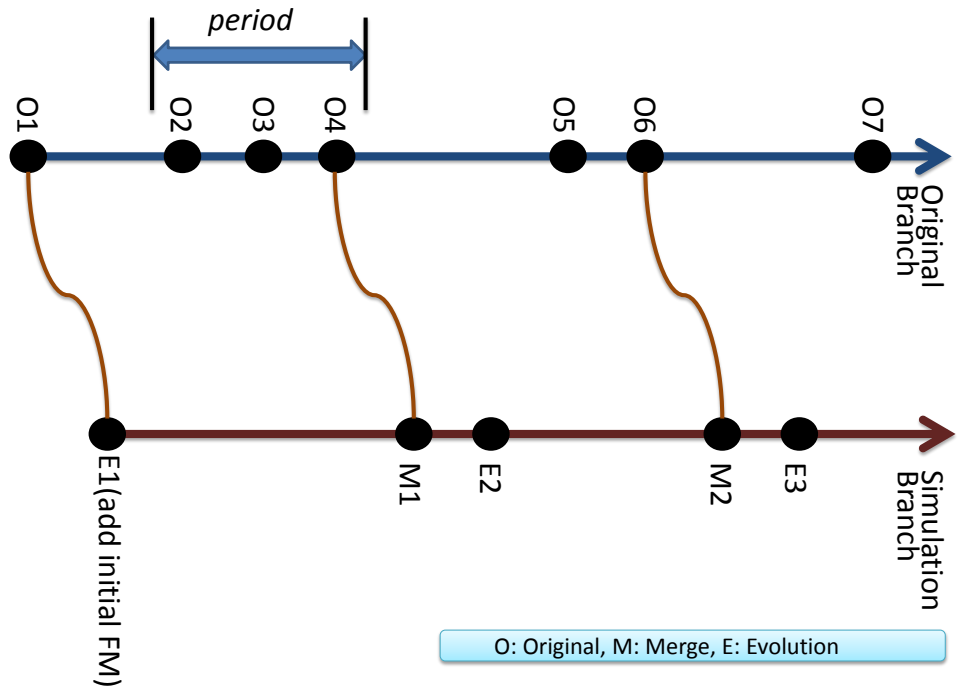

Figure 4.2: Simulation process.

parent-child relationship. For each pair of commits, if there is a direct line (no commits between them) that connects them, then the left-hand-side commit is the parent commit, while the right-hand-side commit is the child commit (e.g., "O3" is the parent commit of "O4"; "O4" and "E1" are both parent commits of "M1"). Commits on the upper horizontal line are original commits of the subject projects, while the commits on the lower horizontal line are made by the researcher during the simulation.

In this example, the researcher created the simulation branch from the first commit ("O1"), then added an initial feature model in commit "E1". After that, the researcher merged three original commits into the simulation branch through commit "M1", evolved the feature model and annotations according to changes made by commits "O2","O3" and "O4", and committed the evolution of the feature model and annotations in commit "E2". This procedure was repeated until all the development history under study had been merged into the simulation branch. Commits made by the researcher in the simulation branch are copies of the original commits, and the copies have a feature model and annotations in it. (In the commit names, "O" stands for "Original," "M" stands for "Merging" and "E"

stands for "Evolving the feature model and annotations.")

4.3.2 Period Selection

In the simulation study, not all commits are merged into the simulation branch via a separate merging commit. For each time of merging, one or a few original commits are merged into the simulation branch together. We use the term *period* to refer to the set of commits that are merged into the simulation branch together in a certain merging, since the set of commits represents a short period of development history (e.g., in Figure 4.2, commits "O2","O3" and "O4" were merged into the simulation branch by one commit, so they formed a *period*).

After a cycle of merging and evolving has finished (e.g., after "E1" in Figure 4.2) the researcher manually looks through the original commits right after last merged commit (e.g., "O2","O3"...) and selects a commit to directly merge into the simulation branch (e.g., "O4"). The commit which is directly merged into the simulation branch becomes the *end point* of a period (e.g., "O4"), and the original commit right after the *end point* becomes the *start point* of the next period (e.g., "O5"). The researcher selects the directly merged commit (e.g., "O4") in the following cases:

1. **When it is necessary to evolve the feature model or annotations.**
   While the researcher is looking through the original commits, if the researcher finds that the software assets in a certain original commit[2] have evolved enough such that the feature model or annotations should be updated in order to keep consistent with the assets, then a merging is performed.

   For example, in Figure 4.3, the researcher found that a new feature "feature1" was added in in commit "O4", which made it necessary to update the feature model and annotations (adding the feature declaration and annotations of "feature1"). So, the merge "M2" was performed, and the feature declaration and annotations of "feature1" was added in commit "E2".

   Sometimes, several commits may be merged into the simulation branch by one merging, although each of them has evolved enough such that the feature model or annotations should be updated. The reason is that these commits belong to the same development task performed in the history, and thus should be put in one period.

   For example, in figure 4.4, a single development task, which is implementing "feature1", was done in three commits ("O2", "O3" and "O4"). So, the researcher

[2]In Git, a commit is a snapshot of the contents in the repository.

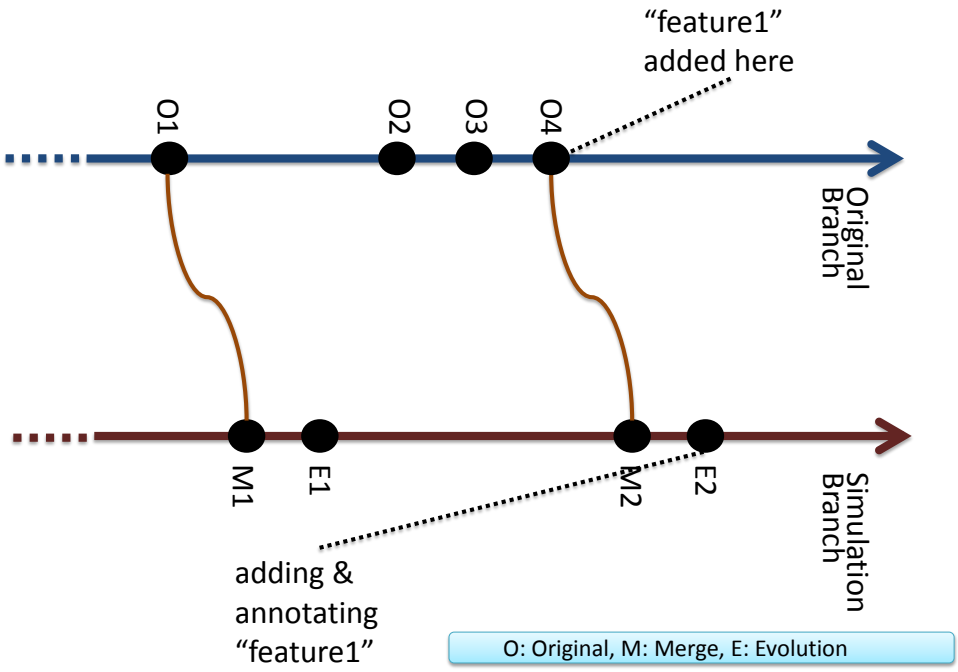

Figure 4.3: Merging after a single commit with feature evolution.

merged the three commits together in "M2", and added the feature declaration and annotations of "feature1" after that.

The researcher identifies the development tasks based on his own judgement, which is based on the information from commit messages, the issue tracker and also original developers. Later, those development tasks are classified into different types, which are later identified as evolution patterns (see Chapter 5).

2. **Synchronizing the simulation branch with the original branch.**
Sometimes, a certain original commit is directly merged into the simulation branch just for synchronizing the simulation branch with the original development branch. Otherwise the simulation branch will not keep up with the original development branch very closely. For each *end point* commit, it has a corresponding commit (annotated mirror commit) in the simulation branch. The annotated mirror commit has exactly the same content with the original commit, except that it has a feature model and annotations in it.

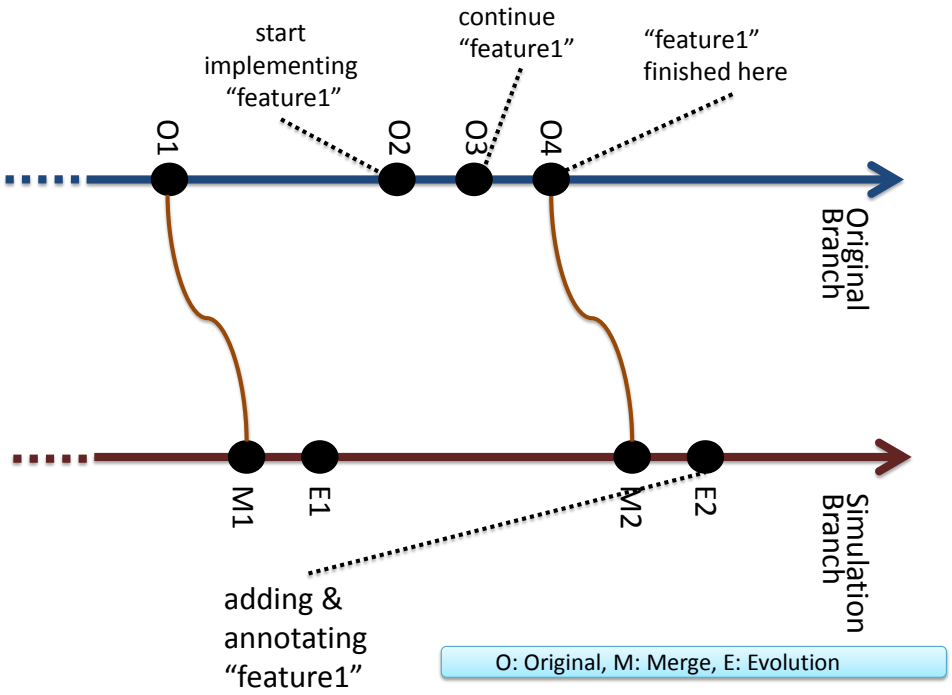

Figure 4.4: Merging after several commits involving feature evolution.

However, for those original commits that are not *end point* commits, none of them has an annotated mirror commit. So, in order to frequently have commits with annotated mirror commits (in other words, to limit the size of each period), the researcher sometimes synchronizes the simulation branch with the original development branch even if there is no need to evolve the feature model or annotations. The benefit of frequently having commits with annotated mirror commits is that we can analyze the evolution history of features in a more fine-grained way (periods are small).

For example, in Figure 4.5, commit “O5” was merged into the simulation branch just for synchronizing the simulation branch at that point. After that merging, commit “O5” has an annotated copy of it (annotated mirror commit, “M2”), while the commits around it do not have such a copy.

The selection of such a synchronizing point follows the following criteria:

- **When an interesting evolution of software assets happens.**
  The researcher synchronizes the simulation branch when some interesting evo-

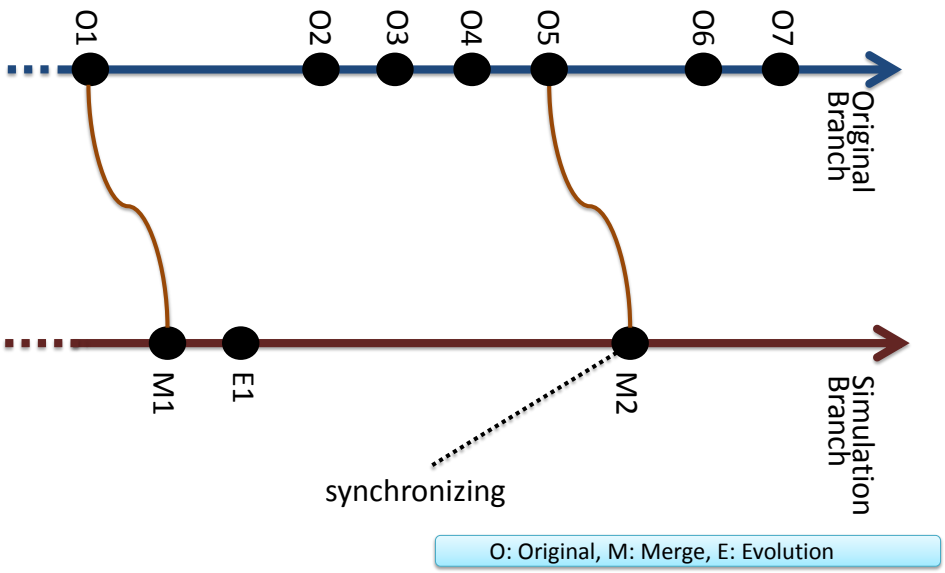

Figure 4.5: Merging for synchronizing.

lution of software assets happens. For example, when code inside an annotated fragment is re-factored. These patterns do not require the feature model or annotations to be manually updated, so they are not the focus of this study, but may be explored in a future study.

- **When an important milestone is reached.**
  Some important milestones are also synchronized, like release points.
- **When the evolution of software assets is significant enough.**
  The researcher also synchronizes the simulation branch when the researcher observes that the original branch has evolved significantly since the last merging. The significance is judged by the researcher subjectively. Cases include when some important changes are made (e.g., an important bug fixing).

In the simulation study we performed, there are 210 periods in total. The largest period contains 18 original commits, while the smallest only contains 1 commit.

4.3.3 Feature Identification

During the simulation process, the researcher needs to manually look at the code or other assets that are added/deleted/modified in original commits, decompose them into features and build the feature model and annotations accordingly. This is essentially a feature identification task. In the simulation study, feature identification was performed based on the information available in the subject projects, including:

1. **Records in commit messages.**
   Commit messages describe what changes were made in certain commits. In the subject projects, changes made to features were usually described in commit messages. So, they can be used to identify features. For example, if a commit has a message like "implemented feature1", then the researcher merges the commit into simulation branch, adds a new feature "feature1" to the feature model, and annotates the code added in the commit with "feature1."

2. **Source code.**
   Source code is the main asset of the projects, wherein features are implemented. The researcher also looks at the source code that is added, deleted, modified during the simulation process, and identifies features based on the researcher's understanding of the code. For example, when a new class/method is added in a commit, and the researcher thinks that this is potentially a unit of reuse, then the researcher merges the commit into the simulation branch, adds a new feature to the feature model, and annotates the class/method.

3. **Feature models in the project Wiki.**
   In the Wiki website[4] of the subject projects, there are feature models for all the subject projects. Those feature models were written down by one of the original developers at a certain time between the 0.3.4 and the 0.3.5 release, before this study began. They represent the features that are implicit in the original developer's mind. The researcher reads through the feature models before the study begins. During the simulation process, the researcher especially watches over the evolution process to see when the features in those feature models are added/evolved, and evolves the feature model and annotations in the simulation branch accordingly.

4. **Issue tracker, original developers.**
   Besides the information sources mentioned above, the researcher also uses information from the issue tracker of subject projects and original developers, which helps the researcher to understand the changes in original commits.

Chapter 5

Evolution Patterns

After the simulation process, we manually looked at the evolution history of the subject projects and identified several feature evolution patterns. An evolution pattern is a type of software development task (can also have sub-types). We manually investigated all the software development tasks that appeared in the simulation study, and classified them into evolution patterns. Each pattern either involves intentionally adding, deleting or modifying the feature model or annotations, which brings extra cost caused by adopting the feature annotation approach, or involves benefit of having explicit feature model and annotations, or both. These patterns are identified based on the characteristic of their changes made to the assets, the feature model and annotations, and their extra cost and/or benefit. Here we will informally describe costs and benefits of adopting the feature annotation approach, and later formalize and measure them.

The following are the evolution patterns we found in this study:

- **P1: Adding new assets.**
  In this pattern, either a new feature and some new assets (not created based on existing assets) of the feature are added to the project, or some new assets are added, which belong to a feature that already exists in the feature model. The newly added assets can be fragments, files or folders, and annotation markers, which map the assets to the feature, should be added immediately or shortly after that.

  This pattern includes the following sub-patterns:

  - **P1.1: Adding a new feature and new assets.**
    In this sub-pattern, a new feature declaration is added to the feature model, and the assets of the feature are added to the project and annotated.

– **P1.2: Adding new assets to an existing feature.**
In this sub-pattern, new assets are added and mapped to an existing feature by annotations. An existing feature means a feature which already has a feature declaration in the feature model.

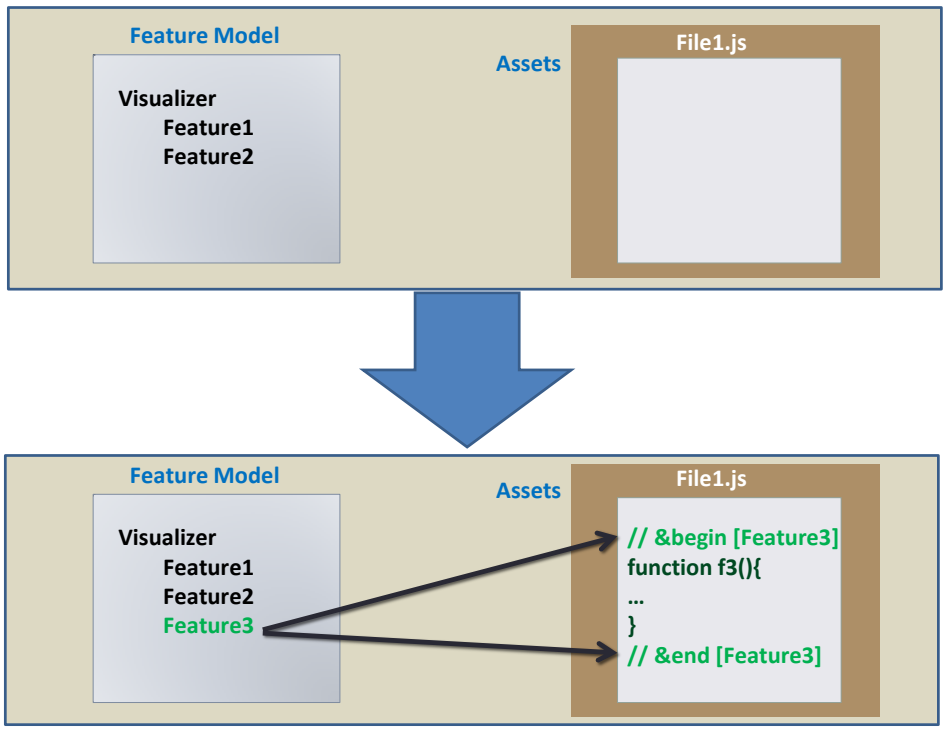

Figure 5.1: Example of pattern "P1.1: Adding a new feature and new assets."

Figure 5.1 shows an example of this pattern. In the example, a new feature declaration Feature3 was added to the feature model, and the asset that implements the feature (function f3()) was added and annotated.

For this pattern, there exists extra cost of adopting the feature annotation approach. The extra cost is introduced by adding a new feature declaration and annotations. This is an extra task for the developer, compared with not adopting the feature annotation approach. We assume that there is no feature location cost in this pattern. Since the developer adds the annotation markers together with the assets implementing the feature, the developer has the feature location in mind and, thus, no cost of finding feature location is needed, compared with recovering feature location in a

reverse-engineering context. Additionally, when using a version control system, the locations of all changes are directly visible before a commit. Future tool support can propose the positions of putting annotation markers based on the change-set.

Under the assumption that there is no feature location cost in this pattern, we argue that the extra cost is relatively low, compared with the development cost. When developing a new feature, the main cost is designing, implementing, and testing the functionality of the feature. The cost of adding a few annotation markers during development is relatively low.

- **P2: Adding re-factored/returned assets.**
  In this pattern, either a new feature and some re-factored/returned assets of the feature are added to the project, or some re-factored/returned assets are added and mapped to an existing feature. Re-factored assets are assets that are re-factored out from other assets, and returned assets are those that used to exist in the project, but were removed at some point later. An example of this pattern is re-factoring a scattered aspect into a class, in which the aspect (e.g., code that handles user logging) is scattered over several classes, and is re-factored into a class called "logging." A feature called "logging" is added to the feature model and the class "logging" is mapped to that feature.

  This pattern includes the following two sub-patterns:

  - **P2.1: Adding a new feature and re-factored/returned assets.**
    In this sub-pattern, a new feature declaration is added to the feature model, and the re-factored/returned assets of the feature are added and annotated immediately.
  - **P2.2: Adding re-factored/returned assets to an existing feature.**
    In this sub-pattern, re-factored/returned assets are added to an existing feature, including their annotations.

  For this pattern, the extra cost is introduced by adding a new feature declaration and annotations. We also assume that there is no feature location cost in this pattern, which is similar to the pattern "P1: Adding new assets." However, since the new feature is built on re-factored or returned assets, the development cost is probably lower than developing something substantially new. So, in this pattern, the percentage of extra cost might be higher than in the pattern "P1: Adding new assets."

- **P3: Removing/disabling a feature.**
  In this pattern, assets of an existing feature are removed or disabled (by commenting

out). The feature annotation markers of the feature need to be removed together with the assets. The feature declaration is either removed or not, depending on whether the developer thinks the feature will possibly return in the future.

This pattern involves some extra cost, but not too much. Fragment and folder annotations can be removed with the assets together, thus, no extra cost is introduced. The removal of file annotations and feature declarations brings extra cost, but could be easily automated by a future tool.

In this pattern, the benefit of having feature annotations is much higher than the cost. The feature annotations provide the location of the feature, which is essential for a developer or a tool to remove the assets of the feature.

- **P4: Structural change within one feature.**
  In this pattern, the structure of the feature assets is changed, and the annotations need to be updated in order to keep the annotations correct and succinct. Here are the sub-patterns of this pattern.

  - **P4.1: Asset splitting.**
    In this sub-pattern, an asset is split into two. For splitting a fragment, the fragments after splitting are not adjacent to each other any more (otherwise they should be annotated as one fragment). The annotation markers are changed correspondingly to map the split assets to the original feature. The rationale behind this pattern is that either part of a fragment is re-factored out into a new fragment (e.g., a piece of code in a method is re-factored into a new method) or some irrelevant code, which does not belong to the same feature, is inserted into a fragment.

    Figure 5.2 shows an example of this sub-pattern. In the example, a part of the code in function f3() was re-factored into a new function f3Util(), and the new function is not adjacent to function f3() (there is some code between f3() and f3Util(), and the code does not belong to feature Feature3). So, new annotations were added to map f3Util() to Feature3. The original fragment is split into two fragments after the evolution.

  - **P4.2: Asset merging.**
    In this sub-pattern, several assets are merged into one, which is basically the reverse of asset splitting. In this study, it happened when several fragments of one feature become adjacent to each other after moving them into one place, and they are merged into one fragment by deleting the annotation markers between them.

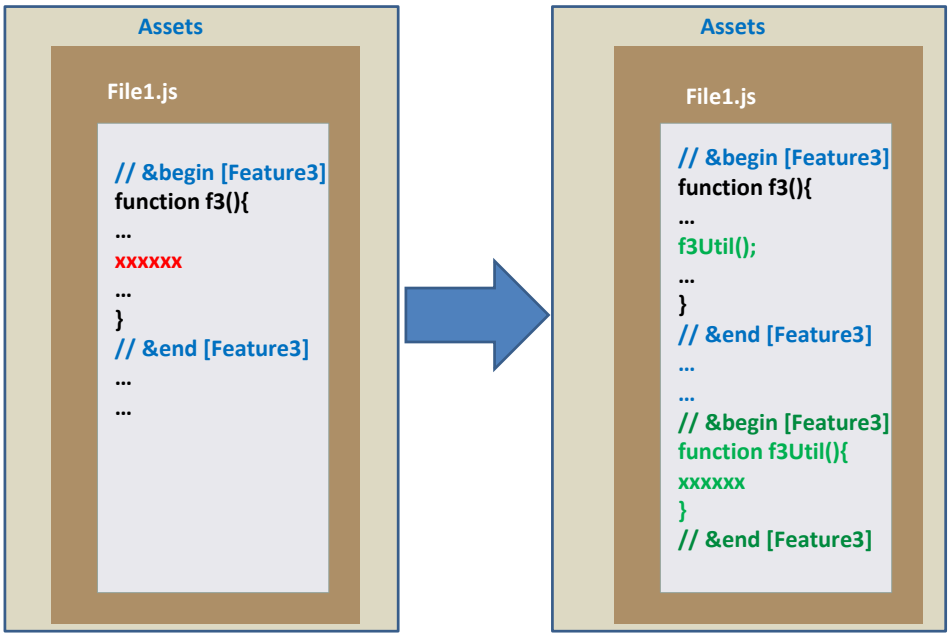

Figure 5.2: Example of pattern "P4.1: Asset splitting."

– **P4.3: Feature modularization into file.**
  In this sub-pattern, a feature whose assets are all fragments is modularized into a module file, and the annotations are changed from fragment annotations to a file annotation. After the modularization, the feature is in a separate module file. During the process, fragment annotation markers are removed, and a file annotation is added, which maps the module file to that feature.

In this pattern, both cost and benefit are involved. The extra cost is introduced by adding/removing/modifying annotation markers. The percentage of extra cost might also be higher than the pattern "P1: Adding new assets." However, the cost of adjusting annotations is still comparably low since there is no feature location cost.

In the sub-pattern "P4.3: Feature modularization into file," feature annotations can bring significant benefit. In order to perform the changes in the sub-pattern, the developer needs to know the location of all the feature assets, which can be provided

by the feature annotations.

- **P5: Feature model and mapping re-factoring.**
  In this pattern, no change is made to the assets in the project. Changes only include modifying the feature model and/or the feature annotations.

  Here are the sub-patterns of this pattern:

  - **P5.1: Identifying a new feature.**
    In this sub-pattern, an existing part of the software system is identified as a new feature. A new feature declaration is added to the feature model, and some existing assets are annotated to the feature. In this case, the feature already exists in the project, but is not explicitly declared and annotated.

    In the simulation study, most of the cases of this sub-pattern are identifying different part(s) of an existing feature F as new sub-feature(s) of F. It is essentially decomposing a feature into smaller sub-features, which could support reuse and variability in a more fine-grained way. For example, if a feature F is further decomposed into three sub-features F1, F2 and F3, then a developer can clone the sub-feature F1 into another project without cloning F2 and F3. However, if the three sub-features are not declared and annotated, the developer can only clone the feature F as a whole. (The developer can still clone a smaller part of the feature F, but in such case the developer can not benefit from the annotations too much, since the smaller part is not annotated as a sub-feature of F.)

    The simulation study is done by a researcher who is not one of the original developers. In reality, if a developer adds and maintains the feature model and annotations alone normal development, some of the cases of this sub-pattern might be avoided because the developer could add the feature declaration and annotations while the feature is being implemented. However, since the developer can not accurately predict the future requirement for reuse and variability, this sub-pattern is still likely to happen. For example, when a developer finds that a smaller part of feature F is need to be cloned separately or be made optional, then the developer will identify the smaller part as a new sub-feature (F1) of F. The developer did not declare and annotate the sub-feature F1 when it was being implemented because he/she failed to predict that the sub-feature F1 would be a unit of reuse or a variation point in the future.

  - **P5.2: Adjusting position of a feature.**
    In this sub-pattern, the position of a feature in the feature model is changed, and constraints of the feature model are possibly changed at the same time.

(In this study, we did not add constraints to the feature model because we focused on the evolution of feature annotations.) If the least-partially-qualified name (which is used to identify the feature) is changed, all the annotations of the feature need to be updated accordingly. The update of feature annotations could be easily automated by a future tool.

  - **P5.3: Feature renaming.**
    In this sub-pattern, a feature is renamed in the feature model, and feature annotations are updated accordingly. The update of feature annotations could also be easily automated by a future tool.
  - **P5.4: Adding feature declaration only.**
    In this sub-pattern, only a new feature declaration is added to the feature model. The assets of the feature are going to be implemented and annotated later.

  For this pattern, the extra cost is introduced by evolving the feature model or annotations. No development cost is involved here. While some sub-patterns in this pattern are simple and can be easily automated (such as "P5.3: Feature renaming"), some might involve higher cost. For example, the main cost of "P5.1: Identifying a new feature" might not be from adding annotation markers, but from identifying which piece of functionality should be a feature, and recovering the location of the feature.

- **P6: Fixing annotations.**
  In this pattern, an error in the feature model or annotations is found and fixed. According to the cost of fixing the error, we identified two sub-patterns:

  - **P6.1: Simple fixes.**
    In this sub-pattern, a simple error is found in feature annotations and fixed by the developer. It could be a syntax error, or some assets that do not belong to a feature were mistakenly annotated with that feature. Some of these cases could also be automated.
  - **P6.2: Fixing missing annotations.**
    In this pattern, the error is that some annotation markers are missing, since they were neglected by the developer. It can be either a begin/end marker of fragment annotations that does not have end/begin marker to pair with it, or some assets of feature are completely not annotated. The cost of fixing missing annotations is potentially higher than in the simple cases, since the developer needs to find out where the missing annotations should be put in, which is essentially a feature location task.

In the simulation study, most of the cases of this sub-pattern happened because the researcher neglected some parts of a feature due to limited familiarity with the code (the researcher is not one of the original developers). In reality, this sub-pattern might happen less often since the original developers are more familiar with the code. Most of the fixes were triggered by modifications to the features. For example, if the researcher found an original commit with a message like "modified feature1," but none of the assets modified in the commit was annotated with feature1, then the researcher realized that there were some missing annotations of feature1. In such case, the researcher merged the original commit into the simulation branch and added the missing annotations.

- **P7: Adjusting file/folder mappings.**
  In this pattern, some file or folder annotation markers need to be updated due to changes made to the corresponding files or folders. In our annotation system design, the mapping between files and features is stored in a separate `.vp_files` file in each folder. So, when files are moved to another folder, renamed or removed, the `.vp_files` files need to be updated, which brings extra cost. For folder mapping adjusting, there was only one case found in the study, in which a folder is moved into a newly created folder that belonged to the same feature, and the folder annotation file (`.vp_folder`) was moved into the newly created folder. Adjusting file mapping can be avoided if we embed the file annotation in each mapped file. The limitation is that this can only be done for text files, and can not be used to map binary files to features.

- **P8: Creating an initial clone.**
  In this pattern, a new project is created by cloning assets from an existing project. Usually, the basic infrastructure (framework, mandatory assets, incl. libraries and documentation) is copied first. After that, individual or all features are propagated. Later, some undesired sub-features of those propagated features are removed or commented out.

  The benefit here is that by looking at the feature models, a developer can get a clear view of which features are implemented in each existing project. This can help the developer decide which project should be used as the base for creating the new one, and what features can be propagated from other projects (other than the base project).

- **P9: Feature propagation.**
  In this pattern, a feature (or part of a feature) in one project is propagated to another project by cloning/moving the assets and manually integrating them. The two

projects are expected to have some similarities (e.g., sharing the same implementation framework), which makes the integration of the propagated feature feasible. Ideally, these two projects should belong to the same product line.

The benefit of feature annotations here is providing the location of the feature to be propagated. With the feature annotations, a developer can easily find the location of the assets of the feature, which is to be propagated into another project.

There is also extra cost of evolving the feature model and annotations. The fragment and folder annotations are propagated together with the assets, and thus no extra cost is introduced. However, the feature declaration and file annotations have to be propagated separately, which brings extra cost. However, the cost is negligible compared to the benefit.

- **P10: Modifying a feature without changing annotations.**
  This is a pattern that happens most often. It involves many activities, such as fixing a bug in the feature's assets, extending an asset of a feature, or re-factoring an asset of a feature. All these activities share one characteristic: the developer needs to know the location of the feature in order to perform it. Feature annotations can facilitate these activities by providing the location.

  This pattern does not require the developer to intentionally change feature annotations, so no extra cost is introduced. For example, if a developer adds some lines of code inside a fragment, the annotation markers of that fragment automatically shift with the code, which does not require the developer to pay attention to the annotations. Thus, no extra cost is introduced.

  Figure 5.3 shows an example of this pattern. In the example, a piece of code was added inside a fragment of Feature3. After the code was added, the end annotation marker automatically shifted with the code. The developer does not need to manually change any annotation.

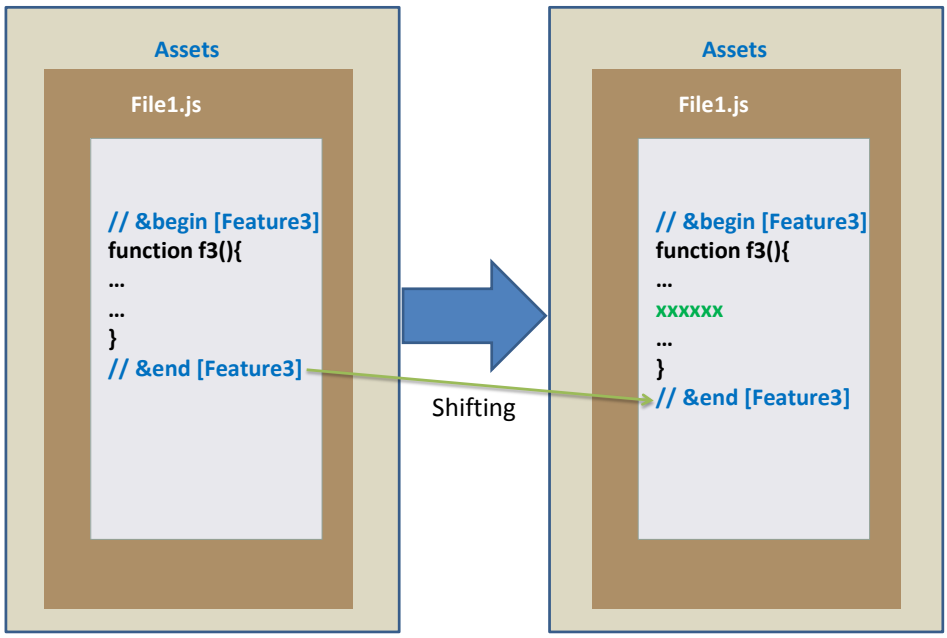

Figure 5.3: Example of pattern "P10: Modifying a feature without changing annotations."

Chapter 6

Analytical and Empirical Evaluation

In this chapter, we present analytical and empirical evaluation of the feature annotation approach, which is based on analyzing the cost and benefit of adopting the approach. We answer the four research questions in this chapter.

6.1 Frequency of Evolution Patterns

In order to know the frequency of different evolution patterns, we collected the number of occurrences of each pattern. An occurrence of a pattern is a period of the simulation that involves the pattern (see Section 4.3.2 for the definition of periods). The frequency of a pattern is the number of periods that involve the pattern.

For example, the evolution pattern "P4: Structural change within one feature" (see Chapter 5) has several sub-patterns, including "P4.1: Asset splitting" and "P4.2: Asset merging." Imagine there is only one period, which involves evolution of both "P4.1: Asset splitting" and "P4.2: Asset merging," then it counts as one occurrence of "P4.1: Asset splitting", one occurrence of "P4.2: Asset merging", and also one occurrence of "P4: Structural change within one feature." The frequency of pattern "P4: Structural change within one feature" (also "P4.1: Asset splitting" and "P4.2: Asset merging") is 1.

For a pattern with sub-patterns, the frequency is not definitely the sum of the frequency of its sub-patterns. In the example mentioned above, there is only one period (p_1) that involves the pattern "P4: Structural change within one feature," but the sum of the frequency of its sub-patterns is 2.

Table 6.1 shows the frequency of all evolution patterns. Figure 6.1 visualizes the frequencies of patterns (not including sub-patterns) in Table 6.1. From the frequency data, we can see that the pattern that happens most often is "P10: Modifying a feature without changing annotations." All the 210 periods in the simulation study involve that pattern.

Patterns	Sub-patterns	Sub-pattern frequency	Pattern frequency
P1: Adding new assets.	P1.1	41	55
	P1.2	14	
P2: Adding re-factored/returned assets.	P2.1	4	8
	P2.2	4	
P3: Removing/disabling a feature.			7
P4: Structural change within one feature.	P4.1	4	7
	P4.2	2	
	P4.3	2	
P5: Feature model and mapping re-factoring.	P5.1	6	16
	P5.2	3	
	P5.3	3	
	P5.4	4	
P6: Fixing annotations.	P6.1	3	11
	P6.2	9	
P7: Adjusting file/folder mappings.			9
P8: Creating an initial clone.			2
P9: Feature propagation.			14
P10: Modifying a feature without changing annotations.			210

Table 6.1: Frequency of evolution patterns (based on number of periods).

6.2 Cost Model

In this study, we formulated a cost model based on the characteristics of extra costs caused by adopting the feature annotation approach. In the model, cost is measured by the number of annotation markers added/deleted/modified, which is essentially a software metric that can not be directly related to the economic cost. So, this model is not intended to be

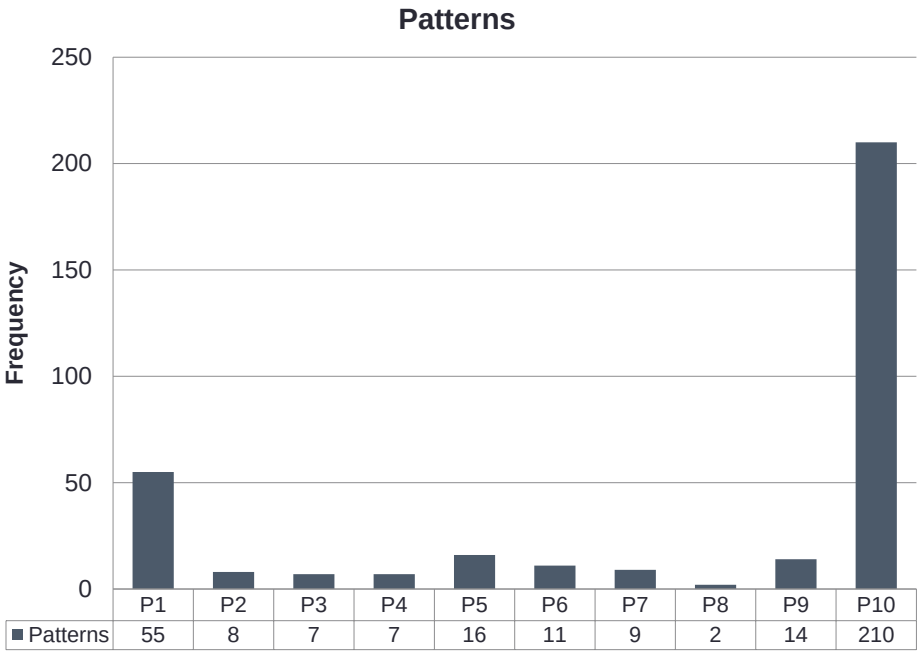

Figure 6.1: Frequency of evolution patterns (based on number of periods).

used for measuring economic cost in reality. We developed this cost model in order to describe our findings in this study (relations between different kinds of cost, and their characteristics), and estimate the amount of extra cost of our feature annotation approach.

$$C_{total} = C_{feature\_model\_structure\_design} + \sum_{f_i \in F} C_f(f_i) \tag{6.1}$$

$$C_f(f_i) = C_{feature\_identification}(f_i) + C_{evo}(f_i) \tag{6.2}$$

$$C_{evo}(f_i) = C_{mdl}(f_i) + C_{annot}(f_i) \tag{6.3}$$

$$C_{mdl}(f_i) = \sum_{a \in A(f_i)} C_{mdl}(a) \tag{6.4}$$

$$C_{annot}(f_i) = \sum_{a \in A(f_i)} C_{annot}(a) \tag{6.5}$$

$$C_{mdl}(a) = \begin{cases} 1 & a \text{ is } \textbf{Feature renaming} \\ 1 & a \text{ is } \textbf{Feature moving} \\ 1 & a \text{ is } \textbf{Adding/removing feature declaration} \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

$$C_{annot}(a) = \begin{cases} \#\text{ of added/modified markers} & a \text{ is } \textbf{Adding annotations together with developing assets} \\ \#\text{ of deleted/modified markers} & a \text{ is } \textbf{Asset merging} \\ \#\text{ of deleted/modified markers} + 2 & a \text{ is } \textbf{Feature modularization} \\ \#\text{ of fixed markers} & a \text{ is } \textbf{Simple bug fixing} \\ \#\text{ of added/modified markers} & a \text{ is } \textbf{Recovering mapping} \\ \#\text{ of adjusted/deleted markers} & a \text{ is } \textbf{File/folder anno adjusting} \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

The cost model consists of a set of formulas. In Formula 6.1, C_{total} means the total cost of adding and maintaining the feature model and annotations during a certain period of development history. It consists of two parts. The first part, $C_{feature\_model\_structure\_design}$, is the cost of designing the structure of the whole feature model. The second part, $\sum_{f_i \in F} C_f(f_i)$, is the sum of the cost of adding and maintaining each feature. The set of features, $F = \{f_1, f_2, ..., f_n\}$, includes all the features that appeared in the period of development history (also includes features that used to exist and later removed at some point).

The cost of adding and maintaining each feature is defined in Formula 6.2. It consists of the feature identification cost and evolution cost (including initial adding and subsequent maintaining). $C_{feature\_identification}(f_i)$ is the cost of identifying feature f_i. It includes the cost of clearly deciding what the intention/function/meaning of the feature is, how the feature is going to be implemented, and whether it is worth annotating (potential variation point or unit of reuse).

$C_{evo}(f_i)$ is the cost involved with evolving feature f_i (see Formula 6.3). It consists of the cost related to evolving the feature model ($C_{mdl}(f_i)$) and the cost related to evolving feature annotations ($C_{annot}(f_i)$).

In order to explain how we estimated the evolution cost in the cost model, we introduce the concept of *annotation change*. An *annotation change* is a change made to either the

feature model or feature annotations by the developer at a certain time, and it must introduce extra cost of adopting the feature annotation approach. For example, adding new annotation markers for newly implemented code fragment is an annotation change, because it introduces extra cost of adopting the feature annotation approach. On the other hand, changes such as that annotation markers shift when code is edited do not require the developer to consciously change the annotation, and thus have no extra cost and do not count as annotation changes.

In Formula 6.4, $C_{mdl}(f_i)$ is the sum of the feature-model-related cost of all annotation changes made to feature f_i. In Formula 6.5, $C_{annot}(f_i)$ is the sum of the annotation-related cost of all annotation changes made to feature f_i. ($A(f_i)$ is the set of annotation changes made to feature f_i.)

Annotation changes happen in the evolution patterns that have extra cost related to the feature model and annotations. An occurrence of a pattern may involve one or several annotation changes. In Formulas 6.6 and 6.7, the cost of an annotation change is measured according to their types. (Table 6.2 presents the annotation change types involved in each evolution pattern. It also presents the benefit of each pattern, which will be explained in Section 6.3.) The following is a list of the types and their cost:

1. **Feature renaming.**
   This type happens in the evolution pattern "P5.3: Feature renaming." It only requires the developer to change one line in the feature model. Update of the references in annotations can be automated. So the cost is 1 in $C_{mdl}(a)$.

2. **Feature moving.**
   This type happens in the evolution pattern "P5.2: Adjusting position of a feature." which means moving the feature declaration within the feature model. It only requires changing one line in the feature model, so the cost is 1 in $C_{mdl}(a)$. Potentially required update of references in feature annotations (LPQ name might change, see Section 3.1) can also be automated.

3. **Adding/removing feature declaration.**
   This type happens in all the patterns that involve adding or removing a feature declaration in the feature model. It only requires adding or removing one line in the feature model, so the cost is 1 in $C_{mdl}(a)$.

4. **Adding annotations together with developing assets.**
   This type happens in all the patterns where new annotations are added together with the development of the asset content, such as "P1: Adding new assets." Under our

assumptions, no extra feature location cost is needed here. Here, we use the number of added/modified annotation markers to measure the cost of evolving annotations in $C_{annot}(a)$.

5. **Asset merging.**
   This type happens in the evolution pattern "P4.2: Asset merging," in which some markers are removed or modified due to asset merging. Here we use the number of deleted/modified markers to measure the cost of evolving annotation in $C_{annot}(a)$.

6. **Feature modularization.**
   This type happens in the evolution pattern "P4.3: Feature modularization into file," in which a feature consisting of fragment assets is modularized into a single file. Here the cost measure includes the number of deleted/modified markers, and two lines of file annotation markers which corresponds to the cost of adding a file mapping for the newly created module file.

7. **Simple bug fixing.**
   This type happens in the pattern "P6.1: Simple fixes." All bug fixing of annotations which does not have feature location cost belongs to this type. Since it does not have extra feature location cost, we use the number of fixed annotation markers to measure the cost.

8. **Recovering mapping.**
   This type happens in evolution patterns "P5.1: Identifying a new feature" and "P6.2: Fixing missing annotations," in which the developer needs to recover the location of a feature and annotate it accordingly. Here, extra feature location cost is required for recovering the positions to put annotation markers.

9. **File/folder annotation adjusting.**
   This type happens in the evolution pattern "P7: Adjusting file/folder mappings" and "P3: Removing/disabling a feature," in which file/folder annotations are moved to another folder, renamed (file name) or deleted. The cost measure is the number of adjusted/deleted annotation markers.

In Formula 6.1, the total cost of evolving the feature model and annotations is aggregated per feature. Alternatively, we can also aggregate the total evolution cost per evolution pattern, which means that the total evolution cost of each evolution pattern is calculated at first, then they are summed up to calculate the total cost. The advantage of this method is that we can compare the cost of different patterns, and, thus, understand which pattern(s) drives costs.

In Formula 6.8, the total cost consists of the feature model structure design cost, total feature identification cost (which is still aggregated per feature in 6.9), and the sum of the evolution cost in different evolution patterns. In Formula 6.10, the cost of a certain pattern is the sum of the cost (including feature model evolving cost $C_{mdl}(a)$ and annotation evolution cost $C_{annot}(a)$) of all annotation changes involved in the pattern. An annotation change is involved in a pattern if and only if the annotation change was performed by the researcher (in reality, by the developer) in an occurrence of that pattern (see Section 6.1).

$$C_{total} = C_{feature\_model\_structure\_design} + C_{total\_feature\_identification} + \sum_{p_i \in P} C_{pattern}(p_i) \tag{6.8}$$

$$C_{total\_feature\_identification} = \sum_{f_i \in F} C_{feature\_identification}(f_i) \tag{6.9}$$

$$C_{pattern}(p_i) = \sum_{a \in A(p_i)} \big(C_{mdl}(a) + C_{annot}(a)\big) \tag{6.10}$$

Table 6.3 shows the cost of all evolution patterns. Figure 6.2 visualizes the cost of patterns in Table 6.3. Based on the cost data, we answer the first two research questions:

RQ1: *What is the annotation recording and editing cost?*
We can simply calculate the annotation recording and editing cost by aggregating the costs of the patterns in which they occur: $C_{record} = \sum_{p_i \in \{P1,P2\}} C_{pattern}(p_i) = 317$ and $C_{edit} = \sum_{p_i \in \{P3,P4,P5,P6,P7,P9\}} C_{pattern}(p_i) = 339$. The total cost is 656[1], which means throughout the development history, 656 lines of feature declarations or annotation markers were added/deleted/modified. From the cost data, we can also see that the pattern that has the highest cost is "P1: Adding new assets," which is also the pattern with the highest frequency among patterns that involve cost.

Opposed to the 656 lines of annotations, throughout the simulated development history, there are 1,798,772 lines of text (including all kinds of text, such as source code or documents) added and 1,251,742 lines of text removed in total. The 0.3.5 release of the four subject repositories has 547030 lines of text in total, and 14794 lines of them are JavaScript source code (excluding libraries). This shows that the cost of adopting our annotation approach is negligible compared with the development cost.

[1]We ignore the cost of "feature model structure design" and "feature identification" in our research questions, since they require separate studies.

The ratio of recording to editing cost $C_{record}/C_{edit} = 107\%$ is interesting, indicating that at least as much—but not much more—effort arises from maintaining annotations as a result of asset evolution. It shows that the annotation maintenance cost does not grow linearly with the amount of evolution in the annotated assets. Compared to storing the traceability information externally, adding or removing a single line in a file will shift the locations of all feature fragments in that file that follow the line and thus require update of the corresponding traceability links.

RQ2: *What percentage of annotation recordings and editings required additional feature location effort?*
We define the number of *annotation omissions* C_{ao} as the number of annotations that were initially omitted when the simulator forgot to annotate or did not anticipate the need to reuse the feature but which were added later on. Adding each omitted annotation requires some additional feature location effort. In our case study, it arose in patterns "P5.1: Identifying a new feature" (25 omitted annotations) and "P6.2: Fixing missing annotations" (36 omitted annotations); in total $C_{ao} = 61$. Thus, in summary, 9.3% of all annotation-related activities incur feature location costs, in addition to recording and editing annotations.

6.3 Benefit Analysis

In this section, we summarize the benefits of adopting the feature annotation approach, and shows quantitative data about the benefit in reuse cases found in the simulation study.

Here is a list of the benefits we found in this case study:

1. **Feature location.**
   Feature annotations can provide the location of the assets that implement a certain feature. The feature location information can be used in many use cases, including removing/disabling a feature, modifying a feature and feature propagation. In all the use cases, the developer needs to know the location of the feature in order to perform the development task.

2. **Feature model.**
   Another important benefit of the feature annotation approach is from the feature models. The feature model of a certain project can provide the information about what features are implemented in the project. By comparing the feature models of different projects, developers and other stakeholders can easily know what functionalities are available in each project.

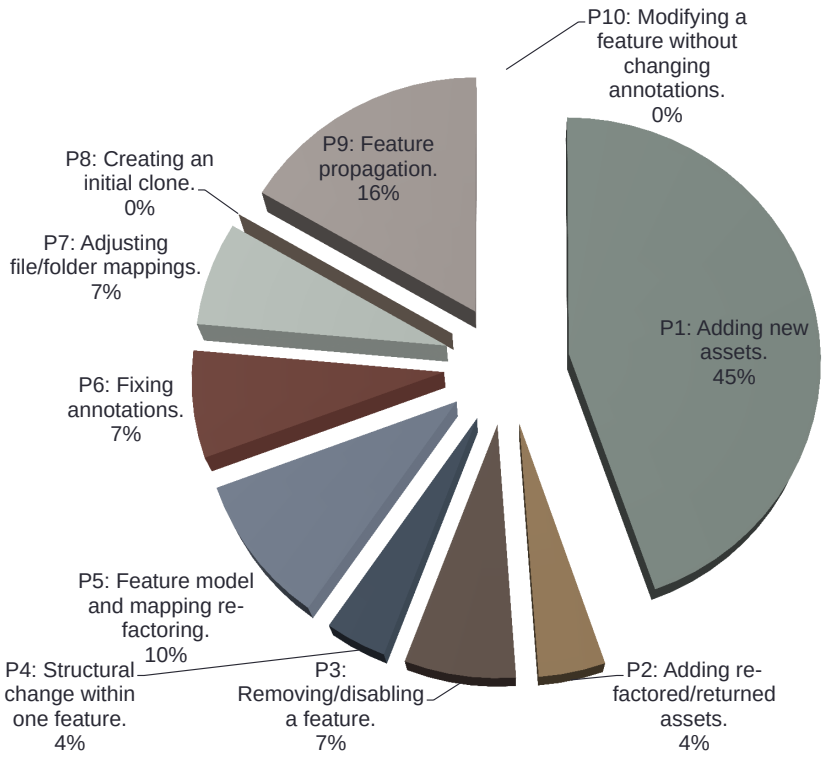

Figure 6.2: Cost of evolution patterns, based on number of markers.

In our feature annotation approach, the benefit also has several limitations. Here is a list of the limitations:

- For feature location, only the whole feature implementation can be directly located by feature annotations. If the developer needs to locate a piece of code that is not a feature, then the benefit of feature location is limited. For example, if the developer wants to propagate a small piece of code (e.g., a small utility function), which is not annotated as a feature, then the developer can not use the feature annotation to directly locate the piece of code. The developer still needs to manually search for the piece of code in the code base. If the developer knows which feature the piece of code belongs to, then the feature annotations can help narrow down the search space, but still can not help the developer to directly locate the piece of code.
- The benefit is also limited by the quality of feature annotations (e.g., accuracy).

Since the feature annotations are basically added and maintained manually without automatic verification or validation, the possibility of having low-quality annotations might be high. The quality of feature annotations can affect the benefit. For example, when a developer is propagating a feature to another project, if the quality of the feature annotations is low (e.g., some assets of the feature are not annotated), then the developer has to manually find those assets that are not annotated, which limits the benefit the developer can get. Inaccuracy of annotations might also make developers not trust the annotations, and, thus, unwilling to use them.

In our study, we investigated feature propagation in detail in order to understand the benefit in reuse cases. We investigated all the cases that a feature was cloned or moved to another project, and analyzed the cost and benefit associated with those cases. We collected the feature propagation cases by manually looking through the development history. During the process, when a feature was added at a certain snapshot C (a commit in Git) in a certain project, the researcher manually looked at the snapshots (at the same time point in history) of other projects to see if the feature existed in them. If the feature existed in other projects at the same time point, then the feature was identified as a propagated feature, and the whole activity was identified as a case of feature propagation. Some cases of feature propagation were identified based on records in commit messages and software documents.

Sometimes, when a feature F is propagated to another project, all the sub-features of F are also propagated as a side-effect of propagating F. In such a case, the developer does not need to care about the sub-features, so it only counts as one case, in which feature F is propagated.

The total number of cases we collected is 55. In each case, one feature was cloned or moved to another project by the original developer. Several features might be propagated together in one original commit, which counts as multiple cases. A certain feature might be propagated to different projects at different moments, which also counts as multiple cases. The cases we collected cover all the feature cloning and feature moving that happened before 0.3.5 release of the subjects, which are shown in Figure 4.1. For each case, we collected the following data:

- The evolution history of the feature declaration and annotations of the propagated feature (before the time when the feature was propagated). This includes all the annotation changes made to the feature declaration and annotations of the propagated feature before it was propagated.

- The total evolution cost of the feature before it was propagated ($C_{evo}(f_i)$, which is in Formula 6.3).
- The benefit of having the feature annotations when the feature was propagated. We counted the number of annotation markers that were used to provide feature location in the feature propagation cases. Annotations inside propagated assets were not counted.

After we collected all the data, we computed several statistics. Table 6.4 shows the statistics of 55 feature propagation cases.

Based on the data of feature propagation cases, we answer the last two research questions:

RQ3: *What percentage of the invested annotation recording and editing cost saved feature location cost during reuse cases?*
We sum up the benefit of all the 55 propagation cases. Overall, 121 annotations markers (99 if not counting multiple uses of some feature annotations) were involved in propagations. Thus, 18% of the overall annotation recording (C_{record}) and editing (C_{edit}) costs in the end saved the lazy feature location costs that would be needed to perform the propagations.

RQ4: *What percentage of feature location cost during reuse could be avoided?*
In our simulation, annotations were surprisingly beneficial for the propagations. For only two features annotations were missing and had to be added—10 and 4 annotations, respectively. We did not observe any inaccurate annotation in the feature propagation cases. Given that 135 annotation markers were involved (including the fixed ones), in total 90% of feature location costs were saved, while such cost was still required for 10% of the propagated markers.

The granularity of the propagated features is also interesting. In 43 of the 55 cases, the propagated feature only has file and/or folder annotations, which means that in those cases the propagated features consist of only files and/or folders, and no code fragments. This indicates that the granularity of features used in feature propagation cases are mainly on file and folder level. Most of the fragment annotations recorded in the simulation study did not provide any benefit for reuse cases.

Patterns	Annotation change types involved	Benefits involved
P1: Adding new assets.	Adding/removing feature declaration. Adding annotations together with developing assets.	
P2: Adding re-factored/ returned assets.	Adding/removing feature declaration. Adding annotations together with developing assets.	
P3: Removing/disabling a feature.	Adding/removing feature declaration. File/folder annotation adjusting.	Feature location.
P4: Structural change within one feature.	Adding annotations together with developing assets. Asset merging. Feature modularization.	Feature location (for feature modularization).
P5: Feature model and mapping re-factoring.	Adding/removing feature declaration. Feature renaming. Feature moving. Recovering mapping.	
P6: Fixing annotations.	Simple bug fixing. Recovering mapping.	
P7: Adjusting file/folder mappings.	File/folder annotation adjusting.	
P8: Creating an initial clone.		Feature model.
P9: Feature propagation.	Adding/removing feature declaration. Adding annotations together with developing assets.	Feature location.
P10: Modifying a feature without changing annotations.		Feature location.

Table 6.2: Summary of the annotation change types and benefits of evolution patterns.

Patterns	$C_{pattern}(p_i)$
P1: Adding new assets.	290
P2: Adding re-factored/returned assets.	27
P3: Removing/disabling a feature.	45
P4: Structural change within one feature.	25
P5: Feature model and mapping re-factoring.	63
P6: Fixing annotations.	46
P7: Adjusting file/folder mappings.	45
P8: Creating an initial clone.	0
P9: Feature propagation.	115
P10: Modifying a feature without changing annotations.	0
(TOTAL)	656

Table 6.3: Cost of evolution patterns, based on number of markers.

	Cost ($C_{evo}(f_i)$)	Benefit
Maximum	22	8
Minimum	0	0
Average	6.145	2.2
Mode	3	2
Median	5	2

Table 6.4: Statistics of feature propagation cases.

Chapter 7

Discussion

In this chapter, we present some discussions about the findings in the study.

7.1 Configuration Semantics of Shared Code

In the annotation system, an asset can be mapped to more than one feature. In such case, the asset is shared by the features it maps to. If the annotations are only used for traceability, then we do not need to care about the configuration semantics of shared assets. However, if the annotations are used for variability, then we need to specify the configuration semantics of the shared assets.

For an asset (A) shared by two features (F1 and F2), there are two kinds of semantics: AND and OR. The semantics AND means that A is present at a variant if and only if both features F1 and F2 are present at the variant. The semantics OR means that A is present at a variant if and only if at least one of the features (F1, F2) is present at the variant. An example of the AND semantics is a piece of code that handles the interaction of two features, so the code piece is present only if both features are present. An example of the OR semantics is a utility method required by both the two features, so as long as one of the features is present, the utility method must be present.

In evolution patterns "P3: Removing/disabling a feature" and "P9: Feature propagation," it might be useful to distinguish the two kinds of semantics of shared code. For example, assume features F1 and F2 share an asset (A), and the developer wants to remove/disable F1 or propagate F1 to another project. In the case of removing/disabling F1, if the semantics of A is AND, then A should be removed/disabled because F1 and F2

will not be both present together. Otherwise, if the semantics is OR, then A should stay because it is still required by F2. In the case of propagating F1 (only F1 is propagated) to another project, if the semantics is OR, then A is propagated because it is required by A, otherwise it is not propagated.

7.2 Potential Benefit

Besides the benefits of feature annotations mentioned in Section 6.3, we also identified several potential benefits:

1. **Providing statistics of features.**
   Given the locations of features provided by the feature annotations, several kinds of statistics can be automatically computed, such as number of lines of feature code and scattering degree of a feature. In addition to the statistics of a snapshot of a feature, statistics of the feature evolution history can also be computed, such as which features change more often and which features have more bugs.

2. **Providing meta-data for future tools to analyze features and automate development tasks.**
   With the information provided by feature annotations, future tools can perform analysis of features and potentially automate some development tasks. For example, in future tools a function called "feature diff" could be implemented, which compares two different versions of a feature and produces a diff. Different versions of a feature could be in two different commits of one project, or in two different projects (caused by feature cloning). Besides analyzing features, future tools can also automate some development tasks. For example, feature propagation could be potentially automated.

3. **Program comprehension.**
   The feature model and annotations could potentially facilitate program comprehension for both original developers and novices. The feature model can provide information about the structure of the functionalities implemented in a certain project, and feature annotations can help the developer navigate while looking at source codes. They provide meta-information about the source code, which is similar with code comments.

There are several existing studies on the benefit of using the feature concept. In [31], the authors investigate an approach of using feature modeling to facilitate program comprehension and software architecture recovery. In [34], the author presents an approach of using feature modeling to link requirements to solutions.

Chapter 8

Threats to Validity

8.1 Internal Validity

Threats to internal validity refers to uncontrolled factors that may influence the analytical and empirical results found in the simulation case study.

- **The researcher.**
  In the simulation study, the researcher who performed the simulation process is not one of the original developers of the subject projects, so the researcher's understanding of the original development (including meanings of the source code, activities happened during the development and rationale behind those activities) is limited. The information resources used by the developer includes commit messages, websites and the issue tracker of the subject projects, in which the information may be potentially inaccurate. One of the original developers was sometimes consulted during the simulation study, but that is also limited due to the large amount of work in this study, and the original developer is not highly involved in this study.

  The limited understanding of the researcher might cause mistakes in the simulation study (e.g., some code mapped to a certain feature by the researcher actually does not belong to the feature). In that case, the amount of cost and benefit measured in the study might be influenced by the potential mistakes.

- **The simulation study.**
  In the simulation study, we simulated the approach of adding and maintaining feature model and annotations during developing. The way of simulating is by re-applying

the commits in the development history, and evolving the feature model and annotations during the re-applying process. This is not an exact simulation, since it does not simulate what the original developer actually did in the development history. Ideally, the simulation should be a re-enactment of the original development, but actually it is not.

8.2 External Validity

Threats to external validity refers to factors that may affect the generalizability of the study results outside the study.

- **The subject projects.**
  In this study, the subject projects we chose were developed in the same lab with the researchers. The reason we chose them is that these projects are developed by the clone-and-own approach, and we can have access to all the resources of it and also the original developers. These projects are developed in an academia environment, and the size of the subjects are small compared to large industry projects, which limits the generalizability to industry environments.

- **The original developers.**
  Both the researcher who performed the simulation study and the original developers are in the same lab, in which the main research topic is about Software Product Line and Feature Model. The researcher and original developers have a certain level of understanding of the feature model concept and other relevant knowledge. This may harm the generalizability in environments in which developers are not familiar with the feature concept and other relevant knowledge.

8.3 Construct Validity

Construct validity is the degree to which the metrics we used to measure cost and benefit reflects the actual cost and benefit. In the study, the metrics we used to measure cost is basically number of annotation markers added/deleted/modified. This is similar with using number of lines of code to estimate development cost. The metric can be used to empirically estimate the amount of cost, but it is not an exact measure.

Chapter 9

Related Work

9.1 Evolution Patterns

The evolution of configurable systems has been investigated before. For instance, in [32], the authors analyze the co-evolution of variability model, asset mapping, and code in the Linux kernel over time. They identify a catalog of re-occurring patterns, describing the nine most-frequent ones in-depth. While their targets are systems that already have an integrated platform, some patterns in fact overlap with ours. For example, the pattern "Add visible optional modular feature" in [32] corresponds to our evolution pattern "P1.1: Adding a new feature and new assets." The patterns in [32] are identified according to the characteristics of the co-evolution of variability models and artifacts, while our patterns are mainly focused on different kinds of the cost and/or benefit of feature annotations.

In [30], the authors present several safe evolution templates that developer can use for evolving software product line. Some of the patterns in that work are also found in our study, such as splitting a software asset.

9.2 Concern Mapping

A concern is any kind of conceptual unit that stakeholders of a software projects may be concerned with. Examples of concerns include features, functionality, and non-functional requirements. The mapping between concerns and software assets are essential for performing tasks related to concerns, such as modifying a concern. The feature annotation approach proposed in this thesis is also a kind of concern mapping approach.

There exist several studies on mapping concerns to software assets. In [35], the authors propose an approach in which the implementation of concerns is documented in software assets using a technique called concern graph, which is a kind of abstract model that describe which parts of software assets are relevant with different concerns. The concern graph is continuously maintained during the software development process, which is similar to our feature annotation approach. They also developed a tool called FEAT (feature exploration and analysis tool) to help developers build concern graphs, and performed several case studies to evaluate the approach they propose. The results of those case studies show that the concern graph is robust and cost-effective to create and use.

Besides the tool called FEAT in [35], there are several other tools or techniques that can be used to map features to software assets, including CIDE[26, 27, 25], FeatureMapper[20] and Spotlight[14]. CIDE is an Eclipse plug-in which can be used to annotate code with the features it implements, to generate variants and perform variability-aware type checking. In CIDE, language structures (such as classes, methods) can be mapped to features via different colors, and the mapping information is stored in separate .color files. It also supports views of features in source code. FeatureMapper is a tool that can map features to solution artefacts expressed in EMF/Ecore-based languages[38]. Spotlight is an editor for *software plans*[13], and can also be used for mapping concerns to assets. Empirical studies were also conducted to evaluate those tools or approaches. In [37], the authors compared the pre-processor approach vs. physical separation of features (feature modules) in a controlled experiment.

Most of the existing approaches we found share a common characteristic: the feature traceability records are maintained external, not embedded in assets. Most also impose specific tools, such as an IDE. In this thesis, the annotation system we designed embeds fragment annotations inside assets, which makes them harder to break compared to external traceability records.

9.3 Feature Location

There are many existing techniques that target the feature location problem. In [36], the authors performed a survey of existing feature location techniques. The result shows that basically all of the existing feature location techniques have very low precision and recall. This means that automated approach of acquiring feature location information is nearly impossible.

Manual feature location is studied by Wang et al. [40]. The study consists of three

experiments in which developers were given unfamiliar software systems and asked to complete six feature location tasks. Based on the results of the study, they propose a conceptual framework for understanding manual feature location process, which consists of a collection of phases, patterns and actions. The study also shows that manual feature location is a human-intensive and knowledge-intensive task which involves significant difficulty.

Finally, we found no work on recording feature traceability manually and continuously, besides an introduction into traceability [17], which classifies traceability maintenance into "continuous" and "on-demand".

9.4 Cost-benefit Analysis

To the best of our knowledge, there is no sound theory of how to model the cost or benefit of software development approaches. Existing approaches usually estimate the cost or benefit by regression method, such as COCOMO[12]. Other methods of estimating software development cost also include Function Point method[39].

In [29], the authors present a pragmatic economic model to perform cost-benefit analysis of the adoption of software reference architectures. The model is based on value-based metrics and other economics-driven models. Commonly, costs are identified, but constants often remain as parameters and they have to be tailored to a specific project or domain context. We express our costs similarly.

We are not aware of a comprehensive cost/benefit model of traceability, although various works propose such. In [22], the authors present an overview of traceability cost and benefit. In [21], the authors introduce a model of traceability cost and benefit, and show that it is useful to estimate the return on investment of tracing approaches. In [16], the authors propose a value-based approach that can be used to understand the cost-benefit trade-off in traceability generation.

9.5 Annotative Variability

Annotative variability is a technique which uses annotations in source code to implement variability, such as using C preprocessor directives to implement compile-time variability. There exist several empirical studies about how annotative variability is adopted in practice. For example, in [28], the authors analyze forty open-source software projects that are

written in C and use C preprocessor directives to implement variability. They explore several characteristics of the annotative variability in those projects, such as the complexity and granularity of variability annotations. They also introduce several metrics to measure different aspects of the variability, such as scattering degree.

9.6 Attention Investment

In our proposed cost model (see Section 6.2), we use the concept annotation change to describe activities that a developer needs to pay attention to. Performing an annotation change to the feature model or annotations is essentially a kind of *attention investment*, which means that the developer invests the attention cost related to evolving the feature model or annotations in the hope that the feature model and annotations will bring some benefits in the future. There exist several studies that apply the attention investment concept. In [10], the authors propose an approach of using the attention investment concept to analyze the cognitive dimensions of notations[18, 19], which are design principles for notations or programming languages. In [9, 8, 11], the authors use the attention investment concept to investigate the aspects of both professional programming and end-user[1] programming, and also propose attention investment models.

[1]End users refer to people who are not professional software developers.

Chapter 10

Conclusion

10.1 Summary of Findings

In this thesis, we presented a case study that aimed at analyzing the cost and benefit of adopting the approach of maintaining a feature model and feature traceability information along normal development process. The study is performed by simulating the approach on a series of subject projects developed by clone-and-own. We classified different kinds of software evolution happened in the simulation process into evolution patterns, and analysed the cost and benefit associated with those patterns. According to the characteristics of the cost and benefit observed in the evolution patterns, we presented a cost model and used it to estimate the cost of adopting the approach. We also used some metric to estimate the benefit.

The cost and benefit data shows that 18% of the feature recording and editing cost saved 90% of feature location cost needed for feature reuse tasks. Feature maintenance tasks (P10: Modifying a feature without changing annotations) constitute the majority of developers' work, which potentially also benefit a lot from the feature annotations. Based on the data, we conclude that, under our assumptions, the cost of creating and maintaining a feature model and annotations with the proposed approach is negligible compared to the development cost. Also, the benefit of the annotations can justify the investment if certain amount of reuse and consistency management is required.

10.2 Future Work

For future work, we need to do an action research, in which the proposed feature annotation approach is adopted in real world. By doing action research, we can get more insights about the actual causes of cost and benefit, challenges of the approach and tool support opportunities.

In this thesis, a cost model is proposed for measuring the extra cost of adopting the feature annotation approach, but it is not validated. For future work, we can do a controlled experiment. In the experiment, two groups can develop the same software project, and one adopts the feature annotation approach while the other does not. By comparing the cost of the two groups, we can measure and analyze the extra cost of the feature annotation approach, and also investigate whether the feature annotation approach will influence the development behavior (e.g., whether the enforcement of recoding feature annotations will make the developer tend to develop features that are less scattered).

In future work, we also need to investigate the tool support opportunities for the feature annotation approach, such as automatically proposing where to put annotation markers, and also implement the tools.

References

[1] http://www.clafer.org/.

[2] http://en.wikipedia.org/wiki/Javadoc.

[3] http://gsd.uwaterloo.ca/claferwebtools.

[4] http://t3-necsis.cs.uwaterloo.ca:8091/ClaferToolsPLE/Intro.

[5] 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 *ICSE Companion*, pages 532–535, 2014.

[6] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Object Technology*, 8(5):49–84, 2009.

[7] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *SLE*, 2010.

[8] Alan Blackwell and Margaret Burnett. Applying attention investment to end-user programming. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 28–30. IEEE, 2002.

[9] Alan F Blackwell. First steps in programming: A rationale for attention investment models. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 2–10. IEEE, 2002.

[10] Alan F Blackwell and Thomas RG Green. Investment of attention as an analytic approach to cognitive dimensions. In *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*, pages 24–35, 1999.

[11] Alan F Blackwell, Jennifer A Rode, and Eleanor F Toye. How do we program the home? gender, attention investment, and the psychology of programming at home. *International Journal of Human-Computer Studies*, 67(4):324–341, 2009.

[12] Barry W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, Ray Madachy, and Bert Steece. *Software Cost Estimation with Cocomo II with Cdrom*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[13] David Coppit and Benjamin Cox. Software plans for separation of concerns. In *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, Lancaster, UK*, volume 22, 2004.

[14] David Coppit, Robert R. Painter, and Meghan Revelle. Spotlight: A prototype tool for software plans. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 754–757, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *CSMR*, 2013.

[16] Alexander Egyed, Stefan Biffl, Matthias Heindl, and Paul Grünbacher. A value-based approach for understanding cost-benefit trade-offs during automated software traceability. In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '05, pages 2–7, New York, NY, USA, 2005. ACM.

[17] Orlena Gotel, Jane Cleland-Huang, JaneHuffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grnbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mder. Traceability fundamentals. In Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, editors, *Software and Systems Traceability*, pages 3–22. Springer London, 2012.

[18] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[19] Thomas RG Green. Instructions and descriptions: some cognitive aspects of programming and similar activities. In *Proceedings of the working conference on Advanced visual interfaces*, pages 21–28. ACM, 2000.

[20] Florian Heidenreich, Jan Kopcsek, and Christian Wende. Featuremapper: Mapping features to models. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 943–944, New York, NY, USA, 2008. ACM.

[21] Matthias Heindl and Stefan Biffl. Modeling of requirements tracing. In Bertrand Meyer, JerzyR. Nawrocki, and Bartosz Walter, editors, *Balancing Agility and Formalism in Software Engineering*, volume 5082 of *Lecture Notes in Computer Science*, pages 267–278. Springer Berlin Heidelberg, 2008.

[22] Claire Ingram and Steve Riddle. Cost-benefits of traceability. In Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, editors, *Software and Systems Traceability*, pages 23–42. Springer London, 2012.

[23] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep., SEI, CMU, 1990.

[24] C. Kapser and M.W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE*, 2006.

[25] Christian Kästner and Sven Apel. Virtual separation of concerns-a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[26] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *ICSE*, 2008.

[27] Christian Kästner, Salvador Trujillo, and Sven Apel. Visualizing software product line variabilities in source code. In *SPLC (2)*, pages 303–312, 2008.

[28] J. Liebig, S. Apel, C. Lengauer, C. Kastner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 105–114, May 2010.

[29] Silverio Martínez-Fernández, Claudia Ayala, and Xavier Franch. A reuse-based economic model for software reference architectures. *Repport: ESSI-TR-12-6, Departament dEnginyeria de Serveis i Sistemes dInformació, Barcelona, Spain*, 2012.

[30] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulezsa, and Paulo Borba. Investigating the Safe Evolution of Software Product Lines. In *Proceedings of the 10th International Conference on Generative Programming and Component Engineering*, pages 33–42. ACM, 2011.

[31] I Pashov and Matthias Riebisch. Using feature modeling for program comprehension and software architecture recovery. In *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*, pages 406–417, May 2004.

[32] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrezj Wasowski, and Paulo Borba. Coevolution of variability models and related artifacts: A case study from the linux kernel. In *17th International Software Product Line Conference*, 2013.

[33] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[34] Matthias Riebisch. Supporting evolutionary development by feature models and traceability links. In *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*, pages 370–377, May 2004.

[35] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1), February 2007.

[36] Julia Rubin and Marsha Chechik. A survey of feature location techniques. In *Domain Engineering*, pages 29–58. Springer, 2013.

[37] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. Comparing program comprehension of physically and virtually separated concerns. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, pages 17–24, New York, NY, USA, 2012. ACM.

[38] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.

[39] C.R. Symons. Function point analysis: difficulties and improvements. *Software Engineering, IEEE Transactions on*, 14(1):2–11, Jan 1988.

[40] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. How developers perform feature location tasks: a human-centric and process-oriented exploratory study. *Journal of Software: Evolution and Process*, 25(11):1193–1224, 2013.