Maintaining Feature Traceability with Embedded Annotations

Wenbin Ji, Thorsten Berger, Michał Antkiewicz, Krzysztof Czarnecki
University of Waterloo, Canada
{w6ji,tberger,mantkiew,kczarnec}@gsd.uwaterloo.ca

ABSTRACT

Features are commonly used to describe functional and non-functional aspects of software. To effectively evolve and reuse features, their location in software assets has to be known. However, locating features is often difficult given their crosscutting nature. Once implemented, the knowledge about a feature's location quickly deteriorates, requiring expensive recovering of these locations. Manually recording and maintaining traceability information is generally considered expensive and error-prone. In this paper, we argue to the contrary and hypothesize that such information can be effectively embedded into software assets, and that arising costs will be amortized by the benefits of this information later during development. We test this hypothesis in a study where we simulate the development of a product line of cloned/forked projects using a lightweight code annotation approach. We identify annotation evolution patterns and measure the cost and benefit of these annotations. Our results show that not only the cost of adding annotations, but also that of maintaining them is small compared to the actual development cost. Embedding the annotations into assets significantly reduced the maintenance cost because they naturally co-evolve with the assets. Our results also show that a majority of these annotations provides a benefit for feature-related code maintenance tasks, such as feature propagation and migrating clones into a platform.

1. INTRODUCTION

The notion of *feature* is commonly used in software development to refer to an increment of functionality [1]. Features are often only used informally; however, many approaches emphasize the explicit and formal use of features, such as, feature-oriented software development (FOSD) [2], feature-driven development (FDD) [3], software product line engineering (SPLE) [4, 5, 6], and virtual platform [7]. In these approaches, features are identified, declared, and used as the primary unit of maintenance and reuse as well as a unit of characterization and comparison between software variants.

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3613-0/15/07...$15.00
DOI: http://dx.doi.org/10.1145/2791060.2791107

In SPLE and FOSD, all possible variants of the software are also integrated in a platform, sharing assets among variants by realizing variability. Features are described in a feature model [8], and the mapping between features and assets is completely recorded, such as by variability annotations (e.g., preprocessor directives). The models are used for product variant configuration and derivation. However, creating such a platform from scratch is unrealistic for many organizations. Instead, they often start with cloning [9, 10, 7, 11] and use it as their preferred reuse strategy as long as they can maintain the clones. When facing scalability problems, they need to migrate all the clones into an integrated platform.

Before the migration, performing tasks on features, such as fixing bugs, modifying, refactoring, disabling, and reusing, requires knowing the *location* of the features in the code, which is often difficult given their crosscutting nature [12]. In fact, feature location is considered one of the most common activities of developers [13, 14, 15, 16]. Later, during the migration, all features and their locations in the code need to be known, in order to re-integrate clones into the platform, to establish a mapping between features and the shared assets, and to introduce variability for optional features.

Thus, organizations that rely on features are faced with the difficult question: *"How to effectively maintain traceability between the features and the corresponding software assets?"* Two strategies are possible: either organizations record feature traceability information during the development of the features (the *eager* strategy), or they retroactively recover such information when needed (the *lazy* strategy). In the eager strategy, developers record the location of features when they perform tasks related to these features or shortly thereafter, when the knowledge is still fresh in their memory. In the lazy strategy, if memory has deteriorated, developers recover the location of features by reading the code or by applying semi-automated feature location techniques.

When choosing the eager strategy, organizations face another question: *"Where to store the feature traceability information?"* Again, two strategies are possible: either the information is stored externally to the corresponding software assets (the *external storage* strategy), or internally as part of the assets (the *internal storage* strategy). The external storage strategy requires a universal way of addressing locations inside the heterogeneous assets and updating the locations when the assets evolve. It also relies on tools (e.g., FEAT [17]) to reduce the burden of constantly maintaining feature locations during evolution. The internal storage strategy requires a feature annotation system for embedding the traceability information into assets.

While significant research exists about the lazy strategy relying on feature location recovery [14], all works lament the lack of precision and the high effort of the task, even if using a semi-automated technique [16] (e.g., FLAT3 [18]). Eagerly recording traceability and storing it externally is also considered expensive, requiring significant effort and heavy-weight tooling to co-evolve the external traceability information with the changing assets [19]. However, no study has been published on the cost and benefit of applying the *eager* and *internal* storage strategies.

We present an empirical investigation of applying an eager annotation approach. We retroactively embed feature annotations into an existing clone-based product line comprising three projects, whose parts are subsequently re-integrated (i.e., merge-refactored) into a shared platform. We simulate the actual development as if annotations were maintained originally. We analyze the evolution of features, their annotations, and qualitatively and quantitatively assess the costs in relation to the benefits of our approach. We compare the cost savings of eager annotations with lazy feature location.

In summary, our contributions comprise:

- Feature evolution patterns together with their costs and benefits (Sec. 3) derived from a simulation study (Sec. 2).
- Quantitative empirical data on these patterns and their actual costs and benefits based on the study (Sec. 4).
- An online appendix [20] with additional statistics and source code repositories of four projects with feature annotations and evolution history, to replicate our results and as a benchmark for future traceability tools.

As we will show, introducing and tracing features early has in fact the potential to support organizations developing a portfolio of systems using clone-based projects. Using our results, organizations can decide whether the investment into an eager feature annotation strategy will pay off for evolving, maintaining, and reusing features across individual software projects that are not integrated in a platform, or for supporting a later migration of these projects into a platform. For researchers, our results show what kinds of activities (illustrated by the patterns) could be supported by tools and processes when using our approach. Our results also create awareness of a surprisingly cheap way of maintaining traceability using embedded annotations, while the majority of existing approaches tries to store this information externally.

2. SIMULATION STUDY

While it is generally accepted that manually recording and maintaining traceability information is costly and error-prone [19], we argue to the contrary.

2.1 Rationale

We hypothesize that the cost of recording and maintaining traceability information embedded into assets is low, since:
H1: the knowledge of the feature and its location is fresh in the developer's memory during implementation, and therefore the cost of feature location is close to zero,
H2: adding a feature annotation is trivial and cheap,
H3: the embedded traceability links naturally co-evolve as the assets evolve and they are much less brittle as compared to the externally stored ones, thus, largely eliminating the maintenance cost, and
H4: the cost of storing traceability information internally is amortized over multiple uses of such information, whereas, in the lazy strategy, the information must be recovered every time it is needed.

We also hypothesize that the value of the embedded traceability information increases over time, since:
H5: the original developers' memory deteriorates with time or when they leave the team,
H6: the precision of such manually recorded information is much higher as compared to traceability information recovered by automated feature location approaches or manually in the future, and
H7: the developers benefit during every task that requires knowledge of feature location, since it is readily available.

In this paper, we take hypotheses **H1**, **H2**, **H4**, **H5**, **H6** as assumptions since they are, in our opinion, reasonable to make given the design of our annotation approach (and testing these hypotheses would require separate studies). We address **H3** and **H7**.

2.2 Research Questions

To gain insights into the applicability of the eager and internal storage strategies as well as the costs and benefits of embedded feature annotations, we conducted a *simulation case study* in which we retroactively applied an embedded feature annotation approach by simulating annotation through an actual development history of a set of software projects. The simulation allowed us to investigate the application process of the annotations and identify the evolution patterns necessary for eager annotation and annotation maintenance. These patterns describe the observed cost and benefit cases of using the approach. Finally, the simulation provided data on the occurrence frequency of the patterns, and contributed supporting evidence towards hypotheses **H3** and **H7**.

The return on investment in recording feature annotations occurs when features are maintained (extended, fixed, removed) and reused (cloned, propagated into platform) later on. When only a subset will be maintained or reused, the investment may be higher than the pay off. Thus, we formulate the following research questions:
RQ1: *What are the annotation recording and editing costs?* The former arise from adding assets, and the latter from evolving assets. We also investigate their ratio to understand the relative evolution costs (to maintain annotations) in our case study. To measure these costs, we rely on a simple metric (number of annotation lines).
RQ2: *What percentage of annotation recordings and edits required additional feature location effort?* We assume that the effort of feature location is zero when the annotations are recorded immediately; however, sometimes the recording can be delayed due to (i) annotation mistakes, (ii) lower eagerness, or (iii) incomplete location knowledge.
RQ3: *What percentage of the invested annotation recording and editing costs saved feature location costs during reuse cases?* We investigate how many of the recorded annotations were beneficial for reuse cases.
RQ4: *What percentage of feature location costs 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 simulated the application of an embedded feature annotation approach—as if it had been used originally during the development of our subject projects. We analyze the behavior of the human simulator, qualitatively

Table 1: Summary of the subject projects

release 01/2014	visualizer	configurator	IDE	platform
started	10/2012	03/2013	10/2013	12/2013
developers	3	3	2	1
features	67	49	38	51
lines of text	5546	2011	1623	5614
lines of code (JavaScript)	1974	908	509	3774
before platform	visualizer	configurator	IDE	-
lines of text	8283	3708	3960	-
lines of code (JavaScript)	3746	2554	2228	-

and quantitatively reflected in the annotations, and investigate whether and to what extent the introduced annotations were beneficial for feature maintenance, evolution, and reuse. We now introduce our subjects and describe our detailed methodology.

2.3 Subject Systems

We study a clone-based product line of web-based tools called Clafer Web Tools [21], which support various use cases of the Clafer modeling language. We selected them, because we were familiar with them and had access to the original developers allowing us to ask about design and implementation details, and their rationale behind some of our observations. Table 1 summarizes the projects, comprising:

- ClaferMooVisualizer ("visualizer") accepts a Clafer model with optimization objectives, runs multi-objective optimization, visualizes the resulting set of optimal configurations, and offers the users an interactive interface for the exploration of the results and comparing solutions for tradeoff analysis.
- ClaferConfigurator ("configurator") accepts a Clafer model, runs a chosen instance generator, and offers the users an interactive interface for the exploration of the results and comparing solutions.
- ClaferIDE ("IDE") offers basic editing, compilation, and instantiation services for Clafer models.

The development of the three projects relied on a clone-and-own approach; later, large parts were migrated into a platform project. Fig. 1 summarizes the development history of these four projects (each represented by a horizontal line). The visualizer was developed first; later, major parts were cloned (propagated) into the configurator. The IDE was developed similarly: most of its features came from the previous two projects. Feature propagation during a migration process later led to a platform that contains common features and shared assets of the projects, used as a shared framework for all of them. Note that propagations in both directions occurred between the projects (indicated by double-arrows).

2.4 Methodology

We simulated the development history of Clafer Web Tools from the very beginning until its 0.3.5 release from January 2014 (cf. Fig. 1), comprising a total of 779 commits and 1.5 years of development. The first author (henceforth called "simulator") inspected commits and annotated code. The simulator was intentionally very free in deciding what and where to annotate, based on his understanding of the codebase and the change history. In case of doubt (e.g., about the rationale behind changes), he was encouraged to consult the original developers.

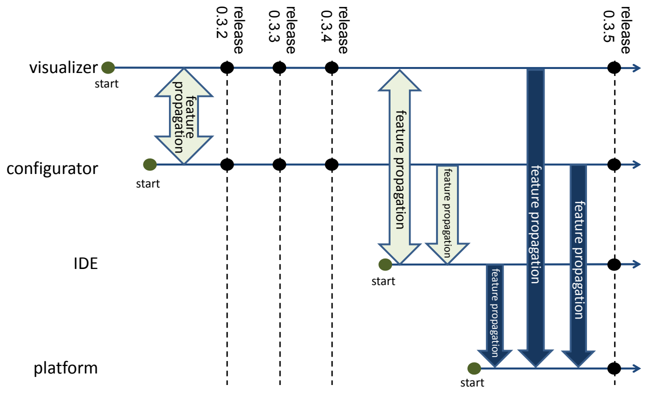


Figure 1: Excerpt of Clafer Web Tools history

2.4.1 Identifying Features

Identifying feature-related development tasks requires identifying features. Therefore, the simulator manually looks at the code or other assets that are added, deleted, or modified in original commits. He decomposes them into features relying on the following information: first, the **project wikis** contain feature models written by one of the original developers between the 0.3.4 and 0.3.5 release, before our study. We use them as an initial understanding of features, as they represent features that are in the original developer's mind. Second, we observe that features were often referred to in **commit messages**, such as "Polling implemented on both sides." Third, although labor-intensive, the simulator also investigated **code and commit diffs** based on his understanding. Candidates were, for instance, adding a new class or method if they looked like a unit of later reuse, or contained a potential variation point. Fourth, project **issue trackers** and **original developers** were consulted.

2.4.2 Simulating Annotations

For each project, the simulator created a new "simulation" branch from the first project commit and added a feature model with a root feature (project name). Fig. 2 illustrates this process. Black dots represent commits, and horizontal lines between them represent the branches and the timeline (e.g., commit `O3` is the parent commit of `O4` because it was done before it; `O4` and `E1` are both parent commits of `M1`). Commit `O1` represents the first commit on the original branch from which the simulation branch was created. Commit `E1` represents the addition of the initial feature model.

The simulator then gradually explored and merged consecutive periods of original commits into the simulation branch, and after each merge, evolved the feature model and the annotations, without making changes to the code itself. In case he later observed that he made an annotation mistake (e.g., missing an annotation), he fixed it in the simulation branch at the time of the observation (not retroactively).

In the example in Fig. 2, the simulator merged three original commits `O2`, `O3`, and `O4` into the simulation branch (`M1`), he then evolved the feature model and added annotations according to changes made by these merged commits. Then, the simulation was committed (`E2`). This procedure was repeated until all commits until release 0.3.5 (January 2014) were merged into the simulation branch.

Based on the simulator's understanding of the original development, he sought for development tasks related to

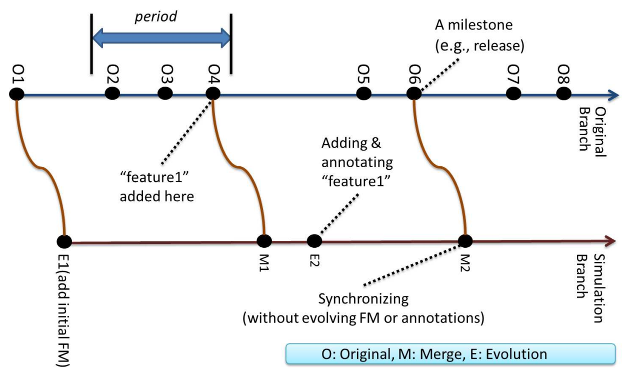


Figure 2: Simulation process

features, such as the implementation of a feature, the evolution of a feature, or feature change propagation. An original commit[1] was merged into the simulation branch when:

- a feature-related development task occurred (identified by the simulator's judgement based on commit messages, issue tracker, and asking the original developers), which requires to evolve feature model or annotations;
- an "interesting" evolution of software assets occurred (e.g., refactoring of the annotated code) without the need to evolve the model or the annotations;
- a project milestone, such as a release, was reached;
- a significant evolution happened and the simulator saw the need to synchronize the simulation branch—to limit the divergence between both branches and to obtain a constant commit granularity.

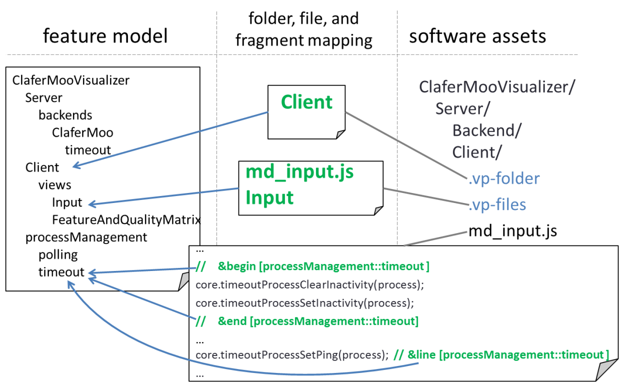
This strategy aims to abstract over the original commits, which often do not correlate with actual development tasks. Instead, we decompose the history into periods that strive to be correlated with development tasks related to features. During the process, the simulator explored the application of annotations. The result was a set of identified evolution patterns which represent re-occurring (at least three times) observations. The patterns were identified by manually investigating and classifying all the software development activities that happened in the simulated development history.

The simulation resulted in 210 periods (i.e., merges to the simulation branch), each comprising between one and 18 original commits, and nine evolution patterns.

2.4.3 Embedded Feature Annotations

The simulator applied a simple embedded feature annotation approach to establish *traceability links* between the feature declarations and the corresponding assets: (i) he created a textual feature model for each project, which contains one feature declaration per line; (ii) he inserted comments to mark source code fragments (blocks of consecutive lines) and single lines with the names of the features they correspond to; and (iii) he mapped entire folders by adding a file (`.vp-folder`) containing a list of features; (iv) he mapped entire files by adding a file (`.vp-files`) listing the files and their corresponding features. In the remainder, we use *annotation marker* to refer to one line of annotation. There are always two annotation markers for a code block, one per code line, one per folder, and two per file.

[1]Git commits are snapshots; thus, all changes since the previous merge were copied into the simulation branch—but the merge commits are more coarse-grained than in the original branch.



```
ClaferMooVisualizer
  Server
    backends
      ClaferMoo
        timeout
  Client
    views
      Input
      FeatureAndQualityMatrix
  processManagement
    polling
    timeout
```

```
ClaferMooVisualizer/
  Server/
    Backend/
    Client/
      .vp-folder
      .vp-files
      md_input.js
```

```
...
//   &begin [processManagement::timeout]
core.timeoutProcessClearInactivity(process);
core.timeoutProcessSetInactivity(process);
//   &end [processManagement::timeout]
...
core.timeoutProcessSetPing(process); // &line [processManagement::timeout]
...
```

Figure 3: Example of embedded feature annotations

Fig. 3 shows an excerpt of the feature model (left) from one of our subject projects (`ClaferMooVisualizer`, see Sec. 2.3) and illustrates references to some of the features (right). In the model, all features implemented in the project are declared (by adding the feature name in a new line) and organized in a hierarchy (using indentation). Fig. 3 (middle, lower half) shows fragment annotations. In the first one, a JavaScript code fragment is mapped to the feature `timeout` by two annotation markers in comments. In the second one, a single line is mapped to the same feature. The contents of the files `.vp-folder` and `.vp-files` are also shown (middle).

3. EVOLUTION PATTERNS

In our study, we identified the following nine evolution patterns that represent re-occurring observations related to the recording and editing of feature annotations. They provide insights into which common development activities (e.g., adding features) require adding or actively maintaining annotations; which additional annotation-related activities (e.g., fixing annotations) are necessary; and which common development activities (e.g., cloning projects) benefit from annotations and how. We explain the patterns using an example if necessary. Thereafter, we discuss the occurrence frequency of each pattern, including sub-patterns. These are summarized in Table 2 based on the number of identified development periods in which they occurred.

3.1 Adding Features

P1: Adding or extending a feature. Features were added or extended by either implementing them from scratch (**P1.1**, **P1.2**) or by using already existing assets (**P1.3**, **P1.4**). In both cases, new assets had to be annotated. The assets were then either mapped to a new feature or an existing one in the model, the former requiring adding a feature declaration. Thus, we define four sub-patterns:

- *Adding new assets as a new feature* (**P1.1**)
- *Adding new assets to an existing feature* (**P1.2**)

In our subject, one quarter (55) of the periods comprised the addition (**P1.1**) or the extension (**P1.2**) of features by developing new assets. Not surprisingly, most features were added at the very beginning of the first project (visualizer).

The other two sub-patterns, which occurred much less frequent (4% of the periods), are:

- *Adding evolved/re-introduced assets as a new feature* (**P1.3**)
- *Adding evolved/re-introduced assets to an existing feature* (**P1.4**)

In contrast to the previous two sub-patterns, assets added are not new, but are either refactored from existing assets (e.g., when scattered aspects are consolidated), or re-introduced because they were temporarily removed. The assets are then either mapped to a new feature (**P1.3**) or to an existing one in the model (**P1.4**).

In our study, an example of **P1.3** was introducing a new feature after merging a specific function (part of another feature) and one of its identical clones as an attempt to reduce duplication. The re-consolidated function was mapped to an existing feature with further assets. An example for **P1.4** was that assets of a feature unique to the visualizer project were removed during the platform migration and later added back to the project. These two sub-patterns occurred in eight periods, comprising the introduction of five new features and a total of twelve assets added. In five of the eight periods, previously removed assets were re-introduced, while in three periods, assets were evolved and added as new features.

3.2 Removing Features

P2: Removing or disabling a feature. In this pattern, assets of an existing feature are removed. Interestingly, we observed that sometimes features are not entirely removed, but temporarily disabled (by commenting out). This pattern has one main variation—whether the feature declaration is also removed, which depends on any anticipated future use.

The pattern involves removing the feature declaration and, if assets are removed, also the corresponding folder and file annotations. While all these removals could be automated, we assume that developers prefer to verify and remove assets manually, since annotations could be inaccurate. Yet, developers still benefit from the feature locations provided by the annotations.

This pattern occurred in seven periods: in six, features were removed, while in one features were temporarily disabled by commenting out. We observed three interesting cases. First, features were moved into a temporary folder and later re-introduced when consolidating common features of projects into the platform. The rationale was to quickly have a running and testable system, since these features (partly developed by another developer) needed integration effort. These disabled features were then either added to the platform or back to their original project. Second, feature removal happened because the developer accidentally thought the functionality of the feature was already covered by another one. It was added back later. Third, in one period, sub-features of cloned features (cf. **P7**) were not needed by the new project and removed.

3.3 Refactoring Features

Without extending functionality, commonly the structure of feature implementations had to be changed.

P3: Structural change within a feature. The feature implementation is re-factored in a way that requires editing feature annotations. We observed this pattern as a result of improving the structure, or as a result of extending or modifying other features in a way that had impact on the structure of the feature. We identified three sub-patterns, each of which causes extra cost, comprising adding, removing, or converting annotation markers. Since the last sub-pattern is feature- rather than asset-centric, only this one of the three benefits from existing annotations, saving feature location costs.

Table 2: Occurrence frequency of evolution patterns.

pattern	frequency[1]	sub-pattern	frequency[1]
P1: Adding or extending a feature	62	P1.1	41
		P1.2	14
		P1.3	4
		P1.4	4
P2: Removing or disabling a feature	7		
P3: Structural change within a feature	7	P3.1	4
		P3.2	2
		P3.3	2
P4: Adjusting file or folder mapping	9		
P5: Evolving the model and the annotations in isolation	16	P5.1	6
		P5.2	3
		P5.3	3
		P5.4	4
P6: Fixing an asset annotation	11	P6.1	3
		P6.2	9
P7: Cloning a project	2		
P8: Propagating a feature	14		
P9: Evolving annotated assets	210		

[1] number of periods of all 210 periods in which the (sub-) pattern occurred

- *Asset splitting* (**P3.1**), where a fragment, file, or folder is split into two non-adjacent parts. We observed only one reason: part of a fragment is re-factored out into a new fragment (e.g., to extract a method). Consequently, the annotation needs to be updated. The majority of these cases happened during the platform migration. For instance, a very core feature implemented in one file had to be split into several files.
- *Asset merging* (**P3.2**), where several assets are merged into one. We observed two occurrences: when code between two fragments of a feature was moved to another place, and when refactoring a scattered feature into one asset.
- *Feature modularization* (**P3.3**), where fragments of a feature are modularized into their own file. Fragment annotations are converted into file or folder annotations. This happened twice, both when the developers aimed at improving modularity.

P4: Adjusting file or folder mapping. File annotations needed to be updated to reflect moves, renames, and removals of files. Cost arises from modifying the respective mapping file (`.vp_files`). For example, when a file is moved to another folder, then the feature references from the `.vp_files` in the old folder need to be moved to the `.vp_files` in the new folder. Folder annotation changes (affecting `.vp_folder` files) were less frequent; in only one case a folder was moved into a newly created folder that maps to the same feature. Both cases show that files that can be modified should be annotated using fragment annotations surrounding the entire contents and only files that cannot be modified (e.g., binary files) should be annotated in `.vp_files`.

3.4 Improving Feature Representation

While all previous patterns were driven by the ordinary development, which had impact on the application of our annotation approach, we also found cases where the representation of features in the model had to be improved. Thus, these cases were driven by the need to modify features, instead of being a consequence of the ordinary development.

P5: Evolving the model and the annotations in isolation. It was often necessary to evolve the feature declarations in the model without changing the code. In some cases, this required adapting the annotations correspondingly:

- *Identifying a new feature* (**P5.1**), where a new feature declaration is added to the model and existing assets are mapped to it. In almost all cases, a feature was refined into sub-features. In one exception, a parent grouping feature was introduced after the addition of a new feature that provides an alternative to an existing one.
- *Adjusting the position of a feature* (**P5.2**), where a feature in the model is moved to another location in the feature hierarchy. Related annotations may need to be adjusted.
- *Renaming a feature* (**P5.3**) in the model, which requires modifying annotations.
- *Adding feature declaration* (**P5.4**), where the assets are added and annotated later. The early declaration aimed at indicating future implementation of the feature.

In all cases, cost arises from adding or changing feature declarations in the model. **P5.1** also imposes costs to annotate assets and costs to locate features, requiring to identify assets that should be mapped to the new feature. In **P5.2** and **P5.3**, however, updating the annotations could be easily automated. In contrast to **P2** (Removing or disabling a feature), we assume that developers would always prefer automated updates. Thus, for **P5.2** and **P5.3**, we will only measure the cost of adjusting or removing the feature declaration in the model.

3.5 Fixing Annotations

P6: Fixing an asset annotation. Sometimes, annotations were incorrect or missing, which required fixing the model or annotations. We observed syntax errors and mistakenly annotated assets (**P6.1**) and, more frequently, missing annotations (**P6.2**). Most of these bug fixes were triggered by modifications to features—specifically when commit messages indicated that. For instance, a commit message indicated that comments were added. Inspecting them led to the annotation of fragments belonging to an existing feature, which simulates fixing an annotation. However, annotation omissions primarily arose from the unfamiliarity of the simulator with the code; thus, such mistakes might be less likely in practice; yet, such fixes will involve feature location costs.

3.6 Cloning and Maintaining Consistency

P7: Cloning a project. A new project is created by cloning assets from an existing project. We observed that usually, the basic infrastructure (framework, mandatory assets, including libraries and documentation) was copied first. After that, individual or all features were propagated. Later, some undesired sub-features of those propagated features were removed or commented out. The benefits arise from having (i) an overview of implemented features in the source project captured in feature models, and (ii) the feature annotations, which provide the same benefit as in **P2** (Removing or disabling a feature).

P8: Propagating a feature. A feature (or part of it) is propagated from one project to another, based on asset cloning and manual integration.

These propagations played a major role when cloning the projects (**P7**) and when consolidating common features of the three projects into the platform. Interestingly, when feature propagations occurred after **P7**, often several features were propagated together in one commit, as well as the basic infrastructure of the project. More precisely, in eight commits, more than one feature was propagated, as opposed to ten commits with only one feature.

Table 3: Aggregated costs of patterns (number of markers)

pattern	$C_{pattern}(p_i)$	$C_{mdl}(p_i)$	$C_{annot}(p_i)$
P1: Adding or extending a feature	317 (48%)	61	256
P2: Removing or disabling a feature	45 (7%)	3	42
P3: Structural change within a feature	25 (4%)	0	25
P4: Adjusting file or folder mapping	45 (7%)	0	45
P5: Evolving the model and the annotations in isolation	63 (10%)	38	25
P6: Fixing an asset annotation	46 (7%)	0	46
P7: Cloning a project	0 (0%)	0	0
P8: Propagating a feature	115 (17%)	30	85
P9: Evolving annotated assets	0 (0%)	0	0
(TOTAL)	656 (100%)	132	524

The clear benefit is that feature location costs can be avoided. Yet, costs arise. First, while fragment and folder annotations are propagated together with the assets, feature declarations and file annotations need to be propagated separately. Second, since annotations might be inaccurate, we assume that developers prefer to check annotations and propagate a feature manually.

3.7 Evolving Assets

P9: Evolving annotated assets. Most frequently, a feature's assets are changed in a way that does not affect annotations or the model. We observed many such activities without imposing annotation costs, including bug fixing and extending or re-factoring assets. For instance, adding code within a fragment indirectly shifts annotation markers. All these activities require knowing a feature's location and, therefore, benefit from annotations. Recall that features are organized in a hierarchy, and that annotations are often nested. Thus, this pattern occurs in all development periods, since the whole project is mapped to the root feature. Yet, the benefit arises only for non-root features. It increases with the scattering degree of a feature, but is independent of the position of the feature in the feature model hierarchy.

4. COST AND BENEFIT

We now quantify costs and benefits of the embedded annotations with respect to the research questions **RQ1**–**RQ4**.

4.1 Cost

The costs of the eager approach comprise annotation recording (C_{rec}) and maintenance costs. The latter involve annotation editing (C_{ed}) and feature location costs (C_{fl}) for delayed annotations (when the need is recognized late). We now analyze C_{rec} and C_{ed} in RQ1, followed by C_{fl} in RQ2.

RQ1: *What are the annotation recording and editing costs?*
We measure recording (C_{rec}) and editing (C_{ed}) costs by resembling the lines of code (LOC) metric. We use it, as it accounts for the varying granularities of different kinds of assets and is highly correlated with development and maintenance effort [22] (cf. our discussion in Sec. 5 and 6).

For each pattern p_i, the costs arise from adding, deleting, or modifying feature declarations in the feature model ($C_{mdl}(p_i)$) and annotation markers ($C_{annot}(p_i)$). We count the number of feature declaration lines and annotation marker lines affected by the activities, so $C_{pattern}(p_i) = C_{mdl}(p_i) + C_{annot}(p_i)$.

Using this strategy, we can simply calculate annotation recording and editing costs by aggregating the costs of the patterns in which they occur: $C_{rec} = C_{pattern}(P1) = 317$ and $C_{ed} = \sum_{p_i \in \{P2,P3,P4,P5,P6,P8\}} C_{pattern}(p_i) = 339$. Thus, throughout the simulated development history, 656 lines of feature declarations or annotation markers were added, deleted, or modified. Table 3 shows the aggregated costs per pattern.

Table 3 also shows the relative cost of individual patterns. Not surprisingly, the most frequent pattern (P1: Adding or extending a feature) also imposes the highest costs, due to the number of annotation recording and editing costs.

Opposed to the 656 lines of annotations, 1,798,772 lines of text (including all kinds of text, such as source code or documents) were added, and 1,251,742 lines removed over the whole simulated development history. The latest (0.3.5) release of the four subject repositories has 547,030 lines of text in total; 14,794 of these are JavaScript code, excluding libraries (cf. Table 1).

Let us revisit our hypothesis from Sec. 2.1, **H3**: the embedded traceability links naturally co-evolve as the assets evolve and they are much less brittle as compared to the externally stored ones, thus, largely eliminating the maintenance cost.

The ratio of recording to editing costs $C_{rec}/C_{ed} = 107\%$ indicates that at least as much—but not much more—effort arises from maintaining annotations as a result of asset evolution. This supports our hypothesis **H3**, because the annotation maintenance cost does not grow linearly with the amount of evolution in the annotated assets. When storing the traceability information externally instead, adding or removing a single line in a file would shift the locations of all feature fragments that follow the line, which would require updating the corresponding traceability links.

RQ2: *What percentage of annotation recordings and edits 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 annotation lines), **P6.2** (Fixing missing annotations: 36 annotation lines), and **P8** (Propagating a feature: 14 annotation lines); in total $C_{ao} = 75$. Thus, in summary, 12% of all annotation-related activities incur feature location costs, in addition to recording and editing annotations. Recall that the former two (**P5.1** and **P6.2**) had feature location costs aiming at having complete annotations; however, in our case study, adding these annotations was not necessary for feature propagation, only those that were added in **P8**. Yet, we sum up these costs, since our approach strives for complete annotations.

4.2 Benefit

In our study, the identified patterns confirm that annotations support both maintenance (**P2**, **P9**) and reuse (**P7**, **P8**). We now look deeper into feature propagation as the most frequent kind of feature reuse, to investigate **RQ3** and **RQ4**. Feature propagation occurred both to propagate features among multiple projects and to migrate the features to an integrated platform.

To obtain detailed statistics, we need to accurately identify instances of **P8**, as opposed to the other patterns, where a more coarse-grained occurrence frequency sufficed (cf. Table 2). We identified feature propagations by inspecting the original commits in each project—primarily the code and annotations, but also the commit message and modified documentations. When a feature addition was identified, we investigated parallel commits of other projects to see if it existed there at the same time. Recall that features can be nested (which is often reflected in the feature hierarchy [23, 24]). Thus, we match propagations to the most coarse-grained feature, ignoring sub-features, which were often propagated together with their parent feature as a side-effect. In contrast, when several features were propagated together in one original commit, we treat them separately.

RQ3: *What percentage of the invested annotation recording and editing costs saved feature location costs during reuse cases?*

We identified 55 feature propagations in the simulation, comprising 26 features, whereas 15 of them were propagated multiple times. In each case, one feature was cloned or moved to another project by the original developer. 40 feature propagations involved propagating whole files; 15 feature propagations involved propagating whole directories; and only 10 feature propagations involved propagating annotated fragments. This indicates relatively coarse-grained propagations.

Overall, 121 annotations markers were involved in propagations (we do not count annotations inside propagated files or fragments). Thus, 18% of the overall annotation recording (C_{rec}) and editing (C_{ed}) costs in the end saved the lazy feature location costs that would be needed to perform the propagations.

RQ4: *What percentage of feature location costs 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 annotations were involved (including the fixed ones), in total 90% of feature location costs during reuse were saved, while such cost was still required for 10% (14) of the propagated markers.

Let us revisit our hypotheses from Sec. 2.1: **H7**: the developers benefit during every task that requires knowledge of feature location, since it is readily available. In the lazy strategy, developers need to perform feature location every time they perform maintenance or reuse of a feature. In our simulation, we only looked at feature reuse cases, and having embedded feature annotations saved 90% of the required feature location effort in feature reuse cases. This partially supports **H7**. We did not look deeper into feature maintenance cases due to the large quantity of them and the difficulty of measuring the benefit (it is difficult to know which annotations would be beneficial in a certain feature maintenance case).

5. DISCUSSION

While we used a simple measurement to quantify investments in feature recording and editing (656 lines), and to obtain that 18% of these saved 90% of lazy feature location costs, the actual costs will be different. More precisely, our eager approach incurs an actual time investment of $(C_{rec} + C_{ed}) \cdot AR + C_{ao} \cdot AL$, where AR (average annotation recording cost) and AL (average cost of finding a location for adding a missing annotation) are constants that need to be tailored to the project context. While concrete values for both constants require additional measurements, we can argue about their relation.

Lazy feature location is commonly considered expensive. It usually comprises source code inspections, together with running and interacting with the systems in order to locate a feature. Recent controlled experiments [13] of feature location in systems with 73k, 2k, 43k, and 19k LOC identified different phases that are commonly performed by developers: "seed search" (find entry points to a feature by interacting with it), "extend" (explore entry points to find related elements, usually by code inspection), "validate" (double-check the locations, usually via debugging), and "document" (record relevant functions in a text document). Three quarters of the 32 participants, who were not familiar with the systems and its source code, completed their three feature location tasks within 45-55 minutes. Given such experimental evidence, we can assume that a typical feature location task is in the order of 15 minutes in systems with similar characteristics. In our study, a feature requires 1.5 annotations on average; thus, we assume an AL of 10 minutes.

This allows a simplified calculation of the break-even point. When investing a number of annotation recording and editing effort $(C_{rec} + C_{ed}) \cdot AR$, the benefit is the number of annotations used in propagations (135) multiplied by the saved location time per annotation, which is diminished by omitted annotations (10%). Thus, $AR = 135/656 \cdot 10\text{min} \cdot 90\%$, so break even is achieved when AR is lower than 1.85 minutes. This clearly shows a big asymmetry in the cost and, thus, a benefit of eager annotation in our case study.

6. THREATS TO VALIDITY

External validity. Our simulation study is based on one set of projects only. However, the identified patterns showed that it has a rich set of feature-related activities, and the size of our projects is similar to other feature location study objects [13]. In fact, we strive to gain in-depth knowledge (patterns and activities) and finding ways to measure cost and benefits. Our methodology is repeatable on other projects.

The subject projects were developed in a research lab of a university, mainly by graduate and co-op students, and they may not be representative of industrial projects. We still believe that many of the identified patterns and insights are applicable to many other code-centric projects. Activities such as feature splitting or feature change propagation likely occur in other projects.

Internal validity. While our research method—retroactive simulation—allowed us to gain in-depth insight into strategies to apply the annotation approach, and to identify and define costs, it may have also limited observations. Our results should be complemented with action research and controlled experiments, which are subject to future work.

The simulator was not an original developer of the subjects. Missing in-depth code knowledge in fact caused some annotation omissions (cf. Sec. 4.1). Yet, these were negligible, and even support our main finding that in practice, annotation recording and editing costs are low. Identifying feature propagations was intricate. Among the 55 identified cases, for some it was in fact hard to tell whether a feature was cloned and customized at the same time, or a similar feature was implemented from scratch. We verified such cases with the original developer. This problem would not occur when using annotations during development, since a developer has the knowledge to differentiate these cases.

Finally, it is possible that using annotations in the original project (not in the simulation) could have influenced future commits. However, our results—that is, the measurements (periods and counting annotation markers) are independent of the actual commit granularity.

Construct Validity. Measurement of pattern frequencies relies on the granularity of commits made to the simulation branch. When no identifiable feature-related development was done, the selection of periods was subjective. However, since it is very costly to identify exact instances of all patterns, and since our main conclusions rely on measuring annotation markers, this simplification is justifiable to estimate the occurrence frequency of patterns. Yet, we identified all instances of feature propagation (P8) exactly to identify the exact benefit of annotations for feature reuse. Further, as discussed (Sec. 5), our primary metric (number of annotation markers) is only indirectly related to the economic cost or benefit. Measuring actual development (i.e., to calibrate the constants AR and AL) requires a controlled experiment, which is out of our scope, but would be valuable future work.

7. RELATED WORK

Traceability of features, requirements, and models. Establishing and maintaining traceability links, especially in the areas of model-driven development and requirements engineering, is a common issue [19, 25, 26]. It usually requires human interaction for meaningful traces, which is considered laborious and error-prone. Many approaches exist to map various kinds of concerns (e.g., features, requirements) to software assets. Robillard et al. [17] propose a continuously maintained concern graph, which is similar to our eager annotation strategy. Their tool FEAT supports developers with building such a graph, which was evaluated as being robust and cost-effective to create and use. Many other tools specific to features exist to map these to assets, such as CIDE [27] (features to code), FeatureMapper [28] (features to models), FeatureIDE [29], or KBuild [30].

Most of the existing approaches all record traceability information externally, with the usual maintenance issues [17]. Most also impose specific tools, such as an IDE. In contrast, our design goal was a generic, lightweight (minimal), and transparent annotation system, not requiring specific tooling.

In fact, only few works propose or evaluate embedding traceability in source code. While Winkler et al.'s survey [19] shows that most UML tools can automatically generate and maintain traceability links between models and code (kept as annotations in code), heavyweight tooling is required, and whether it can be used to realize feature traceability, is not known. Hegedus et al. [31] propose incremental model queries to soft-link EMF model elements in different model resources, as opposed to using hard references. Our approach is not much different: since the traceability information is not stored centrally (together with feature declarations), we also need to query the code on-demand to find all assets of a feature, while also exploiting the natural co-evolution.

Feature location. Rubin et al. [16] survey existing feature location techniques and show that all automated techniques have very low precision and recall, thus, having limited applicability in practice. Manual feature location is studied by Wang et al. [13], whose results show that it is a labor- and knowledge-intensive task. We use these results to discuss the overall costs of our eager approach (cf. Sec. 5).

Finally, we found no work on recording feature traceability manually and eagerly, besides an introduction into traceability [32], which classifies traceability maintenance into "continuous" (i.e., eager) and "on-demand" (i.e., lazy).

Variability evolution. In highly configurable systems with variability, features provide a configuration option. Feature combinations can be selected in interactive configurators to derive a variant. The evolution of such configurable, feature-oriented systems has been investigated before. Passos et al. [33] analyze the co-evolution of feature 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 target are systems that already have an integrated platform, a few patterns in fact overlap with ours. Similarly, Neves et al.'s [34] study identifies "evolution templates" (split asset, refine asset, add new optional feature, delete asset, etc.), some of which are also found in our study (e.g., splitting a software asset).

Yet, traceability differs from variability, as both serve different purposes. Feature traceability is used to locate features and aims to support development processes. Variability aims at the automated derivation of individual products, by defining variation points (where variants differ). While feature traceability is relevant from the very beginning, variability becomes relevant when creating an integrated platform—to make a feature optional or to account for differences among cloned feature implementations. While variability annotations have to be exact and properly structured, our approach tolerates overlaps, omissions, and small inaccuracies.

Cost/Benefit models. Cost and benefit is usually estimated based on regression, such as in COCOMO [35]. As an alternative, Martinez et al. [36] present a pragmatic economic model to perform cost-benefit analyses of software architecture adoption, relying on value-based metrics. 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. Ingram et al. [37] present an overview of traceability cost and benefit. Heindl et al. [38] introduce a model of traceability cost and benefit, and show that it is useful to estimate the return on investment of tracing approaches. Egyed et al. [39] propose a value-based approach that can be used to understand the cost-benefit trade-off in traceability generation.

Attention investment. Performing the various kinds of annotation recording and editing can be seen as *attention investment*—developers invest additional attention, hoping that it brings future benefits. Several studies apply this concept. Blackwell et al. [40] propose to use it for analyzing the cognitive dimensions of notations [41, 42] (design principles for languages syntaxes). Further studies of Blackwell et al. [43] use the concept to investigate the aspects of both professional programming and end-user programming, proposing attention investment models. We strived to minimize the required attention investment costs.

8. CONCLUSION

We presented a simulation study in which we applied an embedded feature annotation approach to establish traceability between features and assets throughout the development history of a clone-based product line. We derived a set of evolution patterns and presented empirical data on the costs and benefits of using the eager and internal storage strategies. Using the patterns, we also illustrated the practical application of such annotations.

In our study, the cost of creating and—more importantly—of maintaining the feature model and the annotations was negligible. Embedding annotations into assets in fact prevented most of the annotation maintenance costs, which were surprisingly low—almost as low as the initial cost of adding the annotations, since they naturally co-evolved together with assets. The cost of lazy feature location is by an order of magnitude greater than that of eagerly recording feature locations using annotations embedded in assets. Although it cannot be predicted which annotations will be beneficial for some task in the future, the eager annotation cost was largely amortized over the actual benefits. In fact, 18% of the recorded feature locations as annotations saved 90% of lazy feature location costs needed for feature reuse tasks, which is better than the break-even point. The benefit of the annotations is likely even higher, since they are also useful in feature maintenance tasks, which we did not measure, but which constitute the majority of developers' work.

The presented work aimed at gaining in-depth insights rather then generality. Future work includes validation of the approach in a case study where the approach is used during development. The study also revealed many opportunities for automation, and we plan to develop tool support for automatically proposing the placement of annotations based on changesets; for consistency management between the feature model and asset annotations; and for offering a feature-oriented project dashboard.

Finally, recall that traceability annotations are different from variability annotations. We also strive to utilize the feature traceability information to later realize variability (i.e., variation points) within an integrated platform. Studying such merge-refactorings and providing techniques and tools to support them is part of our future work [7].

9. REFERENCES

[1] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *SPLC*, 2015.

[2] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Journal of Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, 2009.

[3] S. R. Palmer and M. Felsing, *A practical guide to feature-driven development*. Pearson Education, 2001.

[4] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[5] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[6] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.

[7] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schäfer, "Flexible product line engineering with a virtual platform," in *ICSE*, 2014.

[8] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (FODA)

feasibility study," SEI, CMU, Tech. Rep., 1990.

[9] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A survey of variability modeling in industrial practice," in *VaMoS*, 2013.

[10] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski, "Three cases of feature-based variability modeling in industry," in *MODELS*, 2014.

[11] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *CSMR*, 2013.

[12] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, "Feature scattering in the large: A longitudinal study of linux kernel device drivers," in *MODULARITY*, 2015.

[13] J. Wang, X. Peng, Z. Xing, and W. Zhao, "How developers perform feature location tasks: a human-centric and process-oriented exploratory study," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1193–1224, 2013.

[14] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420–432, Jun. 2007.

[15] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *ICSE*, 1993.

[16] J. Rubin and M. Chechik, "A survey of feature location techniques," in *Domain Engineering*, 2013.

[17] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, Feb. 2007.

[18] T. Savage, M. Revelle, and D. Poshyvanyk, "FLAT3: Feature location and textual tracing tool," in *ICSE*, 2010.

[19] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software & Systems Modeling*, vol. 9, no. 4, pp. 529–565, 2010.

[20] "Online appendix," http://gsd.uwaterloo.ca/embeddedFeatureAnnotations.

[21] M. Antkiewicz, K. Bąk, A. Murashkin, R. Olaechea, J. Liang, and K. Czarnecki, "Clafer tools for product line engineering," in *SPLC*, 2013.

[22] I. Herraiz and A. E. Hassan, "Beyond lines of code: Do we need more complexity metrics?" in *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds. O'Reilly Media, 2010.

[23] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *ICSE*, 2014.

[24] ——, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, 2015, preprint.

[25] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *FOSE*, 2007.

[26] R. E. K. Stirewalt, M. Deng, and B. H. C. Cheng, "UML formalization is a traceability problem," in *TEFSE*, 2005.

[27] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE*, 2008.

[28] F. Heidenreich, J. Kopcsek, and C. Wende, "Featuremapper: Mapping features to models," in *ICSE*, 2008.

[29] C. Kästner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development," in *ICSE*, 2009.

[30] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, "Featureto-code mapping in two large product lines," in *SPLC*, 2010.

[31] A. Hegedus, A. Horvath, I. Rath, and D. Varro, "Query-driven soft interconnection of EMF models," in *MODELS*, 2012.

[32] O. Gotel, J. Cleland-Huang, J. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, J. Maletic, and P. Mäder, "Traceability fundamentals," in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. Springer, 2012.

[33] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wasowski, and P. Borba, "Coevolution of variability models and related artifacts: A case study from the linux kernel," in *SPLC*, 2013.

[34] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba, "Investigating the safe evolution of software product lines," in *GPCE*, 2011.

[35] B. W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, R. Madachy, and B. Steece, *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[36] S. Martínez-Fernández, C. Ayala, and X. Franch, "A reuse-based economic model for software reference architectures," *Repport: ESSI-TR-12-6, Departament d'Enginyeria de Serveis i Sistemes d'Informació, Barcelona, Spain*, 2012.

[37] C. Ingram and S. Riddle, "Cost-benefits of traceability," in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. Springer, 2012.

[38] M. Heindl and S. Biffl, "Modeling of requirements tracing," in *Balancing Agility and Formalism in Software Engineering*, ser. Lecture Notes in Computer Science, B. Meyer, J. Nawrocki, and B. Walter, Eds. Springer, 2008, vol. 5082.

[39] A. Egyed, S. Biffl, M. Heindl, and P. Grünbacher, "A value-based approach for understanding cost-benefit trade-offs during automated software traceability," in *TEFSE*, 2005.

[40] A. F. Blackwell and T. R. Green, "Investment of attention as an analytic approach to cognitive dimensions," in *PPIG-11*, 1999.

[41] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[42] T. R. Green, "Instructions and descriptions: some cognitive aspects of programming and similar activities," in *AVI*, 2000.

[43] A. Blackwell and M. Burnett, "Applying attention investment to end-user programming," in *HCC*, 2002.