CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Design and Assessment of an Engine for Embedded Feature Annotations

Master's thesis in Computer Science and Engineering

Tobias Schwarz

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Design and Assessment of an Engine for Embedded Feature Annotations

Tobias Schwarz

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Design and Assessment of an Engine for Embedded Feature Annotations

Tobias Schwarz

Supervisor: Thorsten Berger, CSE and Wardah Mahmood, CSE
Examiner: Jan-Philipp Steghöfer, CSE

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Design and Assessment of an Engine for Embedded Feature Annotations

Tobias Schwarz
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Features are an inherent unit of development of every software; and are defined as a set of implementation artifacts that constitute a functionality that adds value to the product, and is perceived useful by the customer. Locating features in source code is a typical software developer task, whether it before implementing a new feature, or maintaining and bug fixing of existing ones, as it is essential to know where to make changes. For tracing features to their implementation, two mechanisms can be used; external and internal documentation. As the names imply, external documentation refers to maintaining the traceability links externally, whereas internal documentation involves labeling assets inside the source code (aka embedded annotations). For internal documentation, two strategies are used namely eager and lazy approaches. The former involves annotating the code artifacts during development, whereas the latter requires extracting feature-related information from an un-annotated codebase based on heuristics. The former, although involves some added effort, result in significant returns in terms of accuracy and degree of reuse, also enabling a wider range of analyses. Also, the added effort can be minimal depending on the size of the project but soon begins to prove its worth in the short-run (when aiming to reuse) as well as the long run (when maintaining the code base).

Embedded annotations (with eager strategy) allow a minimally invasive and almost cost-neutral way to document the product features inside source code. This brings some benefits, the significant ones being easier co-evolution of code and traceability links, elimination of feature location, and ease in tasks like feature and artifact reuse (cloning) and maintenance (propagation). Several approaches exist today on how to document features in source code. Different definitions lead to different implementations and therefore, reuse is not directly possible. This work tackles exactly this issue and provides a unified design of embedded annotations with a free-to-use reference library according to the presented specification. The functionality of this library, aka. engine, is shown on the use case of partial feature-based commits. Feature centric development, which is typical for agile projects get the possibility for isolated source code commits based on specific features aka. embedded annotations.

Keywords: Embedded Annotations, Feature, Software Engineering, Software Product Line, Git, Partial Commits

Acknowledgements

First, I would like to use this chance to thank my supervisor, Prof. Dr. Thorsten Berger, who has worked on this thesis topic with me. His enthusiasm for the project, guidance, countless discussions, and encouragement for better results allowed me to create a great research work. Special thanks to my co-supervisor Wardah Mahmood who always had an open ear for me and my general research questions. Without the support of Thorsten and Wardah, I could not have done this work. Allowing me to participate in the research group of Thorsten and find open-minded experts was an enlightening experience.

Thanks to my examiner Prof. Dr. Jan-Philipp Steghöfer for his critical words and push to even better research work.

Without the participants that took part in the survey as well as my thesis opponent Supriya Supriya and peers and friends of my Master Studies in Software Engineering and Management at Chalmers | Gothenburg University this work would not be what it is.

I want to thank in a very special way Liza Reuter and her parents. Without your encouragement and unconditional support, I would not be where and who I am today.

Finally, I would like to give a big thank my parents, as well as my siblings. My life would miss a lot without you.

Tobias Schwarz, Gothenburg, June 2020

Contents

List of Figures

List of Tables

List of Grammars

1

Introduction

Feature-driven development (FDD) structures, as part of the agile methods, the software development process into client-value functionalities, so-called features. FDD focuses on the process to continuously delivering software with an increasing number of functional and non-functional features. (Palmer and Felsing, 2001)
Developing feature-based software requires more than the planning aspects of FDD (Passos, Czarnecki, et al., 2013). Besides adding new features, they often need to be refactored and evolved for their new purpose. For this purpose, the current feature location in the source code must be known.
Locating a feature is a difficult task mainly due to its cross-cutting nature and the deteriorating knowledge about them. Feature knowledge is not only important for variant-rich systems, where they provide a way to distinguish variants, but also for diversified software, such as software product lines. (Passos, Padilla, et al., 2015; Ji et al., 2015)
For feature recovery and feature location, there are research approaches available for full or semi-automated feature locations, but their results are not yet satisfactory for practitioners (Abukwaik et al., 2018). Manual feature recovery and location is more precise but causes at the same time more costs due to labor-intensive work. A solution recently proposed by researchers is to continuously trace features and their locations, using a lightweight technique; embedded feature annotations. Considered as the least expensive technique for feature annotation, the costs for maintenance is reduced and feature propagation and migration is improved for further software variants. (Ji, 2014)
To use the full potential of embedded annotated features, tool support is required. This ensures on the one side to encourage developers to use the annotations, and on the other side to locate features with embedded annotations. Such a tool could also be potentially be enriched with an integration in an existing software version control system, such as Git.
Embedded annotations might find application to many software development areas. By an integration into a programming language itself they could become natural part of that specific language in development IDEs and could also be added as a part of the language's documentation. On the other hand such an integration limits it to a specific programming language. A different approach would be to provide a generic support of embedded annotations into tools and platforms which cover all kinds of programming languages, such as Git or other version control systems.

1.1 Statement of the Problem

Features are commonly used as a way to abstractly and more intuitively describe functional or non-functional parts of software assets. As they are describing the functionality of a software product, they can be used to express the product's functionalities on a common language level. Also, they are used for describing the differences between product variants. When features are not documented, knowledge about their functionality and location in source code often fades out over time and needs to be recovered through labor-intensive work. To document features, there are two possibilities, with embedded annotations in the source code itself or with a separate tool. (Krüger, Mukelabai, et al., 2019). The notion of features can thereby be found in many planning tasks, as well as in agile methods and variant-rich systems.

Relevance of feature annotations Besides the described planning aspect, in project planning, there are more fields where features take a central element. On the highest level, they allow to easily describe characteristics of a variant-rich system. Additional, other tools such as project and issue trackers are using the terminology of the features. The main challenge is the current lack of support of features at the source code level (Ji et al., 2015).
Knowledge of functions is not only important at higher levels, but also at lower levels, such as configuration files, data sets and especially source code. Developing features for further development, maintenance, platform construction or reuse have one thing in common: at which point or points they are coded in the software. Finding a feature is therefore an important task. (Entekhabi et al., 2019)
Ji et al. (2015) unveiled in a study that feature location is "one of the most common activities of developers" and researched the costs and benefits of embedded annotations. There are two kinds of feature location techniques. First, the eager one where features are annotated during development, and the lazy one, where features are annotated after development or even when locating them. In the study of Ji et al. (2015), a cost-saving of 90% from the lazy to the eager strategy could be shown. Also, their work set the foundation for tool developments of FLORIDA (Andam et al., 2017), FeatureDashboard (Entekhabi et al., 2019), and a recommender tool for missing feature locations (Abukwaik et al., 2018). One of the challenges with embedded annotations is that there is currently no standard which unifies their usage and appearance.

Feature annotation management To explore the full potential of embedded annotated features, tool support is required. This ensures on the one side to encourage developers to use the annotations and on the other side to locate features. Such a tool might be usable as a standalone application or integrated into existing tools and platforms.
So far the previous mentioned tools, as well as variant management systems such as FeatureIDE [1] or Pure::Variants[2] are independent standalone, or in other platforms integrated, solutions with different annotation semantics. Latter tools support

[1]`http://www.featureide.com/`

[2]`https://www.pure-systems.com/products/pure-variants-9.html`

thereby variability annotations, while embedded annotations consist of traceability annotations.

1.2 Purpose of the Study

The purpose of this study has several main aspects. First, a standard for embedded annotations shall be proposed. Second, a re-useable parsing engine for locating embedded feature annotations shall be created. And third, the parsing engine should be integrated with Git as an extension for partial feature-based commits.
For the embedded annotations several concepts exist and even when tool implementations are successors of each other the interpretations are slightly different - as evident in (Ji et al., 2015) and (Entekhabi et al., 2019). Therefore as a first step, a reference definition is required. The generation of a unified design for embedded annotations is important beyond the need of a work to reference to. As soon as tools shall be used in different projects, or developers switch projects, an efficient usage is only possible when people and tools work in the same way together.
Due to the different embedded annotation interpretations, different implementations exist to perform the work to extract them. With a reference definition as the second step, a conformal reference and re-useable implementation can be provided.
With the third step, partial feature-based commit, a field is addressed which is little known by mainstream development, even when Git-tooling is available. The extension and simplification of partial commits shall allow developers to easily use them and organize their commits for annotated source code in a better way. Providing tool support for partial feature-based commit reduce the number of steps to be performed and reduce the risk of wrong steps by the developer.

To perform this study as close to practice as possible and to collect real-world requirements, this study is conducted with practitioners. The study concept is done with one specific company from the area of web development. The survey to evaluate the notion of embedded annotations is conducted with practitioners from different companies and industrial fields.

1.3 Structure of the Report

This report is structured such that after this introduction, Chapter 2 presents the relevant background information for this report and the research carried out. Chapter 3 presents the structure of applied research and the application of theory in this context.
The next three chapters describe the results of this work. First, Chapter 4 shows the created and reviewed design for a unified embedded annotation approach and in addition the results of the conducted survey to evaluate it. Chapter 5 shows the results for an engine implementation according to the previously shown embedded annotation design. Lastly, the usage of the created engine in an industrial use case is presented in Chapter 6.
In Chapter 7 the reached results are discussed, followed by the threats to validity

of the conducted research and its limitations (Chapter 8). A summary of the conducted research is given by Chapter 9 and Chapter 10 closes the report with an outlook for potential future research.
The work is appended with further information about the embedded annotations design (Appendix A) and the unmodified results of the survey (Appendix B). Appendix C shows details of the tool evaluation process conducted in Chapter 6.

2
Background and Related Work

This chapter provides the background and refers to related work for the scope of the conducted research and this report.

2.1 Feature Definition

A feature in the field of software product development can be defined as a “logical unit of behavior that is specified by a set of functional and quality requirements” (Bosch, 2000, p.194). Features are used to describe product functionalities and serve as the common language between technical and non-technical persons.
From a user perspective, a software product consists of several functional units within one product or a product family. In the requirements process, these functional units are expressed in functional and non-functional (aka. quality) requirements. A feature covers a specific set of these requirements.
Features are in the first view functional requirements - functionality that is provided by a software product or not. Considering features as non-functional requirements is as important as considering them as functional requirements. The reason for this is the overarching scope of non-functional requirements for the whole system and how it functions. Non-functional requirements can address e.g. reliability, performance, or maintainability of software products.
Features are not only of the type present or not in a software product. They may also have dependencies between each other. The most common relationship is “depends on” where one feature can only be present when the other one is already there. The opposite relation is “mutually exclusive” where features can never be present at the same time.

2.2 Feature Usage

In Chapter 2.1, features were described as a more abstract concept to describe the software’s functionality. There are several approaches which put features in the center of their design such as:

Software Product Line Engineering (SPLE) A software product line has the goal to support a set of similar software products with a shared set of source code. The potential variants are integrated into the shared platform and are selected via a feature model, representing features and their relation.

Clone&Own Describes a procedure to copy a complete software or parts with certain features and use it independent of the original product further on.

Feature-Driven Development (FDD) An agile methodology for project planning whereby the customers' functionalities (features) are put in the center for all planning tasks.

Virtual Platform A tool that supports a set of incremental migration techniques to perform a transformation from clone&own to software product line engineering.

2.3 Feature Location

For developing or changing a feature in the software, its location must be known. Ji et al. (2015) discussed two important questions with feature locations. Firstly, "How to effectively maintain traceability between the features and the corresponding software assets?" and secondly "Where to store the feature traceability information?". Addressing the first question, to map features to source code, two possibilities exist, the eager and the lazy strategy. The eager strategy requires an effort to record feature positions during the actual software development. The lazy strategy retroactively re-constructs the feature location when required afterward. The benefit of the eager strategy is for the developer to have in the moment of development the best understanding of the feature and its relation to the source code. The process to share feature knowledge and even deteriorating feature location knowledge might hinder this work for developers themselves (Ji et al., 2015; Krüger, Mukelabai, et al., 2019; Andam et al., 2017).

For recording feature locations, which is only applicable in the eager strategy, two possible solutions exist. Either to record the feature location in an own external tool or directly with the source code artifacts. The external tool requires a universal way to position the feature locations in different kinds of software artifacts and to handle the evolution of source code. This means that changes must be either detected and mapped or manual work is required to keep the tool up-to-date. The internal (embedded) approach requires a general approach as well to annotate all kinds of software assets without disturbing its functionality or pre-compiler functions.

Krüger, Çalıklı, et al. (2019) showed that small improvements on the source code level have a big impact on software development, as developers primarily focus on it. A lightweight technique, such as embedded feature locations into source code, has an immediate benefit to development and maintenance without tool training or specialist processes to follow.

2.4 Traceability and Variability

Embedded annotations for feature locations may serve two purposes. Either to trace the location of the feature(s) in the source code (traceability) or to control the active parts of a software product which are part of the product's binaries (variability).

To document feature locations in a most flexible way, as well as to link these to other artifacts, the notion of traceability is followed.

Using traceability information for variability purposes and vice versa must be handled with care. Variability information (e.g. #IFDEF pre-compiler) are exclusive on the source code level and contain very specific information which source code parts are pre-compiled into the binaries. Traceability information thereby allows to map source code to features while keeping it unmodified. Furthermore, it enables the link to higher levels such as software architecture or requirements and provides higher flexibility in marking features, due to the non-modifying character.

2.5 Feature tangling and scattering

A goal of software development is to create modular and re-usable source code. Therefore, to follow the design principle separation of concerns is required. This means that concerns - aka. features - are separated into consistent blocks of source code, also known as cohesion. At the same time, these code blocks need to be independent of each other, known as coupling. To reach a good modular and re-usable software it shall have high cohesion and low coupling. With rising complexity, legacy systems, and interconnections between concerns (cross-cutting concerns) the separation into code blocks is difficult or would complicate the overall software unnecessarily. Therefore, software products have always features that are interconnected either in a tangled or scattered way. (Apel et al., 2013)

Feature Tangling means that source code blocks belonging to a certain feature are mixed with source code belonging to different feature(s) inside one logical unit, such as a class, method, or if/switch statement.

Feature Scattering means that a certain feature is separated over multiple different parts of the source code, such as classes or methods.

Example for tangling and scattering The following source code snippets show an illustrative example for tangling and scattering. The code is scattered for feature "Weight" (in red) and feature "Color" (in blue) over all shown classes. Inside class "Edge" both features are tangled.

```
class Graph {
  Vector nv = new Vector();
  Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n);nv.add(m);ev.add(e);
    if (Conf.WEIGHTED) e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w){
    if (!Conf.WEIGHTED) throw RuntimeException();
    Edge e = new Edge(n, m);
    nv.add(n);nv.add(m);ev.add(e);
    e.weight = w; return e;
  }
  void print() {
    for(int i=0; i<ev.size(); i++){
      ((Edge)ev.get(i)).print();
    }
  }
}

class Color {
  static void setDisplayColor(Color c) {...}
}
```

```
class Node {
  int id = 0;
  Color color = new Color();
  void print() {
    if (Conf.COLORED)
      Color.setDisplayColor(color);
    System.out.print(id);
  }
}

class Edge {
  Node a, b;
  Color color = new Color();
  Weight weight;
  Edge(Node _a, Node _b) { a = _a; b
    = _b; }
  void print() {
    if (Conf. COLORED)
      Color.setDisplayColor(color);
    a.print(); b.print();
    if (!Conf.WEIGHTED) weight.print();
  }
}

class Weight { void print() { ... } }
```

Listing 2.1: Code example tangling and scattering (Berger, 2019)

2.6 Tangling Degree and Scattering Degree

To measure the tangling and scattering of code presented in Chapter 2.5, the two measurement values scattering degree and tangling degree exist. Both, scattering degree and tangling degree can be applied to source code and on file level. The following definitions are a combination of the research results of Liebig et al. (2010) and El-Sharkawy et al. (2019).

Metric	Description
SD_{vp}	Scattering degree per individual annotation in source code. Represents the sum of variation points where the individual annotation is used. Variation points are in this context &begin, &end and &line.
SD_{file}	Scattering degree per individual annotation in file level. Represents the sum of files where the individual annotation is used.
TD_{vp}	Tangling degree per individual annotation in source code. Represents the sum of annotations used in one variation point.
TD_{file}	Tangling degree per individual annotation in file level. Represents the sum of used annotations in one source code file.

Usually for all metrics, the average value and standard deviation are given.

Table 2.1: Definition of Scattering and Tangling Degrees

With these metrics, you can make a general assumption about your project and its tangling/scattering situation as well to track if it changes over time.
A factor to consider with such metrics is the way how they are calculated. Liebig

et al. (2010)[p.4] “measure each metric after normalizing the source code of each software system (i.e., removing comments and so on)”. Another factor which might be different handling in metrics “of variation points, namely negating and #else directives, to which we refer to as corner cases, as they are seldom explicitly considered in research” (Ludwig et al., 2019)[p.1].

2.7 Git Version Control Data Flow

Git as distributed version control system stores its to be managed source code online in a “Remote Repository” and creates a full copy of this “Remote Repository” on the users’ machines. The user takes the data from the “Remote Repository” (command “git pull”) to its own “Local Repository” and “Working Directory”. In the “Working Directory” the user can perform its changes.

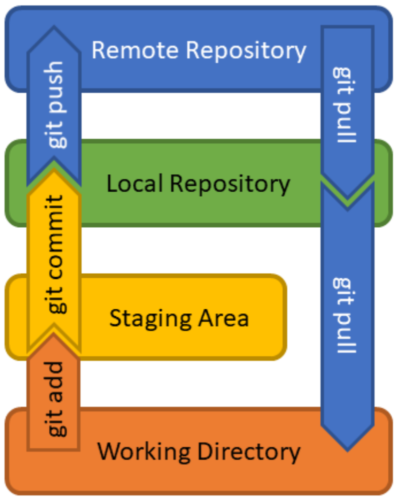


Figure 2.1: Git Data Flow Extract and Storage Level

After completing the changes, the results shall be shared with others and need to be moved from the “Working Directory” to the “Remote Repository”. The first step is to add your changes to the “Staging Area”, aka “Index” (command “git add”). The “Staging Area” serves the purpose to prepare changes in different files and folders for a shared change in source code. After all, changes have been prepared in the “Staging Area”, they are bound together into one commit of changes (command “git commit”) and moved into the “Local Repository”. The final step to share the changes with others is to push one or more commits from the “Local Repository” to the “Remote Repository (command “git push”).

2.8 Git Partial Commits

Git partial commit is a sub-command of the “git add” command. As the name indicates, a “partial commit” versions only a part of a modified git resource. Therefore the “git add” command offers the optional parameter −p or −−patch: “Interactively choose hunks of patch between the index and the work tree and add them to the index. This gives the user a chance to review the difference before adding modified contents to the index.” (Conservancy, 2020)

To prepare the differences and add them to the “Local Repository”, the “Staging Area” is used to collect the results of the individual partial commit steps.

The goal with this process is to enable software developers to work on different features and the base code at the same time, without losing the possibility to create commits in functional fitting units. To create a partial commit with the “−−patch” option, Git is breaking down with an internal algorithm all changes into so-called “hunks”. I.e. a hunk is a small piece of source code, representing a by git expected

belong together difference.
This partial creation of a git commit is possible as git indexes the changes in the staging area before actually committing them. As a developer, you can decide per hunk if you want to put it into the commit, not put it into the commit or break that hunk further down. After evaluating all changes, the commit is complete and can be versioned with a commit message.
Git partial commit is a way to implement parallel different changes in source code and maintain a clean commit history but requires manual and time intense steps. Such a process might be supported with specialized tooling for certain use cases.

2.9 Related Work

The following literature has been analyzed to provide the foundation for this work. The different researchers show the importance of feature documentation and the benefits of using embedded annotations for this purpose. Several works provide as well tool implementations for feature extraction on embedded annotations and take for this an own set of annotation rules on how to use them. It also reveals that only with the right tool support, the concept of embedded annotations be can be used to best effect.

Krüger, Mukelabai, et al. (2019) analyzed in their work two open-source products, “Marlin” (3D printer SW) and “Bitcoin-Wallet” for Android. They identified and located features and provided their results to the research community as a reference software for feature location.
The main aspects taken of this work are the notion of features as well as the notion of embedded annotations, shown in the created annotated projects.

The power of embedded annotations is shown by Ji et al. (2015). In their work, the researchers unveiled that embedded annotations have the main benefit to evolve naturally with the source code itself. The later usage of these annotations allows reduced costs for development, feature propagation, and platform/clone creation as well as maintaining tasks. The saved costs are thereby higher than the spend ones, which were almost zero.
The work with feature annotation can be split into the following tasks: Adding Features, Removing Features, Refactoring Features, Improving Feature Representation, Fixing Annotations, Cloning, and Maintaining Consistency and Evolving Assets. All of them need to be considered when using embedded feature annotations in real projects. (Ji et al., 2015)
The main aspects taken of this work are the challenges arising from working with software features, the different documentation possibilities as well as the cost-efficient usage of embedded annotations.

Entekhabi et al. (2019) proposed the tool FeatureDashboard[1] for feature visualization. The tool is based on textual feature annotation on source code snippet, source

[1] `https://bitbucket.org/easelab/featuredashboard/`

files, and folder level. FeatureDashboard is an Eclipse-based tool and supporting different views about the annotated project. With the available graphical and metric views, developers can identify where the features are located and in addition see their relationship and how they are tangled.
The main aspects taken of this work are the notion of embedded annotations and information about the tool FeatureDashboard to extract feature locations.

The foundation for FeatureDashboard is given by the tool FLORIDA (Andam et al., 2017). Besides setting the two main use cases: Encouraging developers to use embedded feature annotations and feature-location recovery, Andam defines the embedded annotations, the feature views, and metrics as well as the feature location.
The main aspects taken of this work are notion of features and embedded annotations, as well as potential use cases for embedded annotations.

Enabling and encouraging developers to add feature locations into their daily work is challenging. Mainly as the benefit of this work is seen after some time in the maintenance process and even not be the developer himself. Therefore, tool support is necessary to document feature locations. As feature location is a labor extensive working task and fully automated feature locating tools miss the industries required precision, Abukwaik et al. (2018) proposed a machine learning enriched recommender system to enable developers to tag their new created source code.
The main aspect taken of this work is the recommender system to support developers while development and maintenance to document their features.

Hevner et al. (2004) investigate in their work design-science and behavioral-science. Both of them common research methodologies in the Information Systems discipline. Behavioral-science covers research on humans and firms may behave, while design-science searches new ways to create innovative artifacts.
The main aspect taken of this work is the design-science methodology.

3 Methodology

This chapter contains the research focus and describes the underlying research methodology with its concrete adaption for this research. It describes the research questions and how they are covered by the following described research methodology.

3.1 Research Questions

In Chapter 2.9 “Related Work” the current state of research is shown and that currently different definitions and therefore different implementations to extract embedded annotations exist. This research is targeting both conceptual and technical aspects and the following research question and claims are raised for it.

RQ1 What can a unified and intuitive standard for embedded annotations look like?

- **Claim1** A common definition of embedded annotations improves software development efficiency and maintainability.
- **Claim2** Embedded annotations are intuitive to use.
- **Claim3** Embedded annotation location extraction is meaningful for software development

RQ2 How can embedded annotations make an industrial use case more efficient?

3.2 Design Science

The methodology chosen for this thesis is design science. Design science is a problem-solving approach which describes how to conduct, evaluate, and present product development for information system projects. Hevner et al. (2004, p.80) described it as “The goal of design-science research is utility”, which means to extend persons and organizations capabilities by creating new and innovative artifacts. The focus is thereby to plan what will be developed and to reason how it will be used. A goal is to learn from the reasoning when successful, but especially when expectations failed. The difference to the related behavioral-science-research is its search to develop and verify theories, Hevner et al. (2004) proposed in their work seven guidelines for design science in information system projects. These guidelines are in the following listed and contextualized to this work.

Guideline 1: Design as an Artifact

The goal of design science is to create a viable artifact, which can be a software, model, or method. Often the constructed solution is not fully grown and covering in this phase a specific aspect of information systems.

In the context of this thesis, a unified design and an extraction engine for embedded annotation, as well as a Git extension for partial feature-based commits is created.

Guideline 2: Problem Relevance

Design science aims by constructing innovative technology-based solutions to tackle currently unsolved technology and business problems.

In the context of this thesis, we tackle the lack of standardization of embedded annotations as well as the lack of an easy way to perform partial commits based on embedded annotations.

Guideline 3: Design Evaluation

To demonstrate the intended functionality of the developed solutions, design properties for functional and quality aspects are required to evaluate the concept. Due to the iterative and incremental activities, the evaluation phase provides regular updates to the development phase.

The evaluation happens for this research in an empirical dimension as well as a technical dimension. For the unified standard of embedded annotations, personal talks, and a survey is conducted with the supervisor, his research group, and practitioners. To ensure the quality of the implemented tools, technical tests are derived from the specification to ensure the valid detection of embedded annotations and the skipping of non-embedded annotations.

Guideline 4: Research Contributions

As a research methodology, design science targets to provide new and interesting research results to the body of knowledge. The kind of contribution for the designed artifact might be in novelty, generality or significance.

The contribution of this thesis is on the one side the created embedded annotations library - the design artifact - and the specification for embedded annotations - methodology.

Guideline 5: Research Rigor

To determine how well an artifact works, rigor methods need to be designed. Often mathematical formalism on data collection or data analysis is used for this purpose.

Solutions which contain a human factor require more informal methods and interaction with the user.
The level of rigor is also derived from how efficient the solution can be used as well as how applicable it is to the given theory. A high level of rigorous account of generalizability and means to find the right balance between rigor and relevance.

For research question 1, qualitative and quantitative data is collected to evaluate the design of the embedded annotation design. The quantitative data shows thereby if one attribute is in general fulfilled or not.
Research question 2 consists of a tool implementation, which will be evaluated with a set of common development activities. For this the activities are conducted with and without the help of the created tool.

Guideline 6: Design as a Search Process

Searching for the most optimal solution happens in design science as an iterative approach. Starting with a simplified problem or subset allows in the different iterations to learn more about the underlying problem. Searching for the optimal solution requires knowledge about the problem space and solution space. Problem space is the given requirements for the to be solved issue and the solution space covers the technical and organizational aspects.

For this thesis work, we consider different phases of artifact development. Starting with a subset of embedded annotations, these will be tested with defined use cases, defined in collaboration with the industrial partner and research group. Incrementally expanding the functionality allows a deeper understanding of how embedded annotations work is done in this context and how the final solution looks like.

Guideline 7: Communication of Research

To implement and apply the created artifacts, both technical persons and managers need to be convinced of the meaningful purpose of it. The challenge is to provide enough details to technical persons to apply and implement it on a technical level and at the same time to abstract it to an organizational level to allow managers to decide about it for their responsibility area.

In Chapter 4 the "Embedded Annotations Design" is presented and represents the pivot point to use embedded annotations for a project. The design description is written in a way to show technical details and convey the usefulness of the approach. The concrete benefits are shown in Chapter 6 "Industrial Use Case".
All created tools are online accessible and available for later use. It is intended to write a research paper about this work to allow compactly sharing the results.

3.2.1 Adjusted Design Science

Design Science distinguish the research into "Environment", "Design Science Research", and "Knowledge Base". These areas are linked with different cycles: "The

Relevance Cycle bridges the contextual environment of the research project with the design science activities. The Rigor Cycle connects the design science activities with the knowledge base of scientific foundations, experience, and expertise that informs the research project. The central Design Cycle iterates between the core activities of building and evaluating the design artifacts and processes of the research" (Hevner, 2007)[p.2].

In the **Environment** block, different "Application Domains" exist for this research. The first are to be considered "People / Organizational Systems" with the different roles and company processes, which are linked to the design science research, are listed. For this research they are SW-Developer, SW-Architects, Project Management, and Requirements Engineering. Secondly, for the to be considered "Technical Systems", for this research the areas of Feature Documentation, Feature Traceability, Feature Location, and Feature Location Tools are considered. And lastly for "Problems & Opportunities", Standardization, Lightweight tool, and Feature Isolated Development are the important aspects to consider.

The **Relevance Cycle** links the "Environment" and "Design Science Research" blocks and is responsible for input requirements to the research, but also to return the design science research output for field testing back to the environment. For this research the relevance cycle is pass through with the talks to a web development company for the usage of embedded annotations and potential use cases. Also, for feedback to the in the design science research created artifacts this cycle is pass through. For the created Embedded Annotations Design feedback is received by practitioners and the industrial use case is evaluated on typical work tasks of the application domain.

In the **Knowledge Base** block, the theoretical foundation for the design science research is given. As foundation for the definition of embedded annotation, the "Scientific Theories & Methods" of Andam et al. (2017), Entekhabi et al. (2019), Ji et al. (2015), and Krüger, Mukelabai, et al. (2019) are used. The knowledge base is backed with the "Experience & Expertise" of this thesis works supervisor' Thorsten Berger and his research group *easelab*. As existing "Meta-Artifacts" the tools FLORIDA and FeatureDashboard, as well as the git-add sub-command "−−patch" are used.

The **Rigor Cycle** is located between the Knowledge Base and the Design Science Research and provide current knowledge as well as state-of-the-art knowledge to the research. For this research different research works have been conducted especially in the first half of the research to design the embedded annotations design. This cycle was also pass through in weekly meetings with the supervisors, plus talks to the research group. Also, for program comprehension of the tools FeatureDashboard and git partial commit this cycle is used.

Design Science Research has as core element the **Design Cycles**. In this block the research itself is conducted. While Relevance Cycle and Rigor Cycle are conducted for special purposes, the Design Cycles are pass through more frequently. This research has different Design Cycle to evaluate the build artifacts. The *Embedded Annotation Design* as artifact is evaluated in several iterations with the supervisor and his research group. For this the current state as well as potential options were discussed and the next steps defined. After a stable version of the design was reached, external feedback was collected in an empirical survey with industrial par-

ticipants. For this survey persons in different roles and from different companies participated. The design was created based on a set of design properties, which were also used to evaluate the design within this survey. For this the participants had the option to rate a question in a Likert scale from “Completely Disagree” till “Completely Agree” as well as to provide free text answers to the questions. The survey is closed with optional participants industry role and contact information. The received feedback was collected and used to further improve the design. In addition to the theoretical evaluation of the embedded annotation design, the design was used to implement a further artifact, the Reference Engine. While the Reference Engine itself is an artifact, it serves at the same time as evaluation of the design as it is now put into practice and new aspects appeared while implementing and testing. Also, potential options of the design could be eliminated and rose within this Design Cycle. The last Design Cycle is between the Reference Engine, now considered as artifact, and the Industrial Use Case. For the Industrial Use Case several use cases have been evaluated and with the industrial partner two of them where evaluated in more detail. Finally, one use case was implemented and evaluated with this the Reference Engine. This Design Cycle fulfilled thereby especially the purpose to evaluate the interfaces and reliable extracted data. Used to evaluate the Reference Engine, the Industrial Use Case itself has own typical Development Scenarios, e.g. bug fixing, new feature development, or to evaluate its functionality. These typical scenarios have been defined and the result with and without the in the Industrial Use Case created artifact evaluated.
Over the time of the research, the different Design Cycles have been focus of specific **Research Objectives** where a specific aspect of the overall design science research has been worked on.

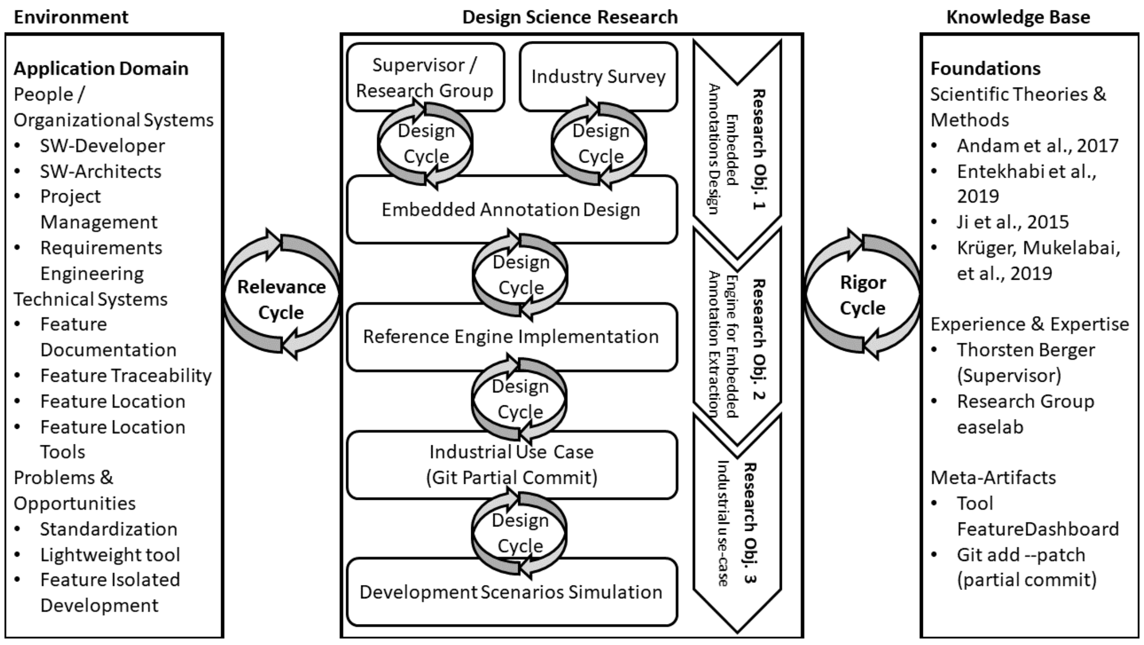


Figure 3.1: Design Science Methodology applied for this research

3.2.2 Project Research Objectives

The research questions are answered with a methodology, based on an adjusted design science process. Different research objectives of the thesis answer thereby different RQs.
The first research objective covers the creation of a unified embedded annotation design and specification and answers thereby RQ1. Research objective 2 takes care of the creation of an engine (aka. library) for extracting embedded annotations out of source code. The last research objective, research objective 3, investigates into industrial use cases for the usage of the created specification and implement one of them. With research objective 3, RQ2 will be answered.

Research objectives 1 - Embedded Annotations Design to answer RQ1 with steps:

- **Literature Review** for knowledge seeking about embedded annotations, and which notions currently are available, is conducted. The literature review is conducted in a lightweight snowball technique with starting literature provided by supervisor, plus a search for embedded annotations and feature documentation on Google Scholar.
- **Specification** to create a notion of embedded annotations in syntax and semantics is created, discussed, and shared.
- **Research Group Feedback** to receive feedback from embedded annotation experts and experienced researchers.
- **Survey Creation** to conduct a survey with industrial practitioners.
- **Practitioner survey** to receive industrial feedback and include their feedback into the embedded annotation design and specification.

Research objectives 2 - Engine for Embedded Annotation Extraction with steps:

- **Embedded Annotation Engine** according to the specification document to implement a reference library that can be re-used in industrial use-cases.

Research objectives 3 - Industrial use-case to answer RQ2 with steps:

- **Define use cases** which can be improved with the usage of embedded annotations.
- **Pick use case** in collaboration with an industrial partner. This decision is taken in an open discussion between representatives from industry and research.
- **Implement use case** , which was collaboratively selected, create a detailed concept to improve use case with embedded annotations and realize it.
- **Evaluate use case** against described use case and present created concept.

4 Embedded Annotations Design

The embedded annotations design serves the purpose to describe how to document software feature locations close to the source code artifacts level. In general, there are two ways to locate features in a software product: First, the “lazy” approach where to locate them when needed and second, the “eager” approach to document feature location while development. The here chosen approach is the “eager” one, which can be either reached with external tooling or as used here, to document the feature locations directly in source code and specialized files close to it. Embedded annotations offer the benefit - to externally documented feature locations - that they evolve naturally with the source code itself (Ji et al., 2015) and while cloning of source code in Clone&Own actions, allow propagating changes over software variants. Embedded annotations cover either blocks or specific lines of source code or file system resources. With this flexible approach, it is possible to annotate projects on a system level, e.g. to benefit from object-oriented programming, folders, and files reflect the internal structure, and at the same time to annotate line-specific feature relations. The way how these annotations work is independent of any project programming languages and can be also applied to non-source code files, such as e.g. configuration or binary files.
Embedded annotations fulfill the purpose of traceability and neither required central management nor to be pre-defined.

Features play a central role in modern software development. In general, agile software development focuses on customer functionality and features, whereby the method “Feature Driven Development (FDD)” takes a special position and puts the feature as the center of every decision and following the agile manifesto. (Wikimedia, 2019)

Locating features in source code is an important work for software developers (Entekhabi et al., 2019). The benefit to document features is seen in most cases only in the long run or with high coverage of the source code but can reduce feature location costs significant (Ji et al., 2015). Currently several slightly different approaches exist to write feature locations in project artifacts, known as embedded annotations (Ji et al., 2015; Andam et al., 2017; Entekhabi et al., 2019; Krüger, Mukelabai, et al., 2019)[1]. The situation of different approaches prevents a general unified working with embedded annotations and therefore reuse of tools and for developers changing projects/companies to use them without potential wrong usage. The here proposed

[1]Details to differences in Table A.1, Chapter A.1 .

notion unifies these approaches and allows the implementation of reference software libraries to it.

4.1 Formal Definition of Embedded Annotations

Embedded Annotations Terminologies

The design for embedded annotations requires some special terminologies:

Feature A distinct functionality or attribute of a software product, usually expressed in functional or non-functional requirements.[2]

Feature Model A feature hierarchy model, describing feature names and their hierarchy in textual form.

Feature Reference Reference to a concrete feature in the feature model.

Annotation Marker Keyword to open/close the annotated scope for one or more feature references.

Annotated Scope Artifacts, source code/files/folders, associated with one or more features. The scope is set with specialized files and annotation markers in source code.

Annotation Concrete usage of one Feature Marker in source code, including all its feature references.

Design Properties

For Design Properties, goals "such as simplicity, aesthetics, expressiveness, and naturalness are often mentioned in the literature, but these are vaguely defined and highly subjective" (Moody, 2009)[p.757]. For this work several design properties are defined and backed up with documenting the design decision flow and reasoning about them. This allows traceability between the final design properties and their origin and helps to justify them. For each design property a unique name is selected and described in a short statement what the property is about. (Moody, 2009)

The following four main- and five sub-Design Properties are used for this embedded annotations design. They are derived from Balzer's "principles of good specification" (Balzer and Goldman, 1981)[p.393] as well as extended with the experience & expertise of the supervisor and co-supervisor. These principles cover the primary use of software specifications: unambiguously and clearly understandable by specifier and implementor (understandability), testability of the specification's implementation, and maintainability to change the specification over time.

Usefulness (Balzer's Principle 1 and 2)

For Usefulness, the design must fulfill its intention to support embedded annotations and provide its user benefit to its working task. For this it defines the necessary elements in functionality and inside the annotation process.

Easy Applicable is part of the Usefulness property and describes how easy embedded annotations can be applied to a specific project.

[2]Detailed analysis about feature definition in Krüger, Mukelabai, et al. (2019)

Flexible to Use is part of the Usefulness property and considers how flexible embedded annotations can be used inside source code and for different projects.

Intuitiveness (Balzer's Principle 5)
For Intuitiveness, the designs level of how natural it feels for the user to use embedded annotations is considered.

Easy to learn describes the process to learn to use embedded annotations and that this time shall be as small as possible.

Robustness (Balzer's Principle 3, 4, and 7)
For Robustness, two dimensions are considered. For the user of embedded annotations, Robustness means that as many annotations as possible survive the evolution of the project, e.g. moving folders/files, removing code, and editing code. For the specification itself, Robustness means that it can be extended and evolved modular and do not require to rework the whole design.

Redundancy (Balzer's Principle 8) is part of the Robustness property and ensures that embedded annotations are designed in that way that annotating a feature in an artifact, the number of added markers and feature references is minimal and not repeated unnecessarily.

Succinctness (Balzer's Principle 6) is part of the Robustness property and balances between readability and that the additional writing effort is minimal.

Negligible Efforts
A design property which results in well working other design properties is Negligible Efforts. Besides having a useful, easy to understand and robust design, the arising costs to create and maintain embedded annotations shall be minimal. This shall avoid that embedded annotations are refused to use because of too high costs.

Embedded Annotations Meta-Model

The Meta-Model shown in Figure 4.1 shows the different attributes, relations and constrains of the embedded annotation notion.

The Meta-Model for embedded annotations contains 16 attributes and 22 relations. The elements used to mark source code artifacts with embedded annotation belong to the attribute type *Artifact* and derive into *Folder*, *File* and *Code Artifact. Code Artifacts* thereby derive into *Code Block* and *Line of Code* annotations. The different types of *Artifact* have all a many-to-many relationship to their feature-mapping counterparts. *File* has thereby the specialty that it contains of a feature-mapping and a *File Reference.*

A concrete *Feature* is represented by a *Feature Reference* which in the following can be used for *Feature to Folder Mapping*, *Feature to File Mapping* or *Code Annotation*; all of them from the type *Feature Mapping.* A concrete *Feature Reference* is used in a concrete mapping, but a mapping might consist of multiple *Feature References. Feature References* in *Code Annotations* are either *Block Annotations* or *Line Annotations.* One *Block Annotations* consist of exactly two *Annotations Marker*: "Begin" and "End". One *Line Annotation* consist of exactly one *Annotation Marker*: "Line".

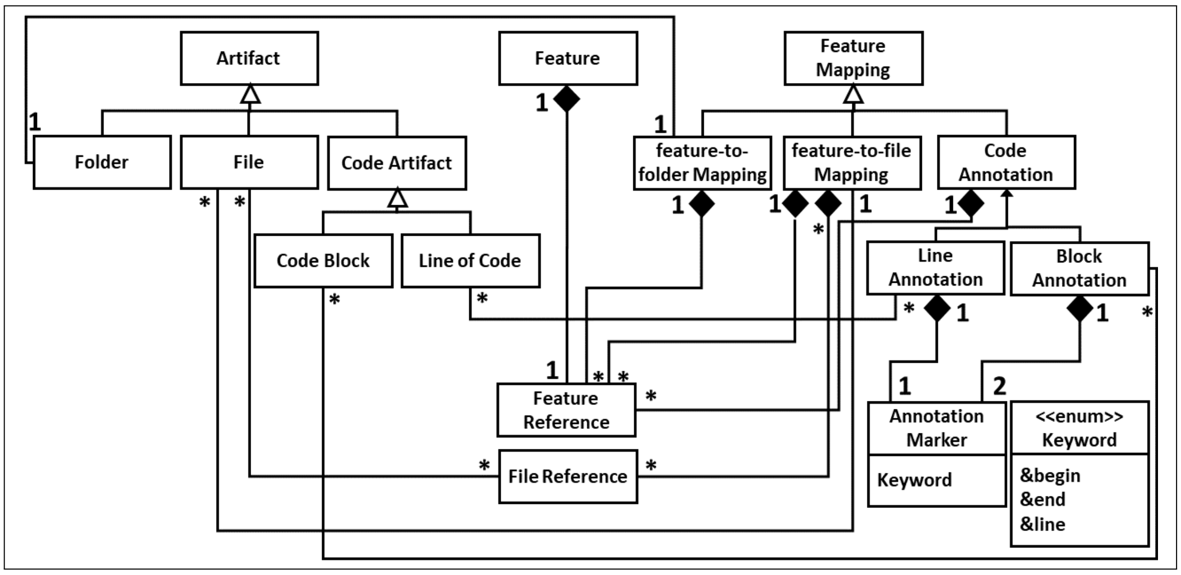


Figure 4.1: Meta-Model for Embedded Annotations

Embedded Annotations Level System

The definition of embedded annotations is split into two levels. This serves the purpose to have the appropriate level of expressiveness for different purposes. The levels are briefly introduced and explained in detail in the further chapters.

Level 1 Begin-, End- and Line-annotations, annotation identifier, Least-Partially-Qualified name, Simple Hierarchy Model, feature-to-file mapping and feature-to-folder mapping

Level 2 Level 1 + Logical operator expressions, Full Hierarchy Model

Keywords

Keywords are reserved words for the usage of embedded annotations and can not be used as annotation names. Note that the limitation is not on a combination of keywords and other words, e.g. the keyword "line" in a concatenation such as "DatabaseLineReading" will not be treated by the parser as a keyword.

Keyword-List:

- &begin
- &end
- &line
- &file

Grammar syntax definition

To express the description of embedded annotations in a regular grammar, the Extended Backus–Naur Form (EBNF) is used. The benefit of this representation is to be free of programming language-specific constraints. The EBNF consists of terminal characters (letters, numbers, spaces, symbols) and non-terminal characters (one or more terminal characters or other non-terminal characters). An expression in EBNF is composed of non-terminal characters and their replacements till only

terminal characters are left in the expression.
For the notion of embedded annotations, there are “Code Annotations”, “File Annotations”, “Folder Annotations” and “Simple Hierarchy Model” available as own EBNF definitions. This is possible as file, folder and hierarchy annotations are located in specialized files with pre-defined names. All remaining documents are checked for “Code Annotations”.

Shared assets between the embedded annotation grammars are “Feature-Reference” and “FEATURENAME”. The representation of the grammars is in the respective EBNF snippets and the full EBNF grammar is attached in the appendix in Chapter A.2 “EBNF Grammar Definitions”.

```
⟨featurereference⟩ ::= ⟨FEATURENAME⟩ (’::’⟨FEATURENAME⟩)*;
```

Grammar 4.1: EA, EBNF of Shared Feature-Reference Expression

```
⟨FEATURENAME⟩ ::= ([A-Z]+
                |  [a-z]+
                |  [0-9]+
                |  ’_’+
                |  ’\”+)+
```

Grammar 4.2: EA, EBNF of Shared FEATURENAME Expression

4.2 Feature Hierarchy Model

The Feature Hierarchy Model defines the available features and their hierarchy relations in a textual format. The feature hierarchy model serves to model features and organized them in a hierarchical structure to keep an understanding of them. This model needs to be maintained by the developers themselves as they have the deepest domain knowledge. This work can be supported by SW-Architects or domain experts.
The syntax is inspired by the Clafer modeling language (Bąk et al., 2011). Feature models allow very detailed descriptions of feature hierarchy and relations in-between. For the purpose of feature modeling, a subset of these options is sufficient and presented in the following as “Simple Hierarchy Model”. The full range of feature models is touched in the “Full Hierarchy Model”.

Simple Hierarchy Model The simple hierarchy model covers the feature references and their hierarchy with one-tab indentation per level. Each feature reference is listed as an independent line. For the simple hierarchy model, the hierarchy-file must be stored at the root node of the project and is exclusive per project.

Example:

```
ProjectName
    FeatureA
        FeatureA1
        FeatureA2
    FeatureB
        FeatureB1
```

Full Hierarchy Model The Full Hierarchy Model supports all language elements of feature hierarchy models. Each feature is listed as an independent line and constraints such as annotations relation, e.g. xor as mutually exclusive selection between annotations, or to mark an annotation with “?” as optional are possible. The full hierarchy model covers extended capabilities, as defined by the Clafer language, and includes also feature inheritance and nesting.
Example:

```
ProjectName
    FeatureA ?
        xor FeatureA1
        FeatureA2
    FeatureB
        FeatureB1 ?
        FeatureB2
```

In concrete implementations, the feature hierarchy file name could be _.cfr or similar as defined for this project.

EBNF representation For the simple hierarchy model, which is used in this project scope, the following EBNF implementation is used. The chosen approach lacks flexibility but fulfills the functional requirements. The proper handling of indents require specialized source code extending the EBNF.

⟨*projectHierarchy*⟩ ::= ⟨*FEATURENAME*⟩ (⟨*subfeature*⟩)*

⟨*subfeature*⟩ ::= ('\n' '\t' ⟨*FEATURENAME*⟩) ⟨*subsubfeature*⟩*

⟨*subsubfeature*⟩ ::= ('\n' '\t\t' ⟨*FEATURENAME*⟩) ⟨*subsubsubfeature*⟩*

⟨*subsubsubfeature*⟩ ::= ('\n' '\t\t\t' ⟨*FEATURENAME*⟩) ⟨*subsubsubsubfeature*⟩*

⟨*subsubsubsubfeature*⟩ ::= ('\n' '\t\t\t\t' ⟨*FEATURENAME*⟩) ⟨*subsubsubsubsubfeature*⟩*

⟨*subsubsubsubsubfeature*⟩ ::= ('\n' '\t\t\t\t\t' ⟨*FEATURENAME*⟩) ⟨*subsubsubsubsubsubfeature*⟩*

⟨*subsubsubsubsubsubfeature*⟩ ::= ('\n' '\t\t\t\t\t\t' ⟨*FEATURENAME*⟩) ⟨*subsubsubsubsubsubsubfeature*⟩*

⟨*subsubsubsubsubsubsubfeature*⟩ ::= ('\n' '\t\t\t\t\t\t\t' ⟨*FEATURENAME*⟩)

Grammar 4.3: EA, EBNF-Snippet of Simple Hierarchy Model

4.3 Feature Reference Names

Inside the feature hierarchy model, features with the same name may appear twice or more often. To reference features uniquely the individual feature is pre-extended by its ancestor till the combined feature reference is unique. This technique is called Least-Partially-Qualified name, short LPQ.
Example:

```
CoffeeShop
    Coffee
        Sugar
        Milk
    Tea
        BlackTea
            Milk
```

The feature "Milk" appears twice in the overall model, the individual entities can be addressed by "Coffee::Milk" and "BlackTea::Milk".
In contrast, the fully-qualified-name is much longer and more likely to change as compared to the least-partially-qualified name when the feature model evolves. In case an annotation appears only once, its LPQ is identical to its name, e.g. "Sugar". The separation of individual annotations to their ancestors is shown via the "::" characters. Approach from Andam et al. (2017).

4.4 Annotation Listing

Annotation identifiers are the concrete embedded annotations or features used in the source code. They are individual or combined words, without spaces or punctuation marks.

The usage of multiple annotation identifiers together is possible. In concrete implementations, the separator of annotations could be a comma, space-character, or similar as defined for this project.
The following syntax applies for the annotations listing:

```
Annotation_1 , Annotation_2 [, Annotation_n]
```

The conjunction of multiple annotations causes the mapping of the marked source code to ALL listed annotations in the same way. I.e. the order of the given annotations is independent.

4.5 Feature expression logic

Besides mapping source code, files, and folders to features or a list of features, it might be required to map code to combinations of features, written in Boolean expressions.

AND-Operator

The marked code part is considered to all given annotations individually. Comparable with multiple begin/end markers.
In concrete implementations the logical operator between individual annotations could be an “AND”, “&&” or similar as defined for this project.
The following syntax applies for combining multiple annotations in one identifier :

```
Annotation_1 AND Annotation_2 [AND Annotation_n]
```

Alternatively, with symbolic characters

```
Annotation_1 && Annotation_2 [&& Annotation_n]
```

OR-Operator

The marked code part is considered to all given annotations individually. Comparable with multiple begin/end markers.
In concrete implementations the logical operator between individual annotations could be an “OR”, “||” or similar as defined for this project.
The following syntax applies for combining multiple annotations in one identifier :

```
Annotation_1 OR Annotation_2 [OR Annotation_n]
```

Alternatively with symbolic characters

```
Annotation_1 || Annotation_2 [|| Annotation_n]
```

NOT-Operator

The marked code part is considered to all annotations individually, except the given one.
In concrete implementations the logical operator for an annotation negation could be an “NOT”, “!” or similar as defined for this project.
The following syntax applies for annotation negation in one identifier :

```
NOT Annotation_1
```

Alternatively with symbolic characters

```
! Annotation_1
```

Order of operators

Annotations logic operators may appear in mixed mode. The general precedence rules of Boolean expressions apply in this case.

4.6 Annotation Markers

There are three kinds of annotation markers: &begin, &end, and &line, each with specific purposes and syntax. Annotation markers are escaped through the programming language specific comment characters, such as e.g. "//" or "#". This avoids unwanted side effects for the project execution.
The individual markers have a leading '&'-symbol to distinguish these keywords from regular comments. Alternatives were considered, but the approach of the '&'-symbol from the basic embedded annotations definition is taken over. For example the '@'-symbol is used for JavaDoc keywords, the '$'-symbol is used in Bash scripts for variable definition and as well in Bash scripts the '#'-symbol as escape character for comments. Also, we consider the start of a comment with an '&'-symbol followed by one of the keywords for a different purpose as unlikely.

4.6.1 The begin-marker

In concrete implementations, this could be #ifdef, &begin, or a similar expression defined for this project.
The following syntax applies for the begin-marker:

```
   //&begin[ <parameter> ] <comment> <cr> /*<cr> carriage return*/
                                        /*is a newline symbol */
```

This marker considers the following consecutive lines of text to be part of this identifier. Identifiers can be defined in a hierarchy feature model but must not. A begin-marker must be closed by an end-marker.

4.6.2 The end-marker

In concrete implementations, this could be #endif, &end, or a similar expression defined for this project.
The following syntax applies for the end-marker:

```
    //&end[ <parameter> ] <comment> <cr>
```

This marker ends the scope of the begin-marker of the given identifier. Identifiers can be defined in a hierarchy feature model but must not. An end-marker must have a fitting begin-marker before.

4.6.3 The line-marker

In concrete implementations, this could be &line or a similar expression defined for this project. The line-marker is a convenient way to use a &begin- and &end-marker for a single line and can be substituted by them.
The following syntax applies for the line-marker:

```
    any source code //&line[ <parameter> ] <comment> <cr>
```

This marker considers exclusively its own line of text to be part of this identifier. If this line is a class or method, still only the annotated line is considered as part of this identifier.

EBNF representation

```
⟨marker⟩   ::= .*? (⟨beginmarker⟩
               |  ⟨endmarker⟩
               |  ⟨linemarker⟩)*

⟨beginmarker⟩ ::= '&begin' ' '* ⟨parameter⟩

⟨endmarker⟩ ::= '&end' ' '* ⟨parameter⟩

⟨linemarker⟩ ::= '&line' ' '* ⟨parameter⟩

⟨parameter⟩ ::= '(' ' '* ⟨lpq⟩ (' '+ ⟨lpq⟩)* ' '* ')' .*?
               |  '(' ' '* ⟨lpq⟩ (' '* ',' ' '* ⟨lpq⟩)* ' '* ')' .*?
               |  '[' ' '* ⟨lpq⟩ (' '+ ⟨lpq⟩)* ' '* ']' .*?
               |  '[' ' '* ⟨lpq⟩ (' '* ',' ' '* ⟨lpq⟩)* ' '* ']' .*?
               |  '{' ' '* ⟨lpq⟩ (' '+ ⟨lpq⟩)* ' '* '}' .*?
               |  '{' ' '* ⟨lpq⟩ (' '* ',' ' '* ⟨lpq⟩)* ' '* '}' .*?
               |  ' '* ⟨lpq⟩ (' '+ ⟨lpq⟩)*
               |  ' '* ⟨lpq⟩ (' '* ',' ' '* ⟨lpq⟩)* ' '*

⟨lpq⟩      ::= ⟨FEATURENAME⟩ ('::'⟨FEATURENAME⟩)*
```

Grammar 4.4: EA, EBNF-Snippet of Annotation Markers

4.6.4 Interleaving of Annotation Markers

Annotation markers appear in the simple case independent of each other and have no cross-cutting relation. That this is unlikely in a practical context is shown by following Listing 4.1. Here the annotation “Codecs” is surrounded by “RequestCoins” and “BIP21” is inside “Codecs”.

Potential interleaving situations are e.g. with overlapping annotation scopes with same begin-multiple end or vice versa.

Overlapping annotation scopes Between different &begin- and &end-markers as well with the &line-markers the marked scope for a certain annotation might overlap. A full overlapping (left), as well as a partial overlapping (right), are possible:

```
//begin[FeatureA]
... (code FeatureA)
//begin[FeatureB]
... (code FeatureA, FeatureB)
//end[FeatureB]
... (code FeatureA)
//end[FeatureA]
```

```
//begin[FeatureA]
... (code FeatureA)
//begin[FeatureB]
... (code FeatureA, FeatureB)
//end[FeatureA]
... (code FeatureB)
//end[FeatureB]
```

This situation might happen also with three and more annotation markers.

Same begin-multiple end or vice versa With the possibility of &begin and &end to support multiple feature references at the same time, there is the option to start multiple features within one &begin- and end them with individual &end-markers - or vice versa.

```
//begin[FeatureA, FeatureB]
... (code FeatureA, FeatureB)
//end[FeatureA]
... (code FeatureB)
//end[FeatureB]
```

```
//begin[FeatureA]
... (code FeatureA)
//begin[FeatureB]
... (code FeatureA, FeatureB)
//end[FeatureA, FeatureB]
```

This situation might happen also with three or more feature references.

Greater flexibility than IFDEFs The approach of interleaving of annotation markers provide greater flexibility to mark features than the notion of IFDEFs. IFDEFs set for their scope a concrete set of features and close them always all together. Usage of inner or simultaneous opening with individual closing is not possible as the IFDEF-precompiler cuts out the code parts and the inner parts are lost.

Example 1: Same begin with multiple ends

```
1  #define FEATURE_FIRST
2  #define FEATURE_SECOND
3
4  #ifdef FEATURE_FIRST &&
      FEATURE_SECOND
5  ...
6  #endif /* FEATURE_FIRST */
7  ...
8  #endif /* FEATURE_SECOND */
```

This setup causes a compiler error “#endif without #if” as #ifdef allows exactly one endpoint about its scope. I.e. multiple-ends or multiple starts are technically not possible.

Example 2: Partially overlapping feature code (lines 4-8 and lines 6-10)

```
1  #define FEATURE_FIRST
2  #define FEATURE_SECOND
3
4  #ifdef FEATURE_FIRST
5  ...
6  #ifdef FEATURE_SECOND
7  ...
8  #endif /* FEATURE_FIRST */
9  ...
10 #endif /* FEATURE_SECOND */
```

Disabling “FEATURE_SECOND” cause to early end of scope at first found #endif:

```
1  #define FEATURE_FIRST
2  //#define FEATURE_SECOND
3
4  #ifdef FEATURE_FIRST
5  ...
6  #ifdef FEATURE_SECOND
7  ...
8  #endif /* FEATURE_FIRST */
9  ...
10 #endif /* FEATURE_SECOND */
```

4.7 Feature Mappings

Embedded annotations are mainly used to annotate source code parts. For the mapping of whole files and folders including their subfolders, two kinds of mappings exist: feature-to-file and feature-to-folder.

4.7.1 Feature-to-code mapping

The feature-to-code mapping serves to link specific blocks and lines of code to one or more features. The parts of the source code which are mapped to a certain feature are called annotation scopes. An annotation scope is surrounded by annotation markers (see Chapter 4.6) and contains at least one feature reference (see Chapter 4.3).

4.7.2 Feature-to-file mapping

The feature-to-file mapping is a specialized file to map one or more file(s) and its/ their content to one or more feature references. All content of the linked file is considered fully to be part of the given feature references. The mapping file must be stored in the same folder as the source code files and covers only the file in this folder. This is especially helpful to map files that don’t contain source code, such as binary or generated files. The feature-to-file mapping is exclusive for files and

folders to be mapped to features by the feature-to-folder mapping - even when some operating systems handle files and folders identical (e.g. Linux).
In case the file name contains spaces or other special characters, the file name can be escaped with leading and ending quotation marks, e.g. “database config.dab”.

For projects with little applicability, or if the files for feature-to-file mapping wants to be avoided, the same effect of mapping the complete content of a file can be reached with a “begin”-annotation marker at the beginning of the file and its “end”-annotation marker counterpart at the end of the file.

The following syntax applies for the mapping file:

Alternative 1

```
File_a (File_b ...) <cr>
Feature_1 (Feature_2 ...) <cr>
File_x (File_y ...) <cr>
Feature_n (Feature_m ...) <cr>
<eof>
```

Alternative 2

```
File_a (File_b ...) <cr>
Feature_1 <cr>
Feature_2 <cr>
...
File_x (File_y ...) <cr>
Feature_n <cr>
Feature_m <cr>
...
<eof>
```

In concrete implementations, this filename could be _.feature-file or similar as defined for this project. To avoid that the file is hidden by the development IDE or file explorer the leading underscore is recommended.

For the file syntax in feature-to-file mapping two solutions with different focuses exist. On the one side, one feature to multiple files and on the other side one file to multiple features. The reasoning for one of these solutions could be given by the scattering degree (SD) and tangling degrees (TD). For this, the SD_{file} and TD_{file} would have been required. Based on the systematic literature review of El-Sharkawy et al. (2019) only two works consider both metrics: Hunsen et al. (2015) and Zhang et al. (2013). Hunsen et al. (2015) analyzed 41 systems and provided both values, but analyzing the system on #IFDEF-variability and not traceability. Zhang et al. (2013) provided in their results only ranges and makes it difficult to compare and as well on variability base.
As no clear tendency can be given the proposed feature-to-file mapping is kept flexible. It supports, therefore, a mapping with multiple files to one or more features and one file to one or more features. The only constrain given is that a file-line is followed by a feature-line and the file must start with a file-line.

Alternative 3 Instead of handling the feature-to-file mapping in an own file, the same result can be reached with the file-marker which should be placed at the file-header, best initial line.
The following syntax applies for the file-marker:

```
    //&file[ <parameter> ] <comment> <cr>
```

This marker considers the whole file to the given features in the parameter. The benefit is the independence of the specialized file and that with a copy/move the feature information is automatically considered. The drawback is that a feature mapping change is not independent possible of the file, e.g. using in another project or scope, and technically not possible for binary files.

The shown “Alternative 2” is dismissed for the implementation part. The reason for this is the challenge to handle files without a file extension, e.g. a file named “database”. The tool will not be able to detect if this is a file or feature. Therefore a safe mapping from files and features is not possible with this approach.
The shown “Alternative 3” is not considered in the implementation part. The reason for this is the late appearance of this option and that the functionality is available with the “Alternative 1”.

EBNF representation

```
⟨fileAnnotations⟩ ::= (⟨fileAnnotation⟩)*

⟨fileAnnotation⟩ ::= ⟨fileReferences⟩ ':'? '\n'+ ⟨lpqReferences⟩

⟨fileReferences⟩ ::= (⟨fileReference⟩ (' '* ⟨fileReference⟩)* ' '*)
                  |  (⟨fileReference⟩ (' '* ',' ' '* ⟨fileReference⟩)* ' '*)

⟨fileReference⟩ ::= ('' <fileName> '')
                  |  (⟨fileName⟩)

⟨fileName⟩ ::= ⟨STRING⟩
                  |  (⟨STRING⟩'.'⟨STRING⟩)

⟨lpqReferences⟩ ::= (⟨lpq⟩ (' '* ⟨lpq⟩)* ' '*)
                  |  (⟨lpq⟩ (' '* ',' ' '* ⟨lpq⟩)* ' '*)

⟨lpq⟩           ::= ⟨STRING⟩ ('::'⟨STRING⟩)*
```

(hint: STRING equals FEATURENAME in Grammar 4.2)

Grammar 4.5: EA, EBNF-Snippet of feature-to-file Mapping

4.7.3 Feature-to-folder mapping

The purpose of this file is to map complete folders and their content to one or more feature references. The mapping of feature references to folders allows linking specific features to the folder, including all its sub-folders and files. With this, the mapping of complete folder structures to features is possible and may substitute the feature-to-file mapping. The mapping file is located on the top level inside the to be annotated folder.
This way of the feature-to-folder has two main benefits. Firstly, the mapping takes care that always all containing artifacts are linked to the feature(s) and provides, therefore, more stability on a folder base than feature-to-file where the developer would need to maintain the file list. And secondly, the location inside the folder is

more stable against renaming or moving the folder than the feature-to-file mapping.

The following syntax applies for the mapping file:

Alternative 1

```
<LPQ> <cr>
<LPQ> <cr>
<eof>
```

Example:

```
Feature_1 <cr>
Feature_n <cr>
<eof>
```

Alternative 2

```
<LPQ> , <LPQ> <cr>
<eof>
```

Example:

```
Feature_1 , Feature_n <cr>
<eof>
```

In concrete implementations, this filename could be _.feature-folder or similar as defined for this project. It is stored inside the folder it annotates to resists better changes of the folder, such as renaming or moving.

EBNF representation

$$
\begin{array}{lcl}
\langle \textit{folderAnnotation} \rangle & ::= & (\text{' '}^{*}\ \langle \textit{lpq} \rangle\ (\text{' '}^{*}\ \langle \textit{lpq} \rangle)^{*}\ \text{' '}^{*}) \\
 & | & (\text{' '}^{*}\ \langle \textit{lpq} \rangle\ (\text{' '}^{*}\ \text{','}\ \text{' '}^{*}\ \langle \textit{lpq} \rangle)^{*}\ \text{' '}^{*}) \\
 & | & (\text{' '}^{*}\ \langle \textit{lpq} \rangle\ (\text{'\\n'}\ \langle \textit{lpq} \rangle)^{*}\ \text{' '}^{*})
\end{array}
$$

Grammar 4.6: EA, EBNF-Snippet of feature-to-folder Mapping

4.8 Embedded Annotation Examples

4.8.1 Annotation Code Examples

To illustrate the defined begin-, end- and line-markers, two real-world examples shall illustrate the usage.

```java
public static PaymentIntent fromBitcoinUri(final BitcoinURI bitcoinUri) {
    final Address address = bitcoinUri.getAddress();
    final Output[] outputs = address != null ? buildSimplePayTo(bitcoinUri.getAmount(), address) : null;

    final String bluetoothMac = (String) bitcoinUri.getParameterByName(Bluetooth.MAC_URI_PARAM); //&line[Bluetooth]

    //&begin[RequestCoins]
    final String paymentRequestHashStr = (String) bitcoinUri.getParameterByName("h");
    final byte[] paymentRequestHash = paymentRequestHashStr != null ? base64UrlDecode(paymentRequestHashStr) : null;

    //&begin[Codecs]
    return new PaymentIntent(PaymentIntent.Standard.BIP21, null, null, outputs, bitcoinUri.getLabel(), //&line[BIP21]
    //&end[Codecs]
            bluetoothMac != null ? "bt:" + bluetoothMac : null, null, bitcoinUri.getPaymentRequestUrl(),
            paymentRequestHash);
    //&end[RequestCoins]
240 }
```

Listing 4.1: Code example embedded annotations: Bitcoin-Wallet, class PaymentIntent. Adjusted from (Krüger, Mukelabai, et al., 2019).

Explanation of used markers and embedded annotations inside Listing 4.1:

- Line 228 “//&line[Bluetooth]” belongs to the feature Bluetooh and the line-marker matches exclusive line 228 to the feature Bluetooth.
- Line 230 “//&begin[RequestCoins]” belongs to the feature RequestCoins and the begin-marker maps lines 230 and following to the feature RequestCoins. This scope is ended with the “//&end[RequestCoins]” at line 239.
- Line 234 “//&begin[Codecs]” belongs to the feature Codecs and the begin-marker maps lines 234 and following to the feature Codecs. This scope is ended with the “//&end[Codecs]” at line 236.
- Line 235 “//&line[BIP21]” belongs to the feature BIP21 and the line-marker matches exclusive line 235 to the feature BIP21.

```
115 public class BlockchainService extends LifecycleService {
116     private WalletApplication application;
117     private Configuration config;
118     private AddressBookDao addressBookDao;  //&line[AddressBook]
        ...
139     private long serviceCreatedAt;
140     private boolean resetBlockchainOnShutdown = false;  //&line[ResetBlockChain]
        ...
157     //&begin[ResetBlockChain]
158     private static final String ACTION_RESET_BLOCKCHAIN = BlockchainService.class.
        getPackage().getName()
159             + ".reset_blockchain";
160     //&end[ResetBlockChain]
161     //&begin[BlockchainSync]
162     private static final String ACTION_BROADCAST_TRANSACTION = BlockchainService.
        class.getPackage().getName()
163             + ".broadcast_transaction";
164     private static final String ACTION_BROADCAST_TRANSACTION_HASH = "hash";
165     //&end[BlockchainSync]
166 
167     private static final Logger log = LoggerFactory.getLogger(BlockchainService.
        class);
```

Listing 4.2: Code example embedded annotations: Bitcoin-Wallet, class BlockchainService. Adjusted from (Krüger, Mukelabai, et al., 2019).

Explanation of used markers and embedded annotations inside Listing 4.2:

- Line 115 “//&line[AddressBook]” belongs to the feature AddressBook and the line-marker matches exclusive line 115 to the feature Bluetooth.
- Line 140 “//&line[ResetBlockChain]” belongs to the feature ResetBlockChain and the line-marker matches exclusive line 140 to the feature Bluetooth.
- Line 157 “//&begin[ResetBlockChain]” belongs to the feature ResetBlockChain and the begin-marker maps lines 157 and following to the feature Reset-BlockChain. This scope is ended with the “//&end[ResetBlockChain]” at line 160.
- Line 161 “//&begin[BlockchainSync]” belongs to the feature BlockchainSync and the begin-marker maps lines 161 and following to the feature Blockchain-Sync. This scope is ended with the “//&end[BlockchainSync]” at line 165.

4.8.2 File Mapping Examples

```
1  Bitcoin-wallet
2  |-- feature-model.cfr
3  |-- src
4    |-- de
5      |-- schildbach
6        |-- wallet
7          |-- data
8          |-- offline
9          |-- service
10         |-- ui
11         |-- util
12         |-- _.feature-file
13         |-- _.feature-folder
14         |-- Configuration.java
15         |-- Constants.java
16         |-- Logging.java
17         |-- WalletApplication.java
18         |-- WalletBalanceWidgetProvider
     .java
19             ...
```

Listing 4.3: Code example mapping files: Bitcoin-Wallet, adapted folder structure. Adjusted from (Krüger, Mukelabai, et al., 2019).

```
1  BitcoinWallet
2      Bluetooth
3      BitcoinBalance
4          DonateCoins
5          SendCoins
6      AppLog
```

Listing 4.4: Code example mapping files: Bitcoin-Wallet, feature-model.cfr. Adapted from (Krüger, Mukelabai, et al., 2019).

```
Logging.java
AppLog
WalletBalanceWidgetProvider.java
BitcoinBalance
```

Listing 4.5: Code example mapping files: Bitcoin-Wallet, _.feature-file. Adapted from (Krüger, Mukelabai, et al., 2019).

```
Main
```

Listing 4.6: Code example mapping files: Bitcoin-Wallet, _.feature-folder. Adapted from (Krüger, Mukelabai, et al., 2019).

Explanation of used mappings inside Listings 4.4 till 4.6. Listing 4.3 shows the folder structure for the mapping files.

- Listing 4.4 shows the feature hierarchy model (Folder structure line 2) with:
  - Line 1 the project name and root node for the hierarchy.
  - Lines 2 - 6 are the features in the project, three in the first and two in the second hierarchy level. Even when more features are used by the project, they are not required to be listed here. Nevertheless its encouraged
- Listing 4.5 shows the feature-to-file mapping (Folder structure line 12) of the files "Logging.java" (Folder structure line 16) mapped to feature "AppLog" and file "WalletBalanceWidgetProvider.java" (Folder structure line 18) mapped to feature "BitcoinBalance".
- Listing 4.6 shows the feature-to-folder mapping (Folder structure line 13) and therefore, inside folder "wallet". This means that the folder wallet and all its content is mapped to the feature "Main".

Fully annotated projects have been created by Krüger, Mukelabai, et al. (2019) and can be found at `https://bitbucket.org/rhebig/jss2018/`.

4.9 Evaluation Embedded Annotation Specification

The in Chapter 4 presented notation for embedded annotations is firstly designed with literature review and expert discussions, see methodology "Research objectives 1" in Figure 3.1. To evaluate the proposed notation an online survey with practitioners is hold. This survey has two main objectives: First, evaluate the acceptance and second, improve the current specification itself. Further outcomes are potential future research topics that are relevant for practitioners. The document reviewed is a compressed and standalone version of the embedded annotation design and therefore called "Embedded Annotation Specification"
The following sub-chapters cover the survey creation, execution, and results.

4.9.1 Survey Creation

The survey consists of two parts: A PDF containing the specification[3] and an online survey.
We define "Design Properties" in Chapter 4.1, each aiming to qualitatively evaluate one aspect of the embedded annotation design. To evaluate those properties, we design the survey, where each question asks about one design property. The target participants are industrial practitioners. For each property, the survey participant is asked to rate its opinion on a Likert scale and support its choice in a free-text field. The Likert scale range from "Completely Disagree" over "Disagree", "Neutral", and "Agree" to "Completely Agree". The free-text field allows to clarify the chosen rating and add further feedback to the survey. The survey continues with an open-ended question to the survey participant about its general opinion on how embedded annotations are beneficial and for specification improvement suggestions. The survey is closed with optional participants industry role and contact information.

The survey questions are:
"To which extent do you agree with the following statements?"

- The notation is useful.
- The notation is intuitive. (e.g. How natural it feels to use it)
- The notation is easy to learn.
- The notation is easily applicable.
- The notation is flexible to use. (imagine the contexts in which you want to use it, do you think it's flexible to use?)
- Using the annotation will avoid redundancies. (i.e. Requires writing more annotations than necessary for mapping assets to features)
- The notation is succinct. (i.e. the additional writing effort is minimal)
- The notion is robust during software evolution and maintenance. (i.e. as many annotations as possible survive the evolution, e.g. moving folders/files,

[3] `https://bitbucket.org/easelab/faxe/src/4548bac7575ae24c1438984e911324f405cb19b6/specification/Embedded_Annotation_Specification.pdf`

removing code, and editing code, etc.)
- The additional effort of using annotations during programming is negligible.
- It will be easy to convince developers to use it while programming.
- What do you think the biggest benefits of using embedded annotations are?
- Any suggestions for improvements?

"Participant information" (optional)
- Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)
- Please add your e-mail address

4.9.2 Survey Results

The survey was conducted in April and May 2020 as an online survey. The invited participants belong to three groups:
1. Industry contacts of the field of software product lines from Thorsten Berger as supervisor
2. Industry partner of thesis work
3. Industry contacts of Tobias Schwarz

For the survey in total 19 persons have been contacted and at least one person forwarded the survey invite in its organization. In total 10 persons replied to the survey, which makes a response rate of 53%. The survey was stopped after this number of participants as the qualitative answers of participants from different companies and backgrounds began to confirm each other and highlight the same factors. Due to the timeline and ongoing Corona pandemic in first half of 2020 the survey was stopped with the good amount of 10 filled survey.

Industry practitioners job roles (9 of 10 answered this optional question):

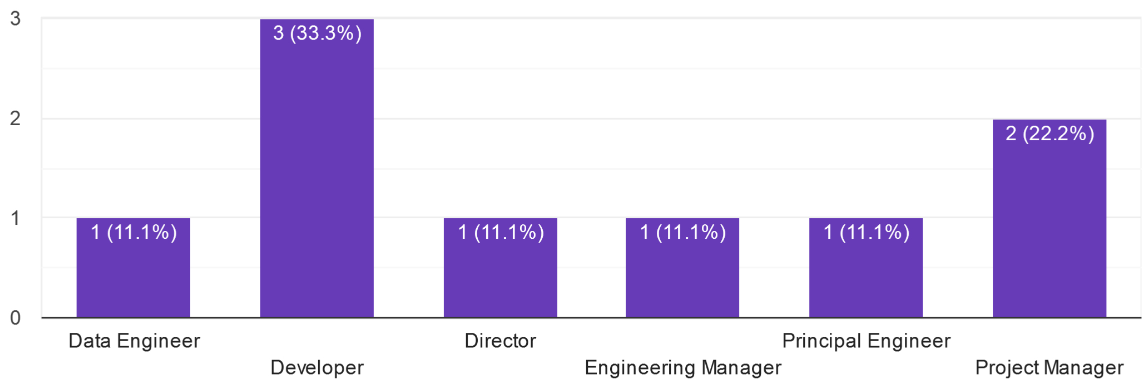


Figure 4.2: EA-Survey, Participants job titles

The summary of the means (Figure 4.3) shows for most design properties a rating between "Neutral" and "Agree". Outliers are the design properties "Effort" and "Convince". All design properties are discussed with their respective detailed results. The selectable range is rated for this analysis from (1) "Completely Disagree" over (3) "Neutral" till (5) "Completely Agree".

Design Property	Mean	SD
Useful	3,5	0,85
Intuitive	3,8	1,03
Learnable	4,2	0,79
Applicable	3,5	0,53
Flexible	3,7	0,95
Redundancy	3,5	0,85
Succinct	3,7	1,34
Robust	3,3	0,48
Effort	2,9	0,88
Convince	2,6	1,17

Table 4.1: EA-Survey, Mean and SD of all Design Properties

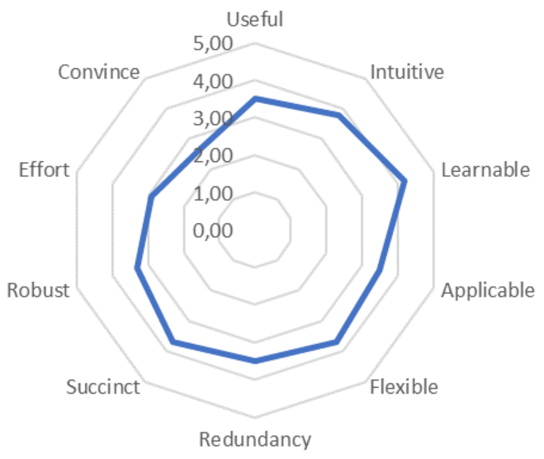


Figure 4.3: EA-Survey, Combined Mean values of design properties

With this result, the notation of embedded annotation has a solid foundation and is understood by practitioners. For an even more convincing result, the survey participants missed information about the return of investment, how teams collaborate with this technique, and tool support. All of which are factors beyond the definition of embedded annotations and belong more to the application of how embedded annotations are used.
The received survey results listed in Appendix B. They are unchanged, except for the removed participant's contact details.

In the following the individual survey questions are presented per question with statistical values of mean and standard deviation, an overall summary of the question in one to two sentences, a bar-chart with the results, and the summarized free-text answers of the participants.

The notation is useful

This question evaluates the design property "Usefulness". The results have a mean of 3.5 with a SD of 0.85 .
The notion can be seen in general as useful. The rationale of this is that different doubts have been seen about e.g. how meaningful for different programming languages it is or a clean code policy.

Annotating features in source code is seen as useful, especially as features and feature thinking is growing with the trends of agile software development concepts. Software projects are growing over time in complexity and team members join over the whole project duration. A project introduction can be supported with embedded annotations for increasingly large and complex development projects. This may be enriched with linking embedded annotations to requirements for general traceability purposes, such as e.g. asked by standards such as ASPICE.
Considerations are in the areas of clean code, i.e. which is intuitive to read and can be understood quickly (Wikimedia, Foundation Inc., 2019), and that embedded

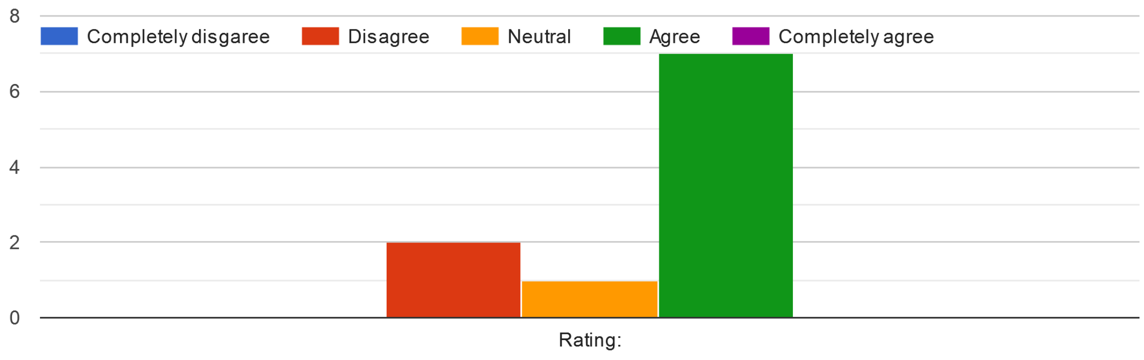


Figure 4.4: EA-Survey Results, The notation is useful

annotation might counteract against this. Besides that, there is the possibility to document features on the software architecture level and structure your source code by code generation from the created architecture. Which at the same time would also allow us to generate embedded annotations directly and support developers while their implementation.

The notation is intuitive

This question evaluates the design property "Intuitiveness (e.g. How natural it feels to use it)". The results have a mean of 3.8 and a SD of 1.03 .
The notion is seen in general as intuitive. The rationale of this is that especially the similarity to other established approaches, such as #ifdef's, support it.

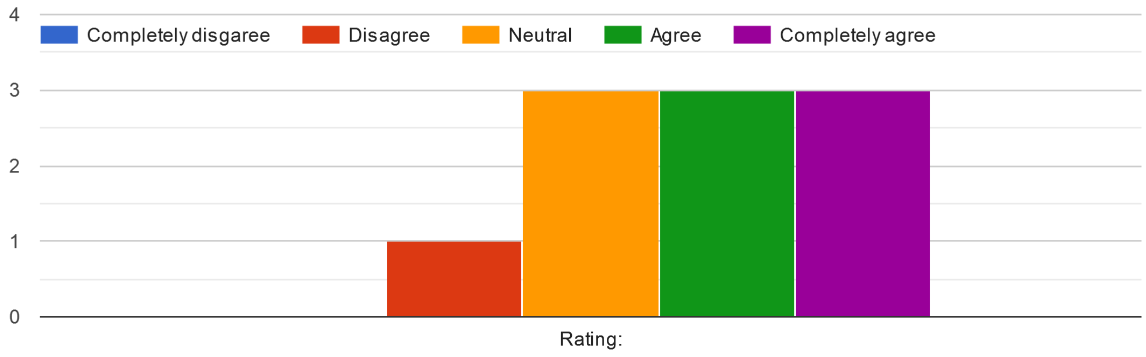


Figure 4.5: EA-Survey Results, The notation is intuitive

The used keywords and syntax are considered as intuitive to use and the proposed structure is appreciated. One doubt is that if embedded annotations are not part of the original syntax, reading and using them might be non-intuitive, especially in more abstract languages such as Python.

The notation is easy to learn

This question evaluates the design property "Easy to learn". The results have a mean of 4.2 and a SD of 0.79 .

The rationale of easy to learn is that with the given keywords and structure the notation is simple to use.

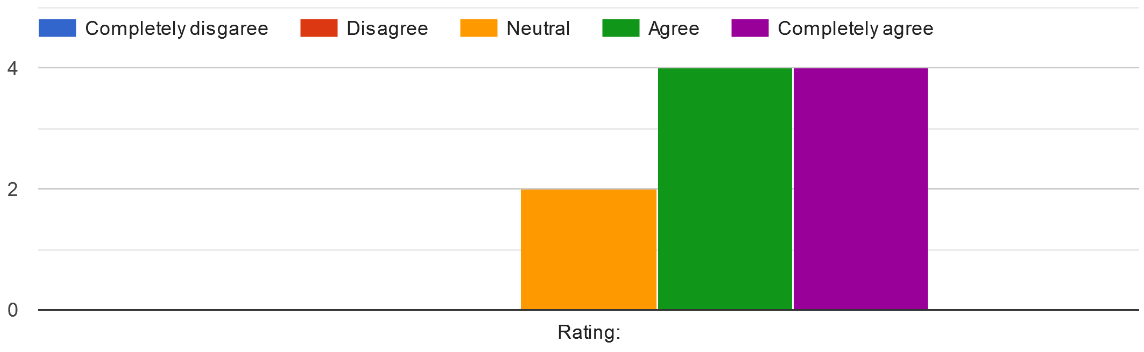


Figure 4.6: EA-Survey Results, The notation is easy to learn

Not only the source code annotations but also the feature-to-file and feature-to-folder mapping is seen as easy to learn. The challenge to learn embedded annotations is currently that there is no way to try them out in a ready to use environment.

The notation is easily applicable

This question evaluates the design property "Easy applicable". The results have a mean of 3.5 and a SD of 0.53 .
The rationale of the notion is highly applicable, but has at the moment a lack in tool support and potential team commitment.

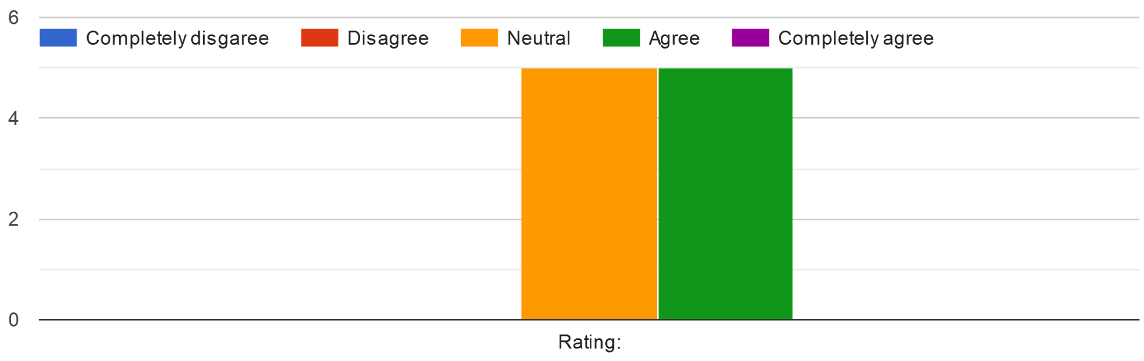


Figure 4.7: EA-Survey Results, The notation is easily applicable

The participants see the notation applicable to all stages of software project development. Best case is early in the project phase, but also for legacy projects or in maintenance phase could be e.g. with feature-to-file or feature-to-folder mapping a low-cost feature mapping be created.
For the usage of embedded annotations in project teams, team-agreements are necessary to commonly define the design and structures features and sub-features naming conventions as well as up to which level features shall be documented. Following such agreements as well as emphasizing their usage and finally benefit from them the support of tools is essential.

The notation is flexible to use

This question evaluates the design property “Flexible to use”. The results have a mean of 3.7 and a SD of 0.95 .
The notion has a positive trend for its flexibility. The rationale of this is that the survey participants have a lack in experience it.

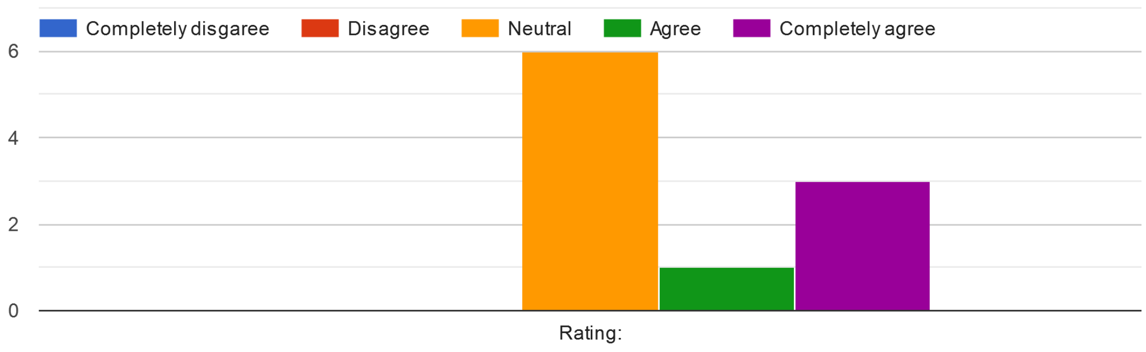


Figure 4.8: EA-Survey Results, The notation is flexible to use

It is recognized that the notation of embedded annotations is independent of the context and development itself and therefore can be applied independently to the actual context. Also, the level to annotate ranges from whole file structures (folder) over files till individual lines of text.
Considerations exist due to the lack to try out embedded annotations on a ready to use the environment as well as doubts on scalability and validity while future development and need to update them exist.

Using the annotation will avoid redundancies

This question evaluates the design property “Redundancy (i.e. Requires to write more annotations than necessary for map-ping assets to features)”. The results have a mean of 3.5 and a SD of 0.85 .
The rationale of this is that the notation is avoiding redundancies.

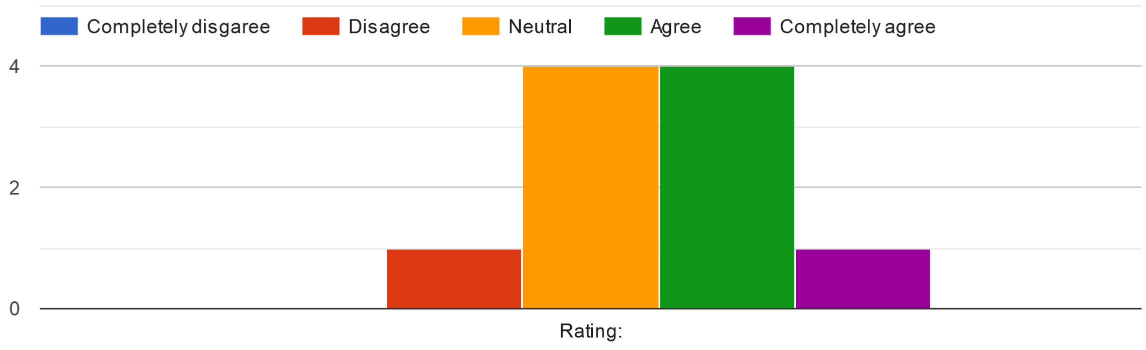


Figure 4.9: EA-Survey Results, The notation avoids redundancies

It is in general seen that embedded annotations avoid redundancies. Redundancy is saved on the one side with how embedded annotations are written and on the other side with the feature knowledge to reduce the risk to implement features twice.
The survey feedback shows that how the question was not fully clear compared to what or with which focus redundancy is avoided.

The notation is succinct

This question evaluates the design property "Succinctness (i.e. the additional writing effort is minimal)". The results have a mean of 3.7 and a SD of 1.34 .
The level of succinct is rated more diverse, with a positive trend. The rationale of this is that the notion is succinct, well-structured, and compact to write, but with the need for tool support.

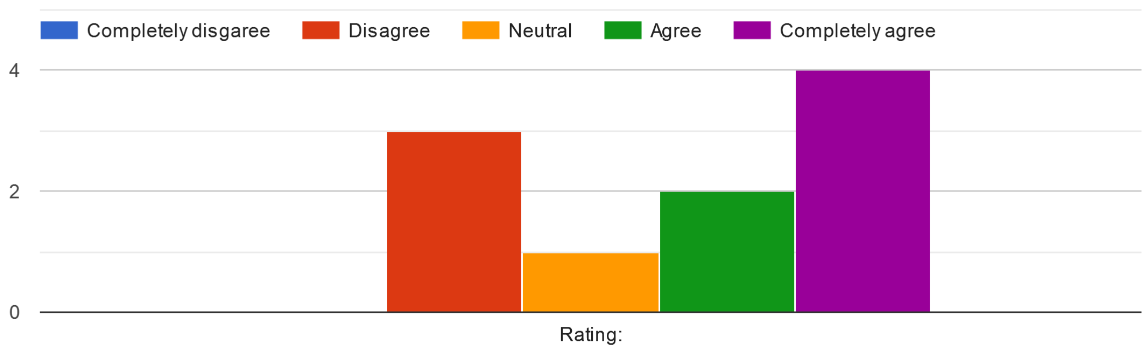


Figure 4.10: EA-Survey Results, The notation is succinct

The strength of the notation for embedded annotations is to act minimal invasive. This is recognized and a compact way to express embedded annotations. At the same time even with this succinct representation tool support is essential and might be in contrast with a clean code approach.

The notation is robust during software evolution and maintenance

This question evaluates the design property "Robustness (i.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code, etc.)". The results have a mean of 3.3 with a SD of 0.48 .
The robustness of the notation is seen as "Neutral" with a positive trend. The rationale of this is that the risk is in the different usage of annotations and lack of updating in the development and maintenance phase. Especially with project time pressure and missing refactoring tooling.

The participants see the notation of embedded annotation robust on all software artifact levels. Feature-to-folder annotations refer to the folder structure which is less likely to change over time and therefore by themselves more stable. Even in the case of renaming or moving to a new place. Feature-to-file are similar to folders, but are liable to refactoring operations such as renaming, moving or deleting. Since most changes happen in source code it is the part of embedded annotations that

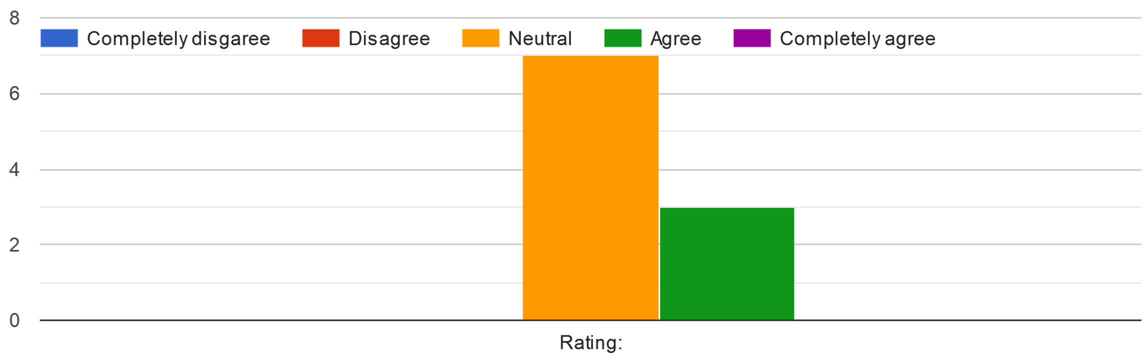


Figure 4.11: EA-Survey Results, The notation is robust

is least robust. Its level of robustness depends on how carefully developers handle them. E.g. in case of adding new content, copy parts of source code, or updating them while refactoring. One risk is that, especially at the end of a project, under pressure, the careful handling is skipped, and no later documentation happens.
For the level of robustness two factors interplay. On the one side the notation itself and on the other how it is used. The proper usage is especially important for large distributed teams, differences in working and cultural backgrounds as well as to be constant over time. For all drawbacks, tool support can reduce the risk of invalid states and remember updating the annotations.

The additional effort of using annotations during programming is negligible

This question evaluates the design property "Negligible effort". The results have a mean of 2.9 and a SD of 0.88 .
The rationale of this is that the additional effort to add embedded annotations is neither seen as given nor as not given.

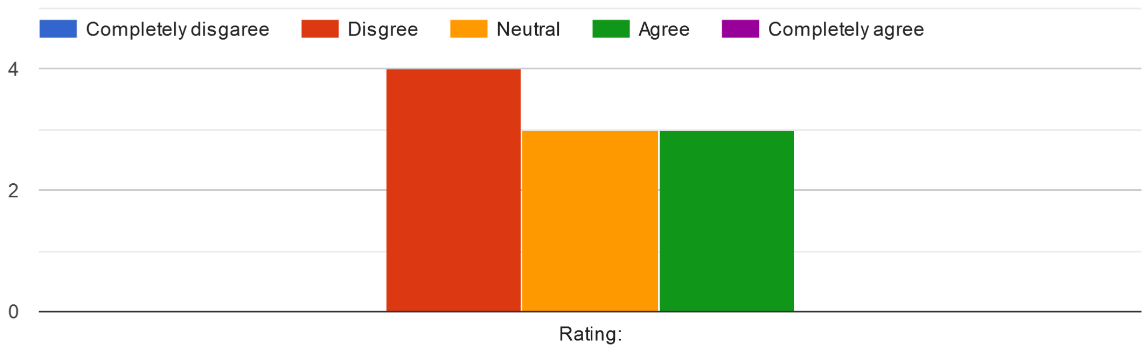


Figure 4.12: EA-Survey Results, The notation is cheap

With the different levels of the embedded annotation notation, the feature locations for folders and files could be created with low additional effort. For well-structured object-oriented-programming (OOP) projects a big part of the feature location documentation is seen to be annotated with this step. With a clean OOP approach,

the actual code annotations should be on a small number.
As for writing readable and understandable source code, using embedded annotations in the best way requires training and experience. Therefore, in the training phase, an extra effort might be required until embedded annotations become a natural habit.

It will be easy to convince developers to use it while programming

As the first additional question to the design properties, an estimation is asked how easy it is to convince developers in the survey respondent's environment to use embedded annotations as well. The results have a mean of 2.6 and a SD of 1.17 .
The rationale of this is that to convince other developers easily to use embedded annotations is rated mainly "Neutral" with a tendency that this is not the case. The survey participants see the lack in the existence of best practice as well as the required tool support to work easily with.

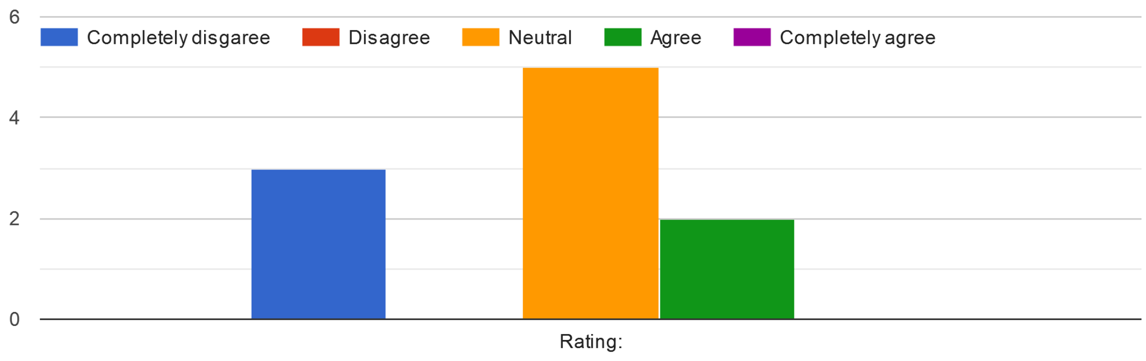


Figure 4.13: EA-Survey Results, The notation is convincing

The notation of embedded annotations is seen new to most practitioners. As there is no own experience or famous companies/tools using it, it is difficult to show the actual advantage to get accepted by developers. Another aspect is the project or product wide usage where guidelines and management support become important. Easy and reliable tolling is an essential part to convince further developers to adapt embedded annotations and apply them.
In the end, a developer or developer group must decide for their scope if the additional effort, independent how small, is worth to spend for more documentation.

What do you think the biggest benefits of using embedded annotations are?

There are several areas of benefits that embedded annotations provide. The first and foremost is the achievement of an overview of the implemented features in the project with a detailed result where to locate them. Applying them over a longer period allows the easier location of features for different tasks as well as supporting effort estimations and planning the implementation of new or changed features. The feature location may serve also for other purposes such as linking them to requirements or trace features for the use of report generation.

Feature location and quickly locating the affected source code part(s) is highly relevant to follow the agile methodologies as here changes over time in already implemented source code parts are likelier than in waterfall projects.

Any suggestions for improvements?

The last question is openly formulated and asks for feedback to improve the specification.

4.9.3 Design Changes

With the received and in Chapter 4.9.2 summarized results, changes in the embedded annotation design has been made. Some examples out of the survey which lead to changes in the specification:

- The introduction was expanded to provide a better understanding of the importance and general usage of embedded annotations.
- The terminology of Clafer was introduced as side note instead of using it in the chapter names
- Special character “<cr>” explained
- For the interleaving of annotation markers, the high flexibility was added to show the benefits to #ifdef notation
- The feature-to-file mapping received an own keyword to annotate file within their own source code

4.9.4 Outcomes

The provided feedback is integrated into the embedded annotation design and part of the latest version of the Specification of Embedded Annotations: **https://bitbucket.org/easelab/faxe/src/master/specification/Embedded_Annotation_Specification.pdf**
With the conducted survey, the notion of embedded annotations is empirically evaluated. For this the notion of embedded annotations is formalized and designed according to defined design properties. The basic foundation of embedded annotations is a summary of existing approaches of which this design is largely inspired.

4.10 Embedded Annotations Workflow and Usage

To receive the full picture, Figure 4.14 shows the workflow of handling embedded annotations and the potential benefits which a project may take out of them. This summary is created as last artifact in “Embedded Annotation Design”, see Figure 3.1. With the creation of this figure at this point in time, the knowledge of Chapter 2.9 “Related Work” and Chapter 4 “Embedded Annotations Design” run into it. The developer, step (1), is responsible to add the source code and embed-

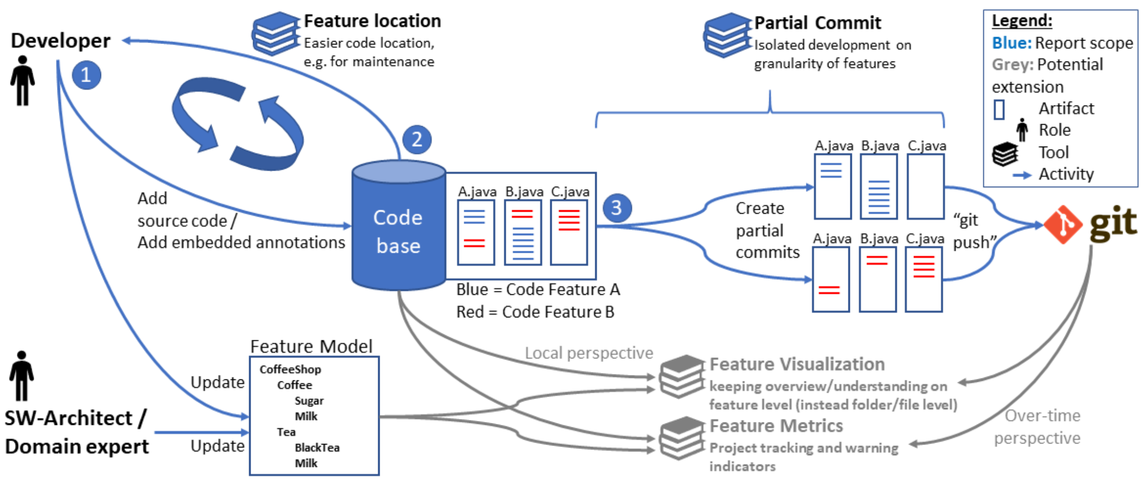


Figure 4.14: Embedded Annotation Workflow and Usage

ded annotations. While continuously developing new software, the added embedded annotations can be re-used for feature location immediately. This allows a closed feedback loop for the effort to add embedded annotations and benefiting of them. Tool support for feature extraction, step (2), can be given via tool support, e.g. FAXE - Feature Annotation eXtraction Engine[4].
Besides the actual development, the developer should be responsible to update the Feature Model. This work might be supported by SW-Architects and Domain experts.
With the knowledge of feature locations, several kinds of other benefits exist. One of the topics on which this work has great expectations is the process of partial feature-based commits. By knowing which parts of a local change belong to which features, isolated code commits can be created. This has e.g. the benefit when on multiple features or feature plus base code changes have been performed parallel and clean feature commits shall be created. This process starts with step (3).
Further, embedded annotations might be used for Feature Visualization (Andam et al., 2017; Entekhabi et al., 2019), i.e. keeping overview/understanding on feature level (instead folder/file level), or for Feature Metrics (Andam et al., 2017; Entekhabi et al., 2019), e.g. for project tracking and warning indicators. Both in taking a local perspective of the source code or based on an online git repository and enrich it with the git history.

[4]`https://bitbucket.org/easelab/faxe`

5 Engine for Embedded Annotation Extraction

In the conducted survey, results shown in Chapter 4.9, a major challenge was seen in proper tool support to use embedded annotations on the one side in the right way and on the other side to gain value from their existence. With the "Engine for Embedded Annotation Extraction" a tool support is created for the core part, the extraction of embedded annotations, out of a given project.
With the approach of a unified design for embedded annotations, it is possible to implement software, which refers to this design, and developers and tool producers can rely on the other's interfaces. This engine functions as the first software to support this unified design and shall allow clean and easy usage for further use.
This reference engine is based on the concept of embedded annotations introduced in Chapter 4 "Embedded Annotations Design" and its capabilities and structure are described in this chapter.
To support this, the foundation of the engine is on a domain-specific grammar, build with the ANTLR4 Parser Generator[1] and uses the in Appendix A.2 "EBNF Grammar Definitions" defined EBNF grammars. The engine itself takes the interpreted grammar, combines the grammar outputs and interacts with the user.
The engine itself is written in Java. The decision for Java is taken to support as many as possible industrial use-cases with one engine implementation. For easier referring and a succinct appearance the engine received the name "FAXE - Feature Annotation eXtraction Engine".
FAXE is hosted on Bitbucket: `https://bitbucket.org/easelab/faxe/`.

5.1 Parser Generator

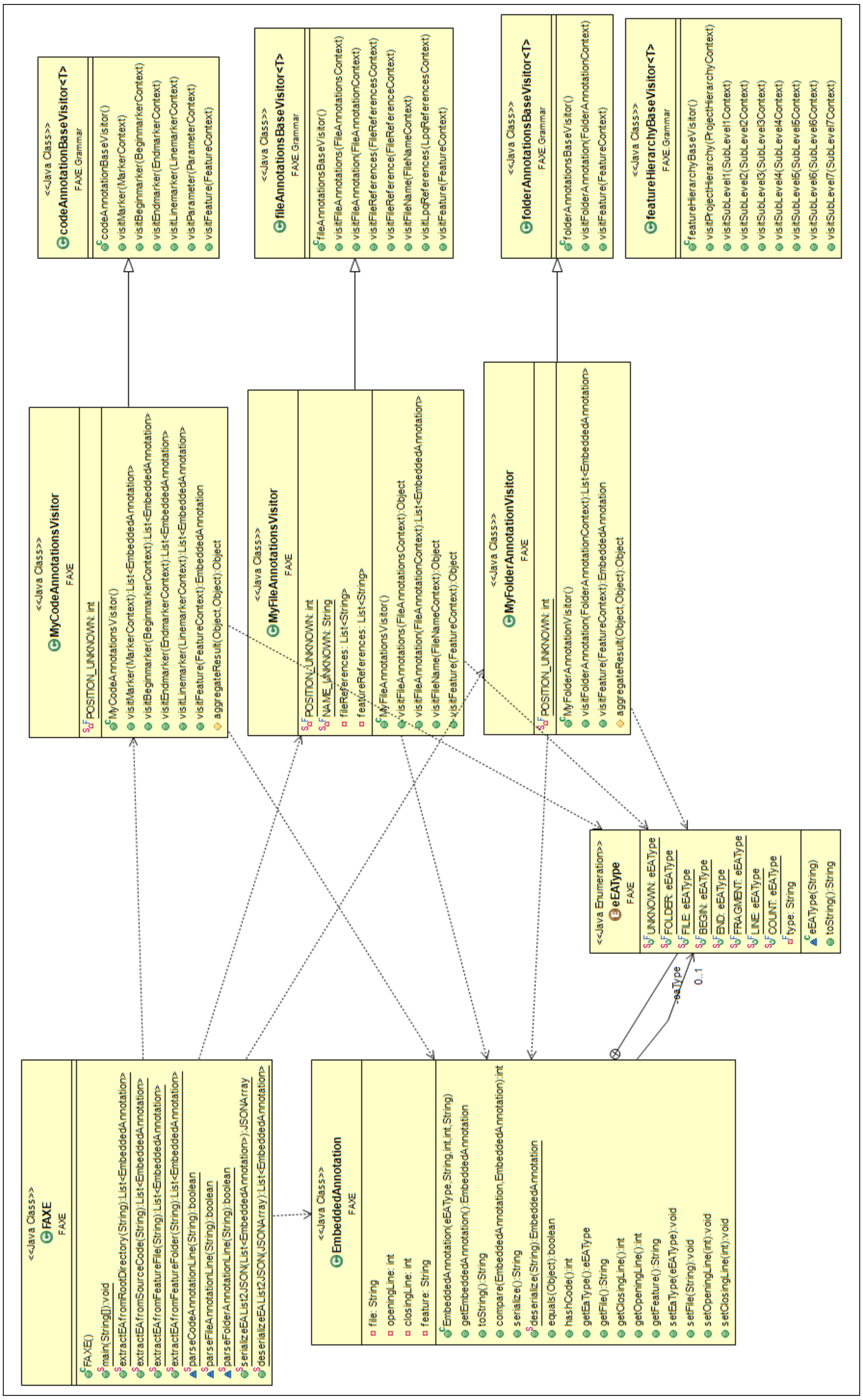
With the definition of the embedded annotations grammar in EBNF, a parser generator is needed to generate source code that can take the projects' source code and "parse" it for the given grammar pattern. The parser for embedded annotations is created with the tool ANTLR4, based on the EBNF grammars for code annotations, file annotations, folder annotations, and simple feature hierarchy. The with ANTRL4 generated parser receives the source code files and embedded annotation files, checks for the defined grammar, and creates an abstract syntax tree (AST). This AST can be accessed by the engine and the required information be taken out.

[1] `https://www.antlr.org`

The tooling used to extract keywords and their surrounding source code is usually using regular expressions (RegEx). RegEx are powerful and easy to use, but becomes difficult to use when searching regular expressions inside another regular expression (recursion) or the number of expressions is growing (scalability). (Tomassetti, 2017)

5.2 Engine Architecture

The engine contains five core classes, shown in Figure 5.1, from which three extend ANTRL4 generated files. The ANTRL4 generated files represent the entry point and consist of more files than shown here.
The class **FAXE** is the entry point into the engine and offers the engine-user methods to check a whole project for embedded annotations or a specific type (source code, feature-to-file, and feature-to-folder). The usage is designed in that way, that the user provides a specific file to analyze or a project path where the engine search iterates through all files and sub-directories. Providing a list of type EmbeddedAnnotations to the user. Inside the class **EmbeddedAnnotations** all necessary information are stored to identify a specific embedded annotation inside the project, i.e. its file, line position, and feature name. With the respective getters, the user can extract the for him/her relevant information. To classify the different types of embedded annotations, the enumeration **eEAType** list all types.
The layer between the publicly available methods and the generated ANTLR4 grammar is filled by the classes **MyCodeAnnotationsVisitor**, **MyFileAnnotationsVisitor** and **MyFolderAnnotationVisitor**. All these classes extend their respective generated BaseVisitor and override the methods where grammar data is extracted. For each visitor its in the grammar defined main rule, e.g. "visitMarker(...)" is called. Internally this triggers a search through the AST for this grammar. The extracted embedded annotations are collected, merged and returned as a list of embedded annotations.
The layer of generated ANTLR4 source code contains more functions that are overridden by the visitor-classes. The methods of the base classes represent all rules in the EBNF grammar and perform the actual detection of the grammar patterns. Figure 5.1 shows one more BaseVisitor than in the engine implementation, this is a preparation for future engine extension and not relevant for the following industrial use case. In the projects' repository unit tests for the engine's functionality and visitor files exist.



<<Java Class>>
codeAnnotationBaseVisitor<T>
FAXE.Grammar
codeAnnotationBaseVisitor()
visitMarker(MarkerContext)
visitBeginmarker(BeginmarkerContext)
visitEndmarker(EndmarkerContext)
visitLinemarker(LinemarkerContext)
visitParameter(ParameterContext)
visitFeature(FeatureContext)
<<Java Class>>
fileAnnotationsBaseVisitor<T>
FAXE.Grammar
fileAnnotationsBaseVisitor()
visitFileAnnotations(FileAnnotationsContext)
visitFileAnnotation(FileAnnotationContext)
visitFileReferences(FileReferencesContext)
visitFileReference(FileReferenceContext)
visitFileName(FileNameContext)
visitLpqReferences(LpqReferencesContext)
visitFeature(FeatureContext)
<<Java Class>>
folderAnnotationsBaseVisitor<T>
FAXE.Grammar
folderAnnotationsBaseVisitor()
visitFolderAnnotation(FolderAnnotationContext)
visitFeature(FeatureContext)
<<Java Class>>
featureHierarchyBaseVisitor<T>
FAXE.Grammar
featureHierarchyBaseVisitor()
visitProjectHierarchy(ProjectHierarchyContext)
visitSubLevel1(SubLevel1Context)
visitSubLevel2(SubLevel2Context)
visitSubLevel3(SubLevel3Context)
visitSubLevel4(SubLevel4Context)
visitSubLevel5(SubLevel5Context)
visitSubLevel6(SubLevel6Context)
visitSubLevel7(SubLevel7Context)
<<Java Class>>
MyCodeAnnotationsVisitor
FAXE
POSITION_UNKNOWN: int
MyCodeAnnotationsVisitor()
visitMarker(MarkerContext):List<EmbeddedAnnotation>
visitBeginmarker(BeginmarkerContext):List<EmbeddedAnnotation>
visitEndmarker(EndmarkerContext):List<EmbeddedAnnotation>
visitLinemarker(LinemarkerContext):List<EmbeddedAnnotation>
visitFeature(FeatureContext):EmbeddedAnnotation
aggregateResult(Object,Object):Object
<<Java Class>>
MyFileAnnotationsVisitor
FAXE
POSITION_UNKNOWN: int
NAME_UNKNOWN: String
fileReferences: List<String>
featureReferences: List<String>
MyFileAnnotationsVisitor()
visitFileAnnotations(FileAnnotationsContext):Object
visitFileAnnotation(FileAnnotationContext):List<EmbeddedAnnotation>
visitFileName(FileNameContext):Object
visitFeature(FeatureContext):Object
<<Java Class>>
MyFolderAnnotationVisitor
FAXE
POSITION_UNKNOWN: int
MyFolderAnnotationVisitor()
visitFolderAnnotation(FolderAnnotationContext):List<EmbeddedAnnotation>
visitFeature(FeatureContext):EmbeddedAnnotation
aggregateResult(Object,Object):Object
<<Java Enumeration>>
eEAType
FAXE
UNKNOWN: eEAType
FOLDER: eEAType
FILE: eEAType
BEGIN: eEAType
END: eEAType
FRAGMENT: eEAType
LINE: eEAType
COUNT: eEAType
type: String
eEAType(String)
toString():String
-eaType
0..1
<<Java Class>>
FAXE
FAXE
FAXE()
main(String[]):void
extractEAfromRootDirectory(String):List<EmbeddedAnnotation>
extractEAfromSourceCode(String):List<EmbeddedAnnotation>
extractEAfromFeatureFile(String):List<EmbeddedAnnotation>
extractEAfromFeatureFolder(String):List<EmbeddedAnnotation>
parseCodeAnnotationLine(String):boolean
parseFileAnnotationLine(String):boolean
parseFolderAnnotationLine(String):boolean
serializeEAList2JSON(List<EmbeddedAnnotation>):JSONArray
deserializeEAList2JSON(JSONArray):List<EmbeddedAnnotation>
<<Java Class>>
EmbeddedAnnotation
FAXE
file: String
openingLine: int
closingLine: int
feature: String
EmbeddedAnnotation(eEAType,String,int,int,String)
getEmbeddedAnnotation():EmbeddedAnnotation
toString():String
compare(EmbeddedAnnotation,EmbeddedAnnotation):int
serialize():String
deserialize(String):EmbeddedAnnotation
equals(Object):boolean
hashCode():int
getEaType():eEAType
getFile():String
getClosingLine():int
getOpeningLine():int
getFeature():String
setEaType(eEAType):void
setFile(String):void
setOpeningLine(int):void
setClosingLine(int):void

Figure 5.1: FAXE Engine Class UML Diagram

5.3 Public Interface Methods and Capabilities

As the design of embedded annotations, the engine's design is driven for intuitiveness, easy to learn, and flexible to use. These concepts have been taken to design the interfaces for public methods. The main goal is the seamless usage of the engine for users, to simplify their work and let them focus on the usage of embedded annotations than bothering to extract them.
The following methods are offered to the engine's users:

Method extractEAfromRootDirectory

```
public static java.util.List<EmbeddedAnnotation>
    extractEAfromRootDirectory(java.lang.String rootDirectory)
```

Method to extract embedded annotations from given root directory. The root directory and all sub-directories are checked for embedded annotations in source code, files and folders. In addition, the hierarchy file is analyzed.

Parameters:
rootDirectory - String of root directory.
Returns:
List of found embedded annotations.

Method extractEAfromSourceCode

```
public static java.util.List<EmbeddedAnnotation>
    extractEAfromSourceCode(java.lang.String fileToAnalyze)
```

Method to extract embedded annotations on source code level of given file.
Parameters:
fileToAnalyze - String of to be analyzed file.
Returns:
List of found embedded annotations.

Method extractEAfromFeatureFile

```
public static java.util.List<EmbeddedAnnotation>
    extractEAfromFeatureFile(java.lang.String fileUnderTest)
```

Method to extract embedded annotations on file level of given file.
Parameters:
fileUnderTest - String of to be analyzed file.
Returns:
List of found embedded annotations.

Method extractEAfromFeatureFolder

```
public static java.util.List<EmbeddedAnnotation>
    extractEAfromFeatureFolder(java.lang.String folderUnderTest)
```

Method to extract embedded annotations on folder level of given folder.

Parameters:
folderUnderTest - String of to be analyzed folder.
Returns:
List of found embedded annotations.

Method serializeEAList2JSON

```
public static org.json.JSONArray serializeEAList2JSON(java.util.
   List<EmbeddedAnnotation> eaList)
```

Transforms list of EmbeddedAnnotation to JSON object.
Parameters:
eaList - List of EmbeddedAnnotation
Returns:
JSON object out of parameter.

Method deserializeEAList2JSON

```
public static java.util.List<EmbeddedAnnotation>
   deserializeEAList2JSON(org.json.JSONArray jsonArray)
```

Transforms JSON object to list of EmbeddedAnnotation
Parameters:
jsonArray - JSON object
Returns:
List of EmbeddedAnnotation out of parameter.

5.4 Engine Usage Example

A project requires adding the engine JAR file into your project or import of it into your source code. Afterwards, the individual engine functions are one-line commands. A small example can be found on Bitbucket[2]. Extract of this example:

```
import FAXE.*;

public class FAXEMiniApplication {
   public static void main(String[] args) {
      ...
      List<EmbeddedAnnotation> eaList =
         FAXE.extractEAfromRootDirectory(projectRoot);
      List<EmbeddedAnnotation> eaListCode =
         FAXE.extractEAfromSourceCode(testFileCode);
      List<EmbeddedAnnotation> eaListFile =
         FAXE.extractEAfromFeatureFile(testFileFile);
      List<EmbeddedAnnotation> eaListFolder =
         FAXE.extractEAfromFeatureFolder(testFileFolder);
   }
}
```

[2] https://bitbucket.org/TobiasOnBitbucket/faxeminiapplication/

6
Industrial Use Case

The embedded annotations specification represents a document, created out of literature review as well as researchers' and practitioners' feedback. Based on that the reference engine was created. To show that this approach is valuable for industry, and to evaluate the FAXE-engine, several actual use cases will be shown (Chapter 6.1) and one of them implemented.

6.1 Potential Use Cases

A subtotal list of use cases which can be improved with the notion of embedded annotations. The fact which all these approaches are using is the knowledge about the feature location in source code.

Maintenance: To quickly locate the feature of interest while bug fixing and implementation work in an existing system.

Feature Metrics: With the knowledge about how features are distributed over the source code and how they interact allows the calculation of metrics such as scattering-, tangling- or nesting-degree.

Visualization: Extracting the feature locations and present them in a graphical way. This could be used e.g. to show which parts of the source code belong to a feature or how features are linked to others.

Version Control: Feature Metrics and Visualization itself would improve with the notion of embedded annotations. Using the version control system, in addition, would allow to calculate/show the evolution of the software over time and especially variants.

Partial Commits: Partial commits allow to commit a subset of changes inside one software project (Chapter 2.8). With the existing git tooling this is a process, requiring many manual steps. By knowing the feature locations inside a project, an automated way is possible, to decide if certain changes are part of a feature. With this the user interactions to perform a partial commit can be made more efficient.

With the industrial partner, one specific company from the area of web development, the potentials of embedded annotations and their software development challenges have been discussed. As embedded annotations where not used in this company so far, mainly the Version Control and Partial Commits options were discussed. The possibility to develop at the same time different features while keeping the commit history clean to specific feature commits, this option was chosen to be implemented. This decision is supported from research side due to its innovative character.

6.2 Use Case “Partial Commit”

The general idea of partial commit has been shown in Chapter 2.8 “Git Partial Commits”. Git provides to the user with “git add −−patch” (alternative ‘-p’) such a functionality with an interactive console and takes the user into the position to decide which parts of the source code shall be added to a commit and which not.
Figure 6.1 shows the workflow which a user must perform for each feature. First, the developer must go through all files and inspect the changes. Per change the user must decide if it shall be part of the partial commit or not. The by git proposed “hunk” blocks might have the need to be split and then added or rejected. After going through all files the user can perform a commit with fitting message and perform the same operation to collect data for the next commit. Finally, with a git push, the data is moved to the server’s git repository.

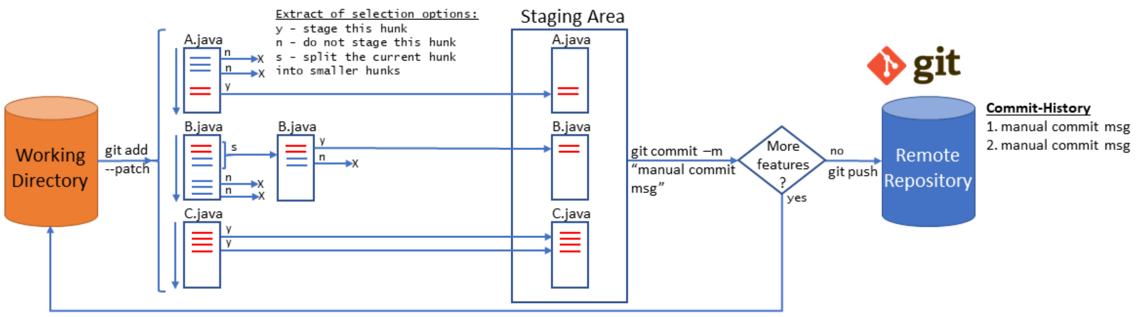


Figure 6.1: Partial Commit Workflow With Interactive Console “git add −−patch”

For the purpose of partial feature-based commits, a fully automated approach is targeted. Conceptually, the user provides the information which feature to commit and the tool takes to map the current software changes to the feature scopes. The precondition is that embedded annotations are present in the to be committed source code.
The application shall be able to accept individual feature names as well as a list of feature names. Latter is required to support product variants which contain a set of features.
A partial feature-based commit shall happen in the following steps:

1. Receive a list of local differences in changed, added and removed files, compared to the git repository
2. Receive list of embedded annotations from local project for a given feature
3. Determine overlapping elements between detected changes and given feature
4. Check if the staging area is empty and in case the user put something in there ask for permission to delete it.
5. Prepare source code in the staging area to perform partial feature commit (“git add”)
6. Perform git commit with either user commit message or tool-generated one

The goal of this application is on the one side to reduce the number of steps for the user to perform a partial feature-based commit and on the other side to avoid manual interactions in the process.

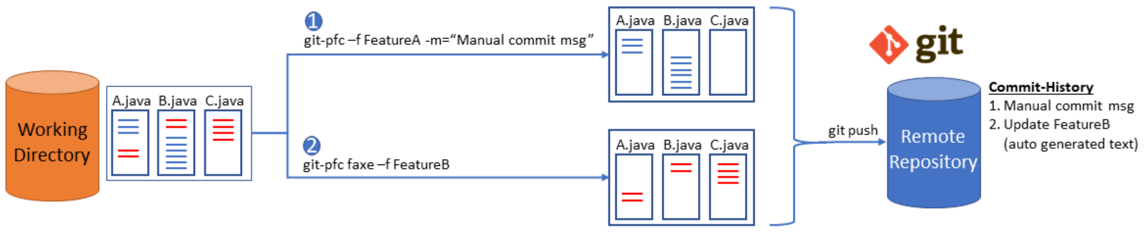


Figure 6.2: Git Partial Commit Workflow With Feature Focus With New Tooling

How the steps of partial feature-based commit work with Figure 6.2. As precondition the developer needs to have installed a git client and created or cloned a git repository. In this working directory the developer performs changes to two different features, namely "FeatureA" and "FeatureB". These changes happen in the working directory, shown in Figure 6.2 as orange cylinder object. In the box to the right of the working directory, the changes of "FeatureA" are shown in blue and the changes of "FeatureB" are shown in red. "FeatureA" and "FeatureB" have changes in files A.java and B.java while C.java contains only changes of "FeatureB".
The intention of partial feature-based commit is now to create own git-commits for "FeatureA" and "FeatureB". This is shown as with the numbers in blue circles

1. to perform a partial feature-based commit for "FeatureA" with a by the developer provided commit message (parameter '-m')
2. to perform a partial feature-based commit for "FeatureB"

For these commits, individually for partial feature-based commit (1) and (2), as Step 1 the changes of the local working directory, compared to the local repository (git version control data flow in Chapter 2.7) is detected. Step 2 takes care to identify all features based on their embedded annotations in the git project root folder and below. As now the two information, local changes, and feature positions, is available, these two lists can be compared in Step 3 and the changes be identified which belongs to the requested feature. Before adding the changes, which belong to the feature, the staging area must be cleaned to avoid manually added source code to be included, Step 4. Step 5 takes care to add all changes of this feature to the staging area, followed by the last Step 6 to perform a git commit and create a new commit in git.
After the developer has performed the wanted partial feature-based commit, the developer must git-push his commits to the Remote Repository, shown as blue cylinder. In the remote repository the commits are added and the individual partial feature-based commits appear as own elements in the "Commit-History".

6.3 State of the Art

To establish a body of knowledge, an online search has been conducted to find possible ways to implement the use case of partial commit. Thereby the search was on the one side conceptional and analyzed the potential solution tracks to follow, and on the other side tools, which offer a partial commit functionality for git. Best results have been made with searching for "git" plus "partial commit"/"partial staging". Most results are forum discussion where further hints are linked and lead to the following list.

The potential solution tracks are:

Shell Program that interacts with "git add -p" As an established and via console interaction available command exists, the first option is to create a tool that interacts with the by git provided tool for partial commits. The process of how to navigate through the menus is known and the structure of the answer message is well structured. Nevertheless, the number of potential scenarios and variances in answer messages is expected to be very high. Also, it remains one inherent issue with the definition of hunk blocks. The source code blocks are split into hunks, which can be further split down. This process is not always possible to break down hunks to individual lines of source code. Therefore, the risk of code blocks exists which contain feature and non-feature code, but which cannot be separated further with the tool.

Git integration libraries Different libraries exist, to control git repositories for different programming languages. E.g. libgit2[1] for C, JGit[2] for Java and GitPython[3] for Python. The benefit of such libraries is the smooth integration of git access and regular source code, such as a feature extraction library. All libraries have in common, that native support of the sub-option “−−patch” for “git add” is missing. Discussions are held for such a feature request and the latest one found is for libgit2 from 2012: `https://github.com/libgit2/libgit2/issues/591`.

Git Hosting Platforms Instead of interacting with the local git repository, one alternative is to interact directly with the git hosting platforms. Hosting platforms such as GitHub or GitLab provide APIs for other tools and websites to interact with them. GitHub provides an API and a CLI (Command Line Interface) tool, where both have the focus to work with GitHub repositories and issues, but not source code. The GitLab API also focuses on repository and issue interaction but allows as well to handle source code. The issue with handling source code directly with GitLab is the divergence of the bypassed local git repository and an additional challenge to handle.

Extension of git As the git tooling provides the partial commit option and as it is available as an open-source project, one option is to extend git itself. By providing a new parameter that triggers its own subroutine, this solution would have the highest integration. Besides the need to understand the extensive git source code and low-level version control handling, an issue is to make this solution available to others. Before the solution is not available in the mainstream git, anybody who would like to use it must replace his official git by the customized version.

For the git tools which are capable to support partial commits, some even on individual line level, in total six tools have been found:

Git-gui - `https://git-scm.com/docs/git-gui`

Git-gui is an in git by default integrated GUI tool to perform commit genera-

[1]`https://libgit2.org/`

[2]`https://www.eclipse.org/jgit/`

[3]`https://gitpython.readthedocs.io/en/stable/#`

tion. It is available as an open-source project, written in TCL, and supports by itself to stage up to individual lines for one commit.

Git-Cola - `https://git-cola.github.io/`
: Git-Cola is a standalone graphical tool to interact with git to stage commits and handle branches. It is available as open-source, written in Python and supports by itself to stage up to individual lines for one commit.

Egit - `https://www.eclipse.org/egit/`
: Egit is a set of Eclipse plugins written with the JGit library. Its source code is available and written in Java. The tool allows to edit the staging area directly and crosscuts with this the need to mark and add individual lines. But with this the inherent risk to add invalid code while manually editing the staging area.

Git-tower - `https://www.git-tower.com/`
: Git-tower is a comprehensive graphical interface to work with git. The tool is proprietary software and no source code is available. The tool allows staging individual lines.

git-crecord by andrewshadura - `https://github.com/andrewshadura/git-crecord`
: The first version of git-crecord is specialized to perform partial commits on the actual git-diff between the working directory and staging area. It is a command-line tool and allows to select individual lines of source code to include in a commit. The implementation is in Python and as open source available.

git-crecord by mbrendler - `https://github.com/mbrendler/git-crecord`
: The second version of git-crecord fulfills the same functionality as the first one. It is written in Ruby and available as open source.

6.4 Tool Design

The internal implementation of the tool for partial feature-based commits is done in Java with the JGit-library to interact with an online git repository. This Java implementation is linked with a custom git command (Bash script) to the local git installation and can be used as own git sub-command from the command line console.
As implementation internal git commands, basic git commands such as git-clone, -add and -commit have been used. The detection of changes happens between a copy of the working directory and the unmodified version of the repository.
The tool is integrated into the user space of local git commands and provides on command line level with a Bash script the functionality to the user. It supports these parameters:

```
$ git pfc -h
Usage: -f featureName -m "message to add to commit" -nc
   -f,--feature <arg>   Single feature to consider [MANDATORY]
   -m,--message <arg>   Text to be added to commit message [OPTIONAL]
   -nc,--no-commit      Suppress git-commit [OPTIONAL]
   -p, --print-embedded-annotations Prints available embedded
    annotations [OPTIONAL]
```

The Java implementation requires a further argument parameter which is automatically filled by the Bash script:

```
  -wd,--working-directory Path to git-folder (with .git) of project
```

6.5 Tool Architecture

The tool is implemented as a Bash script, forwarding the given parameters to the tool. The core implementation is done in Java in a single class. The internal flow of decisions and data transmission is shown in the following figure.

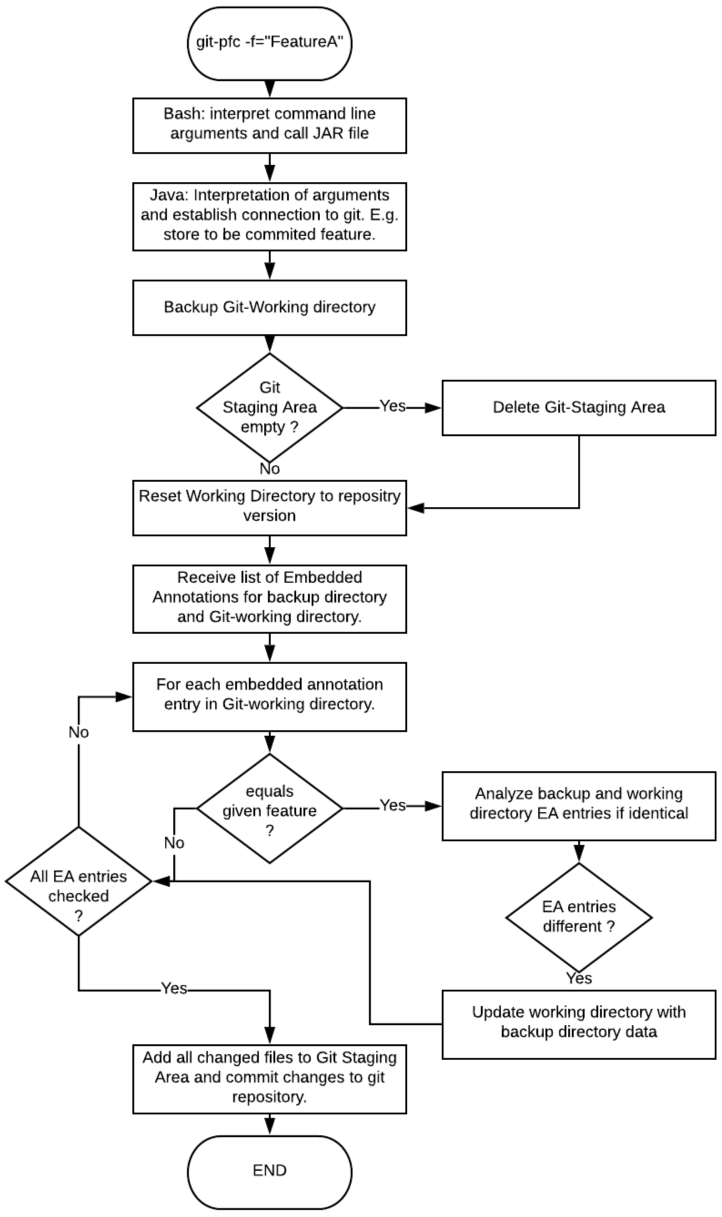

git-pfc -f="FeatureA"
Bash: interpret command line arguments and call JAR file
Java: Interpretation of arguments and establish connection to git. E.g. store to be commited feature.
Backup Git-Working directory
Git Staging Area empty ?
Yes
Delete Git-Staging Area
No
Reset Working Directory to repositry version
Receive list of Embedded Annotations for backup directory and Git-working directory.
For each embedded annotation entry in Git-working directory.
No
equals given feature ?
Yes
Analyze backup and working directory EA entries if identical
No
All EA entries checked ?
EA entries different ?
Yes
Yes
Update working directory with backup directory data
Add all changed files to Git Staging Area and commit changes to git repository.
END

Figure 6.3: Git Partial Feature Commit Tool - Flow Diagram

6.6 Partial Commit Limitations

With the final version of this tool, the handling of changes in existing &begin - &end annotation as well as existing &line annotation has been realized. Both work within the in Figure 6.3 described workflow and analyze the differences based on the embedded annotations lists. Adding and removing annotations is increasingly complex, especially when changes evolve over feature and non-feature source code. An example to visualize this situation is shown in Listings 6.1 and 6.2 and requires advanced diffing as the origin of a change cannot be traced back just by the feature information.
Partial commit in feature base is able to use with changing content of existing embedded annotations and practical experience can be taken out of this.

Present embedded annotations in Listing 6.1:

Name “FeatureA”
Begin 7
End 12
File “HelloPartial.java”

Present embedded annotations in Listing 6.2:

Name “FeatureA”
Begin 7
End 10
File “HelloPartial.java”

Name “FeatureA”
Begin 15
End 19
File “HelloPartial.java”

```
 1 public class HelloPartial {
 2
 3 public void main(String[] args) {
 4    methodA();
 5 }
 6
 7 //&begin(FeatureA)
 8 private void methodA(){
 9    System.out.println("methodA");
10    System.out.println("A Print");
11 }
12 //&end(FeatureA)
13
14 private void methodB(){
15    System.out.println("methodB");
16 }
17
18 private void methodC(){
19    System.out.println("methodC");
20 }
21 }
```

Listing 6.1: HelloPartial.java, Partial Commit Limitations Initial State

```
 1 public class HelloPartial {
 2
 3 public void main(String[] args) {
 4    methodA();
 5 }
 6
 7 //&begin(FeatureA)
 8 private static boolean var1;
 9 private static boolean var2;
10 //&end(FeatureA)
11 private static boolean var3;
12 private static boolean var4;
13 private static boolean var5;
14
15 //&begin(FeatureA)
16 private void methodA(){
17    System.out.println("methodA");
18 }
19 //&end(FeatureA)
20
21 private void methodB(){
22    System.out.println("methodB");
23 }
24
25 private void methodC(){
26    System.out.println("methodC");
27 }
28 }
```

Listing 6.2: HelloPartial.java, Partial Commit Limitations Final State

6.7 Tool Evaluation

The tool evaluation was initially contemplated to be in collaboration with the industrial partner, but due to the mid of 2020 ongoing pandemic a different evaluation approach had to be chosen.
By evaluating the tool for partial feature-base commits, research question 2 ("How can embedded annotations make an industrial use case more efficient?") of this work shall be answered. For this, the tool is demonstrated in different scenarios and each scenario is compared with the currently available techniques and the by this work created tool. The tool is publicly available under `https://bitbucket.org/TobiasOnBitbucket/partialfeaturecommitongit/` and can be used as one line commands within the git console.

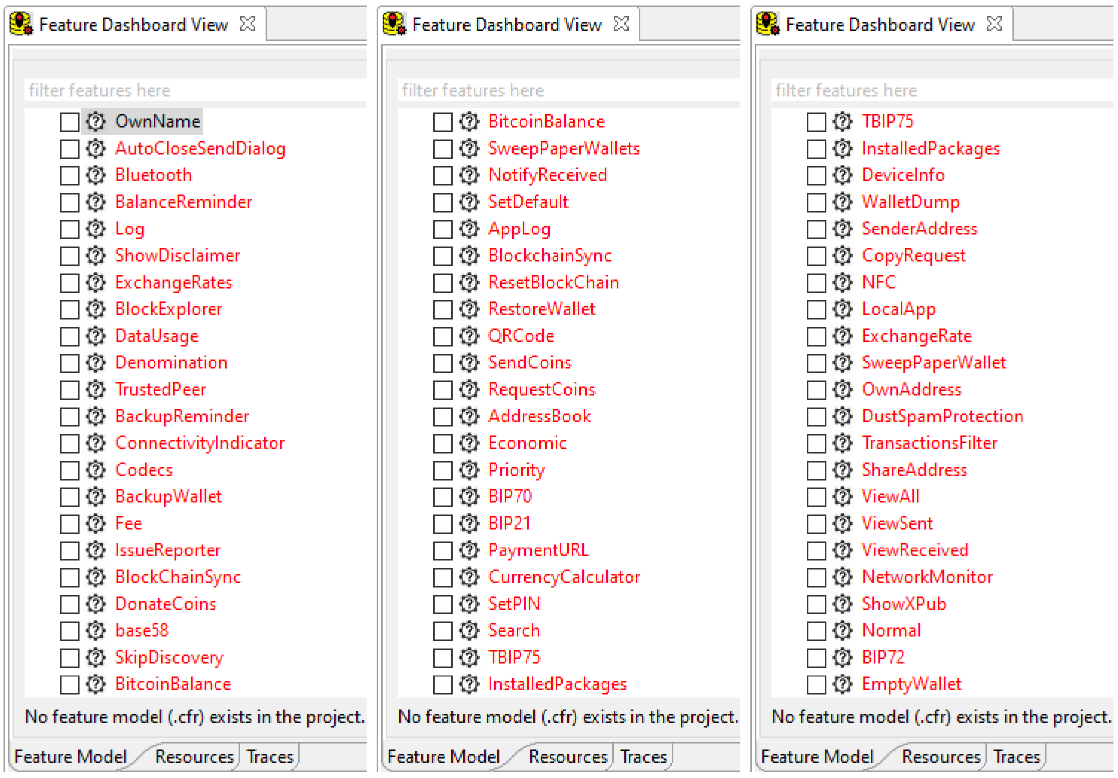
As the Design Properties for the Embedded Annotations Design, the scenarios base on the by Balzer and Goldman (1981) proposed principles of good software specifications: understandability, testability, and maintainability. With these basic principles and the by Ji et al. (2015) presented evolution patterns for embedded annotations, the scenarios have been selected and defined. With the in Chapter 6.6 introduced limitations of the current implementation, the selection of evolution patterns is limited. The scenarios are presented and discussed in this chapter and their execution is documented with screenshots in Appendix C.

The scenarios are:
1. Adding new assets to an existing feature and its unit tests
2. Evolution of source code in embedded annotation and base source code
3. Refactoring features - Structural change within a feature

Before the scenarios can be analyzed in detail, the reliability and scalability of the tool must be ensured. For this the results of FeatureDashboard (original dataset[4]) and FAXE (reworked dataset according specification[5]) are compared. The dataset has around 1500 embedded annotations and due to the rework of the &line annotation (from own line to behind the source code), the line numbers of following embedded annotations are changed. With these two factors, the sequence and feature in &begin/&end and &line embedded annotations are compared.

To extract the data from FeatureDashboard, the original dataset was cloned from its git repository to a local repository and imported to Eclipse with FeatureDashboard. In the "Feature Dashboard View" all features have been selected in "Feature Model" and all files and folders in "Resources". With the option "export traces as CSV format", the detected embedded annotations are extracted to a CSV file.
FeatureDashboard discovers in total 62 features:
The bash script for partial commit supports a print option for all available features in embedded annotations: "git-pfc -p". The outcome of this is a list of 63 features. This is one more feature than shown by FeatureDashboard because FeatureDashboard is

[4]https://bitbucket.org/rhebig/jss2018/src/master/Bitcoin-wallet/
[5]https://bitbucket.org/TobiasOnBitbucket/ea-bitcoin-wallet/src/master/Bitcoin-wallet/



OwnName
AutoCloseSendDialog
Bluetooth
BalanceReminder
Log
ShowDisclaimer
ExchangeRates
BlockExplorer
DataUsage
Denomination
TrustedPeer
BackupReminder
ConnectivityIndicator
Codecs
BackupWallet
Fee
IssueReporter
BlockChainSync
DonateCoins
base58
SkipDiscovery
BitcoinBalance

No feature model (.cfr) exists in the project.

BitcoinBalance
SweepPaperWallets
NotifyReceived
SetDefault
AppLog
BlockchainSync
ResetBlockChain
RestoreWallet
QRCode
SendCoins
RequestCoins
AddressBook
Economic
Priority
BIP70
BIP21
PaymentURL
CurrencyCalculator
SetPIN
Search
TBIP75
InstalledPackages

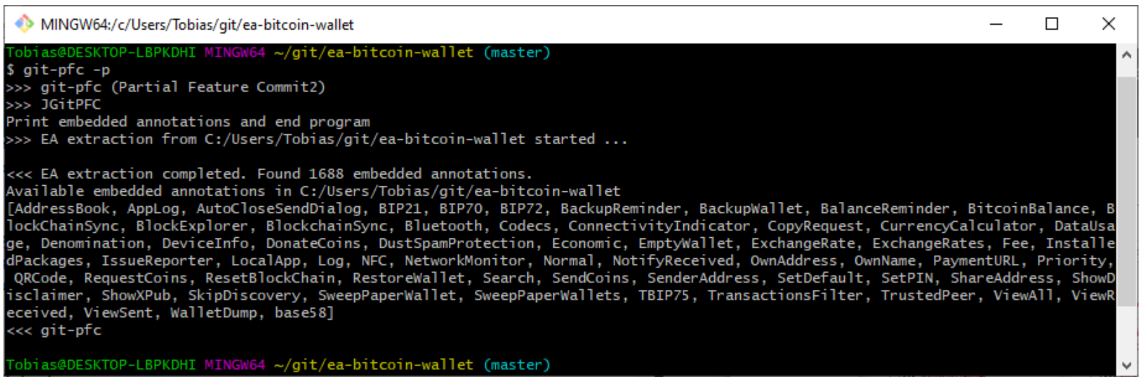
No feature model (.cfr) exists in the project.

TBIP75
InstalledPackages
DeviceInfo
WalletDump
SenderAddress
CopyRequest
NFC
LocalApp
ExchangeRate
SweepPaperWallet
OwnAddress
DustSpamProtection
TransactionsFilter
ShareAddress
ViewAll
ViewSent
ViewReceived
NetworkMonitor
ShowXPub
Normal
BIP72
EmptyWallet

No feature model (.cfr) exists in the project.

Figure 6.4: Tool Evaluation, Available Features in Project, FeatureDashboard

not case sensitive and merges the features "BlockchainSync" and "BlockChainSync".



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/ea-bitcoin-wallet (master)
$ git-pfc -p
>>> git-pfc (Partial Feature Commit2)
>>> JGitPFC
Print embedded annotations and end program
>>> EA extraction from C:/Users/Tobias/git/ea-bitcoin-wallet started ...

<<< EA extraction completed. Found 1688 embedded annotations.
Available embedded annotations in C:/Users/Tobias/git/ea-bitcoin-wallet
[AddressBook, AppLog, AutoCloseSendDialog, BIP21, BIP70, BIP72, BackupReminder, BackupWallet, BalanceReminder, BitcoinBalance, B
lockChainSync, BlockExplorer, BlockchainSync, Bluetooth, Codecs, ConnectivityIndicator, CopyRequest, CurrencyCalculator, DataUsa
ge, Denomination, DeviceInfo, DonateCoins, DustSpamProtection, Economic, EmptyWallet, ExchangeRate, ExchangeRates, Fee, Installe
dPackages, IssueReporter, LocalApp, Log, NFC, NetworkMonitor, Normal, NotifyReceived, OwnAddress, OwnName, PaymentURL, Priority,
 QRCode, RequestCoins, ResetBlockChain, RestoreWallet, Search, SendCoins, SenderAddress, SetDefault, SetPIN, ShareAddress, ShowD
isclaimer, ShowXPub, SkipDiscovery, SweepPaperWallet, SweepPaperWallets, TBIP75, TransactionsFilter, TrustedPeer, ViewAll, ViewR
eceived, ViewSent, WalletDump, base58]
<<< git-pfc

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/ea-bitcoin-wallet (master)
```

Figure 6.5: Tool Evaluation, Available Features in Project, FAXE

To receive an export from FAXE about the details of the embedded annotations (type, feature and location), it must be run from its source code with activated debug information. The created debug print was copied into a CSV file and both data sets have been cleaned for braces, leading project paths and '\' for easier comparison. For the dataset of FeatureDashboard in addition the embedded annotation type (LINE or FRAGMENT (&begin+&end)) is added. The final steps is to sort both data sets according to file name and line number.
The summary about the data shows that there are discrepancies how FeatureDash-

board and FAXE work at the moment. The discrepancies are for the type FRAGMENT and only for certain features. Also, the example data needs to be checked, as FAXE could detect some errors within this data, e.g. 36 &begin without a matching &end. Some discrepancies can be explained with this, but not the fact that FAXE found around 200 FRAGMENTS more than FeatureDashboard (see Table 6.1 and Table 6.2).

```
175 //&begin[PaymentURL]
176 public PaymentIntent(@Nullable final
        Standard standard, @Nullable final
        String payeeName,
177         @Nullable final String
        payeeVerifiedBy, @Nullable final
        Output[] outputs, @Nullable final
        String emo,
178         @Nullable final String
        paymentUrl, @Nullable final byte[]
        payeeData,
179         //&begin[RequestCoins]
180         @Nullable final String
        paymentRequestUrl, @Nullable final
        byte[] paymentRequestHash) {
181         //&begin[RequestCoins]
182     this.standard = standard;
183     this.payeeName = payeeName;
184     this.payeeVerifiedBy =
        payeeVerifiedBy;
185     this.outputs = outputs;
186     this.memo = memo;
187     this.paymentUrl = paymentUrl;
188     this.payeeData = payeeData;
189     //&begin[RequestCoins]
190     this.paymentRequestUrl =
        paymentRequestUrl;
191     this.paymentRequestHash =
        paymentRequestHash;
192     //&end[RequestCoins]
193 }
194 //&line[PaymentURL]
```

Listing 6.3: Bitcoin-Wallet, PaymentIntent.java, Bugs for embedded annotations in line 181 and 194

```
81 public Dialog onCreateDialog(final
       Bundle savedInstanceState) {
82     final Bundle args = getArguments();
83     //&begin[ShareAddress]
84     final Address address = (Address)
       args.getSerializable(KEY_ADDRESS);
85     //&begin[Codecs]
86     //&line[base58]
87     //&end[Codecs]
88     final String addressStr = address.
       toBase58();
89     final String addressLabel = args.
       getString(KEY_ADDRESS_LABEL);
90     //&end[ShareAddress]
91     ...
```

Listing 6.4: Bitcoin-Wallet, WalletAddressDialogFragment.java, Undocumented handling of lines 85 till 87

The results of the comparison of both tools look promising for the &line annotation and several embedded annotations but require future investigation. The original data is not changed to avoid working on the data until just a satisfying result is reached. The first ten lines of the compared data sets/tools (alphabetically sorted):

Embedded Annotations	Count
&line	605
FRAGMENT	835
	====
SUM	1440
ROWS	1440

Embedded Annotations	Count
&line	605
&begin	36
FRAGMENT	1047
	====
SUM	1688
ROWS in dataset	1688

Features	Count
AddressBook	80
AppLog	12
AutoCloseSendDialog	4
BackupReminder	10
BackupWallet	49
BalanceReminder	6
base58	81
BIP21	2
BIP70	14
BIP72	2

Table 6.1: Bitcoin-Wallet, Tool evaluation summary, FeatureDashboard

Features	Count
AddressBook	92
AppLog	12
AutoCloseSendDialog	4
BackupReminder	12
BackupWallet	57
BalanceReminder	6
base58	81
BIP21	6
BIP70	16
BIP72	2

Table 6.2: Bitcoin-Wallet, Tool evaluation summary, FAXE

6.7.1 Scenario 1 - Adding New Assets to an Existing Feature

This scenario covers changes in existing embedded annotations in the project's source code and its unit test. For this purpose, an example test project is used to avoid demonstration commits on a real project. The feature under test is "FeatureTestScenario1". Screenshots to this scenario can be found in Appendix C.1.
The source code for the program and unit test is changed in the following way:

```
public class HelloCommit {
    ...
  //&begin(FeatureA)
    private void commitC(){
        System.out.println("commitC");
    }
    //&end(FeatureA)
    ...

    //&begin(FeatureTestScenario1)
    protected boolean runTestScenario1(
    int i) {
      System.out.println("Run
    runTestScenario1 with i=" +i);

      commitC();

      if(i%2==0) {
        return true;
      } else {
        return false;
      }
    }
    //&end(FeatureTestScenario1)
}
```

Listing 6.5: PartialCommit-Testapplication, HelloCommit.java, Scenario 1 - Unmodified Source Code

```
public class HelloCommit {
    ...
  //&begin(FeatureA)
    private void commitC(){
        System.out.println("commitC");
    }
    //&end(FeatureA)
    ...

    //&begin(FeatureTestScenario1)
    protected boolean runTestScenario1(
    int i) {
      System.out.println("Run
    runTestScenario1 with i=" +i);

      if(i==0) {
        return true;
      }

      commitC();

      if(i%2==0) {
        return true;
      } else {
        return false;
      }
    }
    //&end(FeatureTestScenario1)
}
```

Listing 6.6: PartialCommit-Testapplication, HelloCommit.java, Scenario 1 - Modified Source Code

```java
import static org.junit.jupiter.api.
    Assertions.*;

import org.junit.jupiter.api.Test;

class HelloCommitTest {

  //&begin(FeatureTestScenario1)
  @Test
  void test1() {
    HelloCommit hello = new HelloCommit
    ();
    boolean retVal = hello.
    runTestScenario1(4);
    assertEquals(retVal, true);
  }
  //&end(FeatureTestScenario1)

}
```

Listing 6.7: PartialCommitTestapplication, HelloCommitTest.java, Scenario 1 - Unmodified Unit Test

```java
import static org.junit.jupiter.api.
    Assertions.*;

import org.junit.jupiter.api.Test;

class HelloCommitTest {

  //&begin(FeatureTestScenario1)
  @Test
  void test1() {
    HelloCommit hello = new HelloCommit
    ();
    boolean retVal = hello.
    runTestScenario1(4);
    assertEquals(retVal, true);
  }

  @Test
  void test2() {
    HelloCommit hello = new HelloCommit
    ();
    boolean retVal = hello.
    runTestScenario1(0);
    assertEquals(retVal, true);
  }
  //&end(FeatureTestScenario1)

}
```

Listing 6.8: PartialCommitTestapplication, HelloCommitTest.java, Scenario 1 - Modified Unit Test

To perform the partial feature-based commit the following steps are taken for the currently existing approach (git add) and the new tool approach.

Steps git-add:

1. Call “git add −−patch”
2. Decide for hunk in HelloCommit.java to be staged. Select ‘y’ (yes).
3. Decide for hunk in HelloCommitTest.java to be staged. Select ‘y’ (yes).
4. All changes are now in the staging area. Call “git commit -m “Scenario 1 manual”

Steps partial commit tooling:

1. Call “git pfc -f FeatureTestScenario1 -m “Scenario 1 with tool”

6.7.2 Scenario 2 - Evolution of Source Code in Embedded Annotation and Base Source Code

This scenario covers changes in existing embedded annotations plus the base source code (non-feature source code). Only the changes in the feature shall be committed and change outside is ignored. For this purpose, an example test project is used to avoid demonstration commits on a real project. The feature under test is “FeatureTestScenario2”. Screenshots to this scenario can be found in Appendix C.2.

The source code for the program and unit test is changed in the following way:

```java
public class HelloPartial {

    public void main(String[] args) {
        System.out.println("Make prints in different methods to simulate partial commits.");
        methodA();
    }

    //&begin(FeatureA)
    private void methodA(){
    System.out.println("New Method here1");
    System.out.println("New Method here2");
    System.out.println("New Method here3");
    System.out.println("New Method here4");
    }
    //&end(FeatureA)
    ...

    //&begin(FeatureTestScenario2)
    protected int runTestScenario2(int i, int j) {
      System.out.println("Run runTestScenario1 with i=" +i +" j=" +j);
      return i*j;
    }
    //&end(FeatureTestScenario2)

  //&begin(FeatureA)
  private void methodC1(){
        System.out.println("methodC");
    }
  //&end(FeatureA)
}
```

Listing 6.9: PartialCommitTestapplication, HelloPartial.java, Scenario 2 - Unmodified Source Code

```java
public class HelloPartial {

    //&begin(FeatureA)
    private void methodA(){
    System.out.println("New Method here1");
    System.out.println("New Method here2");
    System.out.println("New Method here3");
    System.out.println("New Method here4");
    }
    //&end(FeatureA)
    ...

    //&begin(FeatureTestScenario2)
    protected int runTestScenario2(int i, int j) {
      System.out.println("Run runTestScenario1 with i=" +i +" j=" +j);
      if(i==0||j==0) {
        return 0;
      }
      return i*j;
    }
    //&end(FeatureTestScenario2)

  //&begin(FeatureA)
  private void methodC1(){
        System.out.println("methodC");
    }
  //&end(FeatureA)
}
```

Listing 6.10: PartialCommitTestapplication, HelloPartial.java, Scenario 2 - Modified Source Code

To perform the partial feature-based commit the following steps are taken for the currently existing approach (git add) and the new tool approach.

Steps git-add:

1. Call “git add −−patch”
2. Decide for hunk (remove main method) in HelloPartial.java to be staged. Select ‘n’ (no).
3. Decide for hunk (change method runTestScenario2) in HelloPartial.java to be staged. Select ‘y’ (yes).
4. All changes are now in the staging area. Call “git commit -m “Scenario 2 manual”

Steps partial commit tooling:

1. Call “git pfc -f FeatureTestScenario2 -m “Scenario 2 with tool”

6.7.3 Scenario 3 - Refactoring Existing Structural Code Within a Feature

This scenario covers the refactoring of existing embedded annotations plus outside of it in the project’s source code. In this example, two features are to be committed. Also, both features are interleaving with each other (line 33). Both created commits shall only contain the changes of one feature. Therefore, the inner feature of the interleaving needs to be committed first. For this purpose, an example test project is used to avoid demonstration commits on a real project. The features under test are “FeatureTestScenario3” and “FeatureTestScenario4”. Screenshots to this scenario can be found in Appendix C.3.

The source code for the program and unit test is changed in the following way:

```java

public class HelloFeature {

  public static void main(String[] args) {
    int res1 = plus(1,2);
    double res2 = multi(5,3);
  }

  //&begin[FeatureTestScenario3]
  private static int plus(int i, int j) {
    int tmp1 = i;
    int tmp2 = j;
    int tmp = tmp1 + tmp2;
    return tmp;
  }
  //&end[FeatureTestScenario3]

  //&begin[FeatureTestScenario4]
  private static int minus(int i, int j) {
    int temp = 0;
    temp = i-j;
    return temp;
  }
  //&end[FeatureTestScenario4]
```

```java

public class HelloFeature {

  public static void main(String[] args) {
    int res1 = plus(1,2);
    double res2 = multi(5,3);
  }

  //&begin[FeatureTestScenario3]
  private static int plus(int i, int j) {
    return i + j;
  }
  //&end[FeatureTestScenario3]

  //&begin[FeatureTestScenario4]
  private static int minus(int i, int j) {
    int temp = 0;
    temp = i-j;
    return temp;
  }
  //&end[FeatureTestScenario4]

  //&begin[FeatureTestScenario3]
  private static double multi(double x,
     double y) {
    if(y==0) {
      return x*0;
    } else if (y==1) {
      return x*1;
    } else if (y==2) {
      x += Math.random()*2; //&line[
    FeatureTestScenario4]
      return x*2;
    } else if (y==3) {
      return x*3;
    } else {
      return -1;
    }
  }
  //&end[FeatureTestScenario3]

}
```

Listing 6.11: PartialCommit-Testapplication, HelloFeature.java, Scenario 3 - Unmodified Source Code

```java

  //&begin[FeatureTestScenario3]
  private static double multi(double x,
     int y) {
    switch(y) {
    case 0:
      return x*0;
    case 1:
      return x*1;
    case 2:
      x = x + Math.random()*2; //&line[
    FeatureTestScenario4]
      return x*2;
    case 3:
      return x*3;
    default:
      return -1;
    }
  }
  //&end[FeatureTestScenario3]

}
```

Listing 6.12: PartialCommit-Testapplication, HelloFeature.java, Scenario 3 - Modified Source Code

To perform the partial feature-based commit the following steps are taken for the currently existing approach (git add) and the new tool approach.

Steps git-add:

1. Call "git add −−patch" with target to commit "FeatureTestScenario4"
2. Decide for hunk (method plus) in HelloFeature.java to be staged. Select 'n' (no).
3. Decide for hunk (method multi) in HelloFeature.java to be staged. Select 's' (split) as hunk is defined over whole method.
4. Decide for hunk (method multi, first sub-hunk) to be staged. Select 'n' (no).
5. Decide for hunk (method multi, second sub-hunk) to be staged. Select 'n' (no).
6. Decide for hunk (method multi, third sub-hunk with "FeatureTestScenario4") to be staged. Select 's' (split) as hunk consists of two lines: one out of this feature and one not.

Steps partial commit tooling:

1. Call "git pfc -f FeatureTestScenario4 -m "Scenario 3 Feature FeatureTestScenario4 manual"
2. Add remaining changes which belong only to FeatureTestScenario3
3. Commit remaining changes which belong only to FeatureTestScenario3

7. Hunk cannot be split further. Limitations of “git add −−patch” reached and full manual approach required.
   (a) Save local copy of file HelloFeature.java
   (b) Reset HelloFeature.java to git repository version
   (c) Manual copy line-annotation of FeatureTestScenario4 to reset file
   (d) Perform “git add HelloFeature.java”
   (e) Perform “git commit -m “Scenario 3 Feature FeatureTestScenario4 manual”
   (f) Copy local copy of file HelloFeature.java back to the original file
8. Call “git add −−patch” with target to commit “FeatureTestScenario3”
9. Decide for hunk (method plus) in HelloFeature.java to be staged. Select ‘y’ (yes).
10. Decide for hunk (method multi) in HelloFeature.java to be staged. Select ‘y’ (yes).
11. All changes are now in the staging area. Call “git commit -m “Scenario 3 Feature FeatureTestScenario3 manual”

7
Discussion

The created solutions for an unified embedded annotation design (Chapter 4), feature extraction engine (Chapter 5), and partial feature-based commit (Chapter 6) show that the chosen approach can be used to realize them. For this research, the design science approach from Hevner et al. (2004) and Hevner (2007) was adjusted to the specific research needs. Difference to the original approach is the fact that for evaluation purposes, software implementations were used, which became afterward artifacts and needed to be evaluated themselves.

My answer to research question RQ1 "What can a unified and intuitive standard for embedded annotations look like?" is the in Chapter 4 provided the unified design of embedded annotations. The research of Ji et al. (2015), Andam et al. (2017), Entekhabi et al. (2019), and Krüger, Mukelabai, et al. (2019) has been analyzed in detail and the unified design been created over many weeks and iterations, discussing the benefits and drawbacks of the different solutions and balancing them. Several aspects with their benefits and drawbacks have been found, discussed and only the most promising variants added to the design in Chapter 4. Important is to conduct such discussions with persons with practical background and in collaboration with domain experts, such as my supervisor and co-supervisor, who know the field of feature development and have seen the usage of embedded annotations before. Especially the design of feature-to-file kept valid alternatives. The available options are either to create the mapping of feature-to-file via a specialized file or to create this mapping as code annotation in the file. Both solutions have thereby their strengths and further research is required to see if one alternative is preferable. The first alternative (specialized file) has the benefit to be able to annotate also non-source code files while renaming or moving the file requires an update in it. The second alternative (code annotation) has the benefit to resist file renaming and movement but does not apply to non-source code or generated files. Latter unless the code generator supports embedded annotations.
The created embedded annotation design is available in a shorted version as self-containing "Embedded Annotation Specification" and can be used independently of the overall research report. To use embedded annotations, the specification alone may serve as a starting point for technical persons and management, but without proper tooling, it is more unlikely that it will be implemented (confirming Andam et al. (2017)). With this unified design for embedded annotations, a central reference point is given and has the potential for tool development referring to it. This may avoid e.g. that derived tools diverge from each other (Entekhabi et al., 2019; Ji et al., 2015).

Software feature traceability is for this research a factor which accompanies constantly this work. Ji et al. (2015) showed how to store and maintain traceability, and this approach is central to the unified design. Against the lazy approach (Ji et al., 2015) to document features after the development or omit feature documentation we follow the eager approach in an as flexible as possible way to annotate source code and other software artifacts. Independent from the design of the embedded annotation itself, the usage of embedded annotations is a challenge itself and brought up as uniform usage in the survey and was discussed in the research of Ludwig et al. (2019) and Abukwaik et al. (2018) that feature annotations are used in the right way.

The factor of proper tool support is one of the key findings of the survey for the embedded annotation design. Based on Java, as one of the most used programming language, a ready-to-use engine to extract feature locations is such a tool.
Different to Entekhabi et al. (2019) this implementation bases on an EBNF grammar. The different types of embedded annotations are described in a generic way, which allows the grammar to be a core element for future non-Java implementations. The usage of grammar is thereby more robust than using regular expressions but at the same time, a different way of definition. Defining a language grammar has some specialties which programmers typically would tackle differently and makes it more difficult to define them without experience. Tomassetti (2017) background to use EBNF grammar, as well as to highlight some general challenges with them, is helpful to define such a grammar.
Core attributes of the engine are ease of use and reusability. The challenge thereby is to define interfaces and return structures in a way to fulfill all kinds of project types. With the help of practitioners and researchers applying the engine, it must be further evolved and updated. Keeping in mind not to overload the engine's interfaces and keeping the implementation stable and reliable. Such an evolution might take a long time and several version iterations.

My answer to research question RQ2 "How can embedded annotations make an industrial use case more efficient?" is provided with the use case of partial feature-based commits, Chapter 6.
Having the ready-to-use engine based on Java allows a larger range of projects to include it in their Java code or import it as Java-libraries into it. One of such implementations and a further tool is the created tool for performing partial feature-based commits.
Partial commit is a known, even when not well known, technique that is supported by Git itself. The new approach is to use the idea of partial commits for feature development to allow a feature-driven development and feature isolated commits. Which might allow projects to avoid feature-branch handling and the management of them.
A lesson learned while developing a tool to interact with Git is that command line git-commands and how a git source code library is used out of source code differs in several aspects. This is especially the case for commands with user interaction, such as "git add −−patch" or how data is received/interpreted in source code.

The tool for partial commit is an independent tool, i.e. not integrated into “git add” (Conservancy, 2020), and can keep with this its independent development from the main git development. It includes the FAXE engine for extracting embedded annotations and shows at the same time the possibility to perform partial feature-based commits and the beneficial usage of the engine.

Figure 4.14 (Chapter 4.10) has among other things shown some concrete potential use cases for embedded annotations. Beyond this list, further use cases are possible, such as feature management or synchronization. Another use case could be in the context of software product line or variant rich system to decide if a certain bug fix must be included in the own derived version.

8 Threats to Validity

With the decisions taken for this research and limitations in the potential scope of such a work, different limitations must be considered. The validity of the presented results depends on the as best as possible objective point of view taken for this analysis. In the following validity threats for *Construct validity*, *Internal validity*, *External validity*, and *Conclusion validity* are discussed.
It is not possible to avoid all threats, but important is to be aware of them and consider potential De-limitations. In general, to reduce the risk/effect of the limitations or at least decrease their impact, the following actions are taken. A regular meeting with the research group of the supervisor takes place where the progress is reported and the next steps are announced. The solutions will be available as open-source products which allow others to review and extend for their needs if necessary.

Construct validity This type of validity considers how the actual research questions have been addressed by this work and ensured to be answered.
First of all, the research questions have been used as core elements for the chosen methodology and based on this the results were presented. To ensure construct validity for RQ1 (unified design for embedded annotations) a set of design principles (Chapter 4.1) has been defined. Based on them the design description has been created and evaluated. For RQ2 (EA in industrial use case) a specific use case has been selected and scenarios have been defined to show the intended functionality.

Internal validity Internal validity covers the different factors, e.g. design principles, and how they relate to each other. When analyzing two factors the risk of an influencing third one is always present.
This threat mainly applies to the conducted survey. To avoid considering connections between the results, the complete survey results have been discussed with the supervisor and co-supervisor, which have high expertise in the domain.

External validity The goal of external validity is the generalizability of the results and to what extent they are valid in other technology/industry fields.
For RQ1 a survey with ten participants from different industries and in different job positions has been conducted. With the number of participants, one risk is to receive too specific requirements and therefore not being representative of the general field of software engineering. RQ2 has been selected based on the need of a specific company and the general lack in research for such tooling. Even when the topic of partial commits is supported by several tools, the limited knowledge and new way of working with features might limit

the applicability of the current solution. For the current situation, the tools have not been applied by practitioners, and by relating to one company, the integration into their IT setup and way of working might not be generalized.

Conclusion validity Conclusion validity covers the situations where researchers might see relations where none are present and miss relations which are present. This research has been discussed in detail between all project members and the supervisor's research group. Nevertheless, relations and bias is unavoidable, even with domain expert knowledge. For the survey in RQ1 the participants had the possibility to provide qualitative answers and show potential links and relations. As the boundaries for RQ2 are set, relations between this tool and others, or its environment, might exist. This is addressed in flexible interfaces and an extendable software architecture.

9 Conclusion

For this research, we used the eager approach to document proactive features in source code. The documented features are a good way to re-use them to trace feature locations. This feature traceability can be achieved with embedded annotations that co-evolve with source code.

Tracing of feature locations is a field in software engineering which will be more and more relevant for practitioners. Especially in safety-relevant fields, such as the certifying body USA-Federal-Aviation-Authority, or the Automotive SPICE process assessment, emphasizes is put on traceability between requirements, source code, and testing.
Niu et al. (2016) pointed out the purpose of traceability as a key factor for its success. The benefit of spending effort in traceability annotations must show early on and also, it must serve the person creating them. If this is not present, the creation and maintenance of traceability will not reach its full potential.
One of such techniques is the in this work analyzed notion of embedded annotations. Documenting features as close as possible to the software artifacts is an approach working on source code and specialized files and therefore known to developers. As embedded annotations are part of the source code, receiving a benefit from them is directly present for future work. With the situation of different definitions for embedded annotations, this work targeted and solved RQ1 "How can a unified and intuitive standard for embedded annotations look like?". The full design is present in this report and a compact form as a self-containing specification document. The evaluation of the design was conducted as an online survey with practitioners and confirmed thereby the general acceptance of embedded annotations.
Answering RQ2 "How can embedded annotations make an industrial use case more efficient?" is done in the context of partial commits. Considering the notion of embedded annotations, the created tool relies on extracting the locations of embedded annotations and performs a Git-commit on all changes for a given feature. The partial-commit-tool as well as the engine to extract embedded annotations (FAXE) are independent tools and can be used for practitioners and researchers in the future.

Beyond the concrete answers to the research questions of this thesis, it provides the scientific contributions to:

- Solves lack of unification of embedded annotations
- Empirically evaluation of unified embedded annotations design
- Feature location extraction engine FAXE on unified embedded annotation design

- Demonstration of FAXE on new use case partial feature-based commit for isolated feature development

This report closes the raised research questions, provides for embedded annotations a unified design, extraction engine, and based on them to perform partial feature-based commits. Potential future extensions, as well as future research, is shown in the next chapter.

10 Future Work

The results of this work may serve for a set of further research areas. Thereby touching a lot of topics which can be further developed by themselves or raise new research questions. This chapter discusses the most interesting questions to work on in the future directly or in an extended scope.

Embedded Annotations Specification - Level 2

For the Embedded Annotations Design, Chapter 4, two levels of embedded annotations are defined. Level 2 is considering in addition to logical operator expressions as well as supporting a Full Hierarchy Model and is not covered by the current results and implementations. A detailed definition of the possibilities and exceptions of embedded annotations with these elements will make the approach more flexible and allow an easier combination with existing annotations, such as #IFDEF.

FAXE engine extensions

While developing the FAXE engine to extract embedded annotations and apply it to the use case of partial commit, different fields of extension showed up. To learn more about the requirements which tools have on such an engine, it could be e.g. applied to existing open-source tools such as FeatureDashboard[1] or FeatureIDE[2]. Another internal field would be the implementation of configuration options for FAXE and to implement the interpretation of feature models. The current engine supports local directories. One next step might be the access to Git repositories and their history.

Extended embedded annotation language support

While there is a set of functionalities that can be added to the engine and extend it, the current version is written in Java. This limits the scope of projects where it can be used. Further research on how to extend this engine for a programming language flexible/independent version of itself is required.

Embedded Annotations without embedded annotations

Adding and maintaining embedded annotations requires to add markers (begin, end, line) into the source code. This additional content might be unwanted, e.g. with a

[1]`https://bitbucket.org/easelab/featuredashboard`
[2]`http://www.featureide.com/`

Clean Code Policy. Future research might be therefore directed into the field of how feature location could be done without embedded annotation markers. This could be e.g. reached with naming conventions for functions and variables.

Usage of embedded annotations outside the technical scope

The focus of this work is how embedded annotations are technical to be realized and can be extracted. A further step to investigate is the usage of how to benefit from this information outside the technical scope, e.g. in requirements engineering or project management. With the usage of embedded annotations for general project purposes, their usage might be enforced with this.

Tool support

One of the main outcomes of the conducted survey for the design of the embedded annotation is that without proper tool support, embedded annotations might not be used, or at least not to the extent they could. Tool development for embedded annotations might look at the following use cases:

- IDE plugin for automated embedded annotations completion
- Static analysis of broken annotation syntax
- Fail compilation if annotations are incorrect
- Support for refactoring, e.g. automation of feature file updates when files or folders are renamed/moved/deleted
- Recommender systems, e.g. to suggest feature annotations based on context and commits

Challenges for embedded annotations in larger development teams

The benefit of embedded annotations is only provided when they are added to the project and maintained over time. Besides the awareness, the different project members must work in a similar way in which level and how embedded annotations are used. This is especially for distributed teams a challenge. It becomes even more relevant when different cultures and focus teams, e.g. platform development and customer teams, come together. Further research is required to understand these needs and propose how to tackle this challenge.

Bibliography

Abukwaik, Hadil et al. (Aug. 2018). "Semi-Automated Feature Traceability with Embedded Annotations". In: DOI: `10.1109/ICSME.2018.00049`.

Andam, Berima et al. (2017). "FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces". In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-Intensive Systems.* VAMOS '17. Eindhoven, Netherlands: Association for Computing Machinery, pp. 100–107. ISBN: 9781450348119. DOI: `10.1145/3023956.3023967`. URL: `https://doi.org/10.1145/3023956.3023967`.

Apel, Sven et al. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated. ISBN: 3642375200.

Bąk, Kacper, Krzysztof Czarnecki, and Andrzej Wąsowski (2011). "Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled". In: *Software Language Engineering.* Ed. by Brian Malloy, Steffen Staab, and Mark van den Brand. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 102–122. ISBN: 978-3-642-19440-5.

Balzer, Robert and Neil Goldman (1981). "Principles of Good Software Specification and Their Implications for Specification Languages". In: *Proceedings of the May 4-7, 1981, National Computer Conference.* AFIPS '81. Chicago, Illinois: Association for Computing Machinery, pp. 393–400. ISBN: 9781450379212. DOI: `10.1145/1500412.1500468`. URL: `https://doi.org/10.1145/1500412.1500468`.

Berger, Thorsten (2019). *Software Engineering Principles for Complex Systems - Implementation -.* Course Material.

Bosch, Jan (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* USA: ACM Press/Addison-Wesley Publishing Co. ISBN: 0201674947.

Conservancy, Software Freedom (2020). *git-add.* URL: `https://git-scm.com/docs/git-add` (visited on 03/12/2020).

Entekhabi, Sina et al. (Sept. 2019). "Visualization of Feature Locations with the Tool FeatureDashboard". In: pp. 1–4. ISBN: 978-1-4503-6668-7. DOI: `10.1145/3307630.3342392`.

Hevner, Alan (Jan. 2007). "A Three Cycle View of Design Science Research". In: *Scandinavian Journal of Information Systems* 19.

Hevner, Alan et al. (Mar. 2004). "Design Science in Information Systems Research". In: *Management Information Systems Quarterly* 28, pp. 75–105.

Hunsen, Claus et al. (Apr. 2015). "Preprocessor-based variability in open-source and industrial software systems: An empirical study". In: *Empirical Software Engineering* 21. DOI: `10.1007/s10664-015-9360-1`.

Ji, Wenbin (2014). *Cost and Benefit of Embedded Feature Annotation: A Case Study.*

Ji, Wenbin et al. (2015). "Maintaining Feature Traceability with Embedded Annotations". In: *Proceedings of the 19th International Conference on Software Product Line*. SPLC '15. Nashville, Tennessee: Association for Computing Machinery, pp. 61–70. ISBN: 9781450336130. DOI: 10.1145/2791060.2791107. URL: https://doi.org/10.1145/2791060.2791107.

Krüger, Jacob, Gül Çalıklı, et al. (2019). "Effects of Explicit Feature Traceability on Program Comprehension". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, pp. 338–349. ISBN: 9781450355728. DOI: 10.1145/3338906.3338968. URL: https://doi.org/10.1145/3338906.3338968.

Krüger, Jacob, Mukelabai Mukelabai, et al. (2019). "Where is my feature and what is it about? A case study on recovering feature facets". In: *Journal of Systems & Software* 152, pp. 239–253. ISSN: 01641212. DOI: 10.1016/j.jss.2019.01.057. URL: http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=buh&AN=135661128&site=eds-live&scope=site&custid=s3911979&authtype=sso&group=main&profile=eds.

Liebig, Jörg et al. (May 2010). "An analysis of the variability in forty preprocessor-based software product lines". In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 1, pp. 105–114. DOI: 10.1145/1806799.1806819.

Ludwig, Kai, Jacob Krüger, and Thomas Leich (2019). "Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?" In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. SPLC '19. Paris, France: Association for Computing Machinery, pp. 218–230. ISBN: 9781450371384. DOI: 10.1145/3336294.3336296. URL: https://doi.org/10.1145/3336294.3336296.

Moody, D. (2009). "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35.6, pp. 756–779.

Niu, Nan, Wentao Wang, and Arushi Gupta (2016). "Gray Links in the Use of Requirements Traceability". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, pp. 384–395. ISBN: 9781450342186. DOI: 10.1145/2950290.2950354. URL: https://doi.org/10.1145/2950290.2950354.

Palmer, Steve R. and Mac Felsing (2001). *A Practical Guide to Feature-Driven Development*. 1st. Pearson Education. ISBN: 0130676152.

Passos, Leonardo, Krzysztof Czarnecki, et al. (2013). "Feature-Oriented Software Evolution". In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '13. Pisa, Italy: Association for Computing Machinery. ISBN: 9781450315418. DOI: 10.1145/2430502.2430526. URL: https://doi.org/10.1145/2430502.2430526.

Passos, Leonardo, Jesús Padilla, et al. (2015). "Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers". In: *Proceedings of the 14th International Conference on Modularity*. MODULARITY 2015. Fort Collins, CO, USA: Association for Computing Machinery, pp. 81–92. ISBN: 9781450332491. DOI: 10.1145/2724525.2724575. URL: https://doi.org/10.1145/2724525.2724575.

El-Sharkawy, Sascha, Nozomi Yamagishi-Eichler, and Klaus Schmid (2019). “Metrics for analyzing variability and its implementation in software product lines: A systematic literature review”. In: *Information and Software Technology* 106, pp. 1–30. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.08.015. URL: http://www.sciencedirect.com/science/article/pii/S0950584918301873.

Tomassetti, Gabriele (2017). *The ANTLR Mega Tutorial.* https://tomassetti.me/antlr-mega-tutorial/. Accessed: 2020-05-11.

Wikimedia, Foundation Inc. (2019). *Clean Code.* https://de.wikipedia.org/wiki/Clean_Code. Accessed: 2020-05-14.

Wikimedia, Foundation Inc. (2019). *Feature-driven development.* https://en.wikipedia.org/wiki/Feature-driven_development. Accessed: 2020-05-15.

Zhang, Bo et al. (2013). “Variability Evolution and Erosion in Industrial Product Lines: A Case Study”. In: *Proceedings of the 17th International Software Product Line Conference.* SPLC ’13. Tokyo, Japan: Association for Computing Machinery, pp. 168–177. ISBN: 9781450319683. DOI: 10.1145/2491627.2491645. URL: https://doi.org/10.1145/2491627.2491645.

A

Embedded Annotation Specification

A.1 Embedded Annotation Definition by Authors

Summary of definitions for embedded annotations by different authors. Out of these differences this work formulates a common definition.

	Feature Model		LPQ	Fragment Annotations			Feature-to-File		Feature-to-Folder	
Ji et al., 2015	Clafer	unspecified	Yes	&begin[LPQs] . . . (LPQ List)	. . . &end[LPQs]	. . . &line[LPQs]	fileName(,fileName)* featureName (,featureName)*	.vp-files	featureName (“ ”featureName)*	.vp-folder
Andam et al., 2017	Clafer	feature-model.cfr	Yes	&begin[LPQ] . . . (single LPQ)	. . . &end[LPQ]	&line[LPQ] . . .	featureName: fileName(,fileName)*	.feature-file	featureName: folderName (,folderName)*	.feature-folder in parent folder
Entekhabi et al., 2019	Clafer	__.cfr	No	&begin[LPQ] . . . (single LPQ)	. . . &end[LPQ]	&line[LPQ] . . .	featureName: fileName(,fileName)*	.feature-file	featureName (\n featureName)*	__.feature-folder
Krüger, Mukelabai, et al., 2019	Clafer	feature-model.cfr .vp-project	unspecified	&begin[Feature] . . . (single Feature)	. . . &end[Feature]	&line[LPQ] . . .	fileName featureName	__.feature-file .vp-files	featureName	__.feature-folder

Table A.1: Overview Embedded Annotation Concepts

A.2 EBNF Grammar Definitions

A.2.1 Feature Hierarchy Model Grammar

```
⟨SPACE⟩  ::= ' '* -> skip

⟨KEYWORDS⟩ ::= ('or'
           |  'xor'
           |  '?') -> skip // Skip Clafer Keywords

⟨projectHierarchy⟩ ::= ⟨FEATURENAME⟩ (⟨subfeature⟩)*

⟨subfeature⟩ ::= ('\n' '\t' ⟨FEATURENAME⟩) ⟨subsubfeature⟩*

⟨subsubfeature⟩ ::= ('\n' '\t\t' ⟨FEATURENAME⟩) ⟨subsubsubfeature⟩*

⟨subsubsubfeature⟩ ::= ('\n'          '\t\t\t'          ⟨FEATURENAME⟩)
           ⟨subsubsubsubfeature⟩*

⟨subsubsubsubfeature⟩ ::= ('\n'         '\t\t\t\t'         ⟨FEATURENAME⟩)
           ⟨subsubsubsubsubfeature⟩*

⟨subsubsubsubsubfeature⟩ ::= ('\n'       '\t\t\t\t\t'       ⟨FEATURENAME⟩)
           ⟨subsubsubsubsubsubfeature⟩*

⟨subsubsubsubsubsubfeature⟩ ::= ('\n'     '\t\t\t\t\t\t'     ⟨FEATURENAME⟩)
           ⟨subsubsubsubsubsubsubfeature⟩*

⟨subsubsubsubsubsubsubfeature⟩ ::= ('\n' '\t\t\t\t\t\t\t' ⟨FEATURENAME⟩)

⟨FEATURENAME⟩ ::= ([A-Z]+
           |  [a-z]+
           |  [0-9]+
           |  '_'+
           |  '\''+)+
```

Grammar A.1: EA, Full EBNF of Simple Hierarchy Model

A.2.2 Source Code Annotations Grammar

```
⟨marker⟩   ::= .*? (⟨beginmarker⟩
                | ⟨endmarker⟩
                | ⟨linemarker⟩)*

⟨SPACE⟩   ::= ’ ’* -> skip ; // ignores all more than one-time space characters

⟨beginmarker⟩ ::= ’&begin’ ’ ’* ⟨parameter⟩

⟨endmarker⟩ ::= ’&end’ ’ ’* ⟨parameter⟩

⟨linemarker⟩ ::= ’&line’ ’ ’* ⟨parameter⟩

⟨parameter⟩ ::= ’(’ ’ ’* ⟨lpq⟩ (’ ’+ ⟨lpq⟩)* ’ ’* ’)’ .*?
                | ’(’ ’ ’* ⟨lpq⟩ (’ ’* ’,’ ’ ’* ⟨lpq⟩)* ’ ’* ’)’ .*?
                | ’[’ ’ ’* ⟨lpq⟩ (’ ’+ ⟨lpq⟩)* ’ ’* ’]’ .*?
                | ’[’ ’ ’* ⟨lpq⟩ (’ ’* ’,’ ’ ’* ⟨lpq⟩)* ’ ’* ’]’ .*?
                | ’{’ ’ ’* ⟨lpq⟩ (’ ’+ ⟨lpq⟩)* ’ ’* ’}’ .*?
                | ’{’ ’ ’* ⟨lpq⟩ (’ ’* ’,’ ’ ’* ⟨lpq⟩)* ’ ’* ’}’ .*?
                | ’ ’* ⟨lpq⟩ (’ ’+ ⟨lpq⟩)*
                | ’ ’* ⟨lpq⟩ (’ ’* ’,’ ’ ’* ⟨lpq⟩)* ’ ’*

⟨lpq⟩       ::= ⟨FEATURENAME⟩ (’::’⟨FEATURENAME⟩)*

⟨FEATURENAME⟩ ::= ([A-Z]+
                | [a-z]+
                | [0-9]+
                | ’_’+
                | ’\”+)+ // restriction from Clafer and follow their definition

⟨OTHER⟩  ::= . -> skip // allows fuzzy parsing
```

Grammar A.2: EA, Full EBNF of Annotation Markers

A.2.3 Feature-to-File Annotations Grammar

```
⟨SPACE⟩   ::= ' '* -> skip // ignores all more than one-time space characters

⟨WS⟩      ::= [̊]+ -> skip

⟨fileAnnotations⟩ ::= (⟨fileAnnotation⟩)*

⟨fileAnnotation⟩ ::= ⟨fileReferences⟩ ':'? '\n'+ ⟨lpqReferences⟩

⟨fileReferences⟩ ::= (⟨fileReference⟩ (' '* ⟨fileReference⟩)* ' '*)
               |  (⟨fileReference⟩ (' '* ',' ' '* ⟨fileReference⟩)* ' '*)

⟨fileReference⟩ ::= ('' <fileName> '')
               |  (⟨fileName⟩)

⟨fileName⟩ ::= ⟨STRING⟩
               |  (⟨STRING⟩'.'⟨STRING⟩)

⟨lpqReferences⟩ ::= (⟨lpq⟩ (' '* ⟨lpq⟩)* ' '*)
               |  (⟨lpq⟩ (' '* ',' ' '* ⟨lpq⟩)* ' '*)

⟨lpq⟩       ::= ⟨STRING⟩ ('::'⟨STRING⟩)*

⟨STRING⟩ ::= ([A-Z]+
               |  [a-z]+
               |  [0-9]+
               |  '_'+
               |  '\"+)+ // -> restriction from Clafer and follow their definition
```

Grammar A.3: EA, Full EBNF of feature-to-file Mapping

A.2.4 Feature-to-Folder Annotations Grammar

```
⟨SPACE⟩  ::= ' '* -> skip // ignores all more than one-time space characters

⟨folderAnnotation⟩ ::= (' '* ⟨lpq⟩ (' '* ⟨lpq⟩)* ' '*)
                 |  (' '* ⟨lpq⟩ (' '* ',' ' '* ⟨lpq⟩)* ' '*)
                 |  (' '* ⟨lpq⟩ ('\n' ⟨lpq⟩)* ' '*)

⟨lpq⟩        ::= ⟨FEATURENAME⟩ ('::'⟨FEATURENAME⟩)*

⟨FEATURENAME⟩ ::= ([A-Z]+
                 |  [a-z]+
                 |  [0-9]+
                 |  '_'+
                 |  '\''+)+
```

Grammar A.4: EA, Full EBNF of feature-to-folder Mapping

B
Survey Data Evaluation Embedded Annotation Specification

This chapter contains the unmodified results of all survey participants. Starting with the given survey introduction and followed by sub-chapters per survey candidate. The only information omitted is the optional given contact email-address of the participant.

B.1 Survey of Embedding Annotations in Code

We are a group of researchers at Chalmers/Gothenburg University, studying the usage of lightweight techniques to trace features in source code. Knowledge of feature locations has several benefits. e.g. saving time, keeping an overview understanding of source code and connecting code to the business side.

The goal of this survey is to receive feedback on the proposed specification for embedded annotations for feature location in source code. Your feedback helps to improve the proposed design for embedded annotations in the industry and your own usage. The survey will take no more than 15 minutes.

The survey consists of two parts. The first part is to read the specification document which explains and illustrates the design of embedded annotations. In the second part, which is this questionnaire, we ask closed and open questions to evaluate the design of embedded annotations. We suggest to briefly skim through the questionnaire first.

Your data will be treated confidentially. Your ratings will only be published in aggregated form and any open-ended response will be shown in an anonymized way. We are pleased to send you a summary of the results of this survey.

Thank you for your contribution! If you have any questions please let us know.

Tobias Schwarz - `tobschw@student.chalmers.se`
Wardah Mahmood - `wardah@chalmers.se`
Thorsten Berger - `thorsten.berger@chalmers.se`

Computer Science and Engineering, Chalmers |University of Gothenburg
*Required

Our notation for embedded annotations

Please read the specification document:
https://drive.google.com/file/d/1gpSxDrXdW7vOxWc19WqHLaE5gUxzuzFZ/view?usp=sharing
To which extent do you agree with the following statements?

B.2 Feedback Participants

Feedback Participant 1

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

What do you think the biggest benefits of using embedded annotations are?

-

Any suggestions for improvements?

-

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

-

Feedback Participant 2

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

The set of keywords (begin, end, AND, etc.) are minimal and similar to other languages and easy to understand.

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

The simplistic design of notations allows developers to learn the framework easily

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

It's easy to apply notations in different stages of software development. Ideally the notations should be applied as early as possible in a project to obtain the maximum value feature locating brings when the project ages. However, even for legacy projects, I believe that only by spending some effort on feature-to-folder and feature-to-file mappings, it would help a great deal in reducing the efforts of locating features for developers, especially newcomers.

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

The ability of annotating down to source code level with logical expressions provides the maximum flexibility. I could not think of any uncovered use cases yet.

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

-

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

The convention is clear and concise so with a well-structured code base where features aren't interleaving one another, the effort of annotating is small.

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Feature-to-folder: since folders are less subjected for change, annotations will stay relatively consistent over time. If a folder is renamed/moved/deleted, the mapping must be updated accordingly. Without proper tooling, this might cause an integrity problem. I'd suggest that we keep the mapping in the same folder instead of from the parent folder as you do now. More specifically, a folder shall have an annotation file which defines all features included within this folder. Feature-to-file: similarly to feature-to-folder, it's subjected to the same problems. I'd suggest in a similar fashion, we keep the annotations on the file headers or class headers instead of a separate file which will be outdated the moment the file is renamed/moved/deleted.

An alternative to the above suggestion is additional tooling which detects discrepancy between the feature files and the actual files/folders. Lastly, feature-to-code is advantageous when the files are copied/moved as the annotations will be replicated. However, in case of source code modification, there will be cases where annotations will become invalid if there's no supported tooling around it. For example, one of the annotation "begin" or "end" is accidentally removed during refactoring. Another example is during refactoring, the annotations might be extracted into different methods without being noticed, as a result, the syntax might still be valid but the annotated piece of code no longer reflects the real feature.

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

It largely depends on the codebase. If it's written in a way that features are interleaving, extensively using feature annotations might require large effort to maintain, especially when it comes to feature-to-code mapping. However, I think it's even a sign of code-smell if one needs to employ feature-to-code at all. In general, feature-to-folder and feature-to-file should be enough in an object-oriented code base which strictly follows programming principals.

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Only when there's sufficient supported tooling, developers will incline to use and maintain feature annotations. Some examples are:

- Static analysis of broken annotation syntax
- Fail compilation if annotations are incorrect
- Automation of feature file updates when files or folders are renamed/moved/deleted
- Suggest feature annotations based on context and commits
- IDE plugins, git plugins or separate installable tools

What do you think the biggest benefits of using embedded annotations are?

The biggest benefit is linking requirements to code, thereby, speeding up the process of development. Nowadays, a lot of projects follow agile methodologies, requirements are usually incomplete or constantly changing over the course of the project. Now if there's a change in requirement, the affected code path will be located instantly.

Any suggestions for improvements?

I've written the suggested improvements in the sections above. I hope with additional tooling and some revision on the structure of feature-to-folder and feature-to-file, the framework will bring great benefit to the software industry.

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

Project Manager

Feedback Participant 3

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

Reading code is harder with additional comments. For bug-fixing readability of code is a must. Also I think with the key goals of identifying features on code level can be easily achieved via architecture and code generation from architecture. I have seen this in ASIL-D and aviation SW projects, where dead-code is strictly not allowed and features are most visible on code level.

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Many special characters are needed. I would not know how to grep them out of the source code.

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

I would need more examples in order to understand. Special cases, like nesting, partial feature distribution, variant handling which are common in my area -> i would not know how to start.

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Can't say without experience.

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

If there eas no feature handling before. This is a valid method to create it.

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

With references (feature annotations) breaking features might be harder on one hand, on the othe hand managing the annotations adds complexity to documentation, making the documentation phase longer. When pressure towards deadline hit is applied during a project, documenation is always the trade off -> see also agile manifesto.

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

Unless automated and somehow derived from existing function names -> No. One way to solve this issue could be to create a annotation language with a regular grammatic (chomsky L3) which has the source code as input alphabet, the language as the automat, and the annotations as output alphabet.

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	●	○	○	○	○

Please elaborate:

more documenation = more effort

What do you think the biggest benefits of using embedded annotations are?

Generating a feature Report out of source code, without architecture

Any suggestions for improvements?

show the complete annotation syntax in a table; show some regular expressions examples -> or how to create the feature reports compare it to architecture driven code creation (where features can be easily identified)

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

Developer

Feedback Participant 4

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

Useful, but perhaps not complete in all development environments. The variability describing the featuers might sometimes be outside of the source code, for example at checkout time (what to checkout), build time (perhaps this notation can be used there as well), load time, run time etc.

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Yes, but I don't think I fully understood the purpose of the clafer hierarchy from the document. Perhaps I just need more time to read about it.

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

Seams to be simple, but can't say for sure until I have used it myself.

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

It is hard say without testing it myself. For example I was thinking about features that spans many files and modules. Sometimes not everything is built at once. I don't know how flexible this is in this context. As the information is in comments, everything is lost at compile time.

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

I am not sure I understand this question.

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

-

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Maybe, but I think this depends on the develipment model, environemnt etc. For example, how to document versions of features? Are a feature always compatible with everything else independent of versions? Of course, the goal is usually yes (?), but in reality maybe not. Is it notion rubust to having information in other files than source code? For example recipies to the checkout system? I don't know.

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

I should be negligible, but I was thinking about the parallell traceability to requirements etc? Is this replacing that, or should/could it be used in parallell without creating ambiguities.

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

I think yes - at least if features are an important part of the development organization.

What do you think the biggest benefits of using embedded annotations are?

To keep the information about features close to the implementation itself. The disadvantage could be in those cases where you would like to separate problem- and solution space more. Maybe this is not a problem, I need to think about it.

Any suggestions for improvements?

Develop a small example of the complete development workflow using the notion (and not only the code examples)

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

Principal Engineer

Feedback Participant 5

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

-

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

-

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	●	○	○	○	○

Please elaborate:

In my opinion, this would require a lot of user-friendliness: e.g., support the annotations by means of coloring within an IDE like eclipse as depicted in the figures (maybe this already exists?), GUIs with file- and folder-choosing dialogs for your mappings, etc.

What do you think the biggest benefits of using embedded annotations are?

It offers arbitrarily fine-grain mappings

Any suggestions for improvements?

-

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

-

Feedback Participant 6

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

I wouldn't put in more comments or notation about features in the code if I didn't feel like I needed to, the code should be self explanatory from the start. I like python because of its simplicity/pseudo like syntax. Maybe a java developer could appreciate these notations more.

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

If it's not part of the original syntax I wouldn't say it's intuitive to use.

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Maybe, I would need to try it myself to give an honest opinion.

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

It's all about being agile nowadays so I don't see a developer using this if it's not very late iteration of a project that focus on reusability and robustness.

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

I haven't tried it so I can't say... But I think software updates will break the notation after a while.

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

I think it might be a redundancy to use it from the start.

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

I think adding any other code than necessary is bad practise.

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

I think it's safe to removing code and edit folders since any developer will hotreload his/her software after any changes.

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

-

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	●	○	○	○	○

Please elaborate:

If it's not part of the original syntax in java so I don't think it will be easy to convince developers to use it.

What do you think the biggest benefits of using embedded annotations are?

It may give a new developer a better chance to understand someone else's code.

Any suggestions for improvements?

I would want a clear example of what you mean by feature. "Feature A distinct functional or non-functional attribute of a software product.1" this quote is really abstract for a common developer. And since I'm in data science I associate feature with something else.

I would also shorten the questionnaire by at least 3 questions. Then I would feel more obliged to answer the questions more thoroughly.

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

Data Engineer

Feedback Participant 7

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

It's good to standardize feature annotations (i.e., have a common understanding).

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

It is close to what we know from #ifdef's. So, yes, it should be intuitive to the majority of (C) programmers.

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

Very limited set of keywords (i.e., should be easy to learn)

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Only with appropriate tool support (e.g., code completion). Otherwise, it could be cumbersome to add.

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

1. The level of granularity is not clear to me. Which AST elements can be annotated. From the examples I see that statements are supported. What about more fine-grained annotations such as individual switch cases, parameter declarations, etc.? What about elements that are non-optional in the AST, such as type and expressions?
If you just parse text: how do you enforce syntactic correctness?
2. What about feature combinations that are not possible according to the feature model (i.e., having two features selected in the annotation and then making them alternative to each other in the feature model)? I know we just talk notation here, but the question is whether there should be support for it? I don't know ;)

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Compared to what?

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

Only with appropriate tool support. Otherwise, I wouldn't use it.

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Hm, renaming files will break it. Also renaming features... Although this is not an issue with notation itself. For instance, if the notation is implemented in a projectional editor or in a textual editor with automated refactorings, we could avoid these issues. Nevertheless, I wouldn't say the "notation" is robust...

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	●	○	○	○

Please elaborate:

Cf. to answer for "The notation is succinct. *".

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

You could be more precise here. Who are the developers using it? Web developers, embedded software developers, etc.? The latter is very hard to convince (since #ifdefs are already in place).

What do you think the biggest benefits of using embedded annotations are?

Being able to trace your features is a big plus.

Any suggestions for improvements?

IMO there should be a more straightforward way for feature mappings, e.g., I don't understand why we need to distinguish files and folders. From a developer's perspective it should be easy to just include/exclude mappings of folders and files. Maintaining two files could be too cumbersome.

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

Engineering Manager

Feedback Participant 8

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

fast finding of features in the source code

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

depends on the discipline of the developer when assigning the parameter names

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

-

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

-

What do you think the biggest benefits of using embedded annotations are?

-

Any suggestions for improvements?

-

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

Director

Feedback Participant 9

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

It may especially be useful for linking SW requirements with corresponding code part, where such code parts spread over sources files and over source file contents.

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

-

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

-

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

This annotation concept looks very flexible and hence I would expect that a project planing to use it should initially agree e.g. schemes and structures of how to identify features, sub-features etc, up to which level, naming conventions etc. Also important seems to be the associated tool(s), which extract and process the annotations and represent the extracted information in a user friendly way.

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

-

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Not yet quite clear to me - the actual annotation syntax is very compact.

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

-

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

Much might depend on the discipline of the involved SW developers. In case of big and larger organizations and changing teams and responsibilities certain code parts are often maintained by different people with different experiences, know-how, ways of working, cultural differences and so on. The added value of the annotations might depend on a consistent usage over time and among different people as much as possible. Thus, it may depend on correspondingly chosen SW development process aspects and their consistent application. (This however is not a topic specific for this annotation scheme.)

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

I would assume that initially a "getting used to" process is required - SW developpers will have to keep in mind using the annotations consistently. This may be similar to the readiness to add to source code "sensible" and "readable" and "understandable" comments in general. Once a certain mindset is achieved and mentally accepted I indeed think that adding such annotations may become a "natural" habit.

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

Would be easy to convince me at least. :-)

What do you think the biggest benefits of using embedded annotations are?

If used and applied consistently over the lifetime of the related SW parts (from the begiining) and if related tool(s) are available to easily extract the annotated information and present it to the user, then the annotation scheme may indeed be really helpful in order to quickly identify source code parts associated with certain features. This may be especially helpful e.g. for bug fixing tasks, for effort estimation and implementation tasks related to new of changed features.

Any suggestions for improvements?

Have you already had a closer look at how the information, which is generated using this annotation scheme, may be represented to potential users, and how users then may actually work with it? This and potential initial experiences might be great information for future interested users of this scheme. Potentially also some best practice recommendations regarding feature granularity and structuring might round up the description.

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

Developer

Feedback Participant 10

The notation is useful. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

The annotations Importance increase with size and complexity of the development project

The notation is intuitive. *

(e.g. How natural it feels to use it)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

The annotation is based on in software development commonly used keywords and syntax. Hence, it is easy to learn and to work with. Furthermore it is compact so that it does not increase the length of the source code.

The notation is easy to learn. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

See answer above.

The notation is easily applicable. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

As single developer the annotation is simple to use. The applicability of the annotation to be used in multidisciplinary and distributed development team should be investigated.

The notation is flexible to use. *

(imagine the contexts in which you want to use it, do you think its flexible to use?)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

The annotation is independent of the context or development problem. Hence it can be applied in a large variety of software development projects.

Using the annotation will avoid redundancies. *

(i.e. Requires to write more annotations than necessary for mapping assets to features)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	●	○

Please elaborate:

It will not avoid redundancies by itself, but it can support the developer to identify those.

The notation is succinct. *

(I.e. the additional writing effort is minimal)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	○	○	●

Please elaborate:

The annotation syntax is very compact and is therefore efficient to be written.

The notion is robust during software evolution and maintenance. *

(I.e. as many annotations as possible survive the evolution, e.g. moving folders/files, removing code, and editing code etc.)

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

That depends a lot on the discipline of the developer. Here it is the risk that with changing source code the annotations get outdated.

The additional effort of using annotations during programming is negligible. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

I would say it is worth the effort, but it requires still time and concentration to write high quality annotations.

It will be easy to convince developers to use it while programming. *

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Rating:	○	○	●	○	○

Please elaborate:

Typically the first thing a developer does is to get the source code working. After this is achieved the developer should improve code quality and documentation. In practice this is the part which lacks on quality. The advantages of this method need to be experienced by the developers to get acceptance.

What do you think the biggest benefits of using embedded annotations are?

Achieving overview of the developed features in the project as well as motivating the developer to think about the written code in more detail resulting in improved code quality

Any suggestions for improvements?

It would be interesting to see the benefits of this method with respect to small-/medium/large development projects.

Participant information

We would ask you to provide your name and email for verification and analysis (e.g., to identify duplicates), to be informed about the study results, and in case of questions we might have. Hint: Your information is voluntary.

Please add your job title. (e.g. Developer, Tester, SW-Architect, Project Manager)

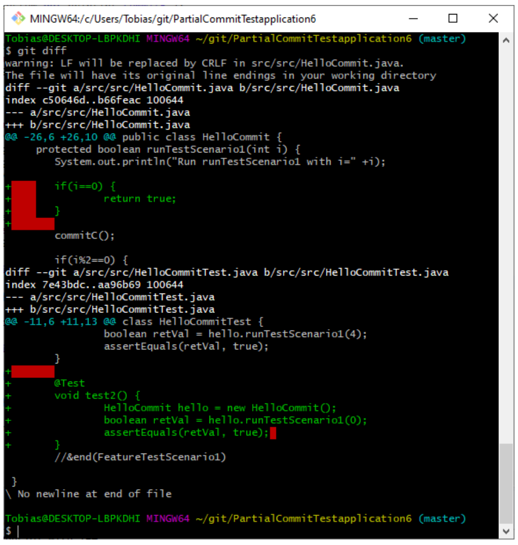
Project Manager

C

Partial Commit Evaluation

C.1 Scenario 1 - Adding New Assets to an Existing Feature

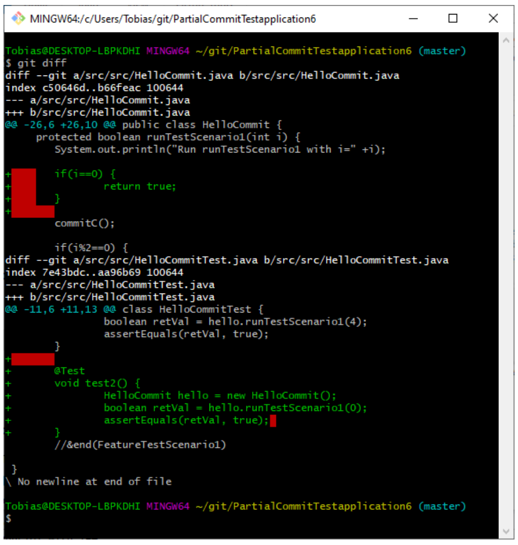
Steps for “git add −−patch” on the left side and the new tool on the right side.



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff
warning: LF will be replaced by CRLF in src/src/HelloCommit.java.
The file will have its original line endings in your working directory
diff --git a/src/src/HelloCommit.java b/src/src/HelloCommit.java
index c50646d..b66feac 100644
--- a/src/src/HelloCommit.java
+++ b/src/src/HelloCommit.java
@@ -26,6 +26,10 @@ public class HelloCommit {
     protected boolean runTestScenario1(int i) {
        System.out.println("Run runTestScenario1 with i=" +i);

+       if(i==0) {
+               return true;
+       }
+
        commitC();

        if(i%2==0) {
diff --git a/src/src/HelloCommitTest.java b/src/src/HelloCommitTest.java
index 7e43bdc..aa96b69 100644
--- a/src/src/HelloCommitTest.java
+++ b/src/src/HelloCommitTest.java
@@ -11,6 +11,13 @@ class HelloCommitTest {
                boolean retVal = hello.runTestScenario1(4);
                assertEquals(retVal, true);
        }
+
+       @Test
+       void test2() {
+               HelloCommit hello = new HelloCommit();
+               boolean retVal = hello.runTestScenario1(0);
+               assertEquals(retVal, true);
+       }
        //&end(FeatureTestScenario1)

 }
\ No newline at end of file

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

(a) Git add −−patch

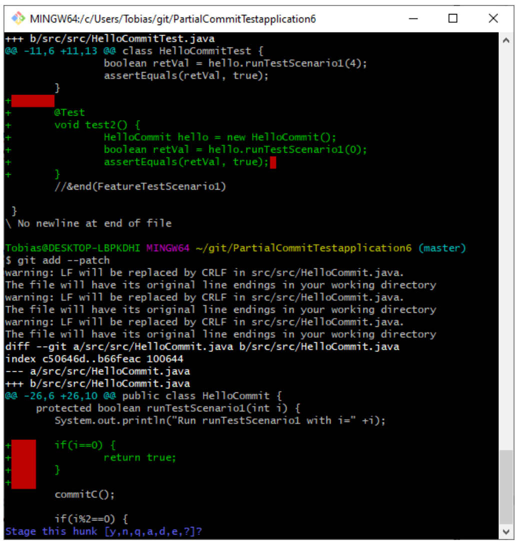


MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff
diff --git a/src/src/HelloCommit.java b/src/src/HelloCommit.java
index c50646d..b66feac 100644
--- a/src/src/HelloCommit.java
+++ b/src/src/HelloCommit.java
@@ -26,6 +26,10 @@ public class HelloCommit {
     protected boolean runTestScenario1(int i) {
        System.out.println("Run runTestScenario1 with i=" +i);

+       if(i==0) {
+               return true;
+       }
+
        commitC();

        if(i%2==0) {
diff --git a/src/src/HelloCommitTest.java b/src/src/HelloCommitTest.java
index 7e43bdc..aa96b69 100644
--- a/src/src/HelloCommitTest.java
+++ b/src/src/HelloCommitTest.java
@@ -11,6 +11,13 @@ class HelloCommitTest {
                boolean retVal = hello.runTestScenario1(4);
                assertEquals(retVal, true);
        }
+
+       @Test
+       void test2() {
+               HelloCommit hello = new HelloCommit();
+               boolean retVal = hello.runTestScenario1(0);
+               assertEquals(retVal, true);
+       }
        //&end(FeatureTestScenario1)

 }
\ No newline at end of file

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$
```

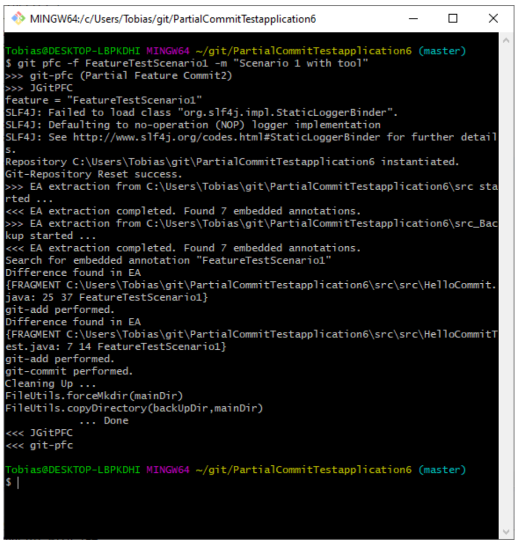
(b) Partial Commit Tool

Figure C.1: Tool Evaluation Partial Commit, Scenario 1, Changes to Commit



```
+++ b/src/src/HelloCommitTest.java
@@ -11,6 +11,13 @@ class HelloCommitTest {
                boolean retVal = hello.runTestScenario1(4);
                assertEquals(retVal, true);
        }
+
+       @Test
+       void test2() {
+               HelloCommit hello = new HelloCommit();
+               boolean retVal = hello.runTestScenario1(0);
+               assertEquals(retVal, true);
+       }
        //&end(FeatureTestScenario1)

 }
\ No newline at end of file

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git add --patch
warning: LF will be replaced by CRLF in src/src/HelloCommit.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/src/HelloCommit.java.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/src/HelloCommit.java.
The file will have its original line endings in your working directory
diff --git a/src/src/HelloCommit.java b/src/src/HelloCommit.java
index c50646d..b66feac 100644
--- a/src/src/HelloCommit.java
+++ b/src/src/HelloCommit.java
@@ -26,6 +26,10 @@ public class HelloCommit {
     protected boolean runTestScenario1(int i) {
        System.out.println("Run runTestScenario1 with i=" +i);

+       if(i==0) {
+               return true;
+       }
+
        commitC();

        if(i%2==0) {
Stage this hunk [y,n,q,a,d,e,?]?
```

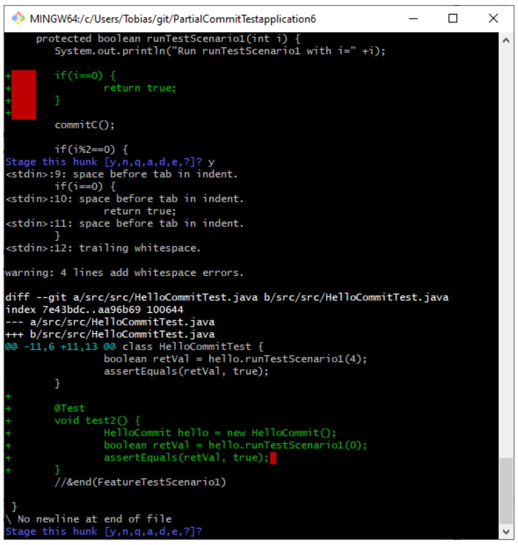
(a) Call "git add −−patch" and Add Hunk in HelloCommit.java



```

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git pfc -f FeatureTestScenario1 -m "Scenario 1 with tool"
>>> git-pfc (Partial Feature Commit2)
>>> JGitPFC
feature = "FeatureTestScenario1"
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further detail
s.
Repository C:\Users\Tobias\git\PartialCommitTestapplication6 instantiated.
Git-Repository Reset success.
>>> EA extraction from C:\Users\Tobias\git\PartialCommitTestapplication6\src sta
rted ...
<<< EA extraction completed. Found 7 embedded annotations.
>>> EA extraction from C:\Users\Tobias\git\PartialCommitTestapplication6\src_Bac
kup started ...
<<< EA extraction completed. Found 7 embedded annotations.
Search for embedded annotation "FeatureTestScenario1"
Difference found in EA
{FRAGMENT C:\Users\Tobias\git\PartialCommitTestapplication6\src\src\HelloCommit.
java: 25 37 FeatureTestScenario1}
git-add performed.
Difference found in EA
{FRAGMENT C:\Users\Tobias\git\PartialCommitTestapplication6\src\src\HelloCommitT
est.java: 7 14 FeatureTestScenario1}
git-add performed.
git-commit performed.
Cleaning Up ...
FileUtils.forceMkdir(mainDir)
FileUtils.copyDirectory(backUpDir,mainDir)
            ... Done
<<< JGitPFC
<<< git-pfc

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

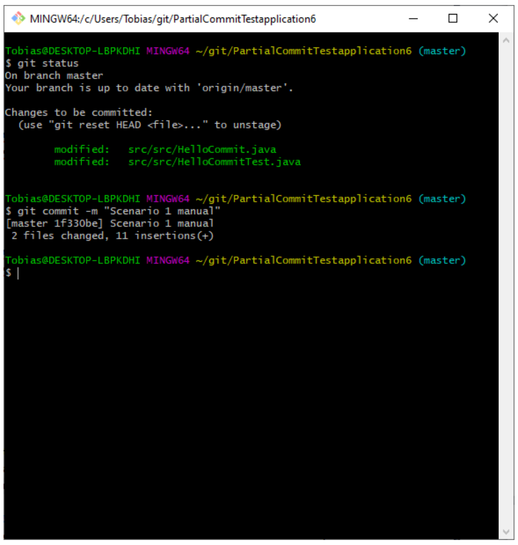
(b) Call Partial Commit Tool

Figure C.2: Tool Evaluation Partial Commit, Scenario 1, Tool Execution



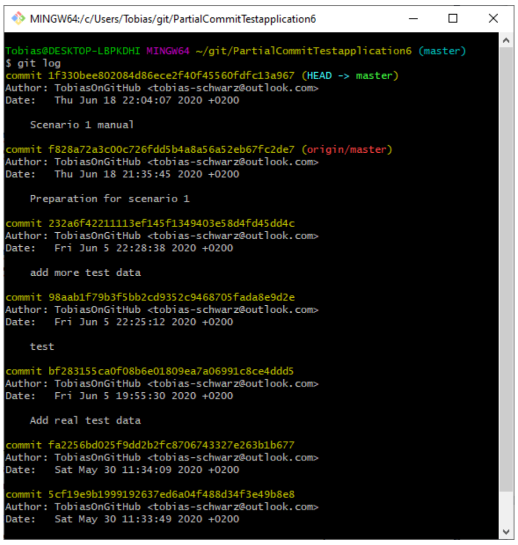
```
     protected boolean runTestScenario1(int i) {
        System.out.println("Run runTestScenario1 with i=" +i);

+       if(i==0) {
+               return true;
+       }
+
        commitC();

        if(i%2==0) {
Stage this hunk [y,n,q,a,d,e,?]? y
<stdin>:9: space before tab in indent.
        if(i==0) {
<stdin>:10: space before tab in indent.
                return true;
<stdin>:11: space before tab in indent.
        }
<stdin>:12: trailing whitespace.

warning: 4 lines add whitespace errors.

diff --git a/src/src/HelloCommitTest.java b/src/src/HelloCommitTest.java
index 7e43bdc..aa96b69 100644
--- a/src/src/HelloCommitTest.java
+++ b/src/src/HelloCommitTest.java
@@ -11,6 +11,13 @@ class HelloCommitTest {
                boolean retVal = hello.runTestScenario1(4);
                assertEquals(retVal, true);
        }
+
+       @Test
+       void test2() {
+               HelloCommit hello = new HelloCommit();
+               boolean retVal = hello.runTestScenario1(0);
+               assertEquals(retVal, true);
+       }
        //&end(FeatureTestScenario1)

 }
\ No newline at end of file
Stage this hunk [y,n,q,a,d,e,?]?
```

Figure C.3: Tool Evaluation Partial Commit, Scenario 1, git-add for Hunk in HelloCommitTest.java



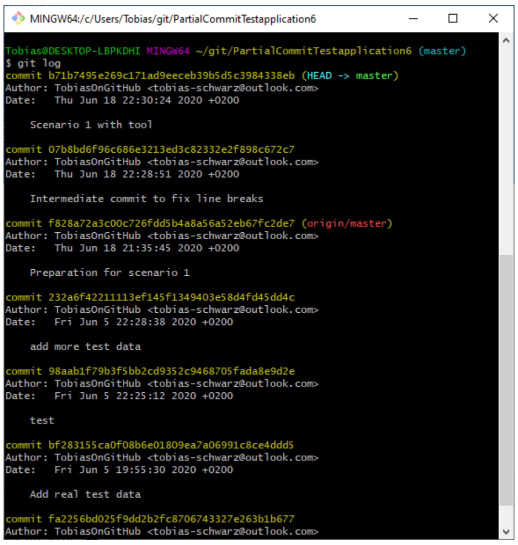
```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   src/src/HelloCommit.java
        modified:   src/src/HelloCommitTest.java


Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git commit -m "Scenario 1 manual"
[master 1f330be] Scenario 1 manual
 2 files changed, 11 insertions(+)

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

Figure C.4: Tool Evaluation Partial Commit, Scenario 1, git-commit for Staged Changes



```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git log
commit 1f330bee802084d86ece2f40f45560fdfc13a967 (HEAD -> master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 22:04:07 2020 +0200

    Scenario 1 manual

commit f828a72a3c00c726fdd5b4a8a56a52eb67fc2de7 (origin/master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 21:35:45 2020 +0200

    Preparation for scenario 1

commit 232a6f42211113ef145f1349403e58d4fd45dd4c
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 5 22:28:38 2020 +0200

    add more test data

commit 98aab1f79b3f5bb2cd9352c9468705fada8e9d2e
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 5 22:25:12 2020 +0200

    test

commit bf283155ca0f08b6e01809ea7a06991c8ce4ddd5
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 5 19:55:30 2020 +0200

    Add real test data

commit fa2256bd025f9dd2b2fc8706743327e263b1b677
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Sat May 30 11:34:09 2020 +0200

commit 5cf19e9b1999192637ed6a04f488d34f3e49b8e8
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Sat May 30 11:33:49 2020 +0200
```

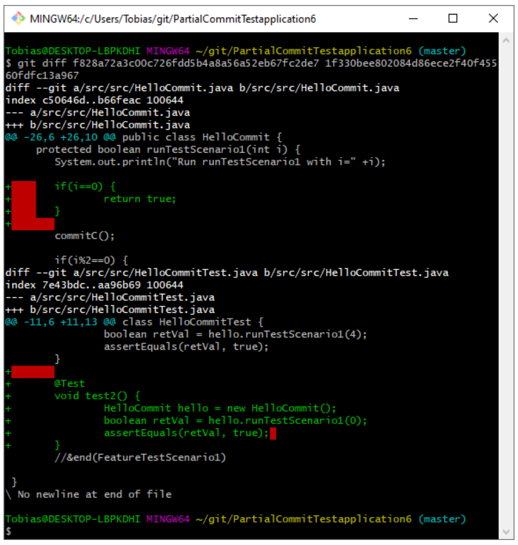
(a) Call git add −−patch



```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git log
commit b71b7495e269c171ad9eeceb39b5d5c3984338eb (HEAD -> master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 22:30:24 2020 +0200

    Scenario 1 with tool

commit 07b8bd6f96c686e3213ed3c82332e2f898c672c7
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 22:28:51 2020 +0200

    Intermediate commit to fix line breaks

commit f828a72a3c00c726fdd5b4a8a56a52eb67fc2de7 (origin/master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 21:35:45 2020 +0200

    Preparation for scenario 1

commit 232a6f42211113ef145f1349403e58d4fd45dd4c
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 5 22:28:38 2020 +0200

    add more test data

commit 98aab1f79b3f5bb2cd9352c9468705fada8e9d2e
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 5 22:25:12 2020 +0200

    test

commit bf283155ca0f08b6e01809ea7a06991c8ce4ddd5
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 5 19:55:30 2020 +0200

    Add real test data

commit fa2256bd025f9dd2b2fc8706743327e263b1b677
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
```

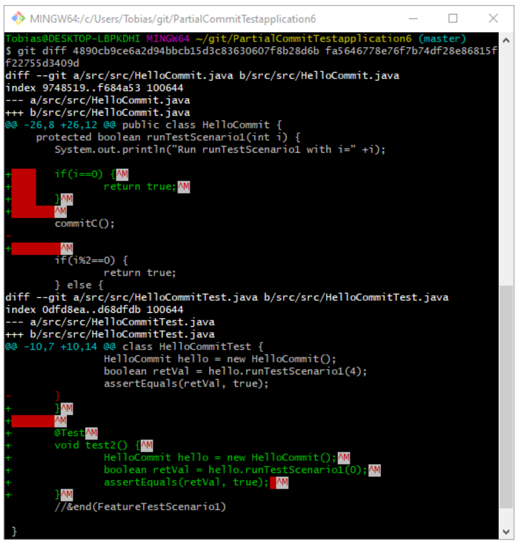
(b) Call Partial Commit Tool

Figure C.5: Tool Evaluation Partial Commit, Scenario 1, git-log After Tool Execution



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff f828a72a3c00c726fdd5b4a8a56a52eb67fc2de7 1f330bee802084d86ece2f40f455
60fdfc13a967
diff --git a/src/src/HelloCommit.java b/src/src/HelloCommit.java
index c50646d..b66feac 100644
--- a/src/src/HelloCommit.java
+++ b/src/src/HelloCommit.java
@@ -26,6 +26,10 @@ public class HelloCommit {
     protected boolean runTestScenario1(int i) {
        System.out.println("Run runTestScenario1 with i=" +i);

+       if(i==0) {
+               return true;
+       }
+
        commitC();

        if(i%2==0) {
diff --git a/src/src/HelloCommitTest.java b/src/src/HelloCommitTest.java
index 7e43bdc..aa96b69 100644
--- a/src/src/HelloCommitTest.java
+++ b/src/src/HelloCommitTest.java
@@ -11,6 +11,13 @@ class HelloCommitTest {
                boolean retVal = hello.runTestScenario1(4);
                assertEquals(retVal, true);
        }
+
+       @Test
+       void test2() {
+               HelloCommit hello = new HelloCommit();
+               boolean retVal = hello.runTestScenario1(0);
+               assertEquals(retVal, true);
+       }
        //&end(FeatureTestScenario1)

 }
\ No newline at end of file

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$
```

(a) Call git add −−patch



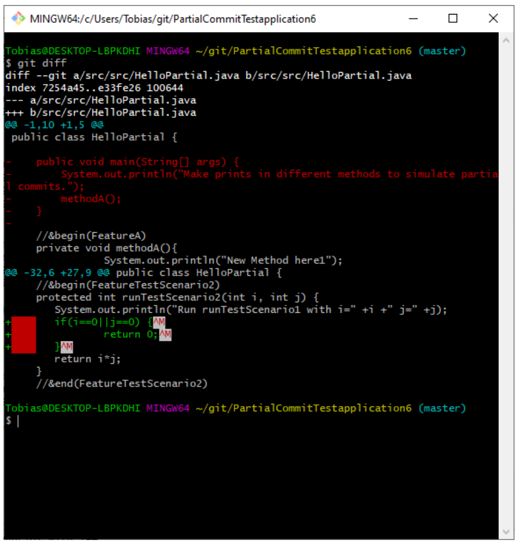
```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff 4890cb9ce6a2d94bbcb15d3c83630607f8b28d6b fa5646778e76f7b74df28e86815f
f22755d3409d
diff --git a/src/src/HelloCommit.java b/src/src/HelloCommit.java
index 9748519..f684a53 100644
--- a/src/src/HelloCommit.java
+++ b/src/src/HelloCommit.java
@@ -26,8 +26,12 @@ public class HelloCommit {
     protected boolean runTestScenario1(int i) {
        System.out.println("Run runTestScenario1 with i=" +i);

+       if(i==0) {^M
+               return true;^M
+       }^M
+       ^M
        commitC();
-
+       ^M
        if(i%2==0) {
                return true;
        } else {
diff --git a/src/src/HelloCommitTest.java b/src/src/HelloCommitTest.java
index 0dfd8ea..d68dfdb 100644
--- a/src/src/HelloCommitTest.java
+++ b/src/src/HelloCommitTest.java
@@ -10,7 +10,14 @@ class HelloCommitTest {
                HelloCommit hello = new HelloCommit();
                boolean retVal = hello.runTestScenario1(4);
                assertEquals(retVal, true);
-       }
+       }^M
+       ^M
+       @Test^M
+       void test2() {^M
+               HelloCommit hello = new HelloCommit();^M
+               boolean retVal = hello.runTestScenario1(0);^M
+               assertEquals(retVal, true); ^M
+       }^M
        //&end(FeatureTestScenario1)

 }
```

(b) Call Partial Commit Tool, line-breaks highlighted due to tool internal handling

Figure C.6: Tool Evaluation Partial Commit, Scenario 1, git-diff for New Commits

C.2 Scenario 2 - Evolution of Source Code in Embedded Annotation and Base Source Code

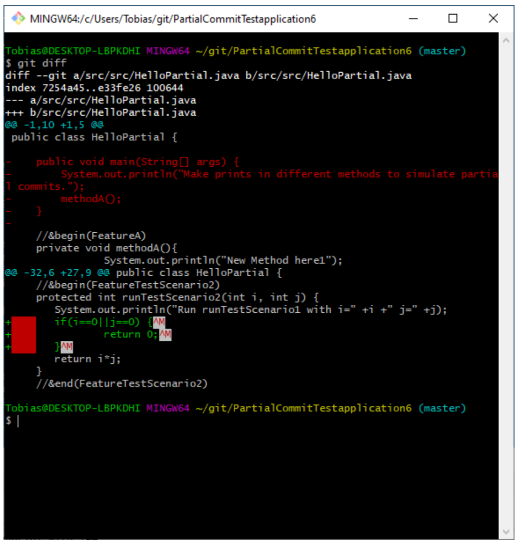
Steps for “git add −−patch” on the left side and the new tool on the right side.



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff
diff --git a/src/src/HelloPartial.java b/src/src/HelloPartial.java
index 7254a45..e33fe26 100644
--- a/src/src/HelloPartial.java
+++ b/src/src/HelloPartial.java
@@ -1,10 +1,5 @@
 public class HelloPartial {

-    public void main(String[] args) {
-        System.out.println("Make prints in different methods to simulate partia
l commits.");
-        methodA();
-    }
-
     //&begin(FeatureA)
     private void methodA(){
                 System.out.println("New Method here1");
@@ -32,6 +27,9 @@ public class HelloPartial {
     //&begin(FeatureTestScenario2)
     protected int runTestScenario2(int i, int j) {
        System.out.println("Run runTestScenario1 with i=" +i +" j=" +j);
+       if(i==0||j==0) {^M
+               return 0;^M
+       }^M
        return i*j;
     }
     //&end(FeatureTestScenario2)

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

(a) Git add −−patch

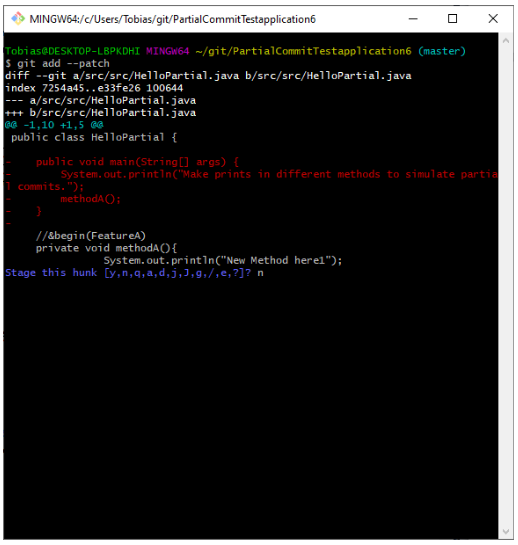


MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff
diff --git a/src/src/HelloPartial.java b/src/src/HelloPartial.java
index 7254a45..e33fe26 100644
--- a/src/src/HelloPartial.java
+++ b/src/src/HelloPartial.java
@@ -1,10 +1,5 @@
 public class HelloPartial {

-    public void main(String[] args) {
-        System.out.println("Make prints in different methods to simulate partia
l commits.");
-        methodA();
-    }
-
     //&begin(FeatureA)
     private void methodA(){
                 System.out.println("New Method here1");
@@ -32,6 +27,9 @@ public class HelloPartial {
     //&begin(FeatureTestScenario2)
     protected int runTestScenario2(int i, int j) {
        System.out.println("Run runTestScenario1 with i=" +i +" j=" +j);
+       if(i==0||j==0) {^M
+               return 0;^M
+       }^M
        return i*j;
     }
     //&end(FeatureTestScenario2)

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

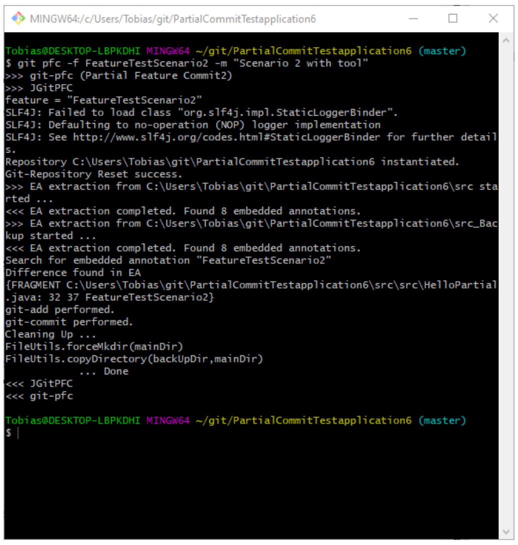
(b) Partial Commit Tool

Figure C.7: Tool Evaluation Partial Commit, Scenario 2, Changes to Commit



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git add --patch
diff --git a/src/src/HelloPartial.java b/src/src/HelloPartial.java
index 7254a45..e33fe26 100644
--- a/src/src/HelloPartial.java
+++ b/src/src/HelloPartial.java
@@ -1,10 +1,5 @@
 public class HelloPartial {

-    public void main(String[] args) {
-        System.out.println("Make prints in different methods to simulate partia
l commits.");
-        methodA();
-    }
-
     //&begin(FeatureA)
     private void methodA(){
                 System.out.println("New Method here1");
Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? n
```

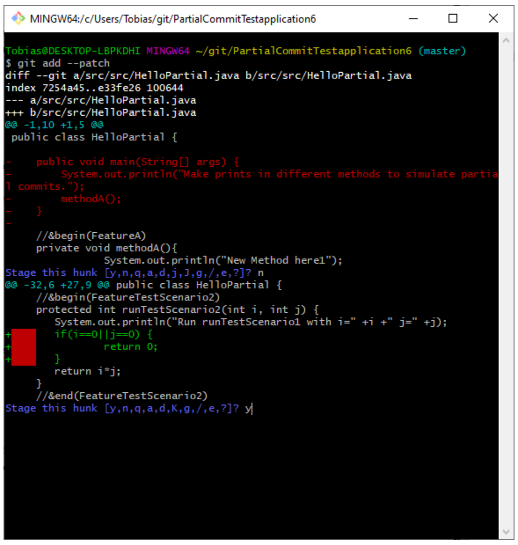
(a) Call “git add −−patch” and Skip Hunk in HelloPartial.java



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git pfc -f FeatureTestScenario2 -m "Scenario 2 with tool"
>>> git-pfc (Partial Feature Commit2)
>>> JGitPFC
feature = "FeatureTestScenario2"
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further detail
s.
Repository C:\Users\Tobias\git\PartialCommitTestapplication6 instantiated.
Git-Repository Reset success.
>>> EA extraction from C:\Users\Tobias\git\PartialCommitTestapplication6\src sta
rted ...
<<< EA extraction completed. Found 8 embedded annotations.
>>> EA extraction from C:\Users\Tobias\git\PartialCommitTestapplication6\src_Bac
kup started ...
<<< EA extraction completed. Found 8 embedded annotations.
Search for embedded annotation "FeatureTestScenario2"
Difference found in EA
{FRAGMENT C:\Users\Tobias\git\PartialCommitTestapplication6\src\src\HelloPartial
.java: 32 37 FeatureTestScenario2}
git-add performed.
git-commit performed.
Cleaning Up ...
FileUtils.forceMkdir(mainDir)
FileUtils.copyDirectory(backUpDir,mainDir)
            ... Done
<<< JGitPFC
<<< git-pfc

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$
```

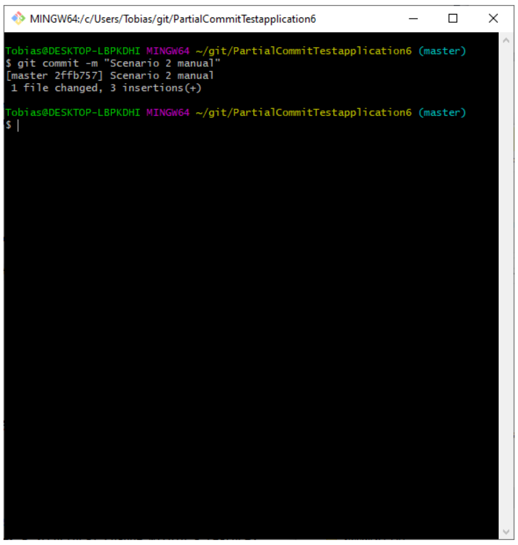
(b) Call Partial Commit Tool

Figure C.8: Tool Evaluation Partial Commit, Scenario 2, Tool Execution



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git add --patch
diff --git a/src/src/HelloPartial.java b/src/src/HelloPartial.java
index 7254a45..e33fe26 100644
--- a/src/src/HelloPartial.java
+++ b/src/src/HelloPartial.java
@@ -1,10 +1,5 @@
 public class HelloPartial {

-    public void main(String[] args) {
-        System.out.println("Make prints in different methods to simulate partia
l commits.");
-        methodA();
-    }
-
     //&begin(FeatureA)
     private void methodA(){
                 System.out.println("New Method here1");
Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? n
@@ -32,6 +27,9 @@ public class HelloPartial {
     //&begin(FeatureTestScenario2)
     protected int runTestScenario2(int i, int j) {
        System.out.println("Run runTestScenario1 with i=" +i +" j=" +j);
+       if(i==0||j==0) {
+               return 0;
+       }
        return i*j;
     }
     //&end(FeatureTestScenario2)
Stage this hunk [y,n,q,a,d,K,g,/,e,?]? y
```

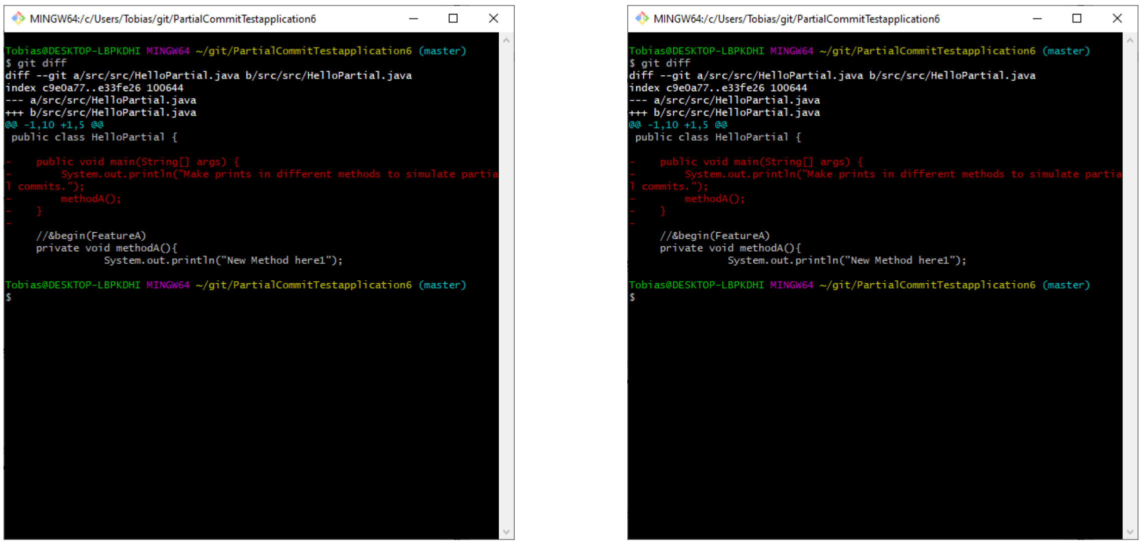
Figure C.9: Tool Evaluation Partial Commit, Scenario 2, git-add for Hunk in HelloCommitTest.java



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git commit -m "Scenario 2 manual"
[master 2ffb757] Scenario 2 manual
 1 file changed, 3 insertions(+)

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

Figure C.10: Tool Evaluation Partial Commit, Scenario 2, git-commit for Staged Changes



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff
diff --git a/src/src/HelloPartial.java b/src/src/HelloPartial.java
index c9e0a77..e33fe26 100644
--- a/src/src/HelloPartial.java
+++ b/src/src/HelloPartial.java
@@ -1,10 +1,5 @@
 public class HelloPartial {

-    public void main(String[] args) {
-        System.out.println("Make prints in different methods to simulate partia
l commits.");
-        methodA();
-    }
-
    //&begin(FeatureA)
    private void methodA(){
                System.out.println("New Method here1");

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$
```

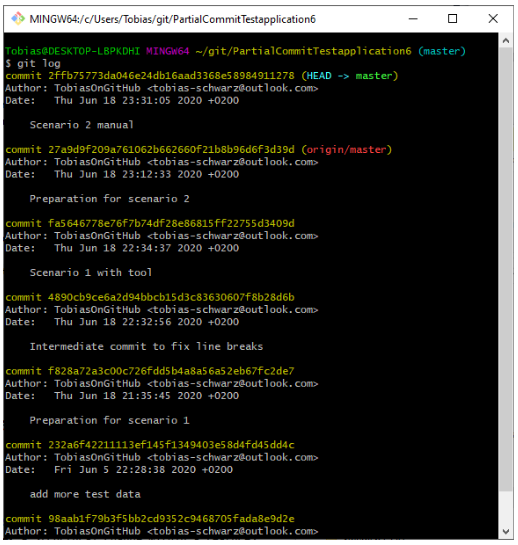
(a) Git add ––patch

MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff
diff --git a/src/src/HelloPartial.java b/src/src/HelloPartial.java
index c9e0a77..e33fe26 100644
--- a/src/src/HelloPartial.java
+++ b/src/src/HelloPartial.java
@@ -1,10 +1,5 @@
 public class HelloPartial {

-    public void main(String[] args) {
-        System.out.println("Make prints in different methods to simulate partia
l commits.");
-        methodA();
-    }
-
    //&begin(FeatureA)
    private void methodA(){
                System.out.println("New Method here1");

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$
```

(b) Partial Commit Tool

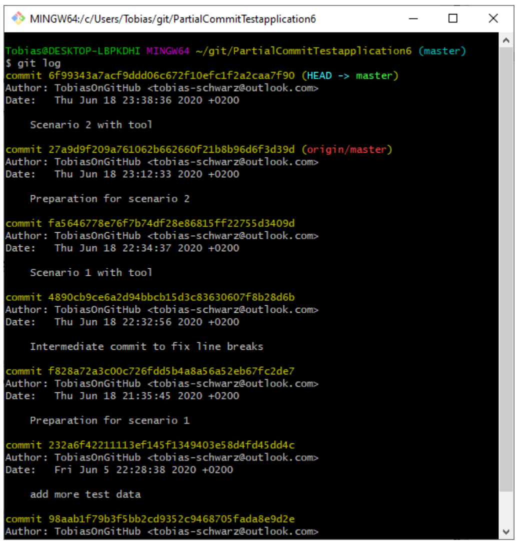
Figure C.11: Tool Evaluation Partial Commit, Scenario 2, Non-feature Changes Unmodified



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git log
commit 2ffb75773da046e24db16aad3368e58984911278 (HEAD -> master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 23:31:05 2020 +0200

    Scenario 2 manual

commit 27a9d9f209a761062b662660f21b8b96d6f3d39d (origin/master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 23:12:33 2020 +0200

    Preparation for scenario 2

commit fa5646778e76f7b74df28e86815ff22755d3409d
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 22:34:37 2020 +0200

    Scenario 1 with tool

commit 4890cb9ce6a2d94bbcb15d3c83630607f8b28d6b
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 22:32:56 2020 +0200

    Intermediate commit to fix line breaks

commit f828a72a3c00c726fdd5b4a8a56a52eb67fc2de7
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 21:35:45 2020 +0200

    Preparation for scenario 1

commit 232a6f42211113ef145f1349403e58d4fd45dd4c
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 5 22:28:38 2020 +0200

    add more test data

commit 98aab1f79b3f5bb2cd9352c9468705fada8e9d2e
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
```

(a) Git add −−patch

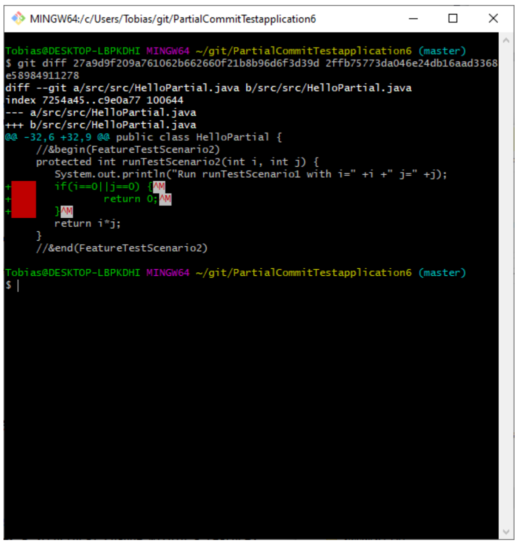


MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git log
commit 6f99343a7acf9ddd06c672f10efc1f2a2caa7f90 (HEAD -> master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 23:38:36 2020 +0200

    Scenario 2 with tool

commit 27a9d9f209a761062b662660f21b8b96d6f3d39d (origin/master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 23:12:33 2020 +0200

    Preparation for scenario 2

commit fa5646778e76f7b74df28e86815ff22755d3409d
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 22:34:37 2020 +0200

    Scenario 1 with tool

commit 4890cb9ce6a2d94bbcb15d3c83630607f8b28d6b
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 22:32:56 2020 +0200

    Intermediate commit to fix line breaks

commit f828a72a3c00c726fdd5b4a8a56a52eb67fc2de7
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 21:35:45 2020 +0200

    Preparation for scenario 1

commit 232a6f42211113ef145f1349403e58d4fd45dd4c
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 5 22:28:38 2020 +0200

    add more test data

commit 98aab1f79b3f5bb2cd9352c9468705fada8e9d2e
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
```

(b) Partial Commit Tool

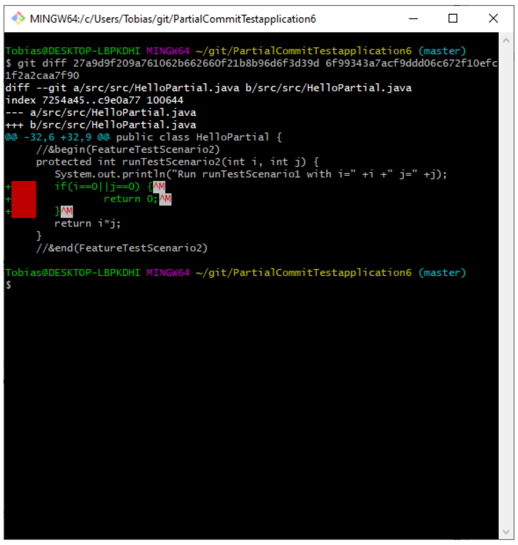
Figure C.12: Tool Evaluation Partial Commit, Scenario 2, git-log After Tool Execution



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff 27a9d9f209a761062b662660f21b8b96d6f3d39d 2ffb75773da046e24db16aad3368
e58984911278
diff --git a/src/src/HelloPartial.java b/src/src/HelloPartial.java
index 7254a45..c9e0a77 100644
--- a/src/src/HelloPartial.java
+++ b/src/src/HelloPartial.java
@@ -32,6 +32,9 @@ public class HelloPartial {
     //&begin(FeatureTestScenario2)
     protected int runTestScenario2(int i, int j) {
        System.out.println("Run runTestScenario1 with i=" +i +" j=" +j);
+       if(i==0||j==0) {^M
+               return 0;^M
+       }^M
        return i*j;
     }
     //&end(FeatureTestScenario2)

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

(a) Git add −−patch



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

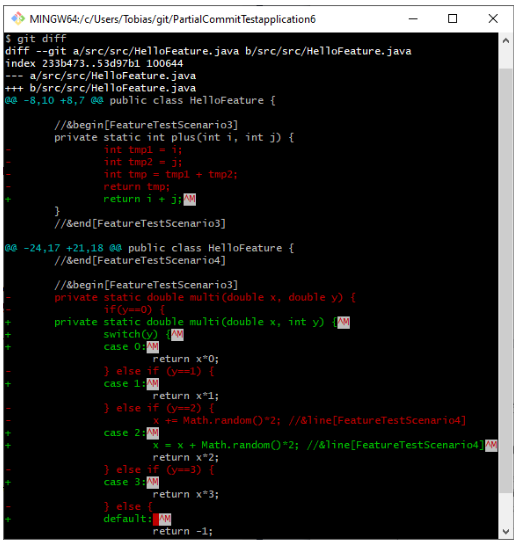
```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff 27a9d9f209a761062b662660f21b8b96d6f3d39d 6f99343a7acf9ddd06c672f10efc
1f2a2caa7f90
diff --git a/src/src/HelloPartial.java b/src/src/HelloPartial.java
index 7254a45..c9e0a77 100644
--- a/src/src/HelloPartial.java
+++ b/src/src/HelloPartial.java
@@ -32,6 +32,9 @@ public class HelloPartial {
     //&begin(FeatureTestScenario2)
     protected int runTestScenario2(int i, int j) {
        System.out.println("Run runTestScenario1 with i=" +i +" j=" +j);
+       if(i==0||j==0) {^M
+               return 0;^M
+       }^M
        return i*j;
     }
     //&end(FeatureTestScenario2)

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$
```

(b) Partial Commit Tool

Figure C.13: Tool Evaluation Partial Commit, Scenario 2, git-diff for New Commits

C.3 Scenario 3 - Refactoring Existing Structural Code Within a Feature

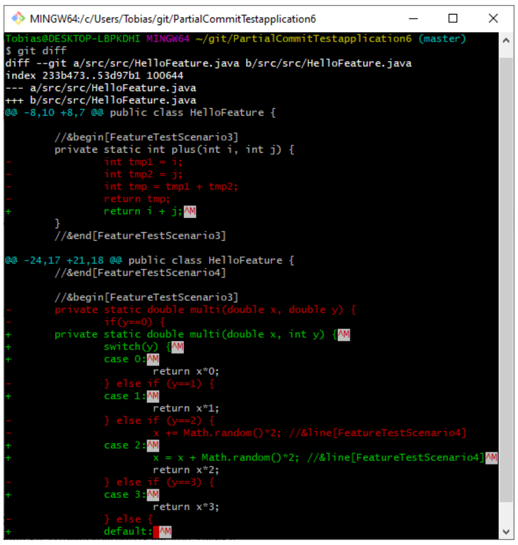
Steps for “git add −−patch” on the left side and the new tool on the right side.



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
$ git diff
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index 233b473..53d97b1 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;^M
        }
        //&end[FeatureTestScenario3]

@@ -24,17 +21,18 @@ public class HelloFeature {
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {^M
+               switch(y) {^M
+               case 0:^M
                        return x*0;
-               } else if (y==1) {
+               case 1:^M
                        return x*1;
-               } else if (y==2) {
-                       x += Math.random()*2; //&line[FeatureTestScenario4]
+               case 2:^M
+                       x = x + Math.random()*2; //&line[FeatureTestScenario4]^M
                        return x*2;
-               } else if (y==3) {
+               case 3:^M
                        return x*3;
-               } else {
+               default: ^M
                        return -1;
```

(a) Git add −−patch

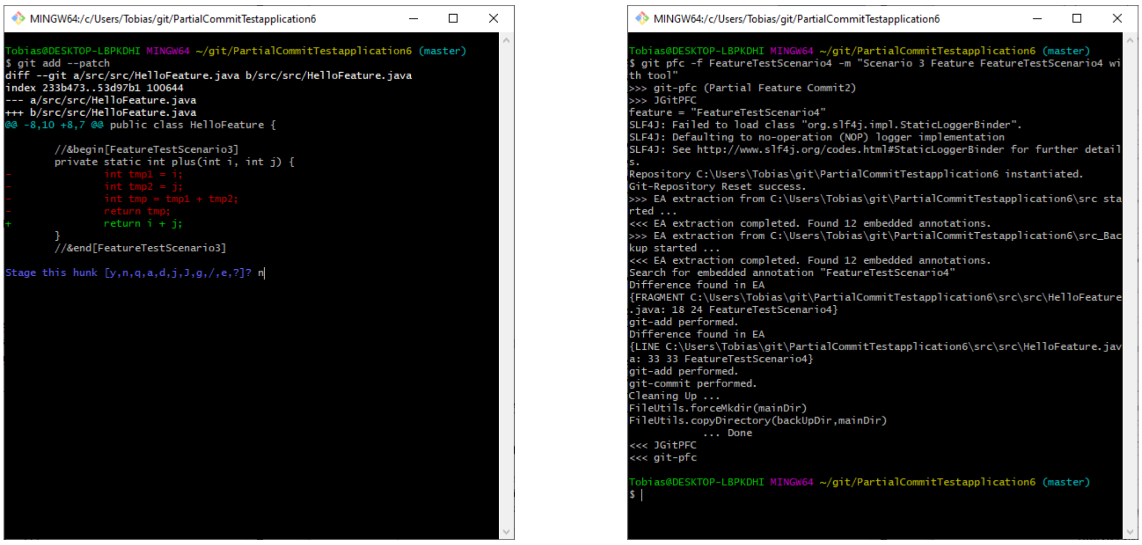


MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index 233b473..53d97b1 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;^M
        }
        //&end[FeatureTestScenario3]

@@ -24,17 +21,18 @@ public class HelloFeature {
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {^M
+               switch(y) {^M
+               case 0:^M
                        return x*0;
-               } else if (y==1) {
+               case 1:^M
                        return x*1;
-               } else if (y==2) {
-                       x += Math.random()*2; //&line[FeatureTestScenario4]
+               case 2:^M
+                       x = x + Math.random()*2; //&line[FeatureTestScenario4]^M
                        return x*2;
-               } else if (y==3) {
+               case 3:^M
                        return x*3;
-               } else {
+               default: ^M
```

(b) Partial Commit Tool

Figure C.14: Tool Evaluation Partial Commit, Scenario 3, Changes to Commit, Feature FeatureTestScenario4



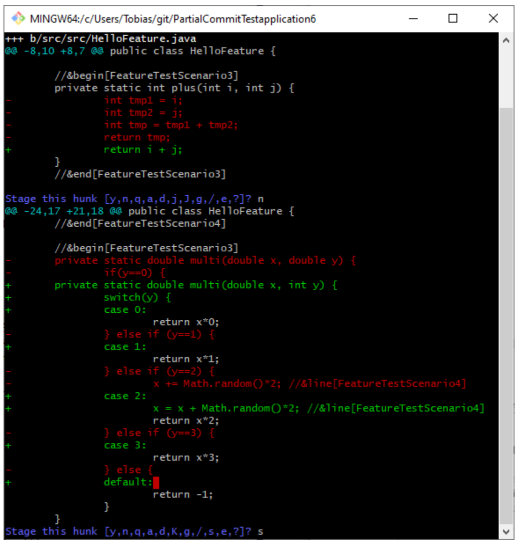
```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git add --patch
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index 233b473..53d97b1 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;
        }
        //&end[FeatureTestScenario3]

Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? n
```

(a) Call "git add −−patch" and Skip Hunk in HelloFeature.java

```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git pfc -f FeatureTestScenario4 -m "Scenario 3 Feature FeatureTestScenario4 wi
th tool"
>>> git-pfc (Partial Feature Commit2)
>>> JGitPFC
feature = "FeatureTestScenario4"
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further detail
s.
Repository C:\Users\Tobias\git\PartialCommitTestapplication6 instantiated.
Git-Repository Reset success.
>>> EA extraction from C:\Users\Tobias\git\PartialCommitTestapplication6\src sta
rted ...
<<< EA extraction completed. Found 12 embedded annotations.
>>> EA extraction from C:\Users\Tobias\git\PartialCommitTestapplication6\src_Bac
kup started ...
<<< EA extraction completed. Found 12 embedded annotations.
Search for embedded annotation "FeatureTestScenario4"
Difference found in EA
{FRAGMENT C:\Users\Tobias\git\PartialCommitTestapplication6\src\src\HelloFeature
.java: 18 24 FeatureTestScenario4}
git-add performed.
Difference found in EA
{LINE C:\Users\Tobias\git\PartialCommitTestapplication6\src\src\HelloFeature.jav
a: 33 33 FeatureTestScenario4}
git-add performed.
git-commit performed.
Cleaning Up ...
FileUtils.forceMkdir(mainDir)
FileUtils.copyDirectory(backUpDir,mainDir)
            ... Done
<<< JGitPFC
<<< git-pfc

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

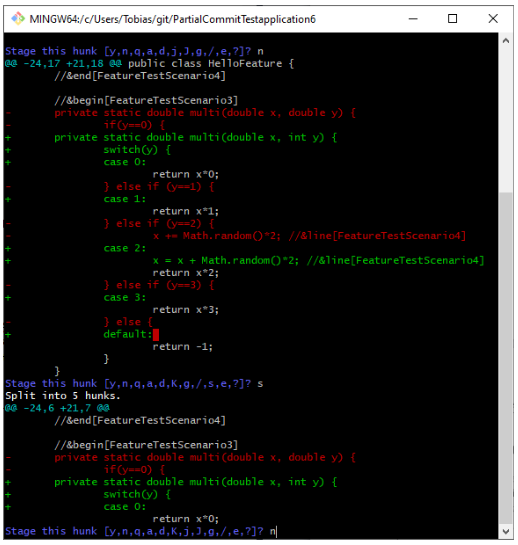
(b) Partial Commit Tool

Figure C.15: Tool Evaluation Partial Commit, Scenario 3, Tool Execution, Feature FeatureTestScenario4



```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;
        }
        //&end[FeatureTestScenario3]

Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? n
@@ -24,17 +21,18 @@ public class HelloFeature {
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {
+               switch(y) {
+               case 0:
                        return x*0;
-               } else if (y==1) {
+               case 1:
                        return x*1;
-               } else if (y==2) {
-                       x += Math.random()*2; //&line[FeatureTestScenario4]
+               case 2:
+                       x = x + Math.random()*2; //&line[FeatureTestScenario4]
                        return x*2;
-               } else if (y==3) {
+               case 3:
                        return x*3;
-               } else {
+               default:
                        return -1;
                }
        }
Stage this hunk [y,n,q,a,d,K,g,/,s,e,?]? s
```

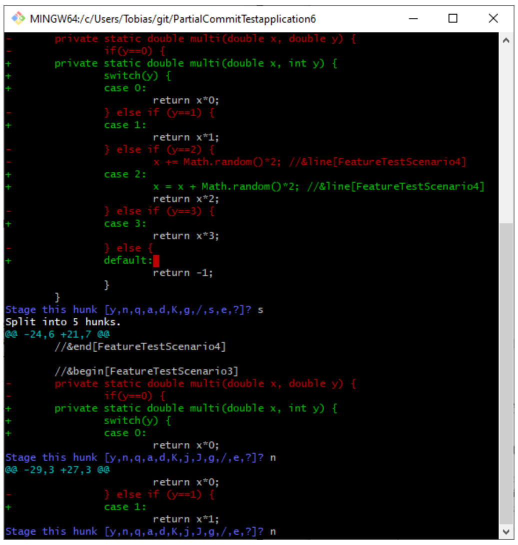
Figure C.16: Tool Evaluation Partial Commit, Scenario 3, Git Split Hunk in HelloFeature.java



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

```
Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? n
@@ -24,17 +21,18 @@ public class HelloFeature {
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {
+               switch(y) {
+               case 0:
                        return x*0;
-               } else if (y==1) {
+               case 1:
                        return x*1;
-               } else if (y==2) {
-                       x += Math.random()*2; //&line[FeatureTestScenario4]
+               case 2:
+                       x = x + Math.random()*2; //&line[FeatureTestScenario4]
                        return x*2;
-               } else if (y==3) {
+               case 3:
                        return x*3;
-               } else {
+               default:
                        return -1;
                }
        }
Stage this hunk [y,n,q,a,d,K,g,/,s,e,?]? s
Split into 5 hunks.
@@ -24,6 +21,7 @@
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {
+               switch(y) {
+               case 0:
                        return x*0;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? n
```

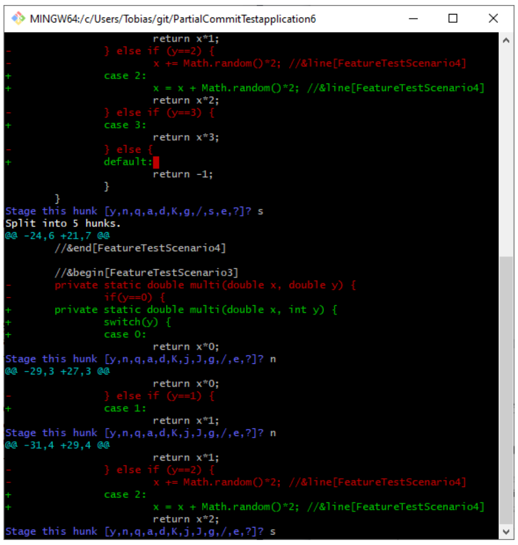
Figure C.17: Tool Evaluation Partial Commit, Scenario 3, Git Skip Hunk in HelloFeature.java



MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

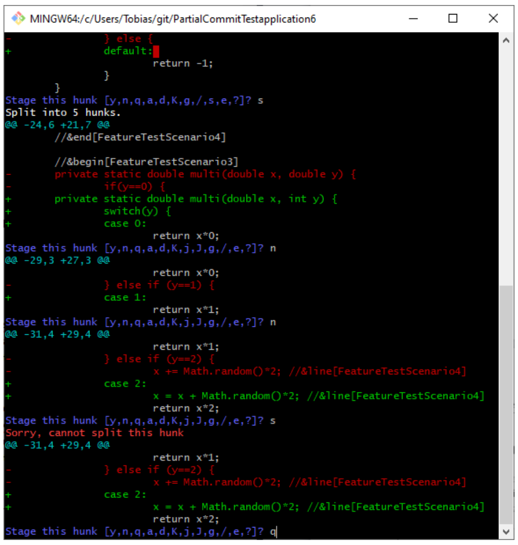
```
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {
+               switch(y) {
+               case 0:
                        return x*0;
-               } else if (y==1) {
+               case 1:
                        return x*1;
-               } else if (y==2) {
-                       x += Math.random()*2; //&line[FeatureTestScenario4]
+               case 2:
+                       x = x + Math.random()*2; //&line[FeatureTestScenario4]
                        return x*2;
-               } else if (y==3) {
+               case 3:
                        return x*3;
-               } else {
+               default:
                        return -1;
                }
        }
Stage this hunk [y,n,q,a,d,K,g,/,s,e,?]? s
Split into 5 hunks.
@@ -24,6 +21,7 @@
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {
+               switch(y) {
+               case 0:
                        return x*0;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? n
@@ -29,3 +27,3 @@
                        return x*0;
-               } else if (y==1) {
+               case 1:
                        return x*1;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? n
```

Figure C.18: Tool Evaluation Partial Commit, Scenario 3, Git Skip Hunk in HelloFeature.java



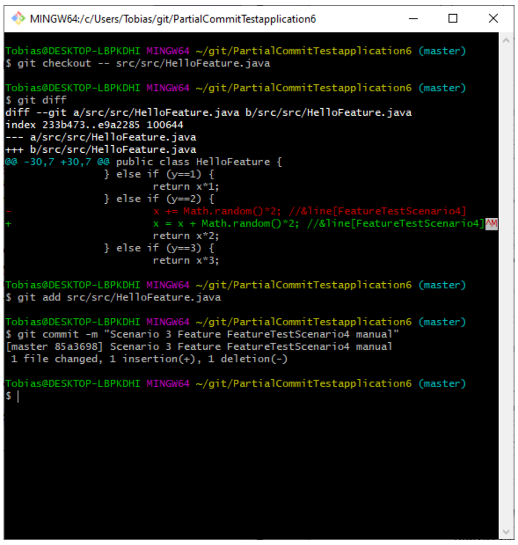
```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6
                                return x*1;
-                       } else if (y==2) {
-                               x += Math.random()*2; //&line[FeatureTestScenario4]
+                       case 2:
+                               x = x + Math.random()*2; //&line[FeatureTestScenario4]
                                return x*2;
-                       } else if (y==3) {
+                       case 3:
                                return x*3;
-                       } else {
+                       default:
                                return -1;
                        }
                }
Stage this hunk [y,n,q,a,d,K,g,/,s,e,?]? s
Split into 5 hunks.
@@ -24,6 +21,7 @@
                //&end[FeatureTestScenario4]

                //&begin[FeatureTestScenario3]
-               private static double multi(double x, double y) {
-                       if(y==0) {
+               private static double multi(double x, int y) {
+                       switch(y) {
+                       case 0:
                                return x*0;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? n
@@ -29,3 +27,3 @@
                                return x*0;
-                       } else if (y==1) {
+                       case 1:
                                return x*1;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? n
@@ -31,4 +29,4 @@
                                return x*1;
-                       } else if (y==2) {
-                               x += Math.random()*2; //&line[FeatureTestScenario4]
+                       case 2:
+                               x = x + Math.random()*2; //&line[FeatureTestScenario4]
                                return x*2;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? s
```

Figure C.19: Tool Evaluation Partial Commit, Scenario 3, Git Split Hunk in HelloFeature.java



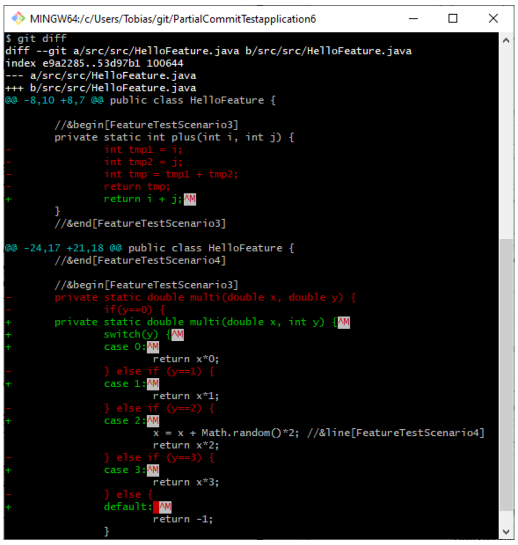
```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6
-                       } else {
+                       default:
                                return -1;
                        }
                }
Stage this hunk [y,n,q,a,d,K,g,/,s,e,?]? s
Split into 5 hunks.
@@ -24,6 +21,7 @@
                //&end[FeatureTestScenario4]

                //&begin[FeatureTestScenario3]
-               private static double multi(double x, double y) {
-                       if(y==0) {
+               private static double multi(double x, int y) {
+                       switch(y) {
+                       case 0:
                                return x*0;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? n
@@ -29,3 +27,3 @@
                                return x*0;
-                       } else if (y==1) {
+                       case 1:
                                return x*1;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? n
@@ -31,4 +29,4 @@
                                return x*1;
-                       } else if (y==2) {
-                               x += Math.random()*2; //&line[FeatureTestScenario4]
+                       case 2:
+                               x = x + Math.random()*2; //&line[FeatureTestScenario4]
                                return x*2;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? s
Sorry, cannot split this hunk
@@ -31,4 +29,4 @@
                                return x*1;
-                       } else if (y==2) {
-                               x += Math.random()*2; //&line[FeatureTestScenario4]
+                       case 2:
+                               x = x + Math.random()*2; //&line[FeatureTestScenario4]
                                return x*2;
Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]? q
```

Figure C.20: Tool Evaluation Partial Commit, Scenario 3, Git Split Hunk rejected



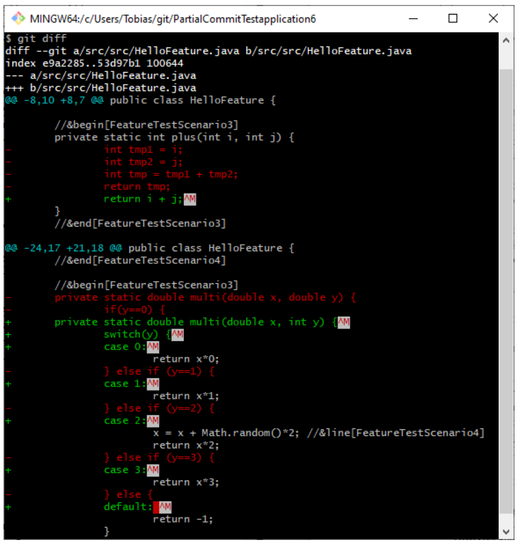
```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git checkout -- src/src/HelloFeature.java

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index 233b473..e9a2285 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -30,7 +30,7 @@ public class HelloFeature {
                } else if (y==1) {
                        return x*1;
                } else if (y==2) {
-                       x += Math.random()*2; //&line[FeatureTestScenario4]
+                       x = x + Math.random()*2; //&line[FeatureTestScenario4]^M
                        return x*2;
                } else if (y==3) {
                        return x*3;

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git add src/src/HelloFeature.java

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git commit -m "Scenario 3 Feature FeatureTestScenario4 manual"
[master 85a3698] Scenario 3 Feature FeatureTestScenario4 manual
 1 file changed, 1 insertion(+), 1 deletion(-)

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

Figure C.21: Tool Evaluation Partial Commit, Scenario 3, Manual Workaround for Rejected Hunk Split



```
$ git diff
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index e9a2285..53d97b1 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;^M
        }
        //&end[FeatureTestScenario3]

@@ -24,17 +21,18 @@ public class HelloFeature {
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {^M
+               switch(y) {^M
+               case 0:^M
                        return x*0;
-               } else if (y==1) {
+               case 1:^M
                        return x*1;
-               } else if (y==2) {
+               case 2:^M
                        x = x + Math.random()*2; //&line[FeatureTestScenario4]
                        return x*2;
-               } else if (y==3) {
+               case 3:^M
                        return x*3;
-               } else {
+               default: ^M
                        return -1;
                }
```

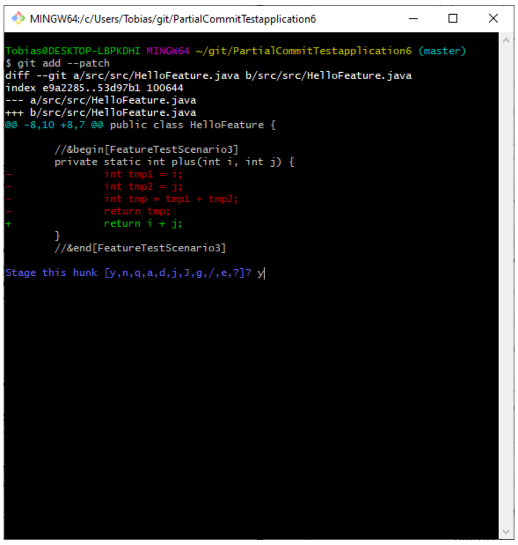
(a) Call git add −−patch



```
$ git diff
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index e9a2285..53d97b1 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;^M
        }
        //&end[FeatureTestScenario3]

@@ -24,17 +21,18 @@ public class HelloFeature {
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {^M
+               switch(y) {^M
+               case 0:^M
                        return x*0;
-               } else if (y==1) {
+               case 1:^M
                        return x*1;
-               } else if (y==2) {
+               case 2:^M
                        x = x + Math.random()*2; //&line[FeatureTestScenario4]
                        return x*2;
-               } else if (y==3) {
+               case 3:^M
                        return x*3;
-               } else {
+               default: ^M
                        return -1;
                }
```

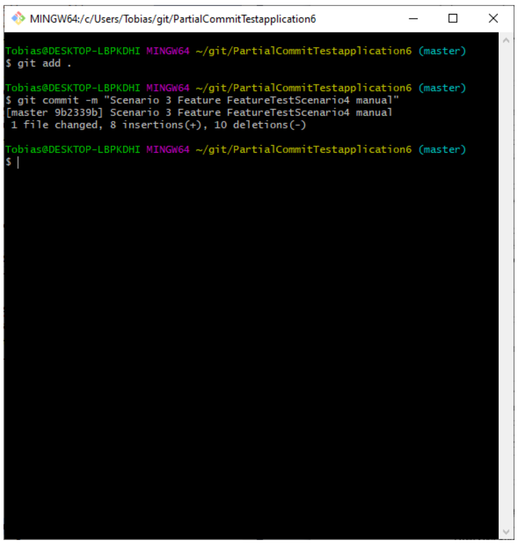
(b) Call Partial Commit Tool

Figure C.22: Tool Evaluation Partial Commit, Scenario 3, Changes to Commit, Feature FeatureTestScenario3



```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git add --patch
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index e9a2285..53d97b1 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;
        }
        //&end[FeatureTestScenario3]

Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? y
```

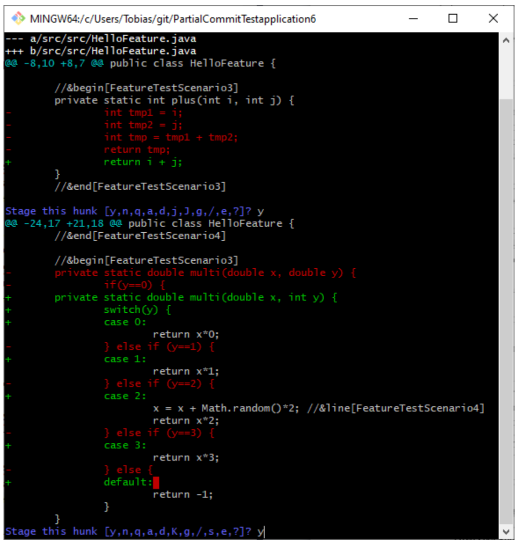
(a) Call "git add ——patch" and Add Hunk in HelloFeature.java



```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git add .

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git commit -m "Scenario 3 Feature FeatureTestScenario4 manual"
[master 9b2339b] Scenario 3 Feature FeatureTestScenario4 manual
 1 file changed, 8 insertions(+), 10 deletions(-)

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

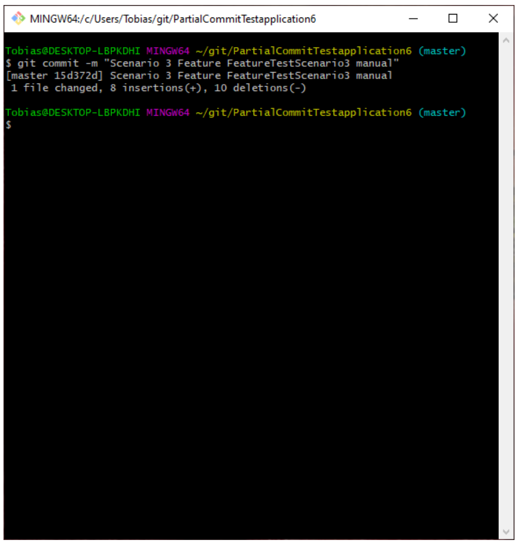
(b) Call Partial Commit Tool

Figure C.23: Tool Evaluation Partial Commit, Scenario 3, Tool Execution, Feature FeatureTestScenario3



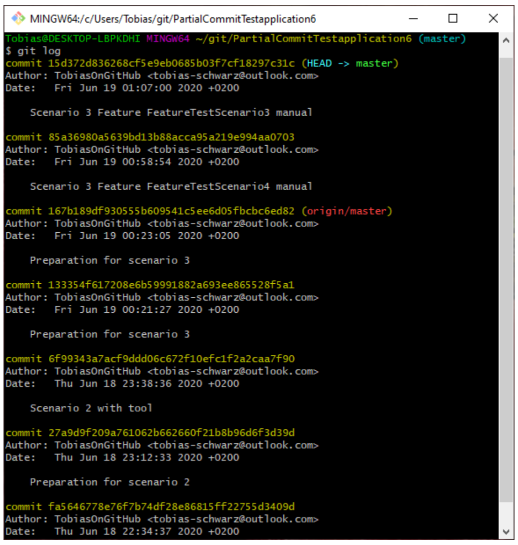
```
MINGW64:/c/Users/Tobias/git/PartialCommitTestapplication6
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;
        }
        //&end[FeatureTestScenario3]

Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? y
@@ -24,17 +21,18 @@ public class HelloFeature {
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {
+               switch(y) {
+               case 0:
                        return x*0;
-               } else if (y==1) {
+               case 1:
                        return x*1;
-               } else if (y==2) {
+               case 2:
                        x = x + Math.random()*2; //&line[FeatureTestScenario4]
                        return x*2;
-               } else if (y==3) {
+               case 3:
                        return x*3;
-               } else {
+               default:
                        return -1;
                }
        }
Stage this hunk [y,n,q,a,d,K,g,/,s,e,?]? y
```

Figure C.24: Tool Evaluation Partial Commit, Scenario 3, Git Split Hunk in HelloFeature.java



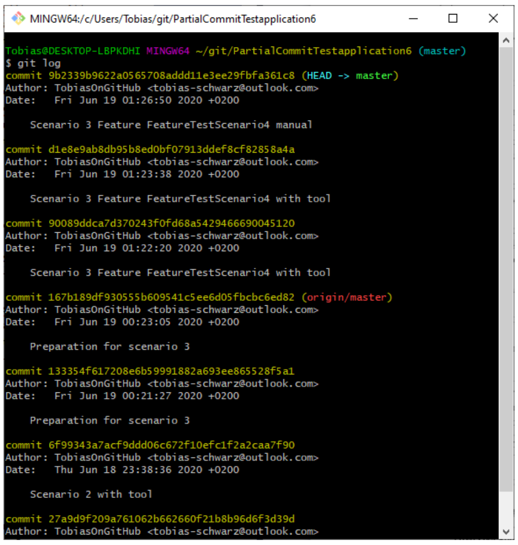
```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git commit -m "Scenario 3 Feature FeatureTestScenario3 manual"
[master 15d372d] Scenario 3 Feature FeatureTestScenario3 manual
 1 file changed, 8 insertions(+), 10 deletions(-)

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$
```

Figure C.25: Tool Evaluation Partial Commit, Scenario 3, git-commit for Staged Changes



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git log
commit 15d372d836268cf5e9eb0685b03f7cf18297c31c (HEAD -> master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 19 01:07:00 2020 +0200

    Scenario 3 Feature FeatureTestScenario3 manual

commit 85a36980a5639bd13b88acca95a219e994aa0703
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 19 00:58:54 2020 +0200

    Scenario 3 Feature FeatureTestScenario4 manual

commit 167b189df930555b609541c5ee6d05fbcbc6ed82 (origin/master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 19 00:23:05 2020 +0200

    Preparation for scenario 3

commit 133354f617208e6b59991882a693ee865528f5a1
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 19 00:21:27 2020 +0200

    Preparation for scenario 3

commit 6f99343a7acf9ddd06c672f10efc1f2a2caa7f90
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 23:38:36 2020 +0200

    Scenario 2 with tool

commit 27a9d9f209a761062b662660f21b8b96d6f3d39d
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 23:12:33 2020 +0200

    Preparation for scenario 2

commit fa5646778e76f7b74df28e86815ff22755d3409d
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 22:34:37 2020 +0200
```

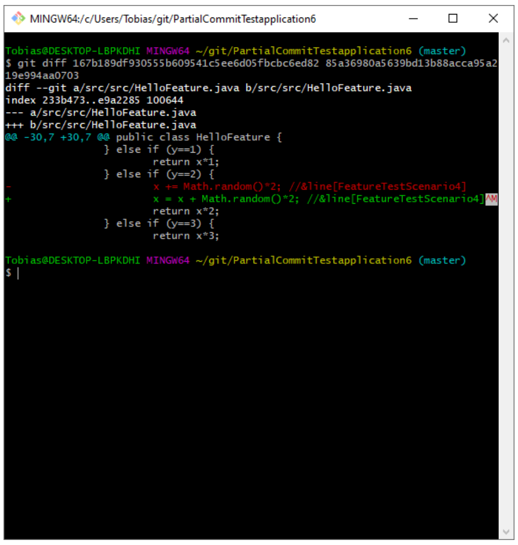
(a) Git add −−patch



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git log
commit 9b2339b9622a0565708addd11e3ee29fbfa361c8 (HEAD -> master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 19 01:26:50 2020 +0200

    Scenario 3 Feature FeatureTestScenario4 manual

commit d1e8e9ab8db95b8ed0bf07913ddef8cf82858a4a
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 19 01:23:38 2020 +0200

    Scenario 3 Feature FeatureTestScenario4 with tool

commit 90089ddca7d370243f0fd68a5429466690045120
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 19 01:22:20 2020 +0200

    Scenario 3 Feature FeatureTestScenario4 with tool

commit 167b189df930555b609541c5ee6d05fbcbc6ed82 (origin/master)
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 19 00:23:05 2020 +0200

    Preparation for scenario 3

commit 133354f617208e6b59991882a693ee865528f5a1
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Fri Jun 19 00:21:27 2020 +0200

    Preparation for scenario 3

commit 6f99343a7acf9ddd06c672f10efc1f2a2caa7f90
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
Date:   Thu Jun 18 23:38:36 2020 +0200

    Scenario 2 with tool

commit 27a9d9f209a761062b662660f21b8b96d6f3d39d
Author: TobiasOnGitHub <tobias-schwarz@outlook.com>
```

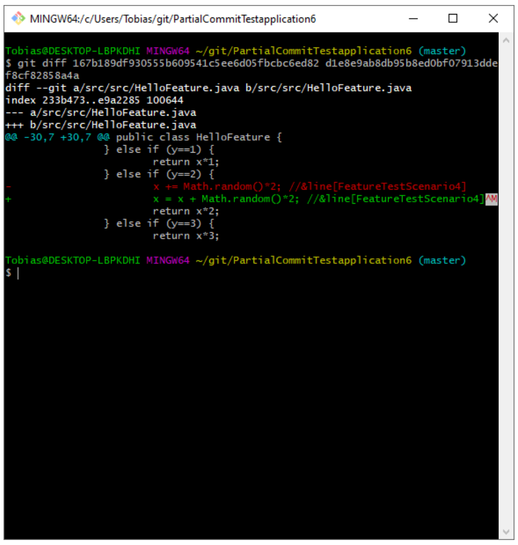
(b) Partial Commit Tool

Figure C.26: Tool Evaluation Partial Commit, Scenario 3, git-log After Tool Execution



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff 167b189df930555b609541c5ee6d05fbcbc6ed82 85a36980a5639bd13b88acca95a2
19e994aa0703
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index 233b473..e9a2285 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -30,7 +30,7 @@ public class HelloFeature {
                } else if (y==1) {
                        return x*1;
                } else if (y==2) {
-                       x += Math.random()*2; //&line[FeatureTestScenario4]
+                       x = x + Math.random()*2; //&line[FeatureTestScenario4]^M
                        return x*2;
                } else if (y==3) {
                        return x*3;

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

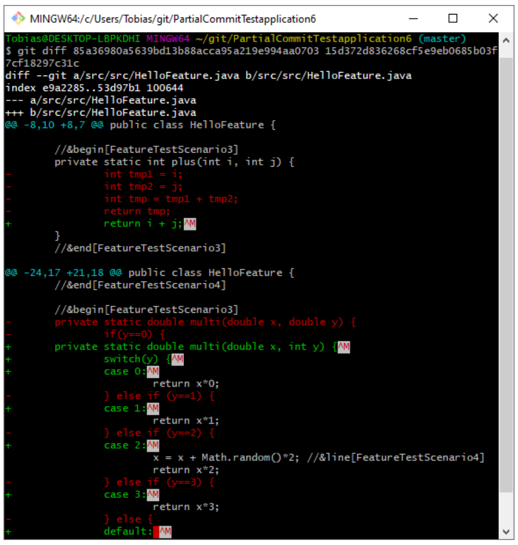
(a) Git add −−patch



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff 167b189df930555b609541c5ee6d05fbcbc6ed82 d1e8e9ab8db95b8ed0bf07913dde
f8cf82858a4a
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index 233b473..e9a2285 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -30,7 +30,7 @@ public class HelloFeature {
                } else if (y==1) {
                        return x*1;
                } else if (y==2) {
-                       x += Math.random()*2; //&line[FeatureTestScenario4]
+                       x = x + Math.random()*2; //&line[FeatureTestScenario4]^M
                        return x*2;
                } else if (y==3) {
                        return x*3;

Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ |
```

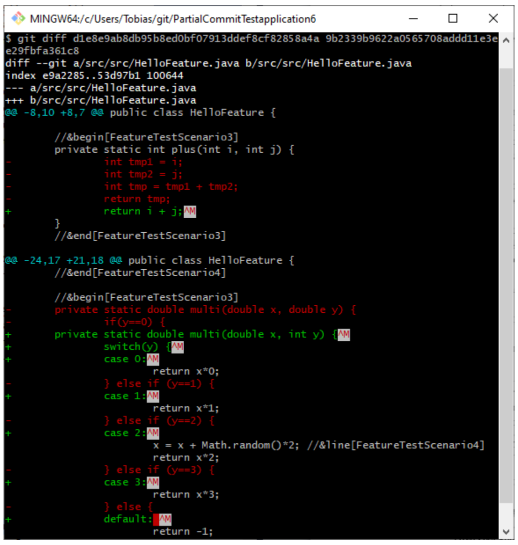
(b) Partial Commit Tool

Figure C.27: Tool Evaluation Partial Commit, Scenario 3, git-diff for New Commits of Feature FeatureTestScenario4



```
Tobias@DESKTOP-LBPKDHI MINGW64 ~/git/PartialCommitTestapplication6 (master)
$ git diff 85a36980a5639bd13b88acca95a219e994aa0703 15d372d836268cf5e9eb0685b03f
7cf18297c31c
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index e9a2285..53d97b1 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;^M
        }
        //&end[FeatureTestScenario3]

@@ -24,17 +21,18 @@ public class HelloFeature {
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {^M
+               switch(y) {^M
+               case 0:^M
                        return x*0;
-               } else if (y==1) {
+               case 1:^M
                        return x*1;
-               } else if (y==2) {
+               case 2:^M
                        x = x + Math.random()*2; //&line[FeatureTestScenario4]
                        return x*2;
-               } else if (y==3) {
+               case 3:^M
                        return x*3;
-               } else {
+               default: ^M
```

(a) Git add −−patch



```
$ git diff d1e8e9ab8db95b8ed0bf07913ddef8cf82858a4a 9b2339b9622a0565708addd11e3e
e29fbfa361c8
diff --git a/src/src/HelloFeature.java b/src/src/HelloFeature.java
index e9a2285..53d97b1 100644
--- a/src/src/HelloFeature.java
+++ b/src/src/HelloFeature.java
@@ -8,10 +8,7 @@ public class HelloFeature {

        //&begin[FeatureTestScenario3]
        private static int plus(int i, int j) {
-               int tmp1 = i;
-               int tmp2 = j;
-               int tmp = tmp1 + tmp2;
-               return tmp;
+               return i + j;^M
        }
        //&end[FeatureTestScenario3]

@@ -24,17 +21,18 @@ public class HelloFeature {
        //&end[FeatureTestScenario4]

        //&begin[FeatureTestScenario3]
-       private static double multi(double x, double y) {
-               if(y==0) {
+       private static double multi(double x, int y) {^M
+               switch(y) {^M
+               case 0:^M
                        return x*0;
-               } else if (y==1) {
+               case 1:^M
                        return x*1;
-               } else if (y==2) {
+               case 2:^M
                        x = x + Math.random()*2; //&line[FeatureTestScenario4]
                        return x*2;
-               } else if (y==3) {
+               case 3:^M
                        return x*3;
-               } else {
+               default: ^M
                        return -1;
```

(b) Partial Commit Tool

Figure C.28: Tool Evaluation Partial Commit, Scenario 3, git-diff for New Commits of Feature FeatureTestScenario3