UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

An Annotation-centric Approach to Similarity Management

Masterarbeit

zur Erlangung des Grades eines Master of Science
im Studiengang Informatik

vorgelegt von

Thomas Schmorleiz

Erstgutachter:	Ralf Lämmel Institut für Informatik
Zweitgutachter:	Martin Leinberger Institut für Informatik

Koblenz, im Februar 2015

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . .

(Ort, Datum) (Unterschrift)

Abstract

Software systems are often developed as a set of variants to meet diverse requirements. Two common approaches to this are ”clone-and-owning” and software product lines. Both approaches have advantages and disadvantages. In previous work [1] we and collaborators proposed an idea which combines both approaches to manage variants, similarities, and cloning by using a *virtual platform* and cloning-related *operators*.
In this thesis, we present an approach for aggregating essential metadata to enable a propagate operator, which implements a form of change propagation. For this we have developed a system to annotate code similarities which were extracted throughout the history of a software repository. The annotations express similarity maintenance tasks, which can then either be executed automatically by propagate or have to be performed manually by the user. In this work we outline the automated metadata extraction process and the system for annotating similarities; we explain how the implemented system can be integrated into the workflow of an existing version control system (Git); and, finally, we present a case study using the *101haskell* corpus of variants.

Zusammenfassung

Um unterschiedliche Anforderungen zu erfüllen, werden Softwaresysteme oft in Form einer Menge von Varianten entwickelt. Zwei gebräuchliche Ansätze für eine solche Softwareentwicklung sind das *clone-and-owning* und die Produktlinienentwicklung. Beide Ansätze haben Vor- und Nachteile. In vorheriger Arbeit mit anderen [1] haben wir eine Idee vorgestellt bei der beide Ansätze verknüpft werden um Varianten, Ähnlichkeiten und Softwareklone zu verwalten. Diese Idee basiert auf einer *virtuellen Plattform* und *Operatoren* für Softwareklone.

In der vorliegenden Arbeit stellen wir einen Ansatz vor um essentielle Metadaten für die Realisierung eines propagate-Operators zu aggregieren. Dafür haben wir ein System entwickelt um Ähnlichkeiten mit Annotationen zu versehen, wobei die Ähnlichkeiten aus der Historie eines Repositories extrahiert werden. Die Annotationen drücken aus wie eine Ähnlichkeit zukünftig gewartet werden soll. Abängig vom Annotationstyp kann diese Wartung automatisiert ausgeführt oder sie muss vom Benutzer manuell betrieben werden. In dieser Arbeit beschreiben wir die automatisierte Extraktion von Metadaten und das System zur Annotation von Ähnlichkeiten; wir erklären wie das System in den Arbeitsfluss eines bestehenden Programms zur Versionierungverwaltung (Git) integriert werden kann; und abschließend stellen wir eine Fallstudie vor, die das *101haskell*-System benutzt.

Acknowledgement

I'd like to thank my supervisor Ralf Lämmel for his support and the great opportunities he has given to me over the years.
The presented work continues previous joint work [1] with Michal Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Stefan Stanciulescu, Andrzej Wasowski, and Ina Schaefer. The work is also inspired by Julia Rubins framework for clone management [2, 3].

I'd like to thank my mother for believing in me and supporting me throughout all my studies.

Contents

Chapter 1

Introduction

Software systems are often developed a set of variants to fulfill potentially conflicting requirements related to aspects like legal frameworks, use cases, or cultural specifics. To create such a set of variants the *clone-and-own* approach is often utilized. That is, assets are copied from existing variants to either start the implementation of a new variant, or to add new features to another existing variant.

Clone-and-owning comes with some advantages [4, 5]. The approach requires no special process, therefore has low adoption costs, and lets developers work independently. On the other hand cloning is associated with some serious disadvantages [4, 5]. Due to the lack of process and monitoring the shared assets are disconnected. Therefore the assets might divergence unintentionally and then have to be synchronized manually. This will not scale well once a large number of variants has to be maintained. These unintentionally divergences might be the result of unpropagated bug fixes, refactoring, or performance improvements.

Another approach to develop as set of variant is *product line engineering* (PLE) [6], which is often proposed to systematically reuse asset across multiple variants. PLE is based on a platform that collects all shared assets and that can be used to derive new variants by selecting certain components like the implementations of features. The advantages of PLE are given by the platform of shared assets. It reduces redundancy and allows for propagation of changes, thereby avoiding unintentional divergence. The disadvantages are mostly related to the migration to PLE. The migration process comes with high risks due to the disruption of the development process [7]. Creating a software product line pro-actively is often not possible since the set of variants, and therefore the set of shared assets is not known at that point [8]. Even with PLE in place clones might still be present [4]. Another issue is that the platform restricts the developers' freedom such that they can not work fully independently.

We therefore need some middle ground that allows for independence of developers and low adoption costs on one hand, but supports systematic clone and similarity monitoring and

management on the other hand. We need an approach that allows cloning while providing means for avoiding unintentional divergence. The approach should additionally allow to detect and remove accidental variation, that is, fragment pairs that were never equal but should be.
In this work, we present an approach to extract information about the sharing of assets from the history of a given repository. More specifically, we store metadata about the similarities of fragments across all variants. We can detect two types of clones [9]. Type-1 clones, where two fragments are equal modulo whitespace. And Type-3 clones, where pieces of code are added, deleted, and modified when comparing the two fragments. We then let the user explore the similarities in a web interface and manage the similarities via annotations that state how each similarity should be maintained. For instance, an annotation can express that the contents two fragments diverged unintentionally and that equality should be restored. An annotation can also express that two fragments that have been equal should indeed be equal, because their difference is based on accidental variation. All maintenance tasks can then either be executed automatically or have to be performed manually by the user.

1.1 Context of this work

In collaboration with others we have proposed a general approach that aims to combine the advantages of PLE and clone-and-owning while diminishing their disadvantages [1]. The work relies on a *virtual platform* which does not store shared assets explicitly like in PLE, but rather holds *metadata* about the project including metadata about the similarities and clones across the variants of the software system. We further identified a set of cloning-related *operators* that both use and modify existing metadata and produce new metadata. Examples of such operators are:

- clone variant to start the implementation of a new variant
- clone feature from one to another existing variant
- propagate changes in one fragment to another fragment in the same or another variant

In this context this work focuses on the aggregation of critical metadata for the propagate operator. It implements a form of change propagation, that is, it synchronizes fragments that should be equal but have diverged over time. The operator is enabled both by the extracted similarities and the annotations of such similarities by the user.

1.2 Related work

Variability. This thesis implements the propagate operator in the context of our paper with others on the notion of a *virtual platform* [1]. The work is also inspired by Julia Rubin's framework for clone management [2, 3]. The paper proposes a new strategy to manage variability in a software system. Berger and others have conducted a survey of variability-modeling practices used in the industry [8]. A systematic review of literature on variability modeling using software product lines has been carried out by Chen et al. [10]. The use of cloning in such software product lines has been addressed by Dubinsky and others in an exploratory study [4]. Fischer et al. have proposed an approach where basic cloning is enhanced by systematic reuse of variants [11].
In our work, cloning and similarities have to be detected and their evolutions extracted and annotated such that they can be systematically maintained further.
Clone detection. For the detection of clones we use an approach by Cordy et al. where fragment tokens are pretty-printed into many lines [12]. Cordy and others have also conducted a survey of the general capabilities of available clone detection techniques [13]. An evaluation of modern clone detection tools was done by Svajlenko et al. [14]. Our detection mechanism extracts fragment pairs that are not equal but just similar. A hybrid approach to detect both type-1 and type-3 clones and a taxonomy for developers' editing activities has been proposed by Roy [15].
Clone evolution. Our approach also extracts the evolution of such similarities from the history of a given repository. Evolutions of cloning groups are called cloning genealogies. Notkin and others have presented and build a *clone genealogy tool* to extract information about the evolution of code clones from a repository's history [16]. The fault-proneness of clone migration in clone genealogies has been studied by Xie et al. [17]. The work by Mondal focuses on the stability of such code clones during software maintenance [18]. Saha et al. have studied the general evolution of clones in an exploratory way [19]. Schneider and others proposed an approach to extract and classify cloning genealogies after tracking fragments through versions of a software system [20].
Reverse engineering. Our metadata extraction process can be seen as a form of reverse engineering. We provide the user with the information extracted from the history of a given repository. Wu et al. have examined the use of reverse engineering from version control systems to help developers to understand, develop, and maintain software systems [21]. While we use annotations to express similarity-maintenance tasks, Brühlmann and others use annotations in a generic approach to enrich reverse engineering with human knowledge [22].
Change propagation. Hemel and others extracted the variability in a set of Linux variants [23]. Their work discusses that changes are often not propagated from the main Linux kernel to its variants. The term "late propagation" refers to a pattern of commits where clones first diverge but are then synchronized again and converge. Our tool discourages such late

propagation and aims to diminish unintentional divergence. Mondal and others have studied the late propagation of near-miss clones [24], while Barbour et al. have defined types of late propagation and discuss why such propagation is indeed harmful [25]. The difficulty of maintaining software in the presence of clones has been studied by Chatterji and others [26].

Clone management. Our approach supports management of similarities and clones. A survey by Roy et al. points out past and current achievements in clone management [27]. Their work mentions annotations as a way for the developer to indicate the intents of cloning. While Nguyen and others utilize annotations in this way in their work on JSync [28], in our study annotations are used to express maintenance tasks. Yamanaka et al. introduce a system for notifying developers about the creation and change of clones [29], we notify developers when they have to act on the fragments of an annotated similarity. The work by Kotschke summarizes the state-of-the-art in clone management with regard to detection, tracking, presentation, and removal [30].

1.3 Research questions

Our research questions are about how useful annotations of similarities are for managing and monitoring sharing of assets. In the following we distinguish between unintentional divergence as an evolution where the underlying fragments were once equal but become unequal unintentionally, and accidental variation where the two fragments have never been equal but should be.

In terms of management we ask about how unintentional divergence and accidental variation can be diminished respectively removed by using annotations:

- *Q1: How much unintentional divergence and accidental variation can be*
  - *annotated*
  - *and finally eliminated guided by annotations?*

  To measure the elimination of unintentional divergence and accidental variation we ask:

  - *Q11: By what percentage can the number of distinct fragments be reduced?*

The second research question is about the added value of being able to monitor similarities and sharing of assets:

- *Q2: How much sharing and similarity can be achieved once unintentional divergence and accidental variation are eliminated with the help of annotations?*

  We measure sharing by two orthogonal questions:

- *Q21: What is the median and average number of variants each fragment is shared in?*
- *Q22: What is the median and average percentage of originality, that is the number of unshared fragments, of each variant?*

To measure similarity we ask:

- *Q23: What is the median and average similarity of fragments?*

The last research question focuses on the insights developers gain by inspecting the final equality classes.

- *Q3: Which inconsistencies can be identified when inspecting the final equality classes?*

  Regarding the emerging equality classes we ask:

  - *Q31: What categories of equality classes, that is, reasons for the underlying sharing, can be identified?*
  - *Q32: When identifying variants that do not contribute to a particular equality class by their fragments, which inconsistencies of these variants can be identified?*

1.4 Roadmap

The rest of this thesis is structured as follows.
The next chapter will explain what kind of metadata needs to be extracted to build the virtual platform about shared assets. The final result will be a set of similarities between fragments.
The third chapter will illustrate the process of annotating the extracted similarities. We will explain both the semantics and the applicability of each annotation category. We will introduce a web application (called ”Ann”) to explore the similarities in a given repository and finally annotate them. Finally, we present some approaches to automate annotation where possible.
The next two chapters will focus on how to act on the annotated similarities. Namely Chapter 4 will describe the use of annotations for automatic change propagation; and Chapter 5 will present the emerging todo list of manual maintenance tasks for the user.
Chapter 6 will illustrate how to integrate our system into the workflow of an existing version control system (Git). Here we will also introduce a set of new Git commands.
Next, we will presents results of a case study using the *101haskell* software chrestomathy [31,32], a corpus of variants for educational purposes and a subproject of the *101companies* project [33].

Finally, we will draw a conclusion, discuss threats to validity, and outline some future work.

Chapter 2

Extracting Metadata

In this chapter we will outline how we extract metadata about the sharing of assets. We will explain the initial extraction process, how we traverse the history of the repository during the extraction, and, finally, how we update metadata when new commits are added to the repository's history.

2.1 Initial extraction process

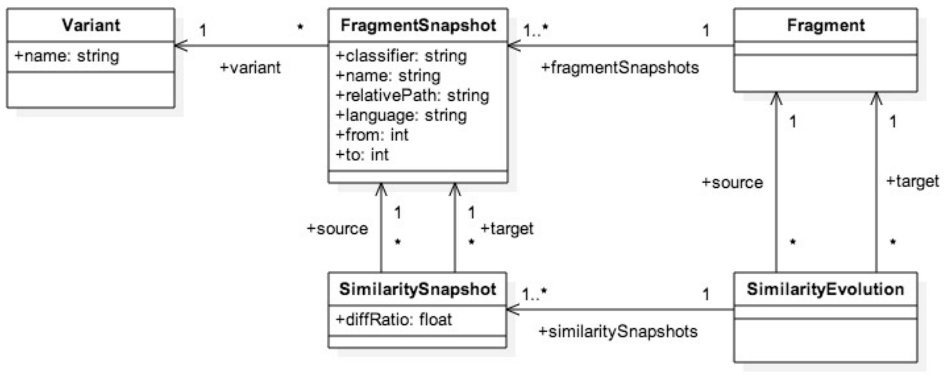


Figure 2.1: Class diagram for pre-annotation metadata.

This section is dedicated to the steps in the initial metadata extraction process that have to be performed before the user can annotate similarities. All steps extract metadata from the given repository and thereby build the basis of a virtual platform. Figure 2.1 shows a model of the extracted metadata. Each of the following extraction steps populates this model, that is, each step creates instances of the classes shown, links them to other instances or enriches existing instances of classes. First, we extract all variant names at each commit point. Then, we will create instances of fragment snapshots, each bound to a commit point and contained in a variant. We then instantiate fragments as series of fragment snapshots. Next, we extract similarity snapshots between fragment snapshots, each associated with a specialized ”diff ratio” as a similarity value. Finally, we create instances of similarity evolutions by listing similarity snapshots.

2.1.1 Variants

The first step extracts the names of variants that can be found throughout the history of the repository. Variants can be organized in many ways including dedicated folders, branches or repositories. Here we present the realization of the extraction of variant names for the case of dedicated folders.

The variant extractor is given a list of paths ps, each pointing to a possible root folder of variation. For each commit c and each path p in ps the extractor checks whether a folder at p exists at c. The first hit (regarding the order given by ps) marks the root folder of variation at c. The names of the sub-folders are then extracted as variant names at c.

This step also attempts to detect possible renaming of variants from each commit c_{i} to next the commit $c_{\mathrm{i+1}}$, represented by a list of pairs (s, t) where s is the source variant name at c_{i} and t is the target variant name at $c_{\mathrm{i+1}}$: If a folder for t did not exist at c_{i}, but files were moved into a new folder for t in $c_{\mathrm{i+1}}$ then we check from which source variant folder s the majority of the files were moved from and we create a renaming (s, t). If s still exists at $c_{\mathrm{i+1}}$ we mark the renaming as a ”split”, indicating that only parts of s were taken to create t.

2.1.2 Fragment snapshots

Now that we have identified the variant names at each commit, we extract the fragments for each commit and each variant. A fragment is a range of consecutive lines of source code that correspond to a hand-selected node in the associated abstract syntax tree (AST) of the containing source file. Each fragment is identified by a classifier and a name. Classifiers correspond to the syntactic category of the hand-selected node. In the case of Haskell examples for classifiers are ’data’, ’type’, or ’function’; and names are the names of data types, type synonyms, or functions. To extract these classifier/name pairs and the corresponding line ranges we need languages-specific technologies we call ”fact extractors” and ”fragment locators”. We have developed such technologies for a board spectrum of software languages, including Java, Haskell, Python, HTML, and JSON. Due to the fact that we have to extract fragments throughout the history of a repository we further refer to fragments at specific commit points as ”fragment snapshots”.

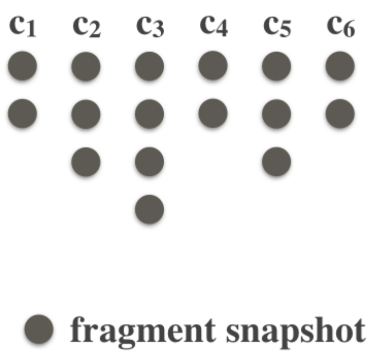


Figure 2.2: Result of extracting fragment snapshots.

Figure 2.2 illustrates the result of this step; for each commit we extract a set of fragment snapshots. Each fragment snapshot is finally stored as a JSON document with the following attributes:

```
{
  "variant"        : "writerMonad",
  "relative_path" : "Cut.hs",
  "classifier"     : "function"
  "name"           : "cutLogCompany",
  "from"           : 8,
  "to"             : 15,
  "sha"            : "02b6b11...",
  "language"       : "Haskell"
}
```

2.1.3 Fragments

Fragment snapshots are bound to commits by their sha attribute. We further refer to ”fragments” as series of fragment snapshots that can be found at consecutive commits. That is, to extract a fragment we need to list its snapshots. More specifically, we need to link snapshots from consecutive commits that we believe to represent the same fragment at the respective commits. To link snapshots we have to be able to deal with renaming at variant, file, and fragment level, with content changes, and with interaction of fragment renaming

and other content changes. Schneider et al. have proposed a framework to extract cloning genealogies. In their work they extract functions as fragments and are able to link fragments from consecutive commits in different scenarios concerning signature, body, and location changes of the function [20]. Our approach is the following.
Given a fragment snapshot $fs_{\mathrm{c_i}}$ at commit c_{i} we attempt to find a fragment snapshot at $c_{\mathrm{i\text{-}1}}$ that represents the same fragment. We first check all fragment snapshots at $c_{\mathrm{i\text{-}1}}$ that can be found in the same variant and file (modulo variant and file renaming). We have identified four cases:

1. **No change.** If we find a fragment snapshot $fs_{\mathrm{c_{i\text{-}1}}}$ at $c_{\mathrm{i\text{-}1}}$ with the exact same content as $fs_{\mathrm{c_i}}$ we link the two fragments.
2. **Pure fragment renaming.** If we find a fragment snapshot $fs_{\mathrm{c_{i\text{-}1}}}$ at $c_{\mathrm{i\text{-}1}}$ with the exact same content except for the fragment name we link the fragments. We also set an attribute $is\_renamed$ of $fs_{\mathrm{c_i}}$ to $true$.
3. **Mass fragment renaming.** If we find a fragment snapshot $fs_{\mathrm{c_{i\text{-}1}}}$ at $c_{\mathrm{i\text{-}1}}$ with the exact same content except for any fragment names found in the same variation and in either files of $fs_{\mathrm{c_{i\text{-}1}}}$ or $fs_{\mathrm{c_i}}$ we link the fragments. We also set an attribute $is\_changed$ to $true$. If the fragment name of $fs_{\mathrm{c_i}}$ has changed we additionally set $is\_renamed$ of $fs_{\mathrm{c_i}}$ to $true$.
4. **Pure content editing.** If we find a fragment snapshot $fs_{\mathrm{c_{i\text{-}1}}}$ at $c_{\mathrm{i\text{-}1}}$ with the exact same classifier and name as $fs_{\mathrm{c_i}}$ we link them and set $is\_changed$ to $true$.

If no linkable fragment snapshot was found we extend the search space from file scope to the variant scope. If we still cannot find a linkable fragment snapshot we have to assume that $fs_{\mathrm{c_i}}$ marks the creation of a fragment, that is, $fs_{\mathrm{c_i}}$ is the first snapshot of a fragment. In this case we set an attribute is_new of $fs_{\mathrm{c_i}}$ to $true$. If $i == 0$, that is we are at the first commit, we set is_new to $true$ for all fragment snapshots.
The order of cases listed above does in fact matter and it is being used when checking each candidate fragment snapshot at $c_{\mathrm{i\text{-}1}}$. We illustrate the importance of the order in the following example: Let the only fragment snapshot that can be found at $c_{\mathrm{i\text{-}1}}$ be fs_1 with the following content:

```
-- sum of two numbers
sum x y = x + y
```

Further, let the only fragment snapshots at c_{i} be fragment fs_2 with content

```
-- sum of two numbers
plus x y = x + y
```

and fragment fs_3 with content

```
-- sum of many numbers
sum xs = foldr plus 0 xs
```

At c_{i} we attempt to find a fragment snapshot from $c_{\text{i-1}}$ for both fs_2 and fs_3 that we can link to. fs_2 can be linked to fs_1 based on case 2. fs_3 can be linked to fs_1 based on case 4. The order of the cases expresses that we prefer matching content (modulo the fragment's name) over sole name matching. We therefore link fs_2 to fs_1. We further extract fs_3 as the first snapshot of a new fragment by setting its is_new attribute to $true$.
While most cases are straightforward, the third case tries to deal with the difficult situation in which a fragment was both renamed and its content was changed. We will come back to this issue in the section about threats to validity in the last chapter.

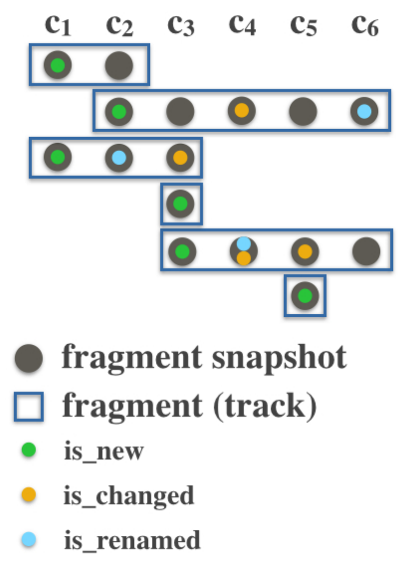


Figure 2.3: Result of extracting fragments.

Figure 2.3 summarizes this step: For each commit we assign each fragment snapshot to a series of snapshots that all represent the same fragment throughout the history of the repository. We further add attributes to each snapshot, indicating whether the snapshot at hand is new, changed, or renamed.

2.1.4 Similarity snapshots

Similarity snapshots are relations of two fragment snapshots at a commit point, associated with a similarity value. That is, for n fragment snapshots at a commit we have $n*n$ possible

similarity snapshots. To reduce the amount of similarities stored and later to be processed by the user we ask the user to provide a threshold, a value each similarity snapshot's value has to pass to be stored.
To compute all similarity snapshots at a commit point we simply compare all fragments snapshots at the commit with each other. To compute the actual similarity value we use an approach introduced by Cordy et al. [12]. The idea is to pretty-print the token sequence of each fragment into many lines and then use sequence matching (diff ratio) as a measure. By using this approach we ignore dissimilarities that are solely based on whitespace. The value of this measure lies between 0 and 1, where 1 indicates a perfect match between two fragments modulo formatting.

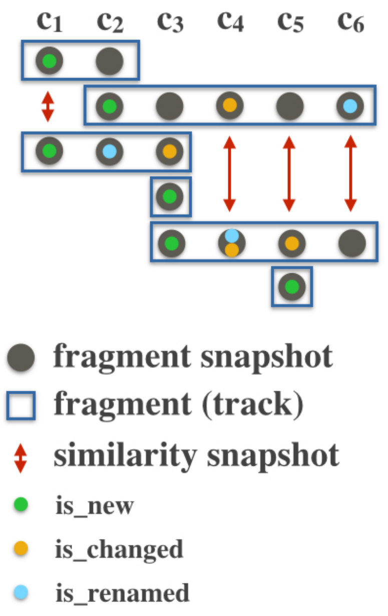


Figure 2.4: Result of extracting similarity snapshots.

Figure 2.4 illustrates the extracted similarities: At each commit point we have collected a set of similarity snapshots between fragment snapshots. This metadata forms the basis of the virtual platform of our approach as it captures the sharing of fragments at each commit point. Each similarity is stored as a JSON document with the following attributes:

```
{
  "fragment_snapshot_1_id" : ObjectId("d165f70d..."),
  "fragment_snapshot_2_id" : ObjectId("d165f74d..."),
  "sha" : "2d3dbab...",
  "diff_ratio" : 0.78
}
```

2.1.5 Similarity evolutions

We have extracted similarity snapshots, each bound to a specific commit. In order to compute the applicability of annotations we have to extract similarity evolutions, series of similarity snapshots from consecutive commits. That is, we extract all similarity snapshots that occurred between two fragments throughout the history. For this we simply have to extract all similarity snapshots between all fragment snapshots of the two fragments.

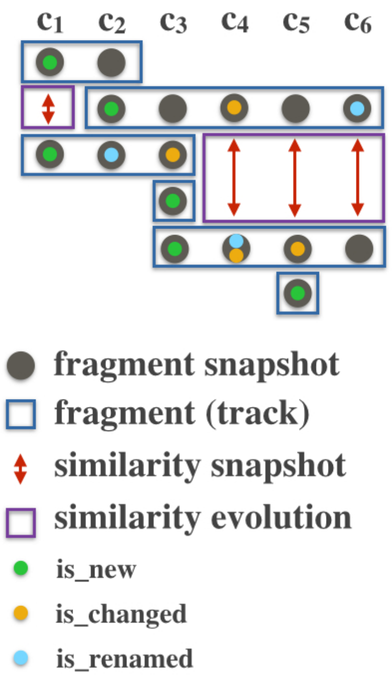
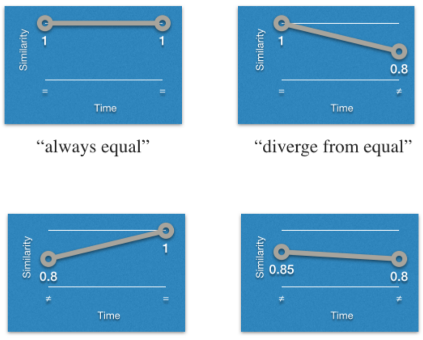


Figure 2.5: Result of extracting fragment snapshots.

Figure 2.5 illustrates the evolutions. For the similarity evolution of two fragments we list consecutive similarity snapshots between the snapshots of the respective fragments at the respective commits. Each similarity evolution is stored as follows:

```
{
  "similarity_ids"      : [...],
  "min_similarity_id"   : ObjectId("...d1694e19"),
  "max_similarity_id"   : ObjectId("...d16943a5"),
  "last_similarity_id"  : ObjectId("...d169434c"),
}
```

The similarity evolutions are what will get annotated, that is, the user specifies how a given evolution should be further maintained. To compute the applicability of annotations we finally associate each similarity evolution with one of the four following categories:



"always equal" "diverge from equal"

"converge to equal" "always non-equal"

Figure 2.6: Simplified illustration of evolution categories

- Always Equal holds similarity evolutions that have always been equalities, that is, the similarity value was always 1.
- Converge to Equal holds similarity evolutions where the similarity value was once below 1 but then increased to 1 at the HEAD of the branch.
- Diverge from Equal holds similarity evolutions where the similarity value was once 1 but then decreased to below 1 at the HEAD of the branch.
- Always Non-equal holds similarity evolutions where the similarity value was constant or changing but always below 1.

We will later use these abstraction over the concrete similarity values to define the applicability of each annotation category. To compute the category of a concrete similarity evolution we use its minimum, maximum, and last similarity value:

- If maximum and minimum similarity value are both 1, assign category Always Equal.
- If minimum similarity value is below 1 and last similarity value is 1, assign category Converge to Equal.
- If maximum similarity value is 1 and last similarity value is below 1, assign category Diverge from Equal.

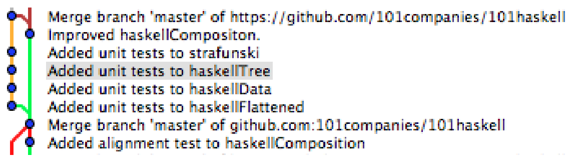
- If maximum and minimum similarity value are both below 1, assign category Always Non-equal.

2.2 Traversing commits

This section explains how we traverse the history of a branch in a repository during the various extraction processes.
When reviewing the extraction steps we can see that each one falls into either of the two following categories:

- **Commit-scoped**. The extraction step extracts metadata that is bound to a specific commit. That is, for a given commit c_i the step only uses information available at c_i to extract new data. The extraction steps for *fragment snapshots* and for *similarity snapshots* fall into this category.
- **Commit-linking**. The extraction step uses metadata that was already extracted at the previous commit and links the metadata to create new metadata. That is, for a commit c_i the step links metadata bound to c_i to metadata bound to c_{i-1}. The extraction steps for *fragments* and for *similarity evolutions* fall into this category.

For the extraction steps that are linking metadata between commits we have to traverse all commits in an appropriate manner. Most Git APIs list all commits sorted by the commits' timestamps. However, commits cannot not simply be traversed in this linear fashion. We explain a correct approach using the following example.



Merge branch 'master' of https://github.com/101companies/101haskell
Improved haskellCompositon.
Added unit tests to strafunski
Added unit tests to haskellTree
Added unit tests to haskellData
Added unit tests to haskellFlattened
Merge branch 'master' of github.com:101companies/101haskell
Added alignment test to haskellComposition

Figure 2.7: Snippet of the history of the 'master' branch's commit graph of 101haskell

Figure 2.7 shows part of the commit graph of the master branch of the *101haskell* project. We can see that the graph is not just a sequence of commits, but contains local branches of users that are eventually merged with the branch of the main repository. Suppose a fragment f was added at the commit c_i ("Added unit tests to strafunski"). The extraction for fragment snapshots creates a new snapshot fs_1 of f and the extraction for fragments starts a new series of fragment snapshots starting with fs_1.
The next commit according to timestamps is c_{i+1} ("Improved haskellComposition"). According to the commit graph it is contained in an other local branch than c_i. Further suppose that fragment f was not added in the commit c_{i+1}.

At $c_{\mathrm{i+2}}$ (”Merge branch of...”) we extract a new fragment snapshot fs_2 of f. However, if we only look at the last commit point according to the timestamps ($c_{\mathrm{i+1}}$) we would not find a snapshot to link to. Therefore, we would start a new series of fragment snapshots at $c_{\mathrm{i+2}}$ for f. To avoid such false metadata we do not look at the last commit according to the timestamps, but at the parent commits according to the commit graph. That is, at $c_{\mathrm{i+2}}$ we have to look at both commits c_{i} and $c_{\mathrm{i+1}}$ and therefore find fs_1 to link to fs_2.

2.3 Updating metadata

As developers add more commits to the history of the repository we have to update the metadata. That is, for each new commit we need to:

- Extract variants
- Extract fragment snapshots
- Extend existing or create new fragments
- Extract similarity snapshots
- Extend existing or create new similarity evolutions

The updating of metadata is incremental, that is, for each commit we reuse metadata from the last commit for files that were not touched.

Chapter 3

Annotating similarities

After the initial metadata extraction process or after updating metadata our system has stored a set of similarity evolutions. This chapter is about the next step: annotating the evolutions by expressing how they should be maintained further. We will first identify the different annotation categories and define their applicability. Then, we present a web application to explore and annotate the similarities. Finally, we will explain how to automatically infer some annotations by making use of the fact that fragments and their similarities form a graph.

3.1 Annotation categories

We identify seven categories of annotations, each stating a maintenance tasks for the underlying similarity evolution:

- Maintain Equality by *automatic* three-way merge when one or both fragments of the similarity evolution change and potentially by *manual* conflict resolution.
- Restore Equality by *automatically* propagating changes from one fragment to the other if a direction of propagation is defined, otherwise by *automatic* three-way merge and potentially by *manual* conflict resolution.
- Establish Equality by *manual* actions on fragments that have never been equal.
- Maintain Similarity by *manual* actions when the similarity changes.
- Restore Similarity by *manual* actions until a target similarity value is reached.
- Increase Similarity by *manual* actions.
- Ignore by not reporting the similarity or equality anymore to the user.

Note that here ”Similarity” always refers to non-equality similarities.

3.2 Structure of annotations

Each annotation has four attributes:

- *category*: This is the name of the category of the annotation.
- *intent*: This is a comment the user can make when creating or updating the annotation. Examples of what this comment can be about include
  - why the annotation category was chosen,
  - which manual actions on fragment-content level are required to perform a manual task (e.g., with line numbers for *Increase Similarity*), or
  - who a manual task is assigned to.
- *propagate_to*: This attribute can only be used for annotations of the category Restore Equality. It expresses in which direction changes should be propagated in. Possible values are $Left$ and $Right$. For a similarity evolution $E_{\mathrm{f_1,f_2}}$ of fragments f_1 and f_2 the value $Left$ indicates that changes should be propagated from f_2 to f_1. $Right$ has an analogous meaning.
- *similarity_evolution_id*: This is the reference to the similarity evolution that is annotated.

3.3 Applicability of annotations

An annotation expresses a maintenance task for the underlying similarity evolution. Therefore, each annotation category is only applicable to a certain subset of the similarity-evolution categories. The following matrix defines the applicabilities:

	Maintain Equality	Restore Equality	Establish Equality	Maintain Similarity	Restore Similarity	Increase Similarity	Ignore
Always Equal	auto	no	no	no	no	no	auto
Converge to Equal	auto	no	no	no	no	no	auto
Diverge from Equal	no	auto	no	manual	manual	manual	auto
Always Non-equal	no	no	manual	manual	manual	manual	auto

The matrix also shows for each annotation category and each evolution category whether the expressed maintenance tasks is automatically executable by default or has to be performed by the user. With this is mind we say that an annotation may be ”executable”.
The applicability for Maintain Equality is only given in the presence of an equality at head. Restore Equality is tailored for the case where a similarity diverged from equality, but should be increased back to equality by using change propagation. Establish Equality

is tailored similarly. All other annotations that are only applicable to evolutions where the current similarity value is not an equality have to be performed manually by the user. Ignore requires no special actions, only instructs the system to no longer report the evolution, and is therefore automatically executable. We will later explain in detail how annotations are executed.

3.4 Web application for similarity exploration and annotation

We have created a web application called ”Ann” for the user to explore and annotate similarities in a systematic manner. We followed some principles regarding user experience (UX) design:

1. Give the user feedback; here about the progress of annotating similarities
2. Present processable amounts of information; here by not giving the user a flat list of similarities, but by letting the user explore similarities in a hierarchy of variants, folder and files and by thereby providing context
3. Avoid repetition of tasks; here by inferring annotations where possible

This section is about the annotation application following these principles. We will first discuss a set of views that inform the user about the state of the repository. Next, we will present two annotators, one commit-centric, one variant-centric, that allow for systematic similarity annotation (principles 1 & 2). In the next section we present two approaches to infer annotations by rules and by utilizing equality classes (principle 3).

3.4.1 Views

When annotating the similarities the user should make informed decisions. Ann therefore provides the user with information about the state of the repository. We have developed two ”views” to enable an overview over both the variants and actions throughout the history of the repository.

Variants

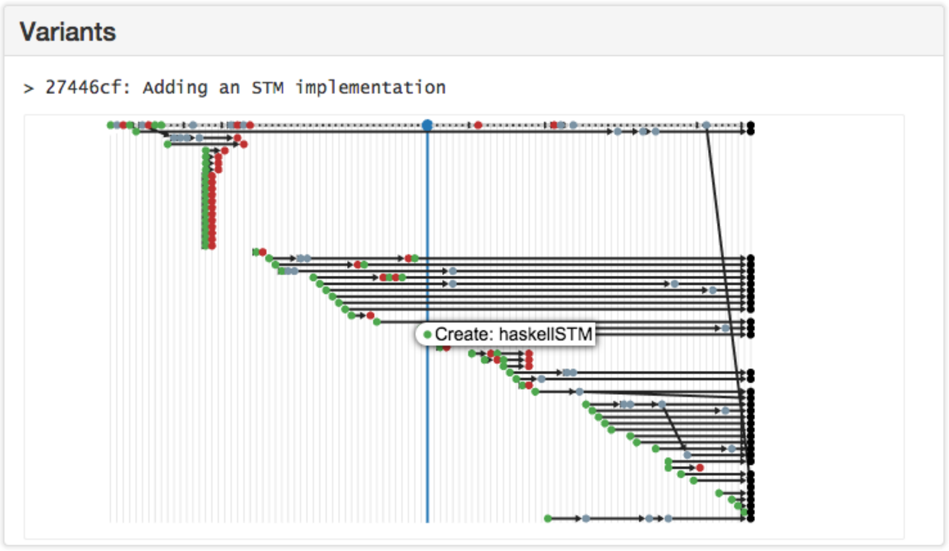
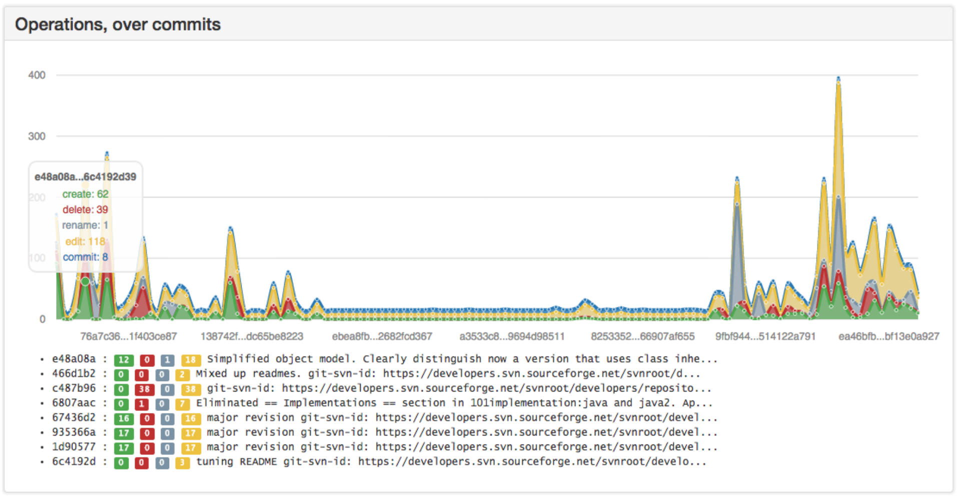


Figure 3.1: Variants throughout the history of the *101haskell* repository.

The first view shows variants over time. On the horizontal axis we show all commits at which at least one variant was created, renamed, or deleted. On the vertical axis we show the variants, sorted by creation time. If a variant was touched at a given commit point we create a color-coded circle where green, gray and red indicate variation creation, renaming and respectively deletion. As discussed in the Chapter 2, we can also detect the splitting of variants. The view indicates such splits by lines connecting different variants. Hovering over a circle additionally reveals details about the underlying commit including its identifier (sha) and message.

Edit operations



```
• e48a08a : 12 0 1 18 Simplified object model. Clearly distinguish now a version that uses class inhe...
• 466d1b2 : 0 0 0 2 Mixed up readmes. git-svn-id: https://developers.svn.sourceforge.net/svnroot/d...
• c487b96 : 0 38 0 38 git-svn-id: https://developers.svn.sourceforge.net/svnroot/developers/reposito...
• 6807aac : 0 1 0 7 Eliminated == Implementations == section in 101implementation:java and java2. Ap...
• 67436d2 : 16 0 0 16 major revision git-svn-id: https://developers.svn.sourceforge.net/svnroot/devel...
• 935366a : 17 0 0 17 major revision git-svn-id: https://developers.svn.sourceforge.net/svnroot/devel...
• 1d90577 : 17 0 0 17 major revision git-svn-id: https://developers.svn.sourceforge.net/svnroot/devel...
• 6c4192d : 0 0 0 3 tuning README git-svn-id: https://developers.svn.sourceforge.net/svnroot/develo...
```

Figure 3.2: Actions throughout the history of the *101haskell* repository.

The second view focuses on the editing actions and Git operations performed throughout the repository's history. The goal is for the user to understand which subsequences of the history are of interest and should be focused on later. At each commit point we show the number of file creations, renaming, edits, and deletions.

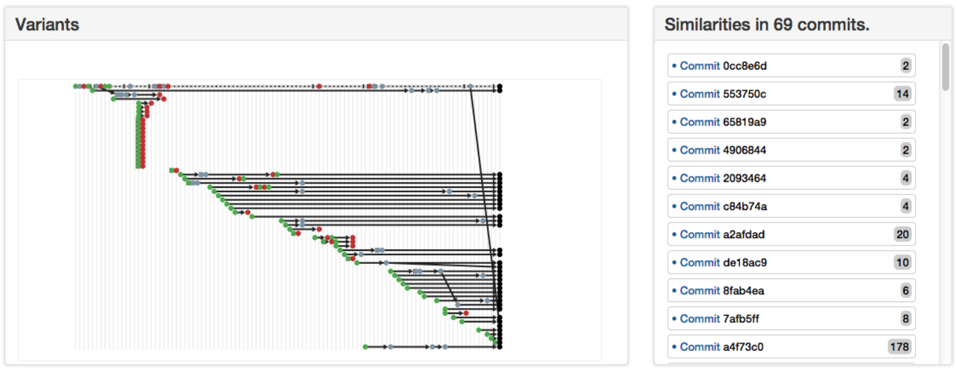
3.5 Annotators

Once the user is informed about the state of a repository, he or she can annotate the extracted similarities. We are providing two annotators which will be discussed in the following.

3.5.1 Commit-centric

This annotator is suited for an annotation process where the user wants to systematically explore the history of the repository.

Initial state



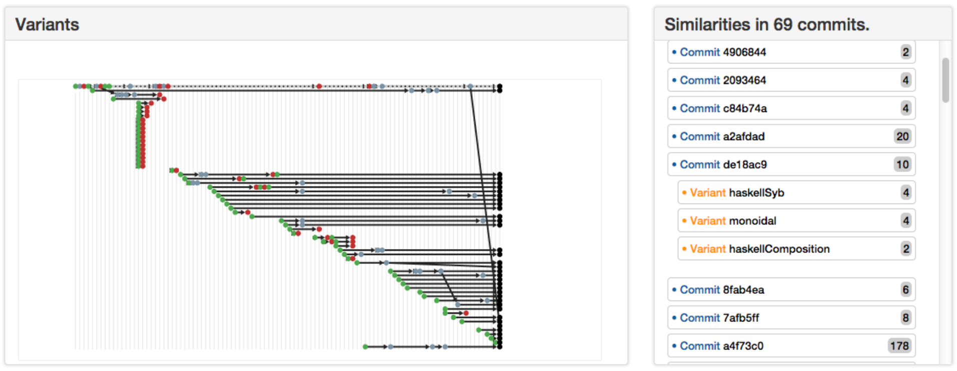
Similarities in 69 commits.

• Commit 0cc8e6d	2
• Commit 553750c	14
• Commit 65819a9	2
• Commit 4906844	2
• Commit 2093464	4
• Commit c84b74a	4
• Commit a2afdad	20
• Commit de18ac9	10
• Commit 8fab4ea	6
• Commit 7afb5ff	8
• Commit a4f73c0	178

Figure 3.3: The commit-centric annotator in its initial state.

Figure 3.3 show the commit-centric annotator it its initial state. On the left we present the variants like we do in the variant view described above. On the right we list all commits at which at least one similarity evolution **starts**. That is, we list commits at which a similarity between two fragments emerged for the first time by passing the user-defined threshold regarding similarity values. At each commit we show the number of similarity evolutions that started at this commit point.

Selecting a variant



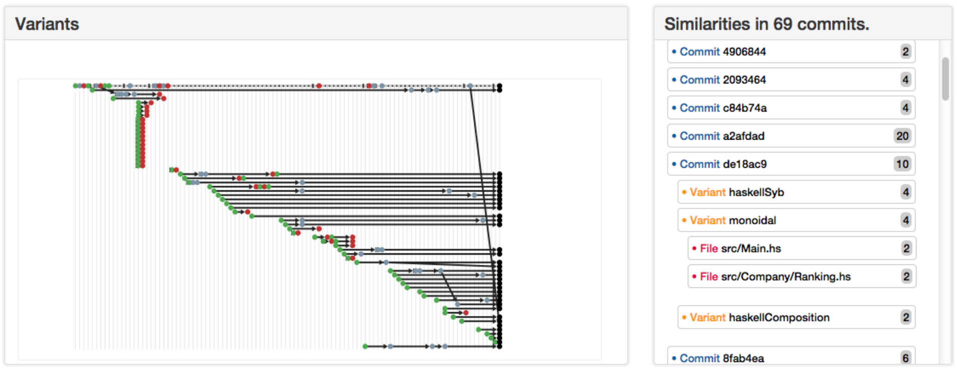
Similarities in 69 commits.

• Commit 4906844	2
• Commit 2093464	4
• Commit c84b74a	4
• Commit a2afdad	20
• Commit de18ac9	10
• Variant haskellSyb	4
• Variant monoidal	4
• Variant haskellComposition	2
• Commit 8fab4ea	6
• Commit 7afb5ff	8
• Commit a4f73c0	178

Figure 3.4: Commit-centric annotator with an expanded commit.

The user can then select a commit he or she wants to explore in terms of similarities. This expands the commit by showing a list variants with their current names at the commit point. Ann only shows variants for which a similarity emerged, again with the number of similarities shown on the right.

Selecting a file



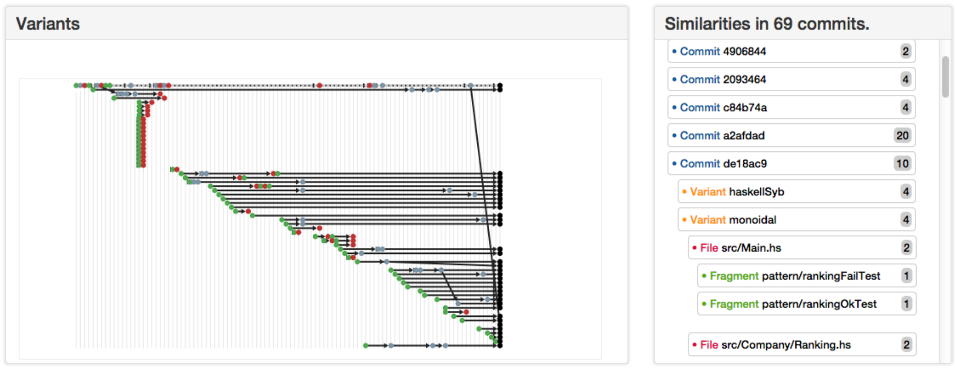
Similarities in 69 commits.

- Commit 4906844 2
- Commit 2093464 4
- Commit c84b74a 4
- Commit a2afdad 20
- Commit de18ac9 10
  - Variant haskellSyb 4
  - Variant monoidal 4
    - File src/Main.hs 2
    - File src/Company/Ranking.hs 2
  - Variant haskellComposition 2
- Commit 8fab4ea 6

Figure 3.5: Commit-centric annotator with an expanded variant.

After selecting a variant Ann shows all the variant's files that contain fragments for which we have extracted a similarity that first emerged at the selected commit point.

Selecting a fragment



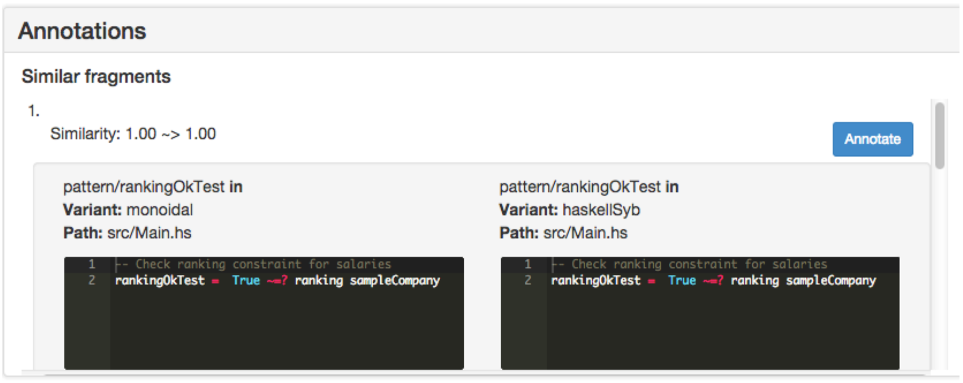
Similarities in 69 commits.

- Commit 4906844 2
- Commit 2093464 4
- Commit c84b74a 4
- Commit a2afdad 20
- Commit de18ac9 10
  - Variant haskellSyb 4
  - Variant monoidal 4
    - File src/Main.hs 2
      - Fragment pattern/rankingFailTest 1
      - Fragment pattern/rankingOkTest 1
    - File src/Company/Ranking.hs 2

Figure 3.6: Commit-centric annotator with an expanded file.

Finally, the user can select a fragment in the file. All fragments are presented as pairs of fragment name and classifier. This and the previous steps provide a systematic way to explore the similarities at a given commit point. In the example above we have navigated to a fragment "pattern/rankingOkTest" in the file "src/Main.hs" in a variant "monoidal" at a commit that is identified by the shortened sha "de18ac9".

Selecting a fragment similarity



1.
Similarity: 1.00 ~> 1.00

pattern/rankingOkTest **in**
Variant: monoidal
Path: src/Main.hs

```
-- Check ranking constraint for salaries
rankingOkTest =  True ~=? ranking sampleCompany
```

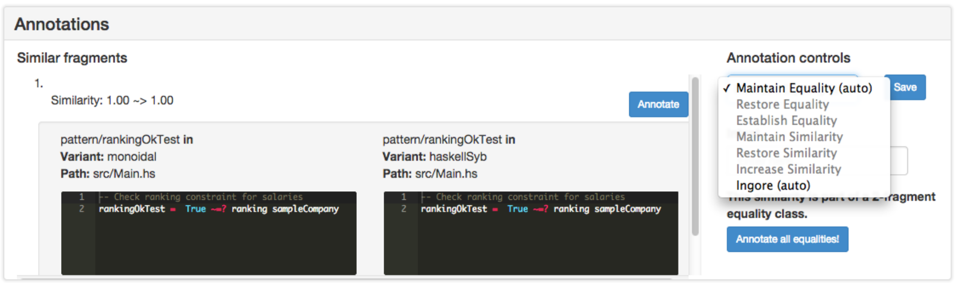
pattern/rankingOkTest **in**
Variant: haskellSyb
Path: src/Main.hs

```
-- Check ranking constraint for salaries
rankingOkTest =  True ~=? ranking sampleCompany
```

Figure 3.7: Commit-centric annotator with similarities for a selected fragment.

After selecting a fragment in the explorer Ann shows all other fragments that first became similar at the selected commit point. For each fragment pair it shows the content of both fragments with their locations (variant and file). The user is also presented with two similarity values in the similarity evolution. The first value is the value at the selected commit point while the second value is the similarity at the HEAD of the repository. If the values differ we show the contents of the fragments both at the current commit point and at the HEAD. Figure 3.7 shows that ”pattern/rankingOkTest” is for example similar to a fragment also called ”pattern/rankingOkTest”, but located in another variant. The fragments are equal both at the selected commit point and at the HEAD.

Annotating a similarity



1.
Similarity: 1.00 ~> 1.00

pattern/rankingOkTest **in**
Variant: monoidal
Path: src/Main.hs

```
-- Check ranking constraint for salaries
rankingOkTest =  True ~=? ranking sampleCompany
```

pattern/rankingOkTest **in**
Variant: haskellSyb
Path: src/Main.hs

```
-- Check ranking constraint for salaries
rankingOkTest =  True ~=? ranking sampleCompany
```

This similarity is part of a 2-fragment equality class.

Figure 3.8: Commit-centric annotator with annotation controls for a selected similarity.

After selecting one of the similarities the user can decide how to annotate it by picking the annotation category in a select list where only applicable categories are enabled. The user can also enter some intent, and, if Restore Equality is selected, the direction in which changes to be propagated in, that is, which fragment's changes should be overwritten by the changes of the other fragment. Ann stores the annotation and additionally keeps track of the current similarity value at HEAD. This value will later be used when Ann updates annotations based on user actions. Updating of annotations will be discussed in chapter 5.

3.5.2 Variant-centric

The variant-centric annotator is suitable when the user wants to systematically explore the variants as they can be found at the HEAD of the repository's branch. This annotator is tailored for the use case where the history is less relevant to the user, but he or she rather wants to focus on the current set of variants.

Selecting a variant

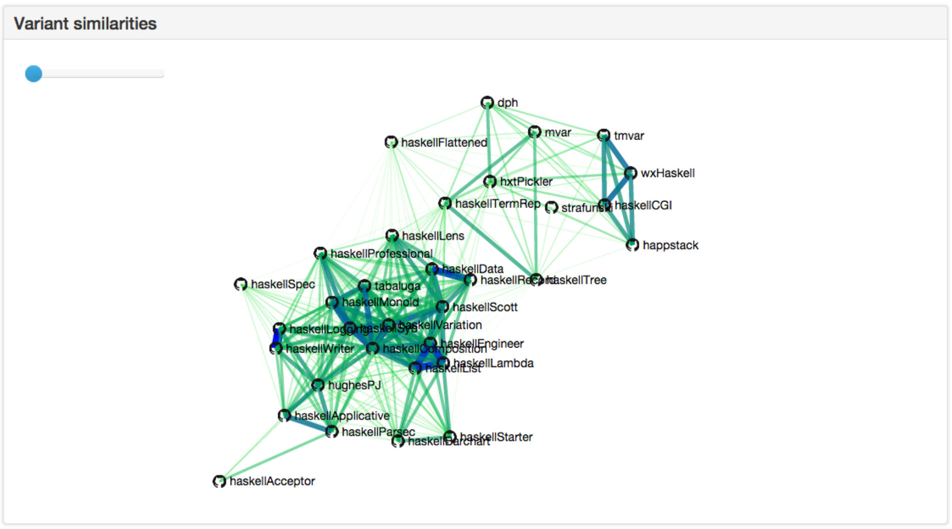
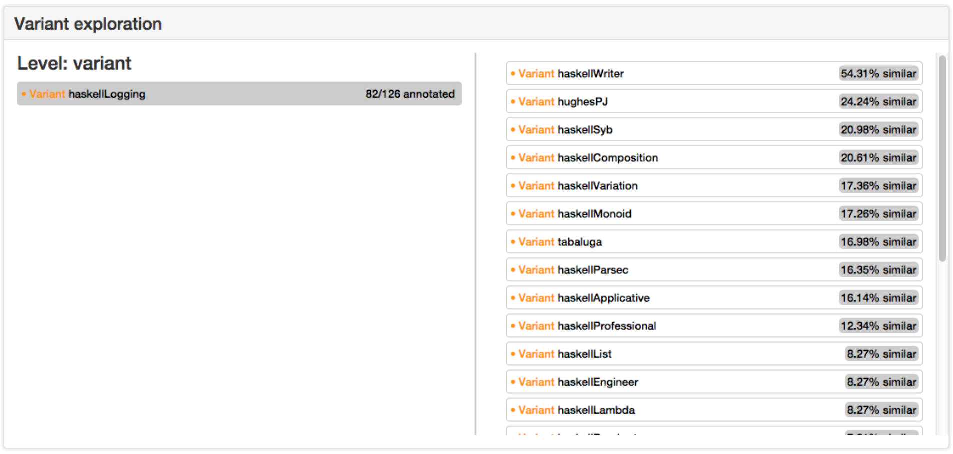


Figure 3.9: Graph of variants with edges weighted and color-coded by the similarities.

Initially, the user is presented with a graph of all variants that can be found at the HEAD of the given repository and branch. Ann uses the similarities at fragment level to compute a similarity at variant level. The presentation uses a force-driven layout such that variants that are more similar to each other are closer to each other than variants that are less similar to each other. By using this layout the user can easily identify "clusters" of highly-similar variants. Eventually, the user selects a variant in the graph.

Exploring a variant



Variant exploration

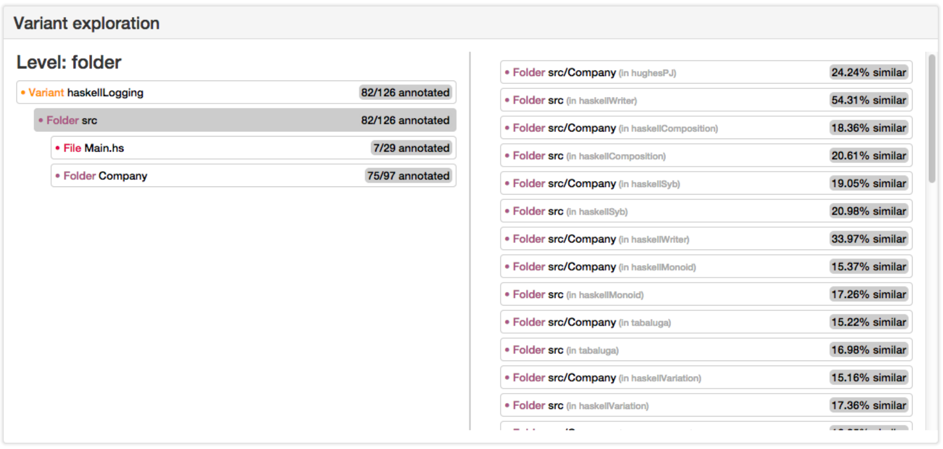
Level: variant

- Variant haskellLogging 82/126 annotated

- Variant haskellWriter 54.31% similar
- Variant hughesPJ 24.24% similar
- Variant haskellSyb 20.98% similar
- Variant haskellComposition 20.61% similar
- Variant haskellVariation 17.36% similar
- Variant haskellMonoid 17.26% similar
- Variant tabaluga 16.98% similar
- Variant haskellParsec 16.35% similar
- Variant haskellApplicative 16.14% similar
- Variant haskellProfessional 12.34% similar
- Variant haskellList 8.27% similar
- Variant haskellEngineer 8.27% similar
- Variant haskellLambda 8.27% similar

Figure 3.10: A variant in the variant-centric annotator.

Once the user has selected a variant we provide a tree-like view to explore the folders, files and fragments of the variant. In figure 3.10 the user is at the variant level. On the right-hand side we then show all similar variants sorted by the variant-similarity value that is computed using the basic fragment similarities. Next to the name of the selected variant Ann shows the progress of annotating by the ratio of annotated similarity evolutions.



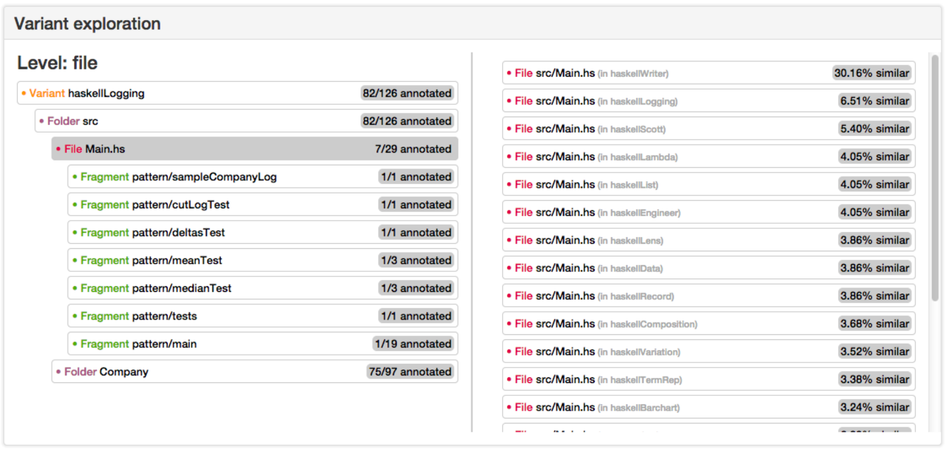
Variant exploration

Level: folder

- Variant haskellLogging 82/126 annotated
  - Folder src 82/126 annotated
    - File Main.hs 7/29 annotated
    - Folder Company 75/97 annotated

- Folder src/Company (in hughesPJ) 24.24% similar
- Folder src (in haskellWriter) 54.31% similar
- Folder src/Company (in haskellComposition) 18.36% similar
- Folder src (in haskellComposition) 20.61% similar
- Folder src/Company (in haskellSyb) 19.05% similar
- Folder src (in haskellSyb) 20.98% similar
- Folder src/Company (in haskellWriter) 33.97% similar
- Folder src/Company (in haskellMonoid) 15.37% similar
- Folder src (in haskellMonoid) 17.26% similar
- Folder src/Company (in tabaluga) 15.22% similar
- Folder src (in tabaluga) 16.98% similar
- Folder src/Company (in haskellVariation) 15.16% similar
- Folder src (in haskellVariation) 17.36% similar

Figure 3.11: A folder in the variant-centric annotator.



Variant exploration

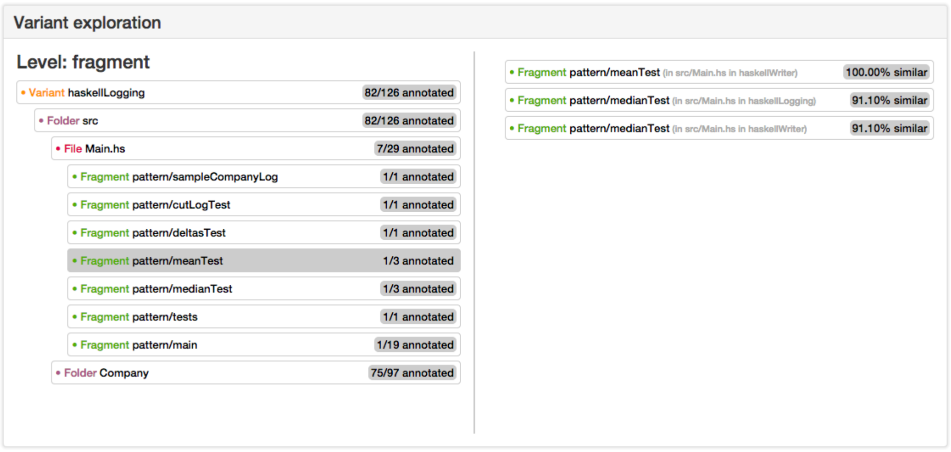
Level: file

- Variant haskellLogging — 82/126 annotated
  - Folder src — 82/126 annotated
    - File Main.hs — 7/29 annotated
      - Fragment pattern/sampleCompanyLog — 1/1 annotated
      - Fragment pattern/cutLogTest — 1/1 annotated
      - Fragment pattern/deltasTest — 1/1 annotated
      - Fragment pattern/meanTest — 1/3 annotated
      - Fragment pattern/medianTest — 1/3 annotated
      - Fragment pattern/tests — 1/1 annotated
      - Fragment pattern/main — 1/19 annotated
    - Folder Company — 75/97 annotated

- File src/Main.hs (in haskellWriter) — 30.16% similar
- File src/Main.hs (in haskellLogging) — 6.51% similar
- File src/Main.hs (in haskellScott) — 5.40% similar
- File src/Main.hs (in haskellLambda) — 4.05% similar
- File src/Main.hs (in haskellList) — 4.05% similar
- File src/Main.hs (in haskellEngineer) — 4.05% similar
- File src/Main.hs (in haskellLens) — 3.86% similar
- File src/Main.hs (in haskellData) — 3.86% similar
- File src/Main.hs (in haskellRecord) — 3.86% similar
- File src/Main.hs (in haskellComposition) — 3.68% similar
- File src/Main.hs (in haskellVariation) — 3.52% similar
- File src/Main.hs (in haskellTermRep) — 3.38% similar
- File src/Main.hs (in haskellBarchart) — 3.24% similar

Figure 3.12: A file in the variant-centric annotator.

Next the user can expand folders and files in the variant. At each level Ann shows similar folders or files, all with their location regarding containing folders and variants.



Variant exploration

Level: fragment

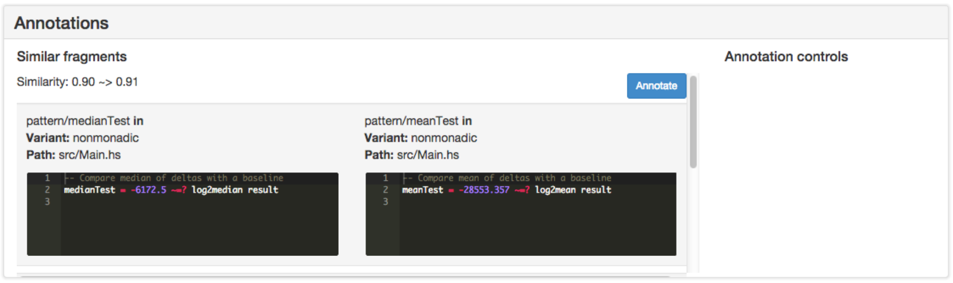
- Variant haskellLogging — 82/126 annotated
  - Folder src — 82/126 annotated
    - File Main.hs — 7/29 annotated
      - Fragment pattern/sampleCompanyLog — 1/1 annotated
      - Fragment pattern/cutLogTest — 1/1 annotated
      - Fragment pattern/deltasTest — 1/1 annotated
      - Fragment pattern/meanTest — 1/3 annotated
      - Fragment pattern/medianTest — 1/3 annotated
      - Fragment pattern/tests — 1/1 annotated
      - Fragment pattern/main — 1/19 annotated
    - Folder Company — 75/97 annotated

- Fragment pattern/meanTest (in src/Main.hs in haskellWriter) — 100.00% similar
- Fragment pattern/medianTest (in src/Main.hs in haskellLogging) — 91.10% similar
- Fragment pattern/medianTest (in src/Main.hs in haskellWriter) — 91.10% similar

Figure 3.13: A fragment in the variant-centric annotator.

Finally, the user can navigate down to the fragment level and is presented with all similar fragments on the right-hand side.

Figure 3.14: A fragment similarity in the variant-centric annotator.



Similarity: 0.90 ~> 0.91

pattern/medianTest **in**
Variant: nonmonadic
Path: src/Main.hs

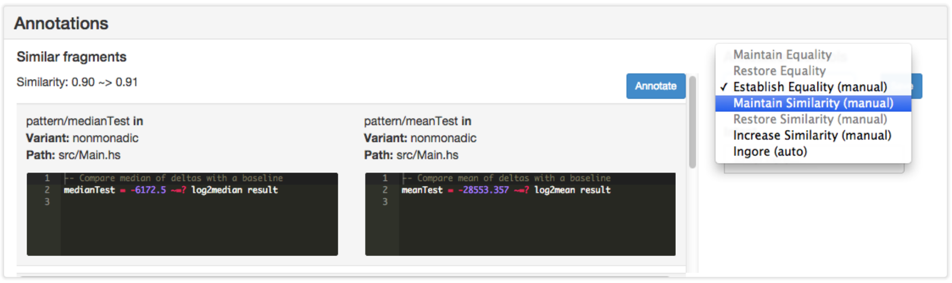
```
-- Compare median of deltas with a baseline
medianTest = -6172.5 ~=? log2median result
```

pattern/meanTest **in**
Variant: nonmonadic
Path: src/Main.hs

```
-- Compare mean of deltas with a baseline
meanTest = -28553.357 ~=? log2mean result
```

After selecting a fragment on the right-hand side we present the similarity of the fragment pair much like we do in the commit-centric annotator.

Figure 3.15: Variant-centric annotator with annotation controls.


Annotations
Similar fragments
Annotate
Maintain Equality
Restore Equality
✓ Establish Equality (manual)
Maintain Similarity (manual)
Restore Similarity (manual)
Increase Similarity (manual)
Ingore (auto)

Similarity: 0.90 ~> 0.91

pattern/medianTest **in**
Variant: nonmonadic
Path: src/Main.hs

```
-- Compare median of deltas with a baseline
medianTest = -6172.5 ~=? log2median result
```

pattern/meanTest **in**
Variant: nonmonadic
Path: src/Main.hs

```
-- Compare mean of deltas with a baseline
meanTest = -28553.357 ~=? log2mean result
```

Finally, the user annotates the similarity. In the example above the user would probably annotate the similarity with Maintain Similarity since the two Haskell functions test different functionalities, should therefore not become equal, but keep their current similarity.

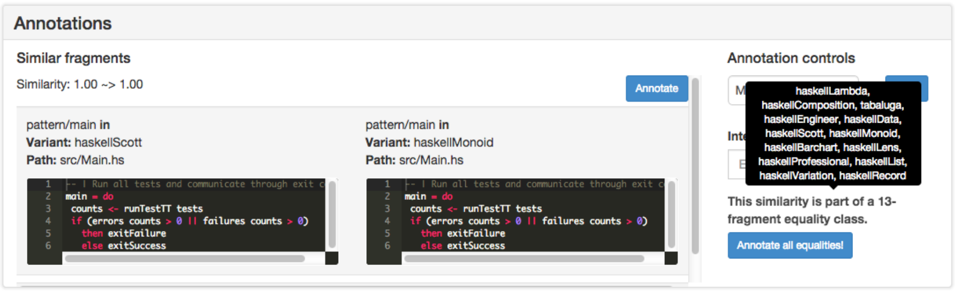
3.6 Automatic annotations

Earlier we have discussed the importance of only providing the user with processable amounts of information. The annotators achieve this by letting the user first explore commits, variants, folders, files, and fragments and by only then showing fragment similarities at the bottom level of exploration. However, the total number of similarities to annotate might still be high. We will later present the case study on the *101haskell* project. In this case the total number of similarities was close to two thousand. In the following we will present two approaches that will reduce the number of actual similarities the user has to annotate.

Both approaches make use of the fact that fragments and similarities can be seen as nodes and edges in a graph. The first approach makes use of specific complete subgraphs; the second approach uses a set of rules to complete missing annotations in triangles in the graph.

3.6.1 Annotating equality classes

In the literature about cloning the terms ”cloning class” or ”equality classes” refer to a set of artifacts that are all pairwise clones by some defined criteria [34]. This criteria might for instance be text-based equality or equality modulo identifier names (called ”parameterized clones”). In our case, we use the criteria of text-based equality modulo whitespace.
If we view fragments and their similarities as a graph, we can extract equality classes by using standard graph algorithms: An equality class in this graph is a complete subgraph where all similarities are equalities. For an equality class with n fragments all fragments are pairwise connected, therefore the equality class has $\sum_{i=1}^{n-1} i$ similarity edges. This is also the number of annotations the user has to create to annotate all similarities in the equality class. However, instead of forcing the user to do that, we can rather provide an option in the annotators to annotate **all** similarities of an equality class with one UI action.



Annotations

Similar fragments

Similarity: 1.00 ~> 1.00

pattern/main **in**
Variant: haskellScott
Path: src/Main.hs

```
-- | Run all tests and communicate through exit c
main = do
  counts <- runTestTT tests
  if (errors counts > 0 || failures counts > 0)
    then exitFailure
    else exitSuccess
```

pattern/main **in**
Variant: haskellMonoid
Path: src/Main.hs

```
-- | Run all tests and communicate through exit c
main = do
  counts <- runTestTT tests
  if (errors counts > 0 || failures counts > 0)
    then exitFailure
    else exitSuccess
```

Annotation controls

haskellLambda, haskellComposition, tabaluga, haskellEngineer, haskellData, haskellScott, haskellMonoid, haskellBarchart, haskellLens, haskellProfessional, haskellList, haskellVariation, haskellRecord

This similarity is part of a 13-fragment equality class.

Figure 3.16: Annotating an equality-class.

Figure 3.16 shows this scenario in the variant-centric annotator. The user has selected a fragment ”pattern/main” in the selected variant. On the right-hand side the user then picked one of the similar fragments. In the annotation controls Ann tells the user that the selected similarity is an edge in an equality class that contains a certain number of fragments (in this example, 13). The user can also see the variant names of fragments of the equality class. Instead of annotating all similarities separately the user can annotate all edges with Maintain Equality by using a dedicated button. That is, for this example, one UI action instead of $\sum_{i=1}^{12} i = 78$ manual annotations.

3.6.2 Rule-based annotation inference

We can make further use of the properties of the similarity graph when we look at triangles of pairwise similar fragments.

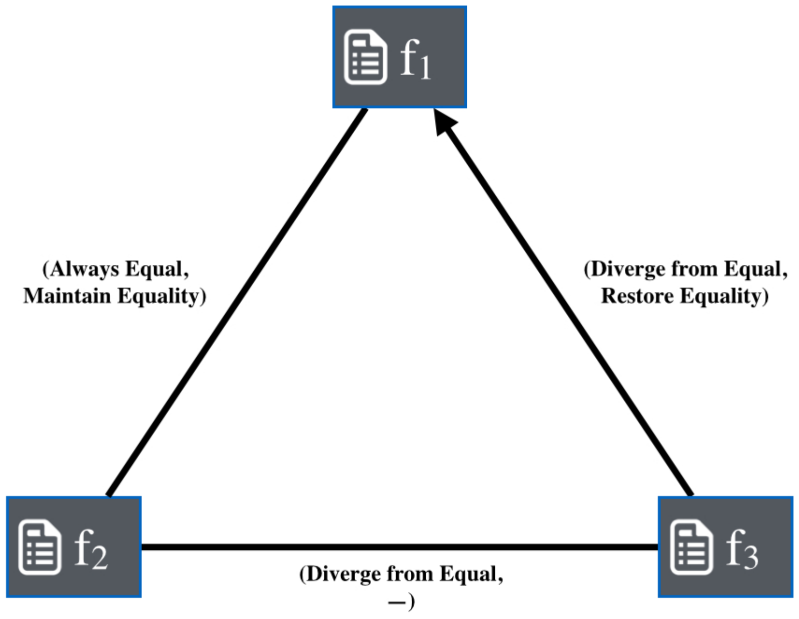

f1
f2
f3
(Always Equal, Maintain Equality)
(Diverge from Equal, Restore Equality)
(Diverge from Equal, –)

Figure 3.17: Triangle of pairwise similar fragments.

In figure 3.17 each node represents a fragment in some variant. All three fragments are pairwise connected because they are all pairwise similar. An edge between two fragments is labeled with the category of the similarity evolution and, potentially, with the category of the similarity's annotation. In this case, two edges were annotated with Maintain Equality respectively Restore Equality while the third annotation is missing. However, maintaining the equality between f_1 and f_2 and restoring an equality between f_1 and f_3 implies that an equality between f_2 and f_3 will be restored, too. We can therefore infer the third annotation automatically and thereby reduce the number of similarities the user has to annotate.

Rule notation and semantics

We can generally describe such inference rules by using the following notation:

$$A(ECs_{\mathrm{f}_1,\mathrm{f}_2}, AC_{\mathrm{f}_1,\mathrm{f}_2}, AD_{\mathrm{f}_1,\mathrm{f}_2}), A(ECs_{\mathrm{f}_1,\mathrm{f}_3}, AC_{\mathrm{f}_1,\mathrm{f}_3}, AD_{\mathrm{f}_1,\mathrm{f}_3}) \rightarrow A(ECs_{\mathrm{f}_2,\mathrm{f}_3}, AC_{\mathrm{f}_2,\mathrm{f}_3}, AD_{\mathrm{f}_2,\mathrm{f}_3})$$

All EC are sets of evolution categories, all AC are annotation categories, and all AD are propagation directions with the possible values $Left$, $Right$, and $None$. Here $Left$ means that for an evolution $E_{\mathrm{f}_1,\mathrm{f}_2}$ changes are pushed to f_1, $Right$ is defined analogously,

and $None$ does not define a direction.
The informal semantics of such a rule are the following. Given a triangle of three fragments f_1, f_2, and f_3 where

- $E_{\mathrm{f_1,f_2}}$ is the similarity evolution of f_1 and f_2,
- $E_{\mathrm{f_1,f_3}}$ is the similarity evolution of f_1 and f_3, and
- $E_{\mathrm{f_2,f_3}}$ is the similarity evolution of f_2 and f_3

and a rule r of the notation above. If

- $E_{\mathrm{f_1,f_2}}$ of fragments f_1 and f_2 is in $ECs_{\mathrm{f_1,f_2}}$, $AC_{\mathrm{f_1,f_2}}$ and $AD_{\mathrm{f_1,f_2}}$ are the category and the direction of the annotation of $E_{\mathrm{f_1,f_2}}$,
- $E_{\mathrm{f_1,f_3}}$ of fragments f_1 and f_3 is in $ECs_{\mathrm{f_1,f_3}}$, $AC_{\mathrm{f_1,f_3}}$ and $AD_{\mathrm{f_1,f_3}}$ are the category and the direction of the annotation of $E_{\mathrm{f_1,f_3}}$,

then annotate $E_{\mathrm{f_2,f_3}}$ of fragments f_2 and f_3 with an annotation of category $AC_{\mathrm{f_2,f_3}}$ and with direction $AD_{\mathrm{f_2,f_3}}$.

Inference rules

We have identified a set of rules for automatic annotation inference:

$$\begin{array}{l} A(\{AlwaysEqual, ConvergeToEqual\}, MaintainEquality, None), \\ A(\{AlwaysEqual, ConvergeToEqual\}, MaintainEquality, None) \rightarrow \\ A(\{AlwaysEqual, ConvergeToEqual\}, MaintainEquality, None) \end{array} \tag{R1}$$

R1 is the rule for closing equality triangles. Though the same result can also be achieved by annotating equality classes, the user may still benefit from this rule when not making use of such equality-class-based annotations.

$$\begin{array}{l} A(\{AlwaysEqual, ConvergeToEqual\}, MaintainEquality, None), \\ A(\{DivergeFromEqual\}, RestoreEquality, Left) \rightarrow \\ A(\{DivergeFromEqual\}, RestoreEquality, Right) \end{array} \tag{R2}$$

That is, we ”mirror” Restore Equality such that changes are pushed in the same direction.

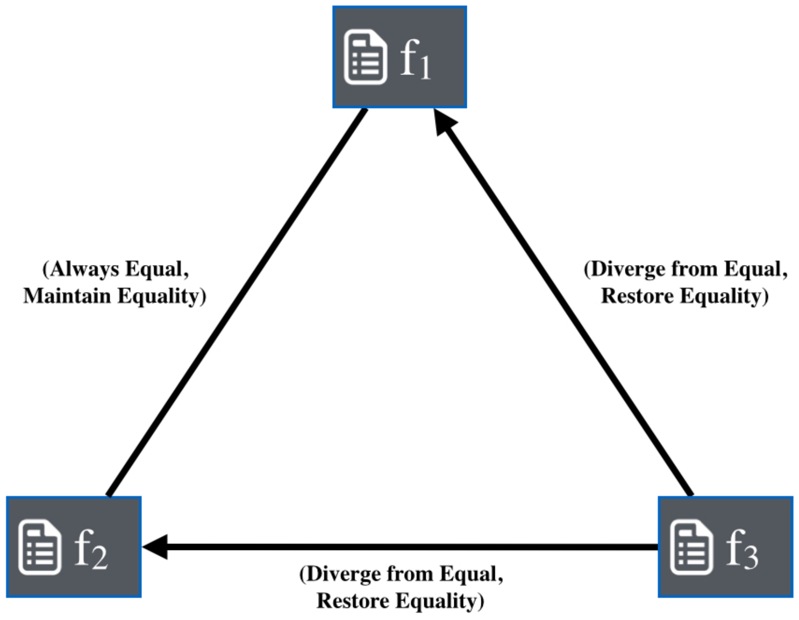


Figure 3.18: Triangle of pairwise similar fragments after applying $R2$.

Figure 3.18 shows the result of applying R2 to the triangle f_1, f_2, and f_3 from figure 3.17. Note that the order of the fragments matters when trying to apply the rule. We have to check six possible orders, though this is optimized in the actual implementation.
Analogously we have the following rule:

$$\begin{aligned}&A(\{AlwaysEqual, ConvergeToEqual\}, MaintainEquality, None),\\&A(\{DivergeFromEqual\}, RestoreEquality, Right) \rightarrow\\&A(\{DivergeFromEqual\}, RestoreEquality, Left)\end{aligned} \tag{R3}$$

We have such "mirroring" rules for all other annotations except Ignore:

$$\begin{aligned}&A(\{DivergeFromEqual, AlwaysNonEqual\}, IncreaseSimilarity, None),\\&A(\{AlwaysEqual, ConvergeToEqual\}, MaintainEquality, None) \rightarrow\\&A(\{DivergeFromEqual\}, IncreaseSimilarity, None)\end{aligned} \tag{R4}$$

Similar rules are analogously defined for Establish Equality, Increase Similarity, and Restore Similarity.

Cascading rule application

The application of annotation-inference rules can cascade due to the fact that triangles can be connected by having edges in common. The following scenario illustrates such cascading.

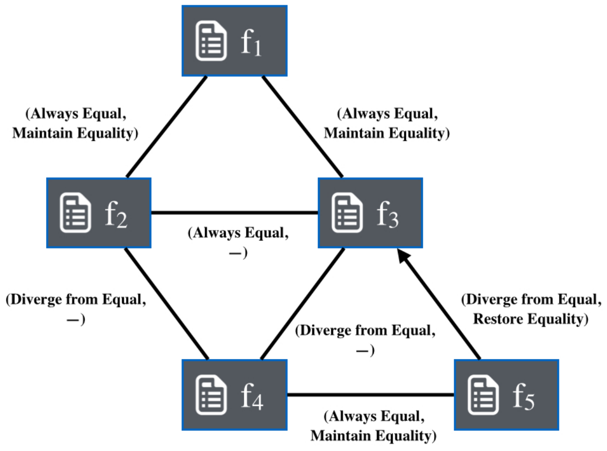

f1
f2
f3
f4
f5
(Always Equal, Maintain Equality)
(Always Equal, Maintain Equality)
(Always Equal, —)
(Diverge from Equal, —)
(Diverge from Equal, —)
(Diverge from Equal, Restore Equality)
(Always Equal, Maintain Equality)

Figure 3.19: Cascading inference-rules applications, step 1

Initially we can apply rule R1 to the triangle of f_1, f_2, and f_3, and rule R3 to close the triangle f_4, f_5, and f_3. The result is the following:

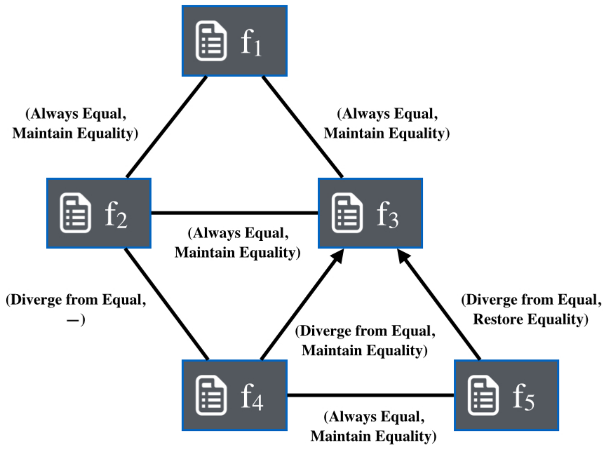

f1
f2
f3
f4
f5
(Always Equal, Maintain Equality)
(Always Equal, Maintain Equality)
(Always Equal, Maintain Equality)
(Diverge from Equal, —)
(Diverge from Equal, Maintain Equality)
(Diverge from Equal, Restore Equality)
(Always Equal, Maintain Equality)

Figure 3.20: Cascading inference-rules applications, step 2

Now, using rule 2, we can close the triangle of f_2, f_3, and f_4, resulting in the following graph:

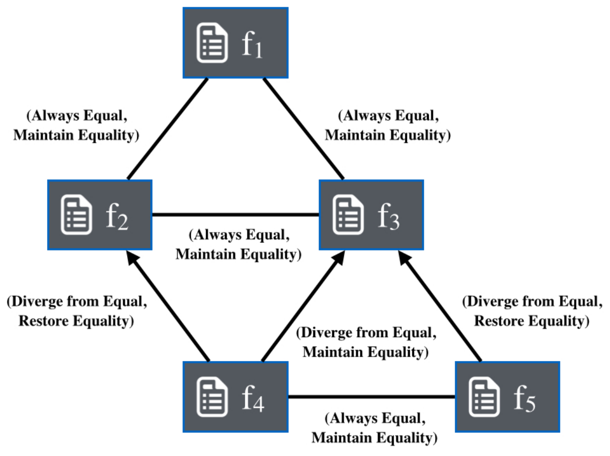

f1
f2
f3
f4
f5
(Always Equal, Maintain Equality)
(Always Equal, Maintain Equality)
(Always Equal, Maintain Equality)
(Diverge from Equal, Restore Equality)
(Diverge from Equal, Maintain Equality)
(Diverge from Equal, Restore Equality)
(Always Equal, Maintain Equality)

Figure 3.21: Cascading inference-rules applications, step 3

In general such rule application can cascade in many steps. When a user annotates a similarity we first try to apply each rule to all triangles the annotated similarity is involved in, and then recursively try to apply all rules to all triangles any automatically annotated edge is involved in. Finally, we update the progress information in the annotator by showing how many similarities have been annotated.

Chapter 4

Automatic change propagation

Annotating similarities results in a set of similarity-maintenance tasks. Therefore, we say that an annotation should be ”executed” because the task stated by the annotation should be executed. For annotations of the categories Maintain Equality or Restore Equality we can try automatic execution based on 3-way-merge. That is, we try to propagate changes to one or both fragments to either maintain or restore an equality.
We earlier discussed the notion of a *virtual platform*, which consists of metadata and cloning-related *operators*. The extracted similarities and the annotations are the metadata while the component which executes annotations is an implementation of the propagate operator.
In this chapter we will discuss how we automatically execute Maintain Equality and Restore Equality annotations, how the user may have to interact, and how these annotations may evolve based on execution.

4.1 Executing Maintain and Restore Equality

We have to act on a similarity evolution $\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}$ that is annotated with Maintain Equality if the fragments f_1 and f_2 diverge with regard to their content. For annotations of category Restore Equality the fragments have already diverged when the annotation was set. In both cases our system performs the following algorithm $execute(\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}})$ for an annotation $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ of a similarity evolution $\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}$:

```
1: $fc_{1,\text{HEAD}} \leftarrow$ content of $f_1$ at HEAD.
2: $fc_{2,\text{HEAD}} \leftarrow$ content of $f_2$ at HEAD.
3: if $tokenlines(fc_{1,\text{HEAD}}) == tokenlines(fc_{2,\text{HEAD}})$ then
4:     $updateSuccess(a_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}})$
5: else
6:     if $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ has a direction then
```

```
 7:         if $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ has direction $Left$ then
 8:             Overwrite content of $f_1$ with $fc_{2,\text{HEAD}}$
 9:         else if $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ has direction $Right$ then
10:             Overwrite content of $f_2$ with $fc_{1,\text{HEAD}}$
11:         end if
12:         updateSuccess($\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$)
13:         return true
14:     else
15:         $\mathsf{SS}_{\mathsf{f}_1,\mathsf{f}_2} \leftarrow$ last similarity snapshot in $\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}$ where $f_1$ and $f_2$ were equal.
16:         $C \leftarrow$ commit point of $\mathsf{SS}_{\mathsf{f}_1,\mathsf{f}_2}$.
17:         $fc_{1,\text{C}} \leftarrow$ content of $f_1$ at $C$.
18:         $fc_{\text{merge}} \leftarrow three\text{-}way\text{-}merge(fc_{1,\text{HEAD}}, fc_{2,\text{HEAD}}, fc_{1,\text{C}})$
19:         if $fc_{\text{merge}}$ has conflicts then
20:             $updateFail(a_{\text{SE}_{\text{f}_1,\text{f}_2}})$
21:             return false
22:         else
23:             if $tokenlines(fc_{1,\text{HEAD}})$ == $tokenlines(fc_{1,\text{C}})$ then
24:                 Overwrite content of $f_2$ with $fc_{\text{merge}}$
25:                 $updateSuccess(a_{\text{SE}_{\text{f}_1,\text{f}_2}})$
26:                 return true
27:             else if $tokenlines(fc_{2,\text{HEAD}})$ == $tokenlines(fc_{1,\text{C}})$ then
28:                 Overwrite content of $f_1$ with $fc_{\text{merge}}$
29:                 $updateSuccess(a_{\text{SE}_{\text{f}_1,\text{f}_2}})$
30:                 return true
31:             else
32:                 if askUserMergeOK() then
33:                     Overwrite content of $f_1$ with $fc_{\text{merge}}$
34:                     Overwrite content of $f_2$ with $fc_{\text{merge}}$
35:                     $updateSuccess(a_{\text{SE}_{\text{f}_1,\text{f}_2}})$
36:                     return true
37:                 else
38:                     $updateFail(a_{\text{SE}_{\text{f}_1,\text{f}_2}})$
39:                     return false
40:                 end if
41:             end if
42:         end if
43:     end if
44: end if
```

That is, the algorithm first checks whether the fragments of the similarity evolution have actually diverged. If so, two cases have to be considered starting in line 6. If the annotation has a direction (only possible for annotations of category Restore Equality) the algorithm just overwrites changes of the fragment the direction is pointing to by the content of the other fragment. If there is no direction, the algorithm looks up the last commit point where the fragments were equal. The algorithm then gets the content of one fragment at that commit point and uses this ”parent content” for a 3-way-merge in line 18. If there are no merge conflicts, the algorithm checks in line 23 and 27 if only one of the fragments has changes since they were equal. If so, the algorithm just adjusts the content of the other fragment. If both fragments changed the algorithm actually asks the user whether both fragments should be adjusted in line 32. If the user accepts this, both fragments' contents are overwritten by the merge.

The algorithm uses some helper methods. First, we assume that an implementation of 3-way-merge is given by means of some library. We have also used $tokenlines$ which implements pretty-printing tokens into many lines to disregard dissimilarities that are solely based on whitespace. We have implemented such a method for various software languages, including Haskell, Java, and Python. We will define $updateSuccess()$ and $updateFail()$ in the next section.

The algorithm returns whether the execution of the annotation has changed the content of any fragment. This information is used when executing all annotations of a repo r:

```
1: done = **false**
2: **while not** done **do**
3:     done = **true**
4:     **for** annotation $a_{SE_{f_1,f_2}}$ in $annotations(r)$ **do**
5:         done = done **and** $execute(a_{SE_{f_1,f_2}})$
6:     **end for**
7: **end while**
```

Since executing annotation may propagate changes over many similarity edges, we have to re-execute all annotations until no change was made. This is simplified here and done more efficiently in the implementation.

4.2 Updating annotation

The methods $updateSuccess(a_{SE_{f_1,f_2}})$ and $updateFail(a_{SE_{f_1,f_2}})$ update an annotation $a_{SE_{f_1,f_2}}$ when its execution succeeded respectively failed.

$updateSuccess(a_{SE_{f_1,f_2}})$ does the following:

```
1: **if** $a_{SE_{f_1,f_2}}$ has category Restore Equality **then**
```

```
2:     set category of $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ to Maintain Equality
3:     remove direction from $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$
4:     remove manual flag from $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$
5: end if
```

That is, only annotations of category Restore Equality have to be updated to category Maintain Equality. $updateFail(a_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}})$ does the following:

```
1: if $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ has category Maintain Equality then
2:     set category of $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ to Restore Equality
3: end if
4: set manual flag of $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$
```

That is, Maintain Equality and Restore Equality can evolve into each other based on successful or failed executions.
We will use the $manual$ flag when computing a todo list of manual tasks for the user.

Chapter 5

Todo list

In the previous chapter we discussed how our system attempts to execute annotations of certain categories automatically. In contrast to that, tasks expressed by annotations of other categories may have to be performed manually. To give the user an overview of what has to be done the Ann web application collects these maintenance tasks in a todo list.
In this chapter we will discuss the reasons why an annotation appears in the todo list, how the user acts on such annotations, and finally how these annotations evolve based on user actions.

5.1 Tasks in the todo list

We can identify two reasons why an annotation a appears in the todo list.

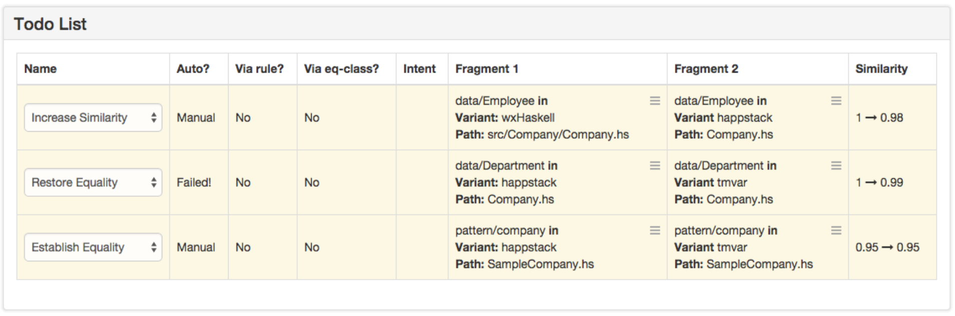
1. Based on the category of a it can be executed automatically by default (see applicability matrix in Chapter 2). However, automatic execution has failed for a, therefore Ann has set a manual flag of a. Because of this annotations of category Restore Equality can appear in the todo list after automatic execution has failed.

2. Based on the category of a it has to be performed manually by default. Additionally the expressed maintenance task calls for user action. Because of this annotations of categories Establish Equality, Increase Similarity and Restore Similarity appear in the todo list. Annotations of category Maintain Similarity are to be performed manually by default, however, they do not require any user action until they evolve to another category like Restore Similarity.

Whenever the user requests the todo list Ann selects annotations based on these two criteria.

5.2 Acting on tasks in the todo list

Ann has a page which shows the todo list for a repository.



Todo List

Name	Auto?	Via rule?	Via eq-class?	Intent	Fragment 1	Fragment 2	Similarity
Increase Similarity	Manual	No	No		data/Employee **in** **Variant:** wxHaskell **Path:** src/Company/Company.hs	data/Employee **in** **Variant** happstack **Path:** Company.hs	1 → 0.98
Restore Equality	Failed!	No	No		data/Department **in** **Variant:** happstack **Path:** Company.hs	data/Department **in** **Variant** tmvar **Path:** Company.hs	1 → 0.99
Establish Equality	Manual	No	No		pattern/company **in** **Variant:** happstack **Path:** SampleCompany.hs	pattern/company **in** **Variant** tmvar **Path:** SampleCompany.hs	0.95 → 0.95

Figure 5.1: Items in the todo list for the user.

Each item in the list shows an annotation with the category, the indent, the two similar fragments, and a summary of the underlying similarity evolution. To allow the user to understand why a particular annotation appears, Ann shows whether the task expressed by the annotation is manual by default, or appears because automatic execution has failed. Additionally, Ann indicates whether the annotation was created based on applying annotation-inference rules or an equality-class annotation.
After selecting an item to act on, the user has two options. Firstly, to perform a chosen task by editing the involved fragments. Secondly, to modify the annotation itself. For instance, the user might realize that an annotation of category Increase Similarity is no longer needed, and that instead the current similarity value is satisfactory. The user can then change the category to Maintain Similarity and the annotation will disappear from the todo list.

5.3 Updating annotations after user actions

After the user has performed a manual maintenance task he or she can execute a specific git command to update the annotations the user has acted on. We will discuss all Ann-specific git commands in the next chapter. An annotation $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ of a similarity evolution $\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}$ is updated according to the following rules:

1. • If $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ is of category Restore Equality, Establish Equality or Increase Similarity and

   • the user has made f_1 and f_2 equal,

   → update the category of $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ to Maintain Equality.

2. • If $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ is of category Maintain Similarity and

- the user has made edits to f_1 and/or f_2 such that their current similarity value at HEAD is not equal to the similarity value at HEAD when the annotation was created or updated,

$\rightarrow$ update the category of $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ to Restore Similarity

3. - If $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ is of category Restore Similarity and
   - the user has made edits to f_1 and/or f_2 such that the current similarity value at HEAD is equal to the similarity value at HEAD when the annotation was created or updated,

   $\rightarrow$ update the category of $\mathsf{a}_{\mathsf{SE}_{\mathsf{f}_1,\mathsf{f}_2}}$ to Maintain Similarity

5.4 Summary of possible annotation evolutions

The following figure summarizes all possible annotation updates performed by Ann.

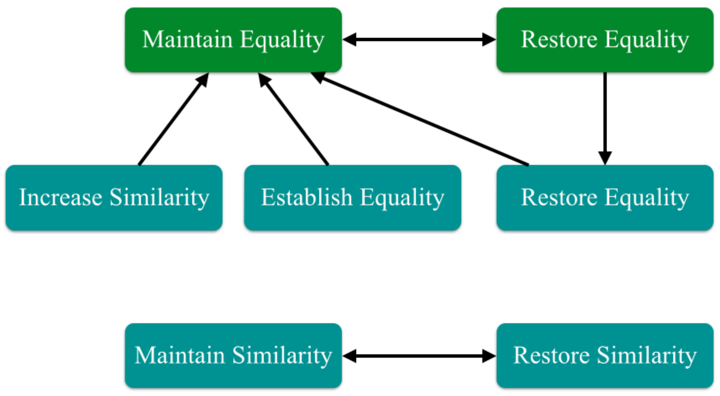


Figure 5.2: Possible evolutions of annotations' categories.

Green backgrounds indicate automatically executable annotations, blue backgrounds highlight manual tasks. We have discussed the evolution between Maintain Equality and automatic Restore Equality in the last chapter.
When the system sets the manual flag for a Restore Equality annotation it becomes a tasks for the user. The annotation can then evolve to Maintain Equality based on rule 1 above. This rule also covers the evolutions of Increase Similarity and Establish Equality

to Maintain Equality. Finally, Maintain Similarity and Restore Similarity can evolve into each based rules 2 and 3.

Chapter 6

Integrating with Git

We make use of a Git API for the implementation of the metadata-extraction process we have discussed in Chapter 2. We also extend Git by a set of new commands that all trigger certain Ann functionality for a given repository.
In this chapter we will discuss a simplified model of the standard Git workflow. We then present a set of new Ann-specific Git commands and how we integrate these into the standard workflow.

6.1 The standard Git workflow

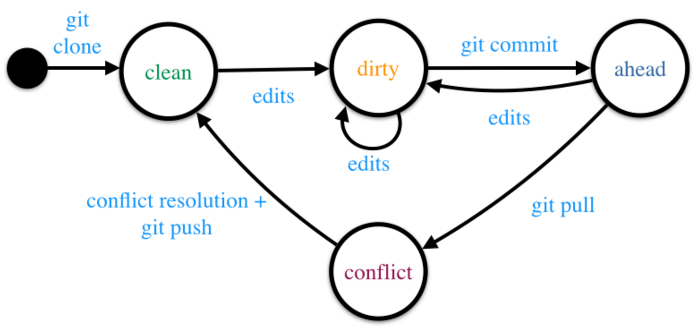


Figure 6.1: Model of a standard Git workflow.

Figure 6.1 shows a standard Git workflow. Note that this model does not include the use of branches since we do not consider branches for representing variants at this point. The simplified model uses the following states:

- *clean*: The workspace is clean and all commits are pushed to the remote repository.
- *dirty*: The workspace contains uncommitted changes.
- *ahead*: The local repository has unpushed commits.
- *conflict*: The workspace contains merge conflicts.

After cloning the repository and workspace are *clean*. Once a user edits, creates, moves, or removes files the workspace gets into a *dirty* state with uncommitted changes. After the user commits these changes the local repository's history is *ahead* of the history of the remote repository. Before pushing these new commits the user should pull commits, possibly resulting in merge *conflict*s.

6.2 Extending git with new commands

To integrate our system with Git we extend the default set of Git commands by three Ann-specific commands:

- `git ann init`: This registers the repository with our system and triggers the initial metadata-extraction process discussed in Chapter 2. After this the user can annotate the extracted similarities as illustrated in Chapter 3.
- `git ann update`: This triggers the extraction of metadata from new commits as discussed in the last section of Chapter 2. Also, this updates all annotations that the user has acted on based on the todo list discussed in Chapter 5.
- `git ann propagate`: This triggers the execution of automatically-executable annotations. That is, the change propagation discussed in Chapter 4 is executed. This command also updates the metadata if new commits were added to the repository since the last metadata extraction.

6.3 Extended Git workflow

With the set of new Git commands in place, we can extend the standard Git workflow to model Ann-specific states and state transitions.
With regard to the state of the metadata we identify two states:

- *synchronized*: The metadata is synchronized with the commits in the history.
- *unsynchronized*: The metadata is out-of-date because of new commits for which the metadata was not extracted yet.

With regard to whether changes have been propagated via automatic annotation execution we identify the following states:

- *propagated*: All changes have been propagated via automatically executing annotations.
- *unpropagated*: Due to file edits or new commits changes may have to be propagated.

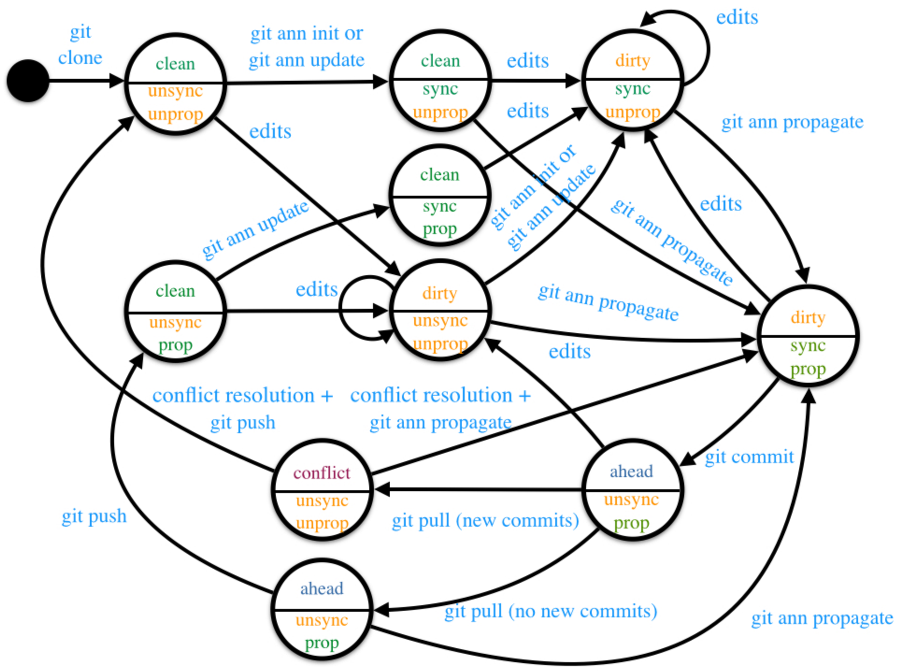


Figure 6.2: Model of the Ann-specific Git workflow.

Figure 6.2 shows a model of the extended Git workflow. After cloning the repository it is not registered with our system and is therefore in states *unpropagated* and *unsynchronized*. The user can then either start editing and will bring the workspace into a *dirty* state, or he or she can execute git ann init to extract all metadata and thereby make it *synchronized* with the history of the repository. Once the user plans to commit changes he or she first calls git ann propagate to trigger automatic annotation execution. This will bring the repository into states *propagated* and *synchronized*. The user can then commit the changes. This action results in a state that is still *propagated* but *unsynchronized* since the new commit was not inspected for new metadata yet. As in the standard Git workflow the user then first pulls before pushing commits. Because pulling may result in file changes the repository might get into an *unpropagated* state. Finally, the user pushes the new commits.
We have not included the annotation process into this workflow since it is independent and can be performed at any time after the initial metadata extraction.

Chapter 7

Case study: *101haskell*

This chapter is dedicated to a case study on the *101haskell* corpus of variants [32].
First, we will introduce the project itself. Next, we will discuss the metrics we use to measure the added value of using our approach to manage clones and similarities. After that, we will present a stepwise scenario of using our system for *101haskell*. For every step we outline what was done and we present new measurements based on our set of metrics. Finally, we will summarize the case study and use the final metrics results to answer our initial research questions.

7.1 *101haskell*

The *101companies* project [33] aggregates knowledge about software languages, technologies, and concepts in a wiki system[1]. It aims to be a useful resource for teaching and learning in these areas. *101companies* is also a software chrestomathy [31], that is, a collection of small software systems useful for teaching programming and software engineering in general. Each software system, called a ”contribution”, implements parts of a common feature model to demonstrate specific languages, technologies, and concepts. That is, *101companies* is a collection of variants where variance between two variants v_1 and v_2 can be achieved (a) by implementing different features and (b) by implementing shared features in different ways. All variants are documented as pages in the *101companies* wiki system.
A specific subset of the variants in the *101companies* project is *101haskell* [32]. While *101companies* covers many software languages in the aggregated variants, *101haskell* focuses on contributions that demonstrate functional programming concepts in and technologies for the Haskell language. The project is hosted as a repository on Github[2] and variants

[1] `http://101companies.org/wiki`
[2] `http://github.com/101companies/101haskell`

are organized by means of folders. The *101haskell* projects hosts 36 variants including the following.

- **haskellStarter**. Contribution with small language footprint.
- **haskellComposition**. Use of recursive data types.
- **haskellVariation**. Use of multiple constructors per type.
- **haskellFlat**. A Haskell-based data model illustrative for data parallelism.
- **wxHaskell**. GUI programming in Haskell with wxHaskell.
- **hxtPickler**. XML data binding for Haskell with HXT's XML pickler.

To start the implementation of a new variant, it is common practice to copy and paste components from existing variants. For instance, a new variant v might illustrate a programming concept by implementing a feature f in a way that the implementation utilizes this concept. Implementations of other features that are unrelated to that concept are then cloned from existing variants and variance in v is achieved by the implementation of f.
In the case study we use three **tactics** to improve the quality of the project.

1. **Restore equalities**. We use automatic change propagation to make fragments equal again that diverged unintentionally.
2. **Establish equalities**. We perform manual editing actions to make fragments equal that were never equal but should be.
3. **Identify inconsistencies**. We inspect equality classes and identify variants not contributing to an equality class due to inconsistencies.

7.2 Metrics for the case study

The next sections will describe a case study on *101haskell*. After each step we will measure the state of the project regarding the following metrics.

7.2.1 Equality classes

We measure

- The **total** number of equality classes.
- The **total** number of non-trivial equality classes, that is, equality classes with more than one fragment.
- The **maximum**, **median**, and **average** size of non-trivial equality classes.

7.2.2 Fragments

We measure

- The **total** number of fragments.
- The number of **unique** fragments. That is, we do not count members of a set fragments separately if all members belong to the same equality class.
- The number of **shared** fragments. That is, the number of fragments that are members of some non-trivial equality class.
- The number of **unshared** fragments. That is, the number of fragments that are members of no non-trivial equality class.
- The **median** and **average** number of variants a fragment is shared in.

7.2.3 Similarities

We measure

- The **median** and **average** similarity of fragments at HEAD where the similarity value passes the user-provided threshold.

7.2.4 Annotations

We measure

- The **total** number of annotations per annotation category.
- The **total** number of **automatic** annotations per annotation category created based on inferring rules or equality classes.

7.2.5 Variants

We measure

- The **uniqueness** of each variant. That is, the ratio of unshared fragments per variant.
- The **median** and **average** uniqueness of all variants.

For some metrics results we show the difference $\Delta_{\text{i-j, i}}$ between the results of a previous step $i - j$ and the current step i.

7.3 Initial status

Equality classes

total	632
total (non-trivial)	95
max size (non-trivial)	13
median size (non-trivial)	3
average size (non-trivial)	4.08

Table 7.1: Equality classes in the initial state.

Fragments

total	925
unique	632
shared	388
unshared	537
variants sharing (median)	1
variants sharing (average)	1.46

Table 7.2: Fragments in the initial state.

Similarities

median	0.98850
average	0.94186

Table 7.3: Similarities in the initial state.

Annotations

No annotations were made at this point.

Variants

variant	**#f**	**#uf**	**u**
hdbc	3	3	100.00%

haskellHxt	4	4	100.00%
haskellDB	7	7	100.00%
strafunski	158	150	94.94%
haskellSpec	29	26	89.66%
haskellFlattened	28	25	89.29%
haskellAcceptor	14	12	85.71%
haskellStarter	14	10	71.43%
mvar	16	11	68.75%
haskellLens	14	9	64.29%
haskellProfessional	41	26	63.41%
happstack	73	44	60.27%
dph	20	12	60.00%
hxtPickler	26	15	57.69%
haskellData	14	8	57.14%
haskellSyb	21	12	57.14%
haskellRecord	14	8	57.14%
haskellTermRep	17	9	52.94%
tabaluga	25	13	52.00%
haskellBarchart	18	9	50.00%
haskellParsec	21	10	47.62%
haskellApplicative	21	10	47.62%
haskellTree	22	10	45.45%
haskellCGI	60	27	45.00%
tmvar	43	19	44.19%
haskellScott	8	3	37.50%
haskellMonoid	22	8	36.36%
hughesPJ	12	4	33.33%
haskellVariation	16	5	31.25%
wxHaskell	47	14	29.79%
haskellList	13	3	23.08%
haskellLambda	13	2	15.38%
haskellEngineer	13	2	15.38%
haskellWriter	22	3	13.64%
haskellComposition	15	2	13.33%
haskellLogging	21	2	9.52%

median uniqueness	52.47%
average uniqueness	53.34%

Table 7.5: Variant uniquenesses in the initial state (f = fragments, uf = unique fragments, u = uniqueness).

7.4 Step 1: Automatically restoring equalities

As recommended by the tool, equalities were annotated first. This advice is given because of the importance of Maintain Equality in the annotation-inference rules for mirroring annotations. To annotate the equalities we systematically navigated variants, looked for non-annotated similarity evolutions leading to an equality, and used the ”annotate equality class” button to maximize the number of automatic annotations.
Next, we used tactic 1. First, we annotated all Diverge from Equality similarity evolutions. After that, git ann propagate was executed to start automatic change propagation. In fact, all annotations of category Restore Equality were given a direction and could successfully be executed automatically. The following metrics show the state of the repository after change propagation, except for the annotations where we show the metrics results for both before and after annotation execution.

7.4.1 Status

Equality classes

		$\Delta_{\mathbf{0,\,1_{after}}}$
total	617	-15
total (non-trivial)	93	-2
max size (non-trivial)	20	$+7$
median size (non-trivial)	4	$+1$
average size (non-trivial)	5.02	$+0.94$

Table 7.6: Equality classes after executing change propagation.

Fragments

		$\Delta_{0,\,1_{after}}$
total	925	± 0
unique	617	-15
shared	401	$+13$
unshared	524	-13
variants sharing (median)	1	± 0
variants sharing (average)	1.51	$+0.05$

Table 7.7: Fragments after executing change propagation.

Similarities

		$\Delta_{0,\,1_{after}}$
median	1.0	+0.01163%
average	0.96495	+0.02452%

Table 7.8: Similarities after executing change propagation.

Annotations

	total	**auto**	$\Delta_{0,\,1_{before}}$
Maintain Equality	953	858	$+953$
Restore Equality	324	212	$+324$
Establish Equality	0	0	± 0
Increase Similarity	3	0	$+3$
Maintain Similarity	44	20	$+44$

Table 7.9: Annotations before executing change propagation.

	total	**auto**	$\Delta 1_{\text{before}}, 1_{\text{after}}$
Maintain Equality	1359	1264	+406
Restore Equality	0	0	−324
Establish Equality	0	0	±0
Increase Similarity	3	0	±0
Maintain Similarity	44	20	±0

Table 7.10: Annotations after executing change propagation.

The metrics reveal the evolution of Restore Equality to Maintain Equality as the result successful change propagation. More specifically, all 324 Restore Equality annotations evolved to Maintain Equality annotations. Note that 82 additional Maintain Equality annotations were created afterwards. These were automatically added based on annotation-inference rule R1, because some fragment that have never been equal before became equal after executing annotations of other fragment pairs. For instance, there were fragments f_1 and f_2 and their equality should be restored. If another fragment f_3 is equal to f_2 but has never been equal to f_1 then after restoring the equality between f_1 and f_2 an additional equality between f_1 and f_3 emerges and should be maintained.

Variants

variant	**#f**	**#uf**	**u**	$\Delta u_{0}, 1_{\text{after}}$
hdbc	3	3	100.00%	±0%
haskellHxt	4	4	100.00%	±0%
haskellDB	7	7	100.00%	±0%
strafunski	158	150	94.94%	±0%
haskellSpec	29	26	89.66%	±0%
haskellFlattened	28	25	89.29%	±0%
haskellAcceptor	14	12	85.71%	±0%
mvar	16	11	68.75%	±0%
haskellLens	14	9	64.29%	±0%
haskellProfessional	41	26	63.41%	±0%
happstack	73	44	60.27%	±0%
dph	20	12	60.00%	±0%
haskellData	14	8	57.14%	±0%
haskellRecord	14	8	57.14%	±0%
hxtPickler	26	14	53.85%	−3.30%
haskellTermRep	17	9	52.94%	±0%

haskellBarchart	18	9	50.00%	±0%
haskellStarter	14	7	50.00%	−21.43%
tabaluga	25	12	48.00%	−4.00%
haskellParsec	21	10	47.62%	±0%
haskellApplicative	21	10	47.62%	±0%
haskellTree	22	10	45.45%	±0%
haskellCGI	60	27	45.00%	±0%
tmvar	43	19	44.19%	±0%
haskellScott	8	3	37.50%	±0%
hughesPJ	12	4	33.33%	±0%
haskellMonoid	22	7	31.82%	−4.54%
haskellVariation	16	5	31.25%	±0%
wxHaskell	47	14	29.79%	±0%
haskellSyb	21	6	28.57%	−28.67%
haskellLambda	13	2	15.38%	±0%
haskellList	13	2	15.38%	−7.70%
haskellEngineer	13	2	15.38%	±0%
haskellWriter	22	3	13.64%	±0%
haskellComposition	15	2	13.33%	±0%
haskellLogging	21	2	9.52%	±0%

		$\Delta_{0,\,1_{\text{after}}}$
median uniqueness	49.00%	−3.47%
average uniqueness	51.39%	−1.95%

Table 7.12: Variant uniquenesses after executing change propagation (f = fragments, uf = unique fragments, u = uniqueness).

7.5 Step 2: Manually establishing equalities

At this point all evolutions leading to an equality had been annotated. After automatic change propagation was used to diminish unintentional divergence, we focused on accidental variation. That is, we used tactic 2 by annotating similarity evolutions that have never been equalities, but where the underlying fragments should be equal nevertheless. To accomplish this, the variants were navigated in the annotator and annotations of category Establish Equality were created. After that, we used the todo list to systematically work on such annotations. For this, we

- first picked a set of fragments that should all become equal,

- then worked on all tasks related to these fragments,
- then committed all changes,
- and, finally, executed *git ann update* to update annotations.

These steps were repeated until no Establish Equality was left in the todo list and thus all equalities were established.
The following metrics show the state of the repository after manually establishing equalities, except for the annotations where we show the metrics results for both before and after these manual actions.

7.5.1 Status

Equality classes

		$\Delta_{1_{after}, 2_{after}}$	$\Delta_{0, 2_{after}}$
total	595	−22	−37
total (non-trivial)	85	−8	−10
max size (non-trivial)	20	±0	+7
median size (non-trivial)	4	±0	+1
average size (non-trivial)	4.89	−0.13	+0.84

Table 7.13: Equality after manually establishing equalities.

Fragments

		$\Delta_{1_{after}, 2_{after}}$	$\Delta_{0, 2_{after}}$
total	925	±0	±0
unique	595	−22	−37
shared	415	+14	+27
unshared	510	−14	−27
variants sharing (median)	1	±0	±0
variants sharing (average)	1.55	+0.04	+0.09

Table 7.14: Fragments after manually establishing equalities.

Similarities

		$\Delta_{1_{\text{after}},\,2_{\text{after}}}$	$\Delta_{0,\,2_{\text{after}}}$
median	1.0	± 0	$+0.01163\%$
average	0.96795	$+0.00310$	$+0.02770\%$

Table 7.15: Similarities after manually establishing equalities.

Annotations

	total	**auto**	$\Delta_{1_{\text{after}},\,2_{\text{before}}}$
Maintain Equality	1359	1264	± 0
Restore Equality	0	± 0	± 0
Establish Equality	120	67	$+120$
Increase Similarity	4	0	$+1$
Maintain Similarity	206	123	$+162$

Table 7.16: Annotations before manually establishing equalities.

	total	**auto**	$\Delta_{2_{\text{before}},\,2_{\text{after}}}$
Maintain Equality	1482	1387	$+123$
Restore Equality	0	0	± 0
Establish Equality	0	0	-120
Increase Similarity	4	0	± 0
Maintain Similarity	272	145	$+66$

Table 7.17: Annotations after manually establishing equalities.

The metrics show the evolution of Establish Equality to Maintain Equality as the result manual user actions. More specifically, all 120 Establish Equality annotations evolved to Maintain Equality annotations. 3 additional Maintain Equality annotations were created, because fragment pairs that were not associated with a similarity evolution before (due to not passing the similarity threshold) became equal. We can also see that 66 additional Maintain Similarity annotation were created after establishing equalities. These were added because new similarity evolutions emerged due to the user actions.

Variants

variant	**#f**	**#uf**	**u**	$\Delta \mathbf{u}_{\mathbf{1}_{\text{after}},\, \mathbf{2}_{\text{after}}}$	$\Delta \mathbf{u}_{\mathbf{0},\, \mathbf{2}_{\text{after}}}$
hdbc	3	3	100.00%	±0%	±0%
haskellHxt	4	4	100.00%	±0%	±0%
haskellDB	7	7	100.00%	±0%	±0%
strafunski	158	149	94.30%	−0.64%	−0.64%
haskellFlattened	28	25	89.29%	±0%	±0%
haskellAcceptor	14	12	85.71%	±0%	±0%
haskellSpec	29	24	82.76%	−6.90%	−6.90%
haskellLens	14	9	64.29%	±0%	±0%
haskellProfessional	41	26	63.41%	±0%	±0%
haskellBarchart	18	11	61.11%	−11.11%	−11.11%
haskellRecord	14	8	57.14%	±0%	±0%
mvar	16	9	56.25%	−12.50%	−12.50%
happstack	73	41	56.16%	−4.10%	−4.10%
dph	20	11	55.00%	−5.00%	−5.00%
haskellStarter	14	7	50.00%	±0%	−21.43%
tabaluga	25	12	48.00%	±0%	−4.00%
haskellParsec	21	10	47.62%	±0%	±0%
haskellApplicative	21	10	47.62%	±0%	±0%
haskellTermRep	17	8	47.06%	−5.88%	−5.88%
hxtPickler	26	12	46.15%	−7.70%	−10.00%
haskellTree	22	10	45.45%	±0%	±0%
haskellCGI	60	26	43.33%	−1.67%	−1.67%
haskellData	14	6	42.86%	−14.28%	−14.28%
tmvar	43	18	41.86%	−2.33%	−2.33%
haskellScott	8	3	37.50%	±0%	±0%
hughesPJ	12	4	33.33%	±0%	±0%
haskellMonoid	22	7	31.82%	±0%	±0%
haskellVariation	16	5	31.25%	±0%	±0%
wxHaskell	47	14	29.79%	±0%	±0%
haskellSyb	21	6	28.57%	±0%	−28.67%
haskellLambda	13	2	15.38%	±0%	±0%
haskellList	13	2	15.38%	±0%	±0%
haskellEngineer	13	2	15.38%	±0%	±0%
haskellWriter	22	3	13.64%	±0%	±0%
haskellComposition	15	2	13.33%	±0%	±0%

haskellLogging	21	2	9.52%	±0%	±0%

		$\Delta_{1_{\text{after}}, 2_{\text{after}}}$	$\Delta_{0, 2_{\text{after}}}$
median uniqueness	47.34%	−1.66	−5.13
average uniqueness	50.00%	−1.39	−3.34

Table 7.19: Variant uniquenesses after manually establishing equalities.

7.6 Step 3: Increasing similarities

Next, we worked on the remaining 4 annotations of category Increasing Similarity. This step did not change any metrics results compared to the last step except for the metrics about annotations and similarities.

7.6.1 Status

Annotations

	total	**auto**	$\Delta_{2_{\text{after}}, 3_{\text{after}}}$
Maintain Equality	1482	1387	±0
Restore Equality	0	0	±0
Establish Equality	0	0	±0
Increase Similarity	0	0	−4
Maintain Similarity	276	145	+4

Table 7.20: Annotations after manually increasing similarities.

Similarities

		$\Delta_{2_{\text{after}}, 3_{\text{after}}}$	$\Delta_{0, 3_{\text{after}}}$
median	1.0	±0	+0.01163%
average	0.96802	+0.00007	+0.02777

Table 7.21: Similarities after manually increasing similarities.

7.7 Step 4: Identifying inconsistencies

The last step was concerned about identifying inconsistencies by using the final equality classes. First, all equality classes were inspected and each one was assigned a category, that is, a reason for the underlying sharing. The following table shows the categories with the number of associated equality classes, total number of fragments, and the number of variants contributing fragments.

Category	**#EC**	**#f**	**#v**
Feature *Company (manager)*	2	21	21
Feature *Company (name)*	3	28	28
Feature *Company (address)*	2	28	28
Feature *Company (department)*	6	18	18
Feature *Company (subunit)*	2	4	4
Feature *Company (company)*	7	25	25
Feature *Company (employee)*	5	27	27
Feature *Company (salary)*	2	28	28
Feature *Total*	4	16	9
Feature *Cut*	5	15	5
Feature *Logging*	5	12	2
Feature *Median*	1	2	2
Feature *Browsing*	1	2	2
Testing *Serialization*	2	15	15
Testing *Cut*	2	13	13
Testing *Total*	2	15	15
Testing *Ranking*	2	6	3
Testing *Logging*	2	4	2
Testing *Median*	1	2	2
Testing *Depth*	1	3	3
Testing (test list)	3	15	15
Main	2	23	23
Sample *Company*	5	24	22
Sample *Log*	1	2	2
Concept *Zipper*	17	68	4

Table 7.22: Categories of reasons for sharing with the associated equality classes (EC = equality classes, f = fragments, v = variants).

After identifying these categories of reasons for sharing, we inspected each category to detect inconsistencies. Three kinds of inconsistencies were distinguished:

1. **Missing implementation**. A variant is missing the implementation of a feature, test, concept, etc.
2. **Unnormalized implementation**. A variant implements a feature, test, concept, etc. in an unnormalized way. After normalizating all such implementations, they would add to existing or create new equality classes.
3. **Mergable equality classes**. Two or more equality classes of a category should be merged. That is, their individual existence is not based on intended variance, but rather missing normalization and inconsistency.

The following table summarizes the findings. For each category it shows numbers of inconsistencies per kind. For kind 1 it shows the number of variants affected. For kind 2 it shows the numbers of variants that should join an existing equality class respectively form a new one. For kind 3 it shows the number equality classes that should merge.

Category	**#k 1**	**#k 2**	**#k 3**	**total**
Feature *Company (manager)*	0	0	2	2
Feature *Company (name)*	0	1	3	4
Feature *Company (address)*	0	1	2	3
Feature *Company (department)*	0	0	2	2
Feature *Company (subunit)*	0	0	0	0
Feature *Company (company)*	0	1	2	3
Feature *Company (employee)*	0	1	2	3
Feature *Company (salary)*	0	0	2	2
Feature *Total*	0	1	0	1
Feature *Cut*	0	0	0	0
Feature *Logging*	0	0	0	0
Feature *Median*	0	0	0	0
Feature *Browsing*	0	0	0	0
Testing *Serialization*	3	0	2	5
Testing *Cut*	8	0	2	10
Testing *Total*	9	0	2	11
Testing *Ranking*	0	0	0	0
Testing *Logging*	0	0	0	0
Testing *Median*	0	0	0	0
Testing *Depth*	0	1	0	1
Testing (test list)	9	0	0	9
Main	9	0	0	9
Sample *Company*	0	0	0	0
Sample *Log*	0	0	0	0

Concept *Zipper*	0	0	0	0

Table 7.23: Categories of reasons for sharing with found inconsistencies (k = kind).

The next table shows the lists of issues for all affected variants.

Variant	**Issues**
dph	Kind 1: missing implementation of Testing *Cut* Kind 1: missing implementation of Testing *Total* Kind 1: missing implementation of Testing (test list) Kind 2: join EC of haskellData for main
happstack	Kind 1: missing implementation of Testing *Cut* Kind 1: missing implementation of Testing *Total* Kind 1: missing implementation of Testing (test list) Kind 1: missing implementation of main
haskellBarchart	Kind 2: join EC of haskellData for Feature *Total* Kind 1: missing implementation of Testing *Total*
haskellCGI	Kind 1: missing implementation of Testing *Cut* Kind 1: missing implementation of Testing *Total* Kind 2: join EC of haskellData for Testing (test list)
haskellDB	Kind 1: missing implementation of Testing *Cut* Kind 1: missing implementation of Testing *Total* Kind 1: missing implementation of Testing (test list) Kind 1: missing implementation of main
haskellFlattened	Kind 2: join EC of haskellData for main
haskellHxt	Kind 2: join EC of haskellData for Testing *Cut* Kind 2: join EC of haskellData for Testing *Total* Kind 2: join EC of haskellData for Testing (test list) Kind 1: missing implementation of main
haskellLens	Kind 2: create EC with haskellRecord for Feature *Company (name)* Kind 2: create EC with haskellRecord for Feature *Company (address)*
haskellLogging	Kind 2: join EC of haskellData for Testing *Serialization*
haskellProfessional	Kind 1: missing implementation of Testing *Median*
haskellRecord	Kind 2: create EC with haskellLens of Feature *Company (name)* Kind 2: create EC with haskellLens of Feature *Company (address)*
haskellTermRep	Kind 1: missing implementation of Testing *Serialization*

hdbc	Kind 1: missing implementation of Testing *Cut* Kind 1: missing implementation of Testing *Total* Kind 1: missing implementation of Testing (test list) Kind 1: missing implementation of main
hughesPJ	Kind 1: missing implementation of Testing *Serialization*
mvar	Kind 1: missing implementation of Testing *Cut* Kind 1: missing implementation of Testing *Total* Kind 1: missing implementation of Testing (test list) Kind 1: missing implementation of main
strafunski	Kind 2: join EC of tabaluga for Testing *Depth*
tmvar	Kind 1: missing implementation of Testing *Cut* Kind 1: missing implementation of Testing *Total* Kind 1: missing implementation of Testing (test list) Kind 1: missing implementation of main
wxHaskell	Kind 1: missing implementation of Testing *Cut* Kind 1: missing implementation of Testing *Total* Kind 1: missing implementation of Testing (test list) Kind 1: missing implementation of main

Table 7.24: Issues found across variants (EC = equality class).

7.8 Summary

Using the final metrics results of our case study we can answer our initial research questions.

Q1. Overall, 324 similarity evolutions were annotated with Restore Equality. That is, the we identified 324 divergences as unintentional. The annotations could all be executed automatically using change propagation. As a result the number of unique fragments was reduced by 2.37%. After that we established equalities manually and the overall number of unique fragments was finally reduced by 5.85%.

Q2. With regards to fragment sharing the following results could be achieved. The median number of variants a fragment is shared was initially 1 and was not changed, the average number could be increased from 1.46 to 1.51 with automatic change propagation (+3.31%), and further to 1.55 by manually establishing equalities (+5.80% in total). The median uniqueness of variants could be decreased from 52.47% to 49.00% using change propagation (-6.61%) and further down to 47.37% after establishing equalities (-9.72% in total). The average uniqueness of variants could be decreased from 53.34% to 51.39% using change propagation (-3.66%) and further down to 50.00% after establishing equalities

(-6.26% in total). The median similarity passing the user-provided threshold of 0.80 could be increased from 0.9885 to 1 (+1.163%) by automatically restoring equalities. The average similarity was increased from 0.94186 to 0.96495 (+2.452%) by using change propagation, further to 0.96795 (+0.310%) by manually establishing equalities, and finally, to 0.96802 (+2.777% in total) by manually increasing similarities.

Q3. We could identify 25 categories of equality classes where each category is a reason for the underlying cloning of fragments. The categories enabled we to identify 47 inconsistencies across 19 variants.

Chapter 8

Conclusion

In this last chapter we will first summarize our work. Then, we identify both internal and external threats to validity. Finally, we outline some directions for future work.

8.1 Summary

Cloning is one approach to develop a software system as a set of variants. One mayor disadvantage of cloning is that the cloned fragments are disconnected. Therefore, whenever one fragment is changed all clones of that fragment have to be synchronized manually. This manual synchronization of fragments is usually unmanaged and therefore error-prone and can lead to unintentional divergence. It also does not scale well when a high number of variants has to be maintained. To manage and monitor such cloning and similarities we have presented a system for extracting similarities from a given repository and for annotating the similarities by expressing how they should be maintained further. We discussed how some categories of annotations can be ”executed” automatically by propagating changes between fragments. We also explained how other annotation categories call for manual actions by the user. We have presented a case study on the *101haskell* corpus of variants where our approach was used to automatically restore and manually establish equalities. A direct result of diminishing unintentional divergence and accidental variation was an increase of sharing of fragments between variants and fragment similarities and a decrease of the uniqueness of variants. By then manually inspecting emerging equality classes we could identify inconsistencies across variants.

8.2 Threats to validity

We identify threats to validity regarding the following four aspects.

8.2.1 Variability

External Validity. We only realized extraction for variant names where variation is organized by means of dedicated folders. Generally other means, such as dedicated branches or repositories, can be used.

8.2.2 Fragment extraction

Internal Validity. In our work fragments are consecutive lines of code that correspond to a hand-selected node in the abstract syntax tree of the given source file. Our work does not provide guidance with regard to the question on which syntactic levels such fragments should be extracted. For instance, for the Haskell case, we currently extract complete top-level functions, but not locally defined functions in where clauses. Selecting another set of syntactic categories for extraction could have resulted in different results in our case study.
External Validity. Another threat is concerned with the extraction of fragments by linking fragment snapshots. When a fragment is both renamed and its content was changed we can currently only detect the case where the name was changed, but all other changes are solely related to renaming of other fragments. In the context of our case study we manually inspected all cases in which a fragment could no longer be tracked and did not find any false negatives. Nevertheless, other approaches [20, 35] need to be studied, compared and potentially utilized.

8.2.3 Clone and similarity detection

Internal Validity. The implemented clone and similarity detection is text-based after pretty-printing tokens into many lines to disregard dissimilarities solely based on whitespace. This approach is only language-specific in that it requires a language-specific pretty-printer. However, the accuracy and completeness of clone detection could potentially be improved by using more language-specific techniques.
External Validity. Our system asks the user to provide a threshold for the similarity value. In our case study only one software language, Haskell, was considered when annotating similarities. Though our approach does support multiple languages, the similarity threshold might have to be different for different languages.

8.2.4 Annotations

External Validity. When extracting fragments we use language-specific technology that returns name/classifier pairs and line ranges for each fragment. For the case study on *101haskell* we have used technology that only returns non-nested fragments. However fragments can generally be nested, which can lead conflicts when annotating similarities. For instance, when we consider the Java classes c_1 with subclass c_{1s} and c_2 with subclass

c_{2s}. If a similarity between c_{1s} and c_{2s} is annotated with Increase Similarity, but a similarity between c_1 and c_2 is annotated with Maintain Similarity the annotations conflict each other. We could solve this by not allowing the extraction of such nested fragments or, preferably, by detecting annotation conflicts.

External Validity. All variants of the *101haskell* project were developed by four people. Only two of those were involved in the annotation process as part of our case study. In general many stakeholders might have to be consulted such as testers, architects and developers. Therefore, the annotation step might be disruptive and time-consuming. Due to the number of people contributing to *101haskell* we could not take this concern into account during our case study.

8.3 Future work

We can identify four possible directions of future work:

Operators. Our work implements the propagate operator in the context of a virtual platform of metadata and operators. Our approach could be extended and generalized to support the implementation of other operators like clone variant or clone feature. Depending on the operator this might require the extraction of additional metadata such as features. Because such operators could also create new metadata and annotations, they could have the benefit that less metadata has be extracted reactively and less annotations have be created manually.

Variability. Our work only supports folders as a way of organizing variants. Other approaches could be supported.

Information in annotations. Currently annotations only consists of the category, a comment by the user, and, potentially a direction changes should be propagated in. However, annotations could hold additional data like an assignment to a specific developer for manual actions required by the annotation. This assignment could then be used to organize collaboration.

Guided annotation process. We have defined a set of rules to automatically infer some annotations based on previous annotations. We could guide the user with regard to which similarities to annotate next, such that the number of inferred annotations can be maximized and thus the number of manual annotations can be minimized.

Bibliography

[1] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stanciulescu, A. Wasowski, and I. Schaefer, "Flexible product line engineering with a virtual platform," in *Proc. of ICSE 2014*, pp. 532–535, ACM.

[2] J. Rubin and M. Chechik, "A framework for managing cloned product variants," in *Proc. of ICSE 2013*, pp. 1233–1236, IEEE.

[3] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: A framework and experience," in *Proc. of SPLC 2013*, pp. 101–110, ACM.

[4] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *Proc. of CSMR 2013*, pp. 25–34, IEEE.

[5] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *Proc. of WCRE 2006*, pp. 19–28, IEEE.

[6] P. Clements and L. Northrop, *Software product lines: practices and patterns*, vol. 59. Addison-Wesley Reading, 2002.

[7] W. A. Hetrick, C. W. Krueger, and J. G. Moore, "Incremental return on incremental investment: Engenio's transition to software product line practice," in *Proc. of OOPSLA 2006*, pp. 798–804, ACM.

[8] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *Proc. of VaMoS 2013*, p. 7, ACM.

[9] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Proc. of ICSTW 2009*, pp. 157–166, IEEE.

[10] L. Chen, M. Ali Babar, and N. Ali, "Variability management in software product lines: a systematic review," in *Proc. of SPLC 2009*, pp. 81–90, Carnegie Mellon University.

[11] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *Proc. of ICSME 2014*, pp. 391–400, IEEE.

[12] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. of ICPC 2008*, pp. 172–181, IEEE.

[13] C. K. Roy and J. R. Cordy, "Scenario-based comparison of clone detection techniques," in *Proc. of ICPC 2008*, pp. 153–162, IEEE.

[14] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *Proc. of ICSME 2014*, pp. 321–330, IEEE.

[15] C. K. Roy, "Detection and analysis of near-miss software clones," in *Proc. of ICSM 2009*, pp. 447–450, IEEE.

[16] M. Kim and D. Notkin, "Using a clone genealogy extractor for understanding and supporting evolution of code clones," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–5, ACM, 2005.

[17] S. Xie, F. Khomh, Y. Zou, and I. Keivanloo, "An empirical study on the fault-proneness of clone migration in clone genealogies," in *Proc. of CSMR-WCRE 2014*, pp. 94–103, IEEE.

[18] M. Mondal, *On the Stability of Software Clones: A Genealogy-Based Empirical Study*. PhD thesis, University of Saskatchewan, 2013.

[19] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry, "Understanding the evolution of type-3 clones: an exploratory study," in *Proc. of MSR 2013*, pp. 139–148, IEEE.

[20] R. K. Saha, C. K. Roy, and K. A. Schneider, "An automatic framework for extracting and classifying near-miss clone genealogies," in *Proc. of ICSM 2011*, pp. 293–302, IEEE.

[21] X. Wu, A. Murray, M.-A. Storey, and R. Lintern, "A reverse engineering approach to support software maintenance: Version control knowledge extraction," in *Proc. of WCRE 2004*, pp. 90–99, IEEE.

[22] A. Brühlmann, T. Gîrba, O. Greevy, and O. Nierstrasz, "Enriching reverse engineering with annotations," in *Model Driven Engineering Languages and Systems*, pp. 660–674, Springer, 2008.

[23] A. Hemel and R. Koschke, "Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices," in *Proc. of WCRE 2012*, pp. 357–366, IEEE.

[24] M. Mondal, C. K. Roy, and K. A. Schneider, “Late propagation in near-miss clones: An empirical study,” *Electronic Communications of the EASST*, vol. 63, 2014.

[25] L. Barbour, F. Khomh, and Y. Zou, “Late propagation in software clones,” in *Proc. of ICSM 2011*, pp. 273–282, IEEE.

[26] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder, “Effects of cloned code on software maintainability: A replicated developer study,” in *Proc. of WCRE 2013*, pp. 112–121, IEEE.

[27] C. K. Roy, M. F. Zibran, and R. Koschke, “The vision of software clone management: Past, present, and future (keynote paper),” in *Proc. of CSMR-WCRE 2014*, pp. 18–33, IEEE.

[28] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, “Clone management for evolving software,” *Software Engineering, IEEE Transactions on*, vol. 38, no. 5, pp. 1008–1026, 2012.

[29] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, “Applying clone change notification system into an industrial development process,” in *Proc. of ICPC 2013*, pp. 199–206, IEEE.

[30] R. Koschke, “Frontiers of software clone management,” *Frontiers of Software Maintenance*, vol. 2008, pp. 119–128, 2008.

[31] R. Lämmel, “Software chrestomathies,” *Science of Computer Programming*, 2013.

[32] R. Lämmel, T. Schmorleiz, and A. Varanovich, “The 101haskell chrestomathy: A whole bunch of learnable lambdas,” in *Proc. of IFL 2013*, pp. 25:25–25:36, ACM.

[33] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich, “101companies: A Community Project on Software Technologies and Software Languages,” in *Proc. of TOOLS 2012*, pp. 58–74, Springer.

[34] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: a multilinguistic token-based code clone detection system for large scale source code,” *Software Engineering, IEEE Transactions on*, vol. 28, no. 7, pp. 654–670, 2002.

[35] S. Kim, K. Pan, and E. J. Whitehead, Jr., “When functions change their names: Automatic detection of origin relationships,” in *Proc. of WCRE 2005*, pp. 143–152, IEEE.

[36] J. Rubin and M. Chechik, “A framework for managing cloned product variants,” in *Proc. ICSE 2013*, pp. 1233–1236, IEEE / ACM.

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

[38] J.-M. Favre, R. Lammel, M. Leinberger, T. Schmorleiz, and A. Varanovich, “Linking documentation and source code in a software chrestomathy,” in *Proc. of WCRE 2012*, pp. 335–344, IEEE.