Maintaining Fine-grained Code Metadata Regardless of Moving, Copying and Merging

Christian R. Prause
Fraunhofer FIT
Schloss Birlinghoven
Sankt Augustin, Germany
christian.prause@fit.fraunhofer.de

Abstract

Source code metadata on a file-level granularity is too coarse for certain applications. But fine-grained metadata (e.g. line-by-line authorship) easily gets lost due to changes like merging, moving or copying code. Enabling metadata to survive code evolution provides valuable insights into program source code. This helps developers to understand the sources and opens up opportunities for advanced tools.

We present a concept that utilizes different search heuristics to identify probable ancestors of source documents, and pair this with clone detection to locate origins of inserted code. Arbitrary kinds of metadata can then be linked to code sections and be preserved automatically while code evolves. We evaluate our approach using code from the Hydra and FreeCol projects, and sketch prospective applications.

1. Introduction

Metadata is data about other data. While metadata is structured in a machine readable way, the data it is about may be of an arbitrary kind in any media. Metadata provides a context for data and is used to facilitate its understanding. Metadata may include descriptive information about context, quality and condition, or characteristics of the data, and can be recorded with high or low granularity.

Code metadata is data about source code. Such metadata would typically include descriptions of the content (e.g. name, size, type, ...), author (developer) and creation date of a source item; usually a file. But often code is developed collaboratively and evolves over years. Pieces of a collectively created source entity (be it a file, function or comment) have entirely different historical contexts. We cannot lump together everything at entity level granularity because we would not be able to eliminate obsolete information later. Here code metadata with a higher, sub-entity granularity is required. Such fine-grained metadata is also called markup.

For example, identification of expertise in software code is an important problem. Origin analysis and ownership determination are hence important aspects. Knowing where code came from and who was involved in its genesis is considered a huge point of interest [1].

Yet the origins of code become blurred as code evolves. Simple renaming (or moving) of an artifact is easy to detect, as is detecting changes in an artifact. But renaming plus changing is not [2]. It is not sufficient to look at the evolution of a single file. Likewise the picture is incomplete when neglecting parallel developments in branches. Data from some versioning systems (like CVS) even lacks change set information of files that changed at the same time. By not providing information on where a file came from, i.e. what previous names it had, some revision tools further complicate preprocessing steps (Section 2).

In consequence we present a concept (Section 3) that treats past and present code files alike, ignoring their names. Instead, we construct an all-encompassing unique version history. This enables fine-grained metadata capable of carrying arbitrary information, e.g. the original author of code, but also semi-automatically or manually generated metadata.

A high-level description of an algorithm with seven processing steps is given. The algorithm harnesses a line-based Levenshtein [3] distance function to reconstruct editing operations between two revisions, a search for the nearest neighbor of an artifact (Section 4), and a clone detection to enable transferring of markup of copied code (Section 5).

After that we provide facts and figures in Section 6. We start with a description of the characteristics of two projects: Hydra, a multi-national research project, and FreeCol, an open-source game. Code from these projects is then used as test corpus for evaluating our tool. Finally, we relate our work to other works in the field (Section 7) and conclude with Section 8, where we sum up and present future work.

A note on the nomenclature in this paper: We use the word *artifact* to refer to one specific revision of a source code file. Artifacts are kept in a database that is independent of the revision repository; except for the fact that source code is copied from the repository to the database. To clarify the distinction between both stores, we speak of the *database* when we mean the store where artifacts (also including their respective metadata) are held, while *repository* denotes the

978-0-7695-3793-1/09 $26.00 © 2009 IEEE
DOI 10.1109/SCAM.2009.20

IEEE computer society

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.

version control repository where source text comes from and that developers work with.

2. Crude Change Information

Source code stored in a revision repository is managed in two dimensions: space (code in diverse directories and files) and time (evolution of each file from revision to revision) [4]. Versioning tools like Subversion help navigating through this space in discrete steps. Many applications slice the space either *vertically*, which means looking at the software configuration at one discrete moment in time, or *horizontally*, which means looking at the evolution of an artifact.

Even without branches that add further dimensions, both slicing approaches are limited. Without time information, on the one hand, it is impossible, for instance, to determine by whom, when or why code was introduced into the repository. On the other hand, evolution information is incomplete when ignoring parallel developments in other parts (meaning in other files or branches) of the software. For a deeper understanding a different view is necessary.

Further problems arise depending on the versioning tool and its correct usage. Subversion does provide the original name of a file when the file was renamed with Subversion's rename facility, but only then. CVS does not record file name changes at all, which leads to misinterpretations [5]. For example, Subversion's `blame` command is useful when origins of code are needed. It annotates each line of code with its respective author and since when it is there. But if two developers use different indentation characters (space vs. tab characters) and their IDE automatically reformats code, Subversion is mislead. Also, the command relies on file names and has problems when a file is renamed without indicating that to Subversion. It is inevitable to deal with file name changes to get correct results. If that is not guaranteed then preprocessing is required.

Still, even if renaming is dealt with, if one developer merely merges files or branches then merge results are completely attributed to the new revision. Origins of the code, which may be in some other file written a long time ago, are neglected. This is especially problematic as large commits are often merges [6]. Even without merges, simply renaming a file results in all code being attributed to the developer who renamed the file.

Atomic commits combine several modifications to different files into one change set. This ensures the coherence of a logical change with its local manifestation in source files. If not supported by the versioning tool, again change information must be preprocessed to find the change sets first. Based on change set information and using the information retrieval vector model, methods for distinguishing code moves from true additions and deletions exist [7]. However, knowing that code was moved without recognizing the actual change impact on code is not enough.

What happens to code that was deleted at some time, and at some later time is brought back into the current version of code? Undoing changes of a previous revision with a new revision is a problem when looking only at change sets.

Assume that the problems of origin detection are solved, this still does not allow to add fine-grained metadata to code. The `blame` command, for example, determines author metadata from the revision since when code is in the repository. Adding arbitrary metadata to code would only be possible by linking it to the origin revision. Selectively adding metadata only to parts of the code or only to revisions after the one with the first appearance of the code is impossible. A method for carrying fine-grained metadata over from revision to revision — no matter in which revision the metadata was first added — is needed.

Therefore, it is necessary to not only monitor the evolution of individual artifacts. Also, the origins of mutations on a sub-artifact/code-text level that developers apply to the code may not be neglected. To overcome the limitations first a new view on source evolution is required. In this view we consider the inheritance relations between source artifacts and how they bequeath code to each other. But this space is much more complex.

3. Fine-grained Code Metadata

We restructure the change space to treat all artifacts equally; no matter if the artifact is in a different branch, obsolete or renamed. New structure is given to this space by a tree graph based on each artifact's ancestor relation to another artifact. This structure is needed so that artifacts can inherit fine-grained metadata from their ancestors. Differences to this ancestor are either inherited from further artifacts, or are new to the artifact and the whole code base. Before presenting an algorithm for passing on metadata, we first present a way for storing it.

3.1. Code Metadata

We want to have additional metadata associated with areas of source code. Artifact-level granularity, where metadata is linked to an entire artifact, would be too crude. We therefore store metadata in *markup* layers that invisibly (meaning *stored somewhere else*) overlie plain text source code. Every single character is thus associable with additional information of any kind and amount.

An example of fine-grained code metadata that we use throughout this paper is a reference to the original author, who added the respective code to the repository. Author information is cheap to generate automatically from commit logs. But it is helpful, too, because implementation expertise can be derived from it [8]. A practical advantage of author metadata is that Subversion also provides it and hence results are comparable. Of course, using other kinds of metadata is

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.

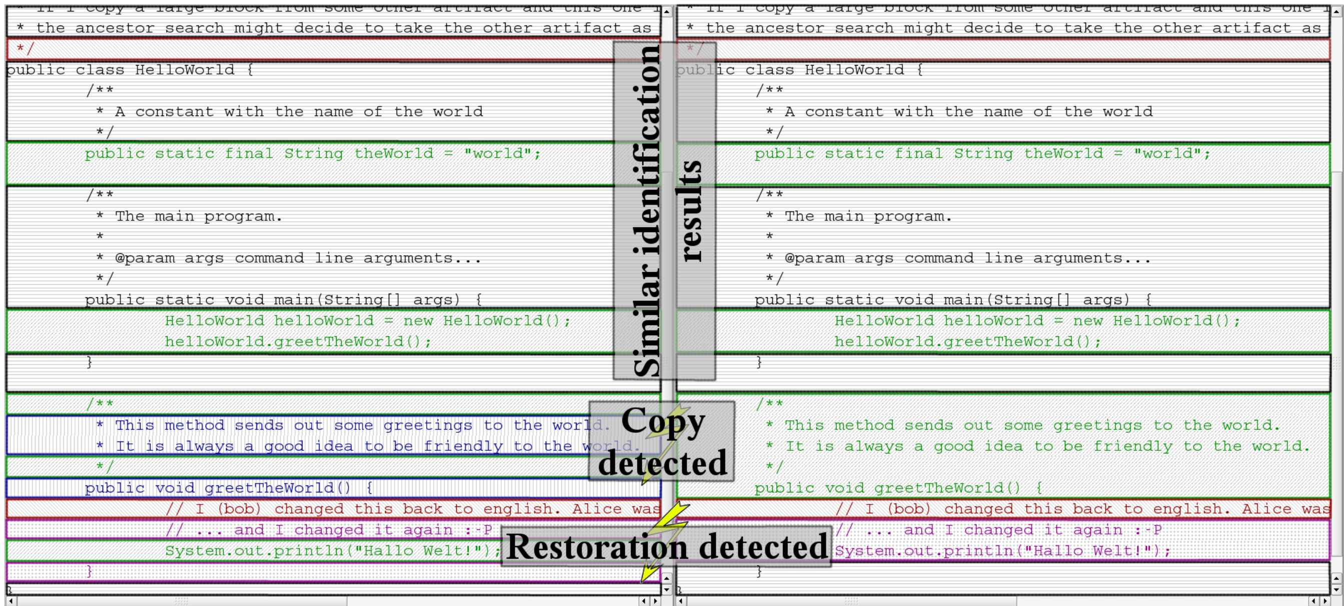

Figure 1. Metadata (e.g. author) as obtained by our tool (left) and `svn blame` (right) for visual comparison

possible. Saving metadata linked as markup to code yields the ability to keep it even if code is moved to a different artifact. It complements artifact-level log information that describes the history of the artifact. Figure 1 shows code with author markup. Different authors renamed, modified, and copied code from another source file. Additionally, one author removed code in one revision which is later reinserted by another author, undoing the previous change. Subversion outputs author/revision information, too, but is less accurate.

Markup sticks to the code it was attached to. When code moves around the markup moves with it. If code gets duplicated its markup is so, too. Analogously the markup disappears when code is deleted. Markup is edited in the same way as the source code.

3.2. Passing on Markup – an Algorithm

Versioning systems like popular CVS or Subversion only make weak assumptions about the artifacts they manage. They only assume that objects they version are files, and that updates are committed to the repository in short intervals to be as precise and small as possible. Obtaining fine-grained and useful change information therefore involves heavy preprocessing steps [9].

We present an algorithm for enhancing artifacts so that artifact evolution in the context of other evolving artifacts is reconstructed from versioning system data. The basic idea is that each artifact has one ancestor from which it inherits everything except for some minor modifications that either come from other artifacts or are all new. Inserting an artifact into the tool's internal database means to copy the artifact's plain text from the repository, and then refining it with markup before it gets stored. This process is made up of seven steps:

1) Get artifact's plain text from the revision repository.
2) Find the most closely related artifact in the database to minimize the number of insert operations. This artifact becomes the new artifact's *ancestor*.
3) Determine an edit script (or *patch*) that transforms the ancestor into the new artifact.
4) Apply the patch to the source code and its markup.
5) For partial strings inserted by the patch determine their origins to maintain markup of copied code.
6) (If applicable) generate new markup for strings where no origin was found; this is new text.
7) Add the new artifact to the database.

We briefly go through all of the algorithm's seven steps in the next paragraphs to give you a better impression of what they do. Steps 2) and 5) are more complex and are therefore explained in the two following sections.

3.2.1. Step 1: Retrieve the Artifact. Retrieving the next artifact from a revision repository is straightforward: Just go through all revisions. For each revision's changes identify changed files and iterate through all files. When the file does not match a certain filter, e.g. if it is not a Java file, insertion is skipped.

3.2.2. Step 2: Find closely related Artifact. This will be the artifact's ancestor. The edit or Levenshtein distance is a metric for the similarity of strings [3]. We employ a modified Levenshtein edit distance that adds or removes entire lines of

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.

code, and assigns different costs to edit operations depending on the number of characters in a respective line.

Finding a close relative of an artifact is a computationally expensive problem due to the sheer amount of comparisons (see Section 4). We discuss this in the next section and present heuristics that find an approximate result with significantly less comparisons than an exhaustive search.

3.2.3. Step 3: Determine Edit Script. An edit script provides a list of insertions and deletions that — when applied in the correct order — transforms one string into another. The script usually denotes the shortest (or least expensive) such list of edit operations. For reasons of computational complexity we only allow insertion and deletion of an entire line of code. The edit script reconstructs the developer's editing operations.

3.2.4. Step 4: Apply Edit Script. Next, the developer's reconstructed edit operations are applied to the nearest neighbor (ancestor) from Step 2. This effectively transforms its text into the text of the new artifact. But more importantly the edit script also updates the artifact's markup so that the markup of the new artifact is again consistent with its text content. Markup disappears when characters are deleted, or moves when the characters it is attached to move.

3.2.5. Step 5: Handle Copied Code. If text is inserted by an edit operation this text does not yet have markup. We try to find out if this code is cloned from another place and, if yes, also clone the missing markup from there. The details are addressed in Section 5. This step handles maintaining code markup if code is copied.

3.2.6. Step 6: Markup New Code. Code is considered totally new if it was not cloned, i.e. no origin from where it has been copied was found. These areas are enriched with *newly inserted* markup. If possible, new metadata can be generated and replace markup of the new areas. In our example where different authors are marked we would assign the current revision's author from the version repository's log to the code.

3.2.7. Step 7: Add to Database. Finally, the artifact is added to the database. At the same time its text is split into lines to update the inverted index of steps two and five.

4. Finding an Artifact's Ancestor

This section describes how we find a closely related artifact in the database for a given new artifact from the revision repository. The aim is to find the artifact in the database that is most closely related to the new artifact, because the amount of work for subsequent processing steps is smaller the more closely both artifacts are related. A better match here means less work in subsequent processing steps. This also reduces the chance to make a mistake in the later step of finding the origins of code. Finding the best matching artifact is equivalent to solving the *nearest neighbor problem* for a Euclidian similarity metric in a high dimensional space.

4.1. Artifact Distance Metric

Before searching for a nearest neighbor we need to define the distance between two artifacts. The search task is then to minimize this distance. In information retrieval a typical distance model is the vector model that compares term frequencies in two texts (or strings) through vector multiplication. This distance aims at semantic or deep similarity.

The similarity of the surface representation — the string itself — not its content is more interesting here because developers write, copy and move lines and characters, not necessarily whole semantic code blocks. This is also the reason why we decided against using tree edit distances. For plain strings the edit or Levenshtein distance is a metric based on the number of edit operations (insertions, deletions and substitutions of single characters) that transform one string into the other. Computation of the Levenshtein distance of strings A and B has space complexity $O(|A|)$ and time complexity $O(|A| \times |B|)$ [10]. Assuming that two artifacts under comparison are of equal length, this algorithm is $O(|A|^2) = O(|B|^2)$. This is problematic with long texts or if calculating lots of distances.

Cleverly devised optimizations exist for approximating character-level edit distances. But we also need the edit script. Therefore we simplify the distance computation by not editing on character granularity, but substring granularity. Some characters like the newline character appear very regularly in source code and make a good separator for substrings, but any other character (like '`;`') would also function. A line also has some limited expressiveness on the amount of functionality in code; remember infamous lines of code (LOC) metric. We therefore approximate the Levenshtein distance by splitting a full source text into lines.

We allow insertion and deletion of full lines only. The cost for inserting a line equals its length, while deleting costs half the line's length. Substitutions are permitted to change the indentation of lines at a reduced cost. This leads to edit scripts somewhat similar to the ones known from common *diff* tools, and is a sufficiently fast metric for determining distances for the nearest neighbor problem.

4.2. Nearest Neighbor Heuristics

In recent years many solutions to the nearest neighbor problem were developed, but they all suffer from poor performance in high dimensional spaces [11]. The set of possible strings is such high dimensional space. A correct result is not required for our concept, though, and an approximate

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.

Occurrences of each line (w/o leading & trailing whitespace) in different artifacts, i.e. low number means high distinctiveness	*Percentage of different lines from a total of x unique different lines*	
	159,580	86,621
1 occurrence	30.3%	5.4%
2 occurrences	15.2%	4.5%
3 occurrences	12.5%	3.8%
4 to 9 occurrences	26.7%	17.9%
10 to 99 occurrences	14.0%	54.0%
100 to 10,000 occurrences	0.1%	14.5%
10,000 or more occurrences	(abs.) 7	(abs.) 15
247,356 (or 962,972 resp.)	(abs.) 1	(abs.) 1

Table 1. Line recurrences in Hydra/FreeCol artifacts

result suffices. There is no need to do an exhaustive search or to guarantee that the result is correct. An important aspect that further simplifies the neighbors problem is that only one result is needed; not many or even a pairwise similarity of all artifacts. Next, we present heuristics that are used to reduce the number of artifact comparisons.

4.2.1. Similar Names. Artifacts in a revision repository are files and files have a name. From this observation we derive three heuristics: There is a chance that the ancestor of an artifact has the same name (SMNAME). Among these the most recently added one is very a probable candidate (SNGLPRDCSSR). Other candidates are files with similar names, especially with those names that differ only in their directory prefix (SIMNAME). It is obvious that cost $C(x)$

$$C(\text{SNGLPRDCSSR}) \leq C(\text{SMNAME}) \leq C(\text{SIMNAME})$$

and for recall $R(x)$

$$R(\text{SNGLPRDCSSR}) \leq R(\text{SMNAME}) \leq R(\text{SIMNAME})$$

hold. So we must trade cost off against recall.

4.2.2. Inverted Index Heuristic. It is possible to seriously reduce the set of potential ancestors for the artifact with an inverted index [12]. We build an inverted index where we store the lines of all artifacts (see Section 5.1). Each line is linked to the artifacts in which it appears. Next, only those artifacts that share similar lines need to be compared.

Source code of computer programs tends to repeat *trivial* lines, i.e. lines that are easy to write or are automatically generated by an IDE. In C-like languages, a typical example of a highly recurrent line is “}” (with an arbitrary amount of white-space) or the blank line. Actually, only those lines with an occurrence count $1 < c_{occur}(x) < n$ for some small n are suitable. Therefore, trivial lines do not provide good hints on what artifacts might be closely related.

But then source code also has lots of lines, which occur rather seldom (see Table 1). A line with only a single occurrence in other artifacts has much expressiveness. Therefore, we first pick lines that are very characteristic. Of course, such line can be so distinctive that it does not appear in any other artifact, which means that it is useless, too.

4.2.3. Tree Search Heuristic. All artifacts in the database (except for the first one) have an ancestor. Adding artifacts one by one to the database implies that all artifacts are integrated in a single finite tree structure. Each leaf (or node) of the tree thus represents one artifact.

The TREESEARCH heuristic exploits this structure: It starts out at the root of the tree. This is the current *search node*. Then it compares the distances between the search node and its children. If the distance between search node and new artifact is minimal then the artifact is added to this node. Otherwise the search continues with the child that has the minimum distance to the new artifact. Figure 2 illustrates how the TREESEARCH finds a node in three iterations using eight comparisons in total.

5. Filling the Gaps

In the previous section you saw how we find the nearest neighbor for an artifact among all other known artifacts. After that we create an edit script that transforms a known artifact into a new one. We apply the edit script. Strings it inserts are without metadata markup. Such string may either have been copied from some other artifact, or it is *all new*. We call inserted areas *(markup) gaps* as there is no markup assigned to them yet. Dealing with gaps is twofold:

- decide if gap contents are all new, and
- if not, determine where the string was taken from.

Every string for which we cannot determine an origin for is assumed all new. If there is more than one possible origin for a line then its context should be honored.

5.1. Line Index

The INVERTEDINDEXn heuristic (Section 4.2.2) relies on an *inverted index*. Before continuing with the description of how markup gaps are filled, we explain further aspects of the *Line Index* in this section. The reason is that the same index is used to find the origins of a line of code in a gap.

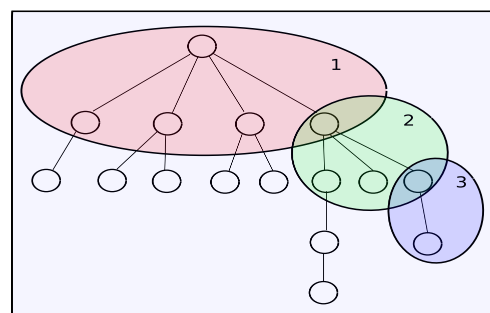

Figure 2. Three iterations of a tree search

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.

String	x_{occur}	$t(x)$
`new Limbo(".", new File(argv[0]));`	1	0.999
`new File(directory).mkdirs();`	93	0.992
`return (java.lang.String) org.ap[...]`	143	0.987
`final String name = owlModel.get[...]`	222	0.980
`import java.io.IOException;`	1069	0.907
`*/`	20612	0.153
`}`	247356	0.000

Table 2. Examples for normalized distinctivenesses

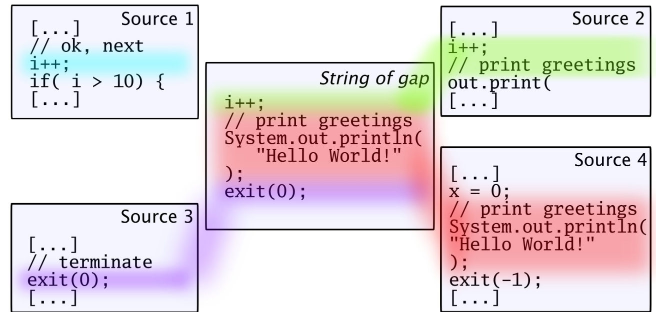

Figure 3. Selecting origin by size of footprint

There are advanced methods for identifying copied code that rely on fuzzy algorithms to abstract from superficial changes. As well on a crude file level as on the finer code level [13]. Different approaches were researched and developed, also including tree models (AST) that identify change types and find similar code [14].

Copy detection always involves a transformation step to bring code into an internal representation [15]. We chose a simple method that only omits leading and trailing white-space when determining origins of a line. With many modern programming languages indentation is automatically changed by the IDE when code moves to a new code block. All remaining formatting of a line is considered the line's fingerprint and helps distinguishing one line from another.

We do not want to find origins for trivial lines because these lines are rather re-written anew every time instead of being copied. A line is considered trivial if its t-value (*normalized distinctiveness*) is less than threshold T:

$$t(x) = 2^{-\frac{x_{occur}}{|A|}} < T = 0.55$$

We begin with the number of appearances divided by total number of artifacts $\frac{x_{occur}}{|A|} > 0$. Here $y = 1 - \frac{x_{occur}}{|A|}$ with $-\infty < y \leq 1$ so that $\left\{ \begin{array}{ll} y \to 1 & \text{for rare} \\ y \to -\infty & \text{for frequent} \end{array} \right\}$ lines. Hence $z = 0 < 2^y/2 = 2^{y-1} \leq 1$ with $z \to 0$ for frequent lines and $z \to 1$ for rare lines.

The threshold $T = 0.55$ was chosen by looking at the distinctivenesses of lines. This corresponds to a distinctiveness of 0.85, which means the line has about 15 appearances in 100 artifacts. Though the optimal value varies for a different source corpus, we hold the view it need not be adapted individually. Table 2 lists some distinctiveness examples of different lines.

5.2. Origins of Copied Code

When the edit script transforms an already inserted artifact into a new one it inserts code at several positions, thereby creating several disjunct gaps. We treat the gaps separately as this reduces the computational effort when dealing with each of the smaller gaps.

In each gap we need to find a mapping between lines in the new artifact and lines in old artifacts. We identify the individual lines in the gap and, for each line, retrieve possible origins from the inverted index. Next, we let each of the coverings grow to also include the next/previous line in the gap, but only if the mapped original line can also grow by consuming its successor/predecessor. If two origin mappings interfere the bigger one prevails.

In Figure 3 you see four potential sources for content of a gap. Although "Source 1" has a match it is overridden by "Source 2" because this is a longer mapping. The second line of the mapping from "Source 2" is in turn overridden by "Source 4" because it is even longer. Notice that the indentation of "Hello World!" does not matter.

In a final step when optimal covering was found markup is copied from the identified origins. Code that has been found to not have an origin is associated with newly created markup if metadata can be generated automatically, e.g. from author information of the Subversion log. Otherwise the user can be requested to provide new metadata, or we leave the markup gap open.

6. Evaluation

A comparable method for maintaining code metadata on a similar level of granularity is not known to the authors. When considering code authorship Subversion provides at least some comparable functionality with its `blame` method. Therefore, besides reporting on the nearest neighbor search and general findings from applying our tool to the two repositories, we compare our approach to Subversion's `svn blame`.

6.1. The Source Corpus

In this section we characterize the projects where we took code from for our analysis. We make a point of that the entire development history beginning from the first revision is available, that several developers are involved, that the code base is not too small/large, and that the projects differ in their nature (not both Free/Libre Open Source Software (FLOSS) projects).

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.

6.1.1. Hydra Subversion Repository.

Hydra[1] is a research project co-funded by the European Commission. It develops a middleware for networked embedded systems. The project duration is four years and involves several developers from commercial and scientific partners from various European countries. At the end of the project sources will be published open source. Yet for now, Hydra has different motivational and social characteristics than a typical FLOSS project [16]: there are no volunteers, no co-developers, no users besides the developers themselves, and developers get paid for their work and know each other from face to face.

The results presented in this section are based on the source corpus in the Hydra development repository after 30 months since project start. It encompasses 350,000 Java Source Lines of Code (SLOC, excluding white-space and comment only lines). At the time of study the repository contained more than 1600 revisions. Hence, an average of 214 SLOC are added with each commit.

6.1.2. FreeCol Subversion Repository.

FreeCol[2] is an open source game project at SourceForge.net[3]. 57 developers have contributed to it since 2002 when Java development replaced more and more of its original C code. It was project of the month at SourceForge in February 2007.

The code base is mature and well-established, and the development team is large and active. 70% (or 114KLOC) of its 123KLOC are Java code[4]. The repository contains over 5300 revisions. Thus the average commit adds 21 SLOC, which is only one 10th of Hydra. Compared to Hydra this can mean that commits are finer and less work happens between two commits.

6.2. Analysis of Neighbor Search Heuristics

The closer the distance between a new artifact and its supposed ancestor, the less and smaller string insertions by the edit script. This reduces the chance of making a mistake during subsequent clone detection. As the exhaustive search is too expensive, we introduced several heuristics (Section 4.2) that do not guarantee finding the nearest neighbor, but deliver good results with significantly less effort.

For the analysis of the different heuristics we added 7500 Hydra (or FreeCol, respectively) artifacts to the database. When adding a new artifact to the database we ran all different heuristics and the exhaustive search in parallel. After that we checked which heuristics found the same artifact as the exhaustive search (or another one with the same distance, of course). Due to the exhaustive search it was not possible to add all artifacts without spending days waiting for results.

1. http://www.hydramiddleware.eu/news.php
2. http://www.freecol.org/
3. http://sourceforge.net/projects/freecol/
4. http://www.ohloh.net/p/3938/analyses/latest

Table 3 lists the results for Hydra/FreeCol: *Hits* is the number of correct nearest neighbors found by the heuristic, *Recall* is this number divided by the total number of insertions, *Badness* is the average factor by which the suspected nearest neighbor's distance is off of the distance to the optimal one, and *Comparisons* is the average number of comparisons done per insertion.

Relying on the inverted index is most promising. But there is only a small increase in the accuracy of INVERTEDINDEXn for greater sample sizes. We studied what sample sizes are required to include the correct result and found that INVERTEDINDEXn becomes worse than random sampling for big sample sizes. It seems that lines with medium certainties are even misleading. Hence, it does not make sense to stick to the inverted index heuristic alone as the tradeoff between precision and speed gets unfavorable soon.

It is possible to increase hit rates by combining different heuristics. We analyzed correlations of heuristics' hits and misses. If it were possible to find negatively correlated heuristics then we could run them in parallel with a good chance to achieve better results. As expected the INVERTEDINDEXn heuristics correlate strongly with about 0.5 (for very different sample sizes, e.g. INVERTEDINDEX1 and INVERTEDINDEX200) to 0.95 (e.g. INVERTEDINDEX100 and INVERTEDINDEX200). The name based heuristics have a strong correlation of 0.95 for SNGLPRDCSSR and SMNAME, and about 0.45 for SMNAME/SNGLPRDCSSR and SIMNAME. Correlations with INVERTEDINDEXn are stronger for smaller $n \leq 10$ (~ 0.7) and weaker for bigger $n \geq 50$ (~ 0.3). For Hydra all correlations are non-negative, though, and we were unable to find a correlation between TREESEARCH and any other heuristic. For FreeCol there is a correlation of about -0.2 between TREESEARCH, and INVERTEDINDEXn and name based heuristics. Running different heuristics in parallel does not guarantee better results, but INVERTEDINDEX100 and SNGLPRDCSSR seem to work sufficiently well together. We found a better recall for TREESEARCH (similar to that for FreeCol) for Hydra revisions until about revision 1000, but did not further investigate this heuristic because of its bad database access characteristics.

6.3. Overall results

Figure 1 shows the results of applying our tool to a test repository. The code was moved to another file, recovered from an obsolete revision, modified by different authors, merged with contents from other files and indentation was changed. As you have seen before, these are the features that our tool can handle. Hence, this leads to different results between our approach and Subversion.

We compared the results for all non-obsolete artifacts in Hydra (revision 1653): Current Hydra artifacts total 927,305 lines including white-space and comments. For 336,965 lines

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.

	Hydra				FreeCol			
Heuristic	Hits	Recall	Badness	Comparisons	Hits	Recall	Badness	Comparisons
TREESEARCH	149	2%	91.5	18.1	2333	31%	132.9	94.3
SNGLPRDCSSR	1533	20%	96.8	1.0	1307	17%	161.2	1.0
SMNAME	1612	21%	91.1	2.0	7144	94%	3.1	27.2
SIMNAME	4360	57%	1.8	7.1	7242	95%	1.1	43.8
INVERTEDINDEX1	4628	61%	4.9	1.0	5977	78%	9.6	1.0
INVERTEDINDEX2	5054	66%	3.1	2.0	6713	88%	2.3	2.0
INVERTEDINDEX5	5414	71%	1.7	4.9	7107	93%	2.2	5.0
INVERTEDINDEX10	5624	74%	1.4	9.8	7182	95%	1.2	9.9
INVERTEDINDEX20	5790	76%	1.1	19.0	7220	95%	1.1	19.7
INVERTEDINDEX50	5995	79%	1.1	48.0	7253	95%	1.1	24.3
INVERTEDINDEX75	6092	80%	1.1	70.4	7275	96%	1.1	50.9
INVERTEDINDEX100	6150	81%	1.1	86.3	7285	96%	1.1	70.2
INVERTEDINDEX200	6299	83%	1.1	101.5	7315	96%	1.1	87.7
All combined	6307	83%	1.1	-	7588	99%	-	

Table 3. Comparison of nearest neighbor heuristics

or 36% a different author is reported (of 33 million characters 12 million or 37% did not match). First, this number appears quite high. Subversion does not detect clones, and as 20% to 30% clones have been reported for projects [17] this can be the primary reason of the differences. We manually checked results for sanity and found that about 14% of the code are auto-generated from WSDLs using Axis and that most areas are therefore attributed to the same other who checked in the first such artifact. Besides this we identified many files that had been moved around without notifying Subversion so that the history was lost.

The average distance between two artifacts is 2172 characters; or 2231 characters if only non-obsolete artifacts are considered. 57% of all artifacts are non-obsolete artifacts, which means that many artifacts are still in their early forms. A high number of 1763 artifacts (40%) have a distance of 0 to their nearest neighbor. We attribute this phenomenon mainly to reorganizations of the repository and to branching, which leads to new file names in subversion. This is supported by the observation that the number is almost cut in half (949 or 21%) when only non-obsolete artifacts are considered. The average number of different authors per artifact is 4.49, or 3.58 if the authors of trivial lines are not considered. Trivial lines are not copied if not in the context of a non-trivial line. Thus the difference indicates that cloning of non-trivial lines was successfully identified. If obsolete artifacts are not considered we get 4.37 and 3.56 respectively. Obsolete artifacts are early artifacts and their history is therefore not so colorful. In Hydra the two main contributors for artifacts usually contribute 85% and 11% (considering only non-obsolete artifacts: 86% and 9%) For results from Subversion there is a shift even more towards the main author. Obviously, there is strong code ownership or not much collaboration.

Non-obsolete FreeCol artifacts total 269,981 lines (9,603,912 characters) including white-space and comments. For 88,848 lines or 33% a different author is reported (2,955,080 characters or 31% did not match). This is quite similar to the results for Hydra, so we also checked manually. One major factor is that the multi-line license information contained in each file is attributed to the same author by our tool. But the biggest difference comes from code reformatting that is not handled by Subversion.

In FreeCol (until revision 3173) the average distance between an artifact and its parent is 1025 (483 for current artifacts). There are 10581 artifacts of which 868 (or 8%) are non-obsolete, thus artifacts have been revised several times. 690 artifacts are not different from their ancestor (branches and renames) and 227 are not yet obsolete. The average document has 6.66 contributors (6.35 excluding trivial lines). When including obsolete artifacts we get 7.44 (7.03 excluding trivial lines) contributors. This could mean that either track of some origins was lost or that a few individuals reworked the code. Two main developers normally contribute 65% and 18% to an artifact (59% and 21% for non-obsolete artifacts only). The average document is moved, renamed or branched 4.24 times.

6.4. Threats to Validity and Lessons Learned

We did extensive testing to verify that our implementation does what it should do: we have unit tests, a test repository with chosen features (like file renaming, code copy, code revival, indentation changes) and manually studied results for two true repositories Hydra and FreeCol. The problem is that defining what correct behavior is, is difficult: We built a system capable of efficiently finding a nearest neighbor, but we cannot guarantee that this is the one the user based his new version on. This information is irrecoverably lost due to the way common versioning systems work. On the one hand you do not want to see trivial lines attributed to the same author. For example, the first author that introduces the line “}” into an artifact should not always be recorded as the author of this line. So the line has to be recognized

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.

as trivial. On the other hand considering too many lines as trivial would not detect code areas that are moved from one artifact to the other. We solve this with an occurrence statistic which is only a crude mechanism, but it is efficient and also works for unformatted source passages like comments.

From looking at code snapshots (revisions) we cannot find out if code was copied from somewhere else or if it was written anew. Whenever results between Subversion and our approach differ it is probable that Subversion errs, but there is no guarantee that this holds true in all cases or that there is not a third author involved that both tools did not identify.

Starting at revision 1928 (in FreeCol) a new developer committed several revisions after reformatting with his IDE. As opposed to Subversion our technique is able to cope with resulting indentation differences. Yet it also has problems with lines that are split into two.

Though it is bad practice to commit artifacts that can be auto-generated, the Hydra repository contains several of these. Auto-generated code has a tendency to look homogeneous, which means that there are few very distinctive lines but lots of semi-distinctive ones, resulting in long search times. Furthermore, until identified as trivial code, large parts of these files will appear cloned and be attributed to the developer who checked them in for the first time.

7. Related Work

If changes to the code were tracked in the IDE they could immediately be applied to the markup as well. For example, Omori and Maruyama propose to record edit operations in the IDE, while the developer is working with the code [9]. This approach provides perfectly accurate information with ease. However, doing this is currently not feasible, because often only an off-the-shelf version control system like Subversion is available for obtaining the data. Additionally, depending on your organization it can be difficult to convince developers to install and use such plug-ins. Imagine multi-national research consortium with different developing partners and flat hierarchies. Therefore we must recover a fine-grained edit script from artifact snapshots instead of recording changes as they happen. Robbes proposes to use a change-based software repository integrated into the IDE to overcome this limitation [18]. Same problem: such repositories are not yet common. Similarly, the decision to write a new tool instead of modifying an existing version control system was a practical one, driven by the need to analyze an existing repository in productive use.

Godfrey and Zou first analyze source code syntax to obtain structured information (e.g. function names) and fingerprints. From the fingerprints they infer merging and splitting of source entities. Their semiautomatic approach (user interaction is required) is similar to the one presented here because it also uses fingerprints [2]. The difference, however, is that we build the fingerprint from a sequence of lines, ignoring semantic structure. Our aim is to recover edit operations on a text level so we can maintain the code metadata. Similarly Weißgerber and Diehl use a fingerprint method to classify revision control transactions into refactorings and changes [19].

That is also why we do not use tree edit distances: A tree edit distance is a metric that describes how one tree can be transformed into another one. Tree editing is interesting for observing the semantic evolution of code because computer program source code has a strict syntax and is thus transformable into an abstract syntax tree (AST). This enables code reuse through finding code with similar functionality [20], or tracking structural evolution of functions by exploiting the observation that function names change rather seldom [21]. But we need text edit operations.

To find the origins of possibly copied code we rely on a very basic form of clone detection similar to the islands-and-water metaphor of Cordy et al. [22]. Clone detection is a research field of its own with many approaches (see [17]) that increase efficiency and add fuzzy matching. For purposes of this paper the inverted index is efficient enough, though. While fuzzyness increases recall, the precision — which is so crucial for our method — is probable to decline.

8. Conclusion

We have presented a concept for associating fine-grained metadata with code. This is opposed to storing metadata on a crude file-level granularity which loses information when code is moved from one file to another. File-level metadata also leads to obsolete information if code disappears from a file but the file itself remains. Our tool considers parallel branches of code, and "dead" code in artifacts that are no longer active. It is aware of the full history of code and does not lose information if an artifact temporarily disappears. Character-based edit scripts allow keeping code metadata synchronized with code while the code evolves.

A modified Levenshtein distance is employed to compute distances between artifacts (i.e. whole source files) in order to find closest relatives and generate short edit scripts. The necessary nearest neighbor search is facilitated by search heuristics, which we compared to each other. Code inserted by the generated edit script is classified into code copied from somewhere else (including deleted files), and new code. It is essential to discriminate between these two kinds of insertions. Our basic clone detection ignores indentation and favors long consecutive blocks. We do not analyze changes on a semantic level but on a character-based *genetic* one to reconstruct the origins of (not reasons for) code *mutations*.

We have studied two projects of rather different natures: an European research project and an open source game. The evaluation shows that projects can be very different and that we cannot claim representativeness of our results for other projects. But we analyzed two quite different projects and

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.

showed that our approach has potential to provide valuable insights into software source code.

In the future we are going to analyze the sources of other projects. We will look for a way to combine our clone detection technique with more advanced methods that use code semantics or fuzziness. It should be possible to improve recall and precision. Additionally, we want to use our algorithms as an improved `svn blame` to make developers statistically responsible for their code and hence reduce the amount of cowboy coding in certain software projects, where this is a problem [23]. And we might investigate something like metadata based code Post-it notes.

We also want to pursue other areas of application. Using our algorithm in the opposite direction, for instance, we mark an area of code in an early revision of a file, and see which parts survive and where they move. We plan to apply the algorithm to less structured artifacts like human language, e.g. texts in a versioned Wiki.

Acknowledgment

The research reported here was supported by the HYDRA EU project (IST-2005-034891). Comments from anonymous reviewers greatly improved this paper.

References

[1] O. Alonso, P. T. Devanbu, and M. Gertz, "Expertise identification and visualization from cvs," in *Proceedings of the 2008 international working conference on Mining software repositories*. New York, NY, USA: ACM, 2008.

[2] L. Zou and M. W. Godfrey, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 166–181, 2005.

[3] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics-Doklady*, vol. 10, no. 8, pp. 707–710, February 1966.

[4] D. Spinellis, *Code Reading – The Open Source Perspective*. Addison Wesley, 2003.

[5] C. Thomson and M. Holcombe, "Correctness of data mined from cvs," in *Proceedings of the 2008 international working conference on Mining software repositories*. New York, NY, USA: ACM, 2008, pp. 117–120.

[6] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us? - a taxonomical study of large commits," in *Proceedings of the 2008 international workshop on Mining software repositories*. ACM New York, NY, USA, 2008.

[7] G. Canfora, L. Cerulo, and M. D. Penta, "Identifying changed source code lines from version repositories," in *International Workshop on Mining Software Repositories*. IEEE, 2007.

[8] J. Anvik and G. C. Murphy, "Determining implementation expertise from bug reports," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society Washington, DC, USA, 2007.

[9] O. Takayuki and M. Katsuhisa, "A change-aware development environment by recording editing operations of source code," in *International working conference on Mining Software Repositories*. New York, NY, USA: ACM, 2008.

[10] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *JournalACM*, vol. 21, no. 1, 1974.

[11] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, 2008.

[12] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[13] H.-F. Chang and A. Mockus, "Evaluation of source code copy detection," in *International working conference on Mining Software Repositories*. New York, NY, USA: ACM, 2008.

[14] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[15] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *International Conference on Software Maintenance*. Los Alamitos, CA, USA: IEEE Computer Society, 1999, p. 109.

[16] K. Crowston and J. Howison, "The social structure of free and open source software development," *First Monday*, vol. 10, no. 2, February 2005.

[17] H. A. Basit and S. Jarzabek, "Efficient token based clone detection with flexible tokenization," in *Proceedings of the 6th joint meeting of the European software engineering conference and the symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2007.

[18] R. Robbes, "Mining a change-based software repository," in *International Workshop on Mining Software Repositories*. IEEE Computer Society Washington, DC, USA, 2007.

[19] P. Weißgerber and S. Diehl, "Identifying refactorings from source-code changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006.

[20] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting similar java classes using tree algorithms," in *Proceedings of the 2006 international workshop on Mining software repositories*. New York, NY, USA: ACM, 2006.

[21] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 international workshop on Mining Software Repositories*. New York, NY, USA: ACM, 2005.

[22] J. R. Cordy, T. R. Dean, and N. Synytskyy, "Practical language-independent detection of near-miss clones," in *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2004.

[23] C. R. Prause and S. Apelt, "An approach for continuous inspection of source code," in *Proceedings of the Sixth International Workshop on Software quality (WoSQ)*. New York, NY, USA: ACM, 2008.

Authorized licensed use limited to: Fraunhofer-Informationszentrum Raum und Bau. Downloaded on November 4, 2009 at 01:20 from IEEE Xplore. Restrictions apply.