Detecting Similar Java Classes Using Tree Algorithms

Tobias Sager, Abraham Bernstein, Martin Pinzger, Christoph Kiefer
Department of Informatics
University of Zurich, Switzerland
tsager@gmx.ch {bernstein,pinzger,kiefer}@ifi.unizh.ch

ABSTRACT

Similarity analysis of source code is helpful during development to provide, for instance, better support for code reuse. Consider a development environment that analyzes code while typing and that suggests similar code examples or existing implementations from a source code repository. Mining software repositories by means of similarity measures enables and enforces reusing existing code and reduces the developing effort needed by creating a shared knowledge base of code fragments. In information retrieval similarity measures are often used to find documents similar to a given query document. This paper extends this idea to source code repositories. It introduces our approach to detect similar Java classes in software projects using tree similarity algorithms. We show how our approach allows to find similar Java classes based on an evaluation of three tree-based similarity measures in the context of five user-defined test cases as well as a preliminary software evolution analysis of a medium-sized Java project. Initial results of our technique indicate that it (1) is indeed useful to identify similar Java classes, (2) successfully identifies the ex ante and ex post versions of refactored classes, and (3) provides some interesting insights into within-version and between-version dependencies of classes within a Java project.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; E.1 [**Data Structures**]: Trees; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*retrieval models*

General Terms

Algorithms, Measurement, Experimentation

Keywords

Tree Similarity Measures, Software Repositories, Change Analysis, Software Evolution

MSR'06, May 22–23, 2006, Shanghai, China.

1. INTRODUCTION

Similarity analysis of source code is helpful during development to provide, for instance, better support for code reuse, faster prototyping, and clone detection. Consider a development environment that analyzes code while typing and that suggests similar code examples or existing implementations from a source code repository. Mining software repositories by means of similarity measures enables, for instance, code reuse and reduces the development effort (and thus cost) by making the shared knowledge base of code fragments in the repository better accessible. As another software evolution-based scenario, consider software project analysis: the detection of similar entities (Java classes in our case) in a complete project can indicate a deficit in the architecture or implementation flaws. Removing or merging similar classes may increase the overall quality as well as maintainability of a software project.

The goal of this paper is to present an approach to detect similar Java classes based upon their abstract syntax tree (AST) representations. These trees are generated using Eclipse's [10] JDT API in which all statements and operations of Java source code are represented. The generated complete ASTs are converted into an intermediary model called FAMIX (FAMOOS Information Exchange Model) [13, 14]. FAMIX is a programming language-independent model for representing object-oriented source code. The similarity between two classes is computed by tree comparison algorithms comparing the FAMIX tree representations of the two classes. We implemented this process as an Eclipse plug-in called *Coogle* (short for Code Google™). Our initial results show that Coogle is indeed useful to find similar Java classes within a Java software project.

The rest of this paper is structured as follows: next, we introduce our current implementation including the preprocessing in Eclipse and the three implemented tree comparison algorithms: bottom-up maximum common subtree isomorphism, top-down maximum common subtree isomorphism, and the tree edit distance. We then evaluate the effectiveness of these algorithms in the context of five constructed test-cases and a real-world Java project (Section 3), which leads to a discussion of our technique in Section 4. Section 5 reflects on our approach in the light of related work. Finally, we close with our conclusions in Section 6.

2. OUR APPROACH: COOGLE

We implemented a first prototype as an Eclipse plug-in [10] called *Coogle* that stands for Code Google™. Coogle essentially implements the following two steps to determine

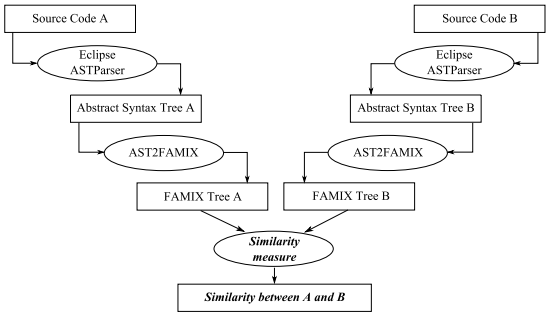

Figure 1: The figure shows the preprocessing steps of Java source code into FAMIX trees which are used as inputs for the similarity measures.

similarity between two or more Java classes: first, it transforms the abstract syntax tree representations of the classes' source code into intermediary FAMIX tree representations, and second, the similarity between these trees is computed based on tree similarity algorithms. The remainder of this section explains both steps in detail.

2.1 Tree Generation

Every piece of code can be represented as an abstract syntax tree (AST). An original, complete AST gets transformed into an abridged FAMIX tree representation using Eclipse's *ASTParser* and our *AST2FAMIX* converter as illustrated in Figure 1. We succinctly explain both of those tools.

The ASTParser is a component of the Eclipse JDT API that processes Java source code into its abstract syntax tree representation.[1] The classes that make up the tree are specified in the package `org.eclipse.jdt.core.dom`. By default, the ASTParser returns complete ASTs out of which the code can be perfectly reconstructed.

Our AST2FAMIX parser traverses the abstract syntax tree as generated by the ASTParser and builds a FAMIX representation from the nodes of the tree. Figure 2 shows a sample Java source code fragment and its corresponding FAMIX tree representation. The elements from the abstract syntax tree are mapped to FAMIX elements according to Table 1. Note that FAMIX does not represent all elements of Java abstract syntax trees, but instead represents a language-independent reduction of complete ASTs. This results in an information loss when converting Java to FAMIX since the level of granularity is reduced in the FAMIX model. However, the benefit from using FAMIX is the ability to perform programming language-independent similarity analyses, i.e., to compare, for instance, classes in C++ or Smalltalk with classes in Java. After the tree generation phase, the resulting FAMIX trees are passed to the similarity measures, which we discuss in the next subsection.

2.2 Tree Similarity Analysis

The current implementation of Coogle is able to detect structural similarity, i.e., similarity between the structure of the FAMIX trees of the source code. When representing source code as trees one important question arises: Is the order of the trees important? A Java compiler, for example, does not necessarily consider the order of top-level class body entities, such as methods or field declarations,

[1] `http://www.eclipse.org/jdt/`

```
package test;

import java.awt.Color;

public class Sample extends Parent {
  private String space = " ";

  public Sample(int input) {
    System.out.println(Color.BLACK);
  }
}
```

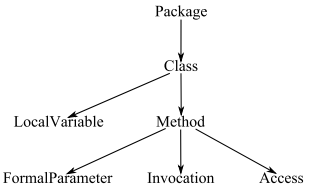

Figure 2: Original Java class source code and its corresponding FAMIX tree representation.

AST Node	*FAMIX Element*
-	FAMIXInstance
-	Model
PackageDeclaration	Package
TypeDeclaration	Class
-	InheritanceDefinition
FieldDeclaration	Attribute
MethodDeclaration	Method
SingleVariableDeclaration	FormalParameter
SingleVariableDeclaration	LocalVariable
ConstructorInvocation, SuperConstructorInvocation, ClassInstanceCreation, MethodInvocation, SuperMethodInvocation	Invocation
FieldAccess, SuperFieldAccess, SimpleName, QualifiedName	Access

Table 1: Eclipse AST elements with the corresponding FAMIX elements.

as relevant, whereas instructions in the bodies of these entities depend on the order of appearance in the source code. Hence, we need an algorithm that disregards the order between top-level entities but takes the order of low-level entities, such as method bodies, into consideration. As the FAMIX model does not represent all instructions which can occur in the body of an entity, e.g., it does not represent control structures, we would prefer using algorithms that perform a matching of unordered trees. Unfortunately, not all of the similarity measure algorithms that we have chosen have efficient solutions for unordered trees. For example, an unordered solution for the tree edit distance (see 2.2.1) is NP-complete as shown in [18]. We, therefore, implemented unordered tree matching for the bottom-up maximum common subtree search only and otherwise use algorithms for ordered trees.

2.2.1 Tree Similarity Algorithms

The literature on tree searching/editing is very elaborate. Shasha et al., for instance, describes his work on general tree and graph searching using exact and approximate search algorithms in [12]. Wang et al. presents a tool called *TreeRank* that does a nearest neighbor search for detecting similar patterns in a given phylogenetic tree [17]. These algorithms are highly specialized/optimized and, therefore, complex. Valiente [16], in contrast, discusses a number of standard tree searching and editing algorithms in detail providing efficient code implementation examples. To ensure a

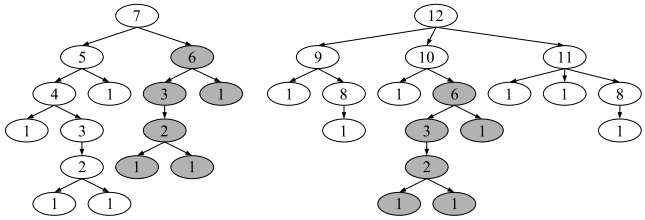

Figure 3: Bottom-up maximum common subtree isomorphism for two ordered trees (adapted from Fig. 4.15 in [16], page 225).

quick prototyping approach we decided to first implement three different algorithms from Valiente's work for measuring tree similarity: bottom-up maximum common subtree isomorphism, top-down maximum common subtree isomorphism, and tree edit distance.

Bottom-up Maximum Common Subtree Isomorphism. The goal of this algorithm is to find the largest isomorphic subtree of two given trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$. Valiente reduces this problem to the problem of partitioning the nodes $V_1 \cup V_2$ into equivalence classes. If two nodes v and w belong to the same equivalence class, the bottom-up subtree of T_1 rooted at node $v \in V_1$ is isomorphic to the bottom-up subtree of T_2 rooted at node $w \in V_2$. The equivalence classes of two ordered trees are illustrated by the numbers in the nodes in Figure 3, where the bottom-up maximum common subtree for the trees is highlighted in gray. We determine the isomorphism code of a given node by recursively building an isomorphism string consisting of the isomorphism codes of all children of the node. This string gets then compared to a collection of existing isomorphism strings. If the string is already in the collection, the corresponding equivalence class is read from the collection. If the isomorphism string is not contained in the collection, we add it to the collection and assign a new equivalence class code to the string. After collecting the equivalence classes of both trees T_1 and T_2, the algorithm searches for the biggest equivalence class by using a queue with the size of the nodes as priority. The first element in the queue is the node with the biggest size. This ensures that the matched subtree is indeed a maximum common subtree.

Valiente describes this algorithm for unlabeled trees only. We extended the algorithm to use labeled trees by assigning a unique integer value to each FAMIX node type (Package, Class, Method, etc.). The equivalence classes are then matched based on this value and the already defined equivalence class code. This solution is also suggested in [15]. We implemented the comparator pattern [3] for this label comparison.

To use this algorithm for unordered trees as well, the isomorphism codes of the children of a processed node are sorted based on their assigned FAMIX node type before searching for already existing code sequences in the equivalence class collection. This ensures that all children of a node only differing in order are treated the same, thus unordered.

Note that so far, we have only identified the maximum common subtree of both of the input trees. In order to get a similarity score between the two trees, we apply the following procedure: the size (number of nodes) of the first input tree T_1 is denoted by $|V_1|$. The cardinality $|V_2|$ stands for the size of the tree T_2 representing the second class. Furthermore, T_m denotes the maximum matched subtree of size $|V_m|$. An efficient similarity measure needs to satisfy the following properties: first, the more of T_1 is matched, the higher the similarity score of T_1 and T_2 is. This is expressed by $\frac{|V_m|}{|V_1|}$. This results in low values for complete matches of T_2, e.g., in the case if T_2 is much smaller than T_1. Second, complete matches should get higher values than non-complete ones, i.e., not the whole tree T_2 can be matched to T_1. We experimented with different possibilities. Finally, we decided to use a solution also described in [1] that results in a similarity value between 0 and 1 (1 for identical trees T_1 and T_2).

$$sim_{MaxCommonSubtree}(T_1, T_2) = \frac{2 \times |V_m|}{|V_1| + |V_2|} \qquad (1)$$

Having two trees T_1 and T_2 with $|V_1|$ and $|V_2|$ nodes, where $|V_1| \leq |V_2|$, the algorithm for ordered trees runs in $O(|V_1| \log |V_2|)$ time using $O(|V_1| + |V_2|)$ additional space (see Theorem 4.56 in [16]). The algorithm for unordered trees takes $O((|V_1|+|V_2|)^2)$ time and also uses $O(|V_1|+|V_2|)$ additional space (Theorem 4.60 in [16]).

Top-down Maximum Common Subtree Isomorphism. An algorithm to find a top-down maximum common subtree isomorphism for ordered and unordered trees is defined in [16]. This algorithm finds the largest common subtree of two given trees T_1 and T_2 under the prerequisite that the root of the common subtree is identical (same node type) with the root nodes of the compared trees. The differences between the algorithm for ordered trees and the algorithm for unordered trees are fundamental. For this paper, we implemented the algorithm for ordered tree matching.

Starting from the root nodes of T_1 and T_2, the algorithm recursively processes all children in preorder and compares each pair of nodes for equality. If two nodes match, they are added to a mapping $M \subseteq V_1 \times V_2$ that contains the complete subtree after the recursion finishes. Note that the recursion stops at nodes which do not match, i.e., the children of non-matching nodes are not getting compared to each other.

The comparison of the nodes during the recursive processing again allows for an extension of the algorithm to labeled trees, returning a successful match only when the labels (node types) match. Again, we used Equation 1 to get a similarity score from the size of the maximum common subtree and the two trees T_1 and T_2 under comparison.

This algorithm is very efficient with a running time of $O(|V_1|)$ and $O(|V_1|)$ additional space for two ordered trees T_1 and T_2, where $|V_1| \leq |V_2|$ (see Lemma 4.52 in [16]).

Tree Edit Distance. Calculating the tree edit distance is a completely different approach for tree analysis than the maximum common subtree isomorphism algorithms. The tree edit distance algorithm answers the question of how many steps it takes to transform one tree into another tree by applying a set of elementary edit operations to the trees: insertion, substitution, and deletion of nodes. For the ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ we denote a *deletion* of a leaf node $v \in V_1$ by $v \mapsto \lambda$ or (v, λ). The *substitution* of a node $w \in V_2$ by a node $v \in V_1$ is denoted by $v \mapsto w$ or (v, w) and an *insertion* of a node $w \in V_2$ as a new leaf into T_2 is de-

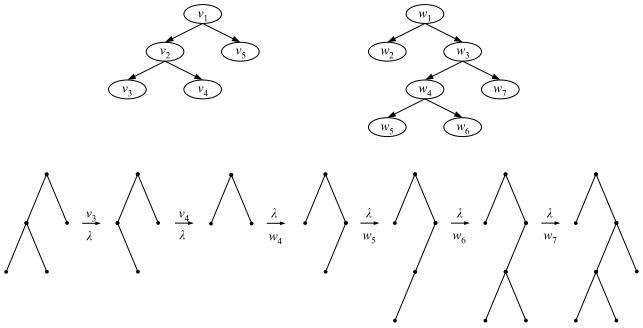

Figure 4: Transformation between two ordered trees (adapted from Fig. 2.1 in [16], page 56).

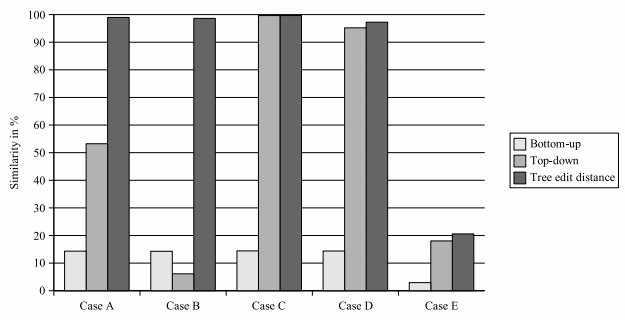

Figure 5: Results with constructed test cases for each similarity measure.

noted by $\lambda \mapsto w$ or (λ, w). Deletion and insertion operations are performed on leaves only. When deleting a non-leaf node v, every node in the subtree rooted at v has to be deleted first. The same applies to the insertion of non-leaves. A tree is transformed into another tree by using a sequence of elementary edit operations as illustrated in Figure 4. Note that in this figure, substitution of corresponding nodes is not indicated. The complete transformation script is: $[(v_1, w_1), (v_2, w_2), (v_3, \lambda), (v_4, \lambda), (v_5, w_3), (\lambda, w_4), (\lambda, w_5), (\lambda, w_6), (\lambda, w_7)]$. Costs are assigned to all elementary edit operations. Our current implementation uses a cost function of $\gamma(v, w) = 1$ if $v = \lambda$ or $w = \lambda$ and $\gamma(v, w) = 0$ otherwise. The function reflects that node substitutions usually denote relabelings which have little structural significance and should, therefore, not be weighted. The edit distance then is the least-cost transformation of T_1 to T_2 normalized by the sum of nodes in T_1 and T_2. The lower the normalized edit distance of two trees, the higher their similarity.

$$sim_{TreeDistance}(T_1, T_2) = \frac{TreeDist(T_1, T_2)}{|V_1| + |V_2|} \qquad (2)$$

Finding the least-cost transformation of an ordered tree T_1 and T_2 by determining shortest paths in an edit graph runs in $O(|V_1||V_2|)$ time using $O(|V_1||V_2|)$ additional space (see Lemma 2.20 in [16]).

3. EXPERIMENTAL EVALUATION

To evaluate the ability of our approach to detect similar entities in a software project, we ran the evaluation on two datasets. First, we constructed a set of special test cases capturing frequently occurring changes which happen during software development. Evaluating our similarity detection algorithms for these constructed changes, we were able to analyze how specific changes affect structural similarity. In a second experiment we chose a well known Java project as the dataset for our implemented similarity measures. We used Eclipse's compare plug-in `org.eclipse.compare`[2] and measured the similarity of the classes within the same version as well as between different versions of the plug-in. This analysis focused on the efficiency of the measures for detecting structural similarities between two classes in the former and on software evolution considerations in the latter case. The remainder of this section shows our obtained results of this two experiments.

[2]`http://dev.eclipse.org`

3.1 Experiment #1: Constructed Test Cases

As a basis for the construction of the test cases we took the class `AzureusCoreImpl` from the Azureus project (except in test case E).[3] This class was chosen as it uses both "normal" and `static` attributes and methods. Additionally, it defines getter and setter methods for its attributes. In addition to the test cases we compared `AzureusCoreImpl` with an empty class to ensure the plausibility of our implemented measures. The defined test cases are the following:

Test Case A: Add Constructor. This test case adds a new constructor with a single `this()`-invocation as body to the class.

Test Case B: Add Attribute. We add a new attribute with its respective getter and setter methods to the class. The rest of the class is left unchanged.

Test Case C: Add Invocation. We insert an invocation, i.e., a method call into the body of an existing method.

Test Case D: Method Extraction. This case models the movement of code statements from an existing method into a new method. An invocation of the new method is added to the original one. This change often happens during a code refactoring to remove duplicated code or when pulling-up code into parents [2].

Test Case E: Implement Interface. The interface programming pattern is one of the most important design patterns in object-oriented programming [3]. This test case measures the similarity between classes implementing the same interface. We expect this case to have very low structural similarity, as the structural similarities between classes implementing the same interfaces are limited to the implementation of the methods defined in the interface, which may be implemented in structurally completely dissimilar ways.

3.1.1 Results of Test Cases

This section presents our findings, discusses major drawbacks, and details the performance of the implemented algorithms. The results of this experiment are shown in Figure 5. Confirming our expectations, none of the algorithms

[3]`http://azureus.sourceforge.net`

detected any significant similarity in Test Case E. A similarity measure finding classes implementing the same interface would have to put special emphasis on the interface definitions, which none of our measures does.

Bottom-up Maximum Common Subtree Isomorphism. The obtained results show clearly that the bottom-up subtree isomorphism algorithm performs worst for detecting similar Java classes. The similarity score remains at about 14%. The reason for this is that this measure uses equivalence classes for checking equality of different nodes. For a better match, the measure would have to include the unchanged parts of the tree as well, requiring the equivalence class of the root to remain the same. This is not the case in our tests as a node insertion/deletion at tree depth level 1 changes the equivalence code of the root. Hence, the measure matches the biggest subtree from level 2, which usually is the biggest method.

Top-down Maximum Common Subtree Isomorphism. We obtain mixed results with this measure. Case C, D, and partly case A show good scores for detecting similarity with the top-down algorithm. In case B, similarity is not well detected because the insertion of a variable stops the matching process too early. Hence, this algorithm overvalues small changes near the root, such as simple insertions, deletions, and relabelings. This limitation could possibly be lessened by continuing the comparison even when nodes have different names or by using a top-down algorithm for unordered trees.

Tree Edit Distance. The tree edit distance algorithm performed best for our test cases. The similarity scores in each test (except E) are over 97%, which is sufficient for establishing a similarity relation between two classes with a high accuracy. The big advantage of this algorithm in comparison to the two maximum common subtree algorithms is that it is not as susceptible to node insertions/deletions.

3.2 Experiment #2: Java Project

The `org.eclipse.compare`-plug-in of Eclipse is used as sample, real-world Java project to test the similarity measures. The analysis demonstrates the ability of using the implemented similarity measures in a non-laboratory environment and critically highlights shortcomings. "The plug-in provides support for performing structural and textual compare operations on arbitrary data" (from its Javadoc). The goal of this experiment was to visualize the project's code evolution steps by comparing different versions of the plug-in. I.e., how strongly is class similarity affected when changes/refactorings occur to the code from one version to the next. To achieve this goal, we have compiled a heatmap illustrated in Figure 6 showing similarities between classes of two different versions of the plug-in. For our experiment, we used versions 3.0 and 3.1 of the compare-plug-in.

3.2.1 Results of org.eclipse.compare-Plug-in

Based on our findings of Experiment #1 (see Section 3.1), we chose to perform a similarity analysis using the tree edit distance measure to determine class similarity within the `org.eclipse.compare`-plug-in. Consider the heatmap depicted in Figure 6: class similarity is computed between any class of version 3.0 (on the x-axis) and 3.1 (on the y-axis).

Figure 6: Heatmap showing similarities between all classes of versions 3.0 and 3.1 of the `org.eclipse.compare`-plug-in. Black squares indicate similarity above 90%, dark-gray above 75%, and light-gray above 50%.

The similarity score between two classes is visualized as a shaded square. Similarity above 90% is indicated by black squares. Squares shaded in dark-gray denote class similarity above 75%, whereas squares in light-gray indicate similarity above 50%. Similarities below 50% are not shaded.

The elements on the diagonal indicate how strong the software has changed between versions. I.e., the diagonal is clearly visible as a dark "line" showing that the same classes of two versions have very high similarity—more than 75% in most cases. As shown in Figure 6, a few classes have changed from version 3.0 to 3.1. As an example, we picked the class `MergeMessages` from the package `org.eclipse.compare.internal.merge` (indicated in Figure 6 by a circle on the line marked with the arrow labeled with a "2") that has a similarity of 74% between the two versions. Examining the source code of the two versions, we found that three more static fields were added and one method was removed and replaced with a static initializer in version 3.1. Hence, three more attribute nodes and one method node with an invocation node are added to the tree representation in version 3.1 resulting in a similarity score of 74%.

The map of Figure 6 is not only useful to visualize software evolution but also to measure and visualize similarity between any two classes of a software project. Again, we explain this with an example: "The interface `ICompareNavigator` is used to navigate through the individual differences of a `CompareEditorInput`" (from its Javadoc). It is a simple interface defining exactly one method with one argument. Similarity between this interface and 25 other classes/interfaces out of 114 is very high, i.e., above 75% (refer to Figure 6 that shows an arrow labeled with a "1" pointing to the line corresponding to `ICompareNavigator` in this

heatmap). The entities with high similarity are, for instance, `INavigatable` (sim=75%) also specifying one method with one parameter but, in addition, defining a static field. Hence, the corresponding tree representations of these interfaces are very similar, which is correctly reflected by their high similarity value. Another interface similar to `ICompareNavigator` is `IViewerDescription` (sim=75%) also defining one method but with 3 arguments instead of 1 as in the case of `ICompareNavigator`. Note that Figure 6 shows the similarities of classes in version 3.0 with those in 3.1. To compare the similarity of classes within the same version, we double-checked our findings in a heatmap comparing classes from version 3.0 with all other classes from version 3.0 (omitted due to space constraints). Naturally, all similarities on the diagonal in this within-version analysis were 100% equal since classes where compared with themselves. The line comparing `ICompareNavigator` with the other classes within the package showed a similar (but not the same) behavior as in the between-version analysis above.

4. DISCUSSION

In this section we discuss the results of our experiments, highlight shortcomings of the tree similarity algorithms and propose possible improvements for future work.

4.1 Comparison of Implemented Measures

Our obtained results showed that a bottom-up maximum common subtree isomorphism match is not a good measure for similarity when evaluated on the user-constructed test cases. It is too susceptible to subtle code modifications in methods, which usually cause changes at the bottom-level of the tree.

The top-down maximum common subtree algorithm produces promising results on the test cases. The measure is a good indicator for similarity as it is able to detect classes with similar structures. One negative characteristic of this algorithm is that simple changes at the top of the tree, such as adding a new attribute or inserting an attribute between existing methods, reduces the reliability of the measure. The top-down approach is, however, not as sensitive to changes at the bottom of the trees as the bottom-up approach.

The best-performing similarity algorithm turned out to be the tree edit distance on both datasets. It detected the small changes specified in the test cases and provided itself as a good indicator for structural similarity when tested on the compare-plug-in. Note that the price to pay for it is high: it runs in quadratic time complexity $O(|V|^2)$ of its tree input size, which resulted for the $n \times n$-comparison of 114 classes of the `org.eclipse.compare`-plug-in in more than an hour to finish.

4.2 Limitations and Future Work

Field or Method Name Matching. Additional information can be gained from class, field, or method names in similarity comparisons. Methods and fields used for similar tasks are (hopefully) named with similar names if the code was created abiding reasonable naming standards. This helps detecting cloned parts of classes. Text-based similarity measures can then be used to calculate similarity between names. The current implementation of Coogle treats nodes representing `class X` and `class Y`, for instance, as equal since only the types ("Class" in this case) of nodes are compared. Note that because Coogle currently does not compare names, also access control qualifiers, such as `public`, `protected`, and `private`, are not taken into consideration since those are not proper FAMIX entities, but simple string attributes of entities.

FAMIX Limitations. The FAMIX model represents a fixed set of elements (see Table 1), i.e., invocations, declarations, attributes, etc., but does not include assignments, mathematical operations, and control structures. This limits the detection of small changes to basic, top-level instructions. Different results are to be expected when bypassing FAMIX and directly generating the comparisons from complete abstract syntax trees or when extending the FAMIX model with a more fine-grained hierarchical structure. This might help to detect "real" functionally similar classes and diminish the detection of classes only structurally similar. On the other hand, because the size of the input trees would be much bigger, the algorithm's performance would get worse.

Surrounding String Matching. We plan to include surrounding text in similarity comparisons, as statements in source code are frequently surrounded by free text, such as Javadoc. Analyzing the similarity of this text and including this textual similarity in the measures will probably further boost the precision of the similarity algorithms.

Similarity Measure Combinations. We think it is useful to investigate different combination approaches of structure-based as well as text-based similarity measures to further increase the precision of matches. We, therefore, postponed the implementation and evaluation of such combinations to the future.

5. RELATED WORK

Both Mishne & Rijke [7] and Neamtiu et al. [9] define a conceptual model for source code representation that partially resembles the abstract syntax tree as defined by Eclipse. Mishne & Rijke use code similarity for retrieving similar code fragments from an existing repository of code documents based on classifying instructions in the code with varying weights. They do, however, not apply tree similarity measures to retrieve similar code fragments from the repository. Neamtiu et al. extract similarity by mapping corresponding AST elements of two code documents. Again, this algorithm does not use a generic tree similarity algorithm. Their focus lies on the mapping algorithms to measure similarity between AST representations, which relies on node names within the ASTs.

A different approach is introduced by Kontogiannis who defines a Program Description Tree (PDT) that is generated from code fragments [5]. These fragments are treated as behavioral entities, i.e, as independent components interacting with resources and other entities of the software project. The PDT, therefore, not only represents structural information like an AST does, but also contains information such as interactions and read or write accesses, i.e., behavioral information. Similar fragments are detected by searching for entities with similar characteristics of these PDTs. In our current implementation of Coogle we do not examine attribute accesses, but take interactions, such as method invocations, into account. It would be interesting to extend

Coogle to operate on PDTs and compare the performance of the two.

A similar approach to ours is described by Holmes & Murphy in [4]. Their tool, *Strathcona*, is used to find source code in an example repository by matching the code a developer is currently writing. In contrast to Coogle, this approach is not based on tree similarity algorithms, but on multiple structural matching heuristics, such as examining inheritance relationships, method calls, and class instantiations. The measures are applied to the currently typed code. Matched examples from the repository are retrieved and displayed to the developer for selection. We have not yet implemented such a feature in Coogle, which could clearly help to foster code reuse. Again, it would be interesting to compare their approach that makes heavy use of domain knowledge (in the form of heuristics) with ours that solely relies on the power of the similarity algorithms.

A promising approach to help developers navigate source code efficiently is presented by Robillard in [11]. The presented technique is able to find relevant (in our context similar) elements in source code to a given query element by examining the topological properties of the structural dependencies of the query element. Elements are considered as relevant if they fulfill well-defined criteria, such as specificity and reinforcement to other elements. In that context, it would be interesting to know about the performance of our implemented tree similarity algorithms as another criteria to determine relevant elements of interest.

Finally, various other approaches exist for detecting similarity in trees and source code. Baxter & Manber describe a tool that analyses projects for duplicated code [1]. Their implemented algorithm is based on abstract syntax trees and employs a hashing function on code fragments for detecting exact and near-miss clones. Myles & Collberg take a similar approach by using a birth-marking technique, deducing unique characteristics from the instruction set of a program to detect software theft [8]. While it would make perfectly sense to use Coogle as a tool to find code clones and possible software plagiarisms, we currently only employ it to find similarity between classes to, for instance, comprehend software evolution steps.

6. CONCLUSION

In this paper we presented our approach to detect similarities between different Java classes based on abstract syntax trees. Similarity is calculated by means of three tree similarity algorithms: bottom-up maximum common subtree isomorphism, top-down maximum common subtree isomorphism, and the tree edit distance. We chose the FAMIX model of object-oriented programming languages to represent the compared trees.

The trees under comparison are generated in two steps: first, the abstract syntax tree representation of the source code is built through Eclipse's ASTParser, and second, our AST2FAMIX parser traverses the abstract syntax tree and builds a FAMIX representation from the nodes of the tree. Finally, the similarity algorithms are used to determine the degree of similarity between different FAMIX trees.

We validated our approach on two datasets: user-constructed test cases and the `org.eclipse.compare`-plug-in. Of the three tree similarity measures, we found the tree edit distance to produce the best results, followed by the top-down maximum common subtree isomorphism algorithm. Especially, when measuring the effects of small changes, the tree edit distance measure proved to be very robust. When evaluating our approach on the `org.eclipse.compare`-plug-in, we successfully detected structural similarities between classes of the same version. In addition, our approach proved to be very promising when comparing different versions of the project, i.e., when focusing on questions concerning software evolution, for instance, how did the classes change from one release to the next. Visualizing class similarities by the use of heatmaps, we were able to illustrate this code evolution within the `org.eclipse.compare`-plug-in.

As our initial results have shown, our approach is very promising. It (1) indeed identified similar Java classes, (2) successfully identified the ex ante and ex post versions of refactored classes, and (3) provided some interesting insights into the within-version and between-version dependencies of classes within a medium-sized Java project.

7. REFERENCES

[1] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke,and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[4] R. Holmes and G. C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proceedings of the 27th International Conference on Software Engineering*, pages 117–125, 2005.

[5] K. Kontogiannis. Program Representation and Behavioural Matching for Localizing Similar Code Fragments. In *Proceedings of the 1993 Conf. of the Center for Advanced Studies on Collaborative Research*, pages 194–205, 1993.

[6] A. Michail and D. Notkin. Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships. In *Proceedings of the 21st International Conference on Software Engineering*, pages 463–472, 1999.

[7] G. Mishne and M. de Rijke. *Source Code Retrieval using Conceptual Similarity*. 2004.

[8] G. Myles and C. Collberg. K-Gram Based Software Birthmarks. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 314–318, 2005.

[9] 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*, pages 1–5, 2005.

[10] Object Technology International, Inc. Eclipse Platform Technical Overview. 2003.

[11] M. P. Robillard. Automatic Generation of Suggestions for Program Investigation. In *Proceedings of the 10th European Software Engineering Conference*, pages 11–20, 2005.

[12] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 39–52, 2002.

[13] S. Tichelaar. *FAMIX Java Language Plug-in 1.0*. 1999.

[14] S. Tichelaar, P. Steyaert, and S. Demeyer. *FAMIX 2.0: The FAMOOS Information Exchange Model*. 1999.

[15] G. Valiente. Simple and Efficient Tree Pattern Matching. Technical Report LSI-00-72-R, Technical University of Catalonia, Dec. 2000.

[16] G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, Berlin, 2002.

[17] J. T.-L. Wang, H. Shan, D. Shasha, and W. H. Piel. TreeRank: A Similarity Measure for Nearest Neighbor Searching in Phylogenetic Databases. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management*, pages 171–180, 2003.

[18] K. Zhang, R. Statman, and D. Shasha. On The Editing Distance Between Unordered Labeled Trees. *Information Processing Letters*, 42(3):133–139, 1992.