Classifying Change Types for Qualifying Change Couplings

Beat Fluri and Harald C. Gall
s.e.a.l. – software evolution and architecture lab
Department of Informatics
University of Zurich, Switzerland
{fluri,gall}@ifi.unizh.ch

Abstract

Current change history analysis approaches rely on information provided by versioning systems such as CVS. Therefore, changes are not related to particular source code entities such as classes or methods but rather to text lines added and/or removed. For analyzing whether some change coupling between source code entities is significant or only minor textual adjustments have been checked in, it is essential to reflect the changes to the source code entities. We have developed an approach for analyzing and classifying change types based on code revisions. We can differentiate between several types of changes on the method or class level and assess their significance in terms of the impact of the change types on other source code entities and whether a change may be functionality-modifying or functionality-preserving. We applied our change taxonomy to a case study and found out that in many cases large numbers of lines added and/or deleted are not accompanied by significant changes but small textual adaptations (such as indentation, etc.). Furthermore, our approach allows us to relate all change couplings to the significance of the identified change types. As a result, change couplings between code entities can be qualified and less relevant couplings can be filtered out.

1. Introduction

One effective way to overcome or avoid the negative effects of software aging is by placing *change* in the center of the software development process. In particular, understanding the nature of fine-grained source code changes can help comprehending software evolution.

Current change history analysis approaches rely on information provided by versioning systems such as CVS. Versioning systems usually track changes for files on a text basis and are unable to enrich changes with source code structure information. That means, changes in CVS are not explicitly associated with particular source code entities such as methods or classes without reconstructing the change location as done by [28]. Moreover, changes may be reported although no source code entity was modified. For understanding source code changes, reconstructing the changed method or class is insufficient, because changes on a particular source code entity can modify its behavior and further have different impacts on other entities. Such a classification of source code changes is desirable, bot not possible with a text *diff* approach. Existing works such as [19] and [22] have shown that classifying source code changes is needed to help increasing the change impact awareness.

In this paper, we present a *taxonomy of source code changes* to be used for our further change analysis. We define source code changes according to tree edit operations in the abstract syntax tree and classify each change type with a *significance level* that expresses how strong a change may impact other source code entities and whether a change may be functionality-modifying or functionality-preserving. This classification allows us to assess source code entities enabling further software evolution analysis: it can be used to qualify error-proneness of source code entities, assess change couplings [18], or identify programmer clean-up patterns [6].

In particular, we focus on the definition of change types as well as the significance level classification and describe the potential of our approach using ArgoUML as case study. We implemented our approach as the Eclipse plugin CHANGEDISTILLER, and show how the significance level of changes relates to the lines added/removed information provided by CVS. Additionally, we show how change couplings can be better qualified with our classification.

The remainder of this paper is organized as follows. In Section 2 we describe the concepts and the taxonomy of source code changes. Section 3 defines a set of source code changes and assigns a significance level to each of them. The implementation of CHANGEDISTILLER is presented in Section 4 and is used for a validation of our taxonomy in Section 5. We discuss related work in Section 6 and finalize with our conclusions in Section 7.

2. A Taxonomy of Source Code Changes

Our taxonomy of source code changes defines source code change types with tree edit operations on the abstract syntax tree. Further, it classifies these change types according to a significance level schema. In this section we present how the change types are defined and how they are classified. The definitions and corresponding significance level of the chosen set of source code changes is presented in Section 3.

2.1. Change Types

The taxonomy of source code changes given in this paper focuses on object-oriented programming languages (OOPLs)—in particular for Java. By adjusting the change descriptions the taxonomy can also be used for other OOPLs.

In most OOPLs the concept *class* defines the framework for encapsulating functionality and state. In our taxonomy, a class is divided into *body*- and *declaration*-parts: *class body* and *method body* as well as *class declaration*, *attribute declaration*, and *method declaration*. We are interested in changes in both parts, *e.g.*, the parameter type of the method declaration has changed or the assignment was moved outside an if-statement.

Level of Granularity

Our taxonomy of source code changes is based on the abstract syntax tree (AST). The smallest entities used are statements; structure statements such as loop or control structures are more coarse-grained than normal statements and are treated separately.

We use the term *source code entity* representable for all language constructs provided by an OOPL, such as single and composed statements as well as method or class declarations. In an AST these source code entities are either sub-ASTs or leafs. For our taxonomy, ASTs consist of *entity nodes* with labels and values. The label represents the kind of source code entity, the value its textual representation depending on the kind of entity. For instance, the *method invocation* statement is a leaf node. The label of the corresponding entity node is 'method_invocation' and the value is the method invocation as a string.

For the technical notation of entity nodes we use the terminology of Chawathe *et al.* [2]. That means, for a node x, $l(x)$ denotes the label of x, $v(x)$ denotes the value of x, and $p(x)$ denotes the parent of x, if x is not the root. Children of an ordered entity tree node u are indexed, $\langle v_1, ..., v_m \rangle$, *i.e.*, a sequence of nodes. We call v_i the ith child of u.

We distinguish between ordered and unordered entity trees. In most object-oriented languages such as Java or C++, methods or attributes of a class, *i.e.*, its children entity trees, do not have a particular order. Statements inside a method must have an order.

Basic Operations

Since ASTs are rooted trees and since source code changes transform an AST, the basis for source code changes are elementary tree edit operations. The detection of source code changes falls into the tree edit distance problem.

According to [24] the *elementary tree edit operations* are *insert*, *delete*, and *substitute* of a tree node. Insert and delete operations are only allowed on leaf nodes. Substitution is a form of replacement of a tree node v with another, existing node w. In our taxonomy of source code changes, we use a weaker kind of substitution. The statements (*i.e.*, nodes) are not replaced with other statements, but their values are *updated*. Consider an if-statement with a certain condition. By changing the condition of the if-statement, the value of the entity node is *updated* with the new condition. The advantage of this terminology is that the entity tree (entity node and its subtrees) of the if-statement remains the same. The update operation is applied on the value of an entity node. That means, for instance, that changing the then-block of an if-statement is not an update of the if-statement.

Move operations can be described as a combination of an insert and a delete operations. In the context of source code change we may give the move operation more importance, *e.g.*, changing the order of statements to gain performance. We include the move operation in the elementary tree edit operations.

Our current implementation of the source code change classification is based on the change detection algorithm of Chawathe *et al.* in [2]. The change types used for our classification are built upon the output of this algorithm. We use a similar definition of the elementary tree edit operations as Chawathe *et al.*.

In our taxonomy we use the subscripts $_{old}$ and $_{new}$ to describe the values of the subscripted variable before and after the change.

- **Insert**: $\text{INS}((l, v), y, k)$; new leaf node with label l and value v as kth child of node y.
- **Delete**: $\text{DEL}(x)$; delete node x from its parent $p(x)$.
- **Move**: $\text{MOV}(x, y, k)$; node x becomes the kth child of y and is deleted from $p(x)$ iff $p_{old}(x) \neq y$.
- **Update**: $\text{UPD}(x, val)$; update $v(x)$ with val, *i.e.*, $val = v_{new}(x)$ and $v_{old} \neq v_{new}$.

Each change type is defined in the following format:

X *denotes the Y Z*.

"X" is written in small caps and is the name of the defined change type. "Y" is the name of the elementary tree edit operation and "Z" the tree edit operation applying the change.

2.2. Significance of Changes

Our classification of source code changes defines how significant a certain change is. The *significance level of a change* is defined as the impact of the change on other source code entities, *i.e.*, how likely is it that other source code entities have to be changed, when a certain change is applied. Additionally, whether a change is *functionality-preserving* or *functionality-modifying* also influences the significance level of a change. If a change modifies the functionality of the enclosing entity, it is functionality-modifying, otherwise functionality-preserving. For instance, although a renaming strongly induces other changes, it does not (or should not) change the functionality of a program. Note, that functionality-modifying or -preserving relates to a single change. By all means it is possible that a set of functionality-modifying changes is functionality-preserving for the corresponding program.

To classify the significance of a source code change, we use the significance levels *low*, *medium*, *high*, and *crucial*. Local changes, such as changes in a method body, are considered to have a low or medium significance level, whereas changes on the interface of a class have a high or crucial significance level. For instance, if a parameter name of a method signature is changed, each access of this parameter inside the method body has to be changed. Indeed, such a change induces many other changes and according to its impact the significance level is high. However it does not change the functionality of the method. Therefore, we define the significance level of parameter renaming as *medium*.

3. Source Code Changes

This section presents the chosen set of source code change types. Each source code change is defined as a single or a set of basic tree edit operations. In each definition the significance level of a change is motivated and defined. We start with body-part changes from fine- to coarse-grained entities and finish with declaration changes. A comprehensive overview of the change types and their classification is given in Appendix A.

3.1. Body-Part Changes

Method Body Changes

Method body changes are changes on the entities control structure, loop structure, and statement. Although programming languages may have different loop structures, they are all reducible to the simplest loop form (*e.g.*, while) and are not treated separately; similar for control structures, *e.g.*, switch-case structure can be expressed as if-else structures.

For method-body changes we assume: 1) the parent of a statement s, $p(s)$, is either a statement or the method declaration; and 2) statements are ordered children of their parents, *i.e.*, if statements $\langle s_1, ..., s_m \rangle$ are children of statement t, then s_i is the ith child of t.

STATEMENT ORDERING CHANGE *denotes the move operation* $\text{MOV}(s, p(s), k)$.

A statement ordering change may induce a change of the postcondition of the method and impact calling methods. However, changing the ordering of the statements may also be applied according to other functionality-preserving criteria, such as performance. We define the significance level of the statement ordering change as *low*.

STATEMENT PARENT CHANGE *denotes the move operation* $\text{MOV}(s, y, k)$*, with* $p_{old}(s) \neq p_{new}(s)$.

It is interesting to know, what the old and new parent of the statement is. Moving a statement from an then-block to a loop may have more effect than from the else-block to the then-block. Since the parent of a statement and accordingly the label of the parent $l(p(s))$ is known, we are able to distinguish between different statement parent changes. As changing the parent of a statement has more impact on the postcondition of the method than a simple ordering change, we define the significance level of a statement parent change as *medium*.

STATEMENT INSERT *denotes the insert operation* $\text{INS}((l(s), v(s)), y, k)$.

STATEMENT DELETE *denotes the delete operation* $\text{DEL}(s)$.

STATEMENT UPDATE *denotes the update operation* $\text{UPD}(s, val)$.

With the label of the statement $l(s)$ and the one of its parent $l(p(s))$ we are able to describe the insert and delete operation in more detail, *e.g.*, a method invocation was inserted/deleted in/from a loop. This applies for updates as well.

Inserting and deleting statements are mostly functionality-modifying operations, whereas updating statements may also be applied because of renaming of a variable, parameter, or method. We define the significance levels of insert and delete changes as *medium* and that of update changes as *low*.

Structure Statements. The statement changes discussed above also apply to the structure statements loop L and con-

trol structure CS, but have further impact. Inserting, deleting, or moving structure statements may change the overall nested depth $\delta(M)$ of a method M. We distinguish between *increasing*, iff $\delta_{old}(M) < \delta_{new}(M)$, insert and move changes; and *decreasing*, iff $\delta_{old}(M) > \delta_{new}(M)$, delete and move changes.

As the overall nested depth of a method is an indicator for the complexity (sometimes also for the error-proneness) of a method [16], we define the significance level of these increasing and decreasing changes as *high*.

In our tree representation of source code entities, we chose the condition expression, $CE(\cdot)$, of a structure statement as the value of the corresponding entity node. We call the update operation of a structure statement as *loop/control structure condition expression change*: $\text{UPD}(L, v(CE_{new}(L)))$ and $\text{UPD}(CS, v(CE_{new}(CS)))$.

Changing the condition expression of a structure statement is functionality-modifying and may also impact other changes inside the method-body. We define the significance level of this change as *medium*.

A control structure has an alternative path, a so called else-part without a condition.[1] Inserting or deleting an else-part does not have any impact on the overall nested depth of a method, but is functionality-modifying: *else part insert* and *else part delete* with significance level *medium*.

Class Body Changes

We have already used and described changes on the body of a class in [4]. Inserting and deleting of attributes and methods fall into this category.

For attributes, we call these operations *additional object state* and *removed object state* changes, because attributes describe the state of an object unless it is not declared as static. An additional object state change has not any impact and is functionality-preserving. It has a *low* significance level. A removing object state change is functionality-modifying and all accesses on the attribute have to be deleted, such a change has a *crucial* significance level.

We call the basic changes of methods *additional functionality* and *removed functionality* changes. Their significance levels are defined analogously as the basic changes of attributes: *low* and *crucial*.

Since in most object-oriented programming language the entities in a class must not be ordered, move operations are not considered.

3.2. Declaration-Part Changes

Each declaration part may have modifiers. Changes on modifiers can be declared in general. We distinguish between access and final modifiers. Static modifiers are currently not considered.

Access Modifier Changes

Access modifiers describe how restricted the access is to a class C, method M, or attribute A. We use the terminology of Java for access modifiers: $\mu_A = \{\text{private}, \kappa_A, \text{protected}, \text{public}\}$, with κ_A as the default access modifier, meaning that no access modifier is given.

Changes on access modifiers are defined once for all declaration-parts, because they have the same meaning for a class, method, and attribute.

INCREASING ACCESSIBILITY CHANGE *, e.g., changing a method from* protected *to* public*, denotes the insert operation* $\text{INS}((l(\mu_A), v(\mu_A)), y, 1)$*, where either* $v(\mu_A) =$ protected *or* $v(\mu_A) =$ public*; the delete operation* $\text{DEL}(\mu_A)$ *with* $v(\mu_A) =$ private*; or the update operation* $\text{UPD}(\mu_A, val)$*, where either* $v_{old}(\mu_A) =$ private *and* $val =$ protected $\vee$ $val =$ public*, or* $v_{old}(\mu_A) =$ protected *and* $val =$ public.

The increasing accessibility change is functionality-modifying. Its impact on other changes is rather low, because accesses on an entity may remain unmodified. We define the significance level of this change as *medium*.

DECREASING ACCESSIBILITY CHANGE *, e.g., changing an attribute from* protected *to* private*, denotes the insert operation* $\text{INS}((l(\mu_A), v(\mu_A)), y, 1)$ *with* $v(\mu_A) =$ private*; the delete operation* $\text{DEL}(\mu_A)$*, where either* $v(\mu_A) =$ protected *or* $v(\mu_A) =$ public*; or the update operation* $\text{UPD}(\mu_A, val)$*, where either* $v_{old}(\mu_A) =$ public *and* $val =$ protected $\vee$ $val =$ private*, or* $v_{old}(\mu_A) =$ protected *and* $val =$ private.

Decreasing accessibility change has a deep impact on other changes. In the worst case all accesses on an entity have to be changed. The significance level of this change is defined as *crucial*.

Final Modifier Changes

Besides access modifiers, a class, method, or attribute may also be declared as *final*, $\mu_F = \{\text{final}, \kappa_F\}$ where κ_F denotes the empty final modifier, meaning that no final modifier is given.

Possible final modifier changes are the basic operations insert and delete. Update or move are not reasonable for this modifier.

FINAL MODIFIER INSERT *denotes the insert operation* $\text{INS}((l(\mu_F), v(\mu_F)), y, 2)$ *with* $y \in \{C, M, A\}$.

FINAL MODIFIER DELETE *denotes the delete operation* $\text{DEL}(\mu_F(x))$ *with* $x \in \{C, M, A\}$.

[1] 'else if' is modeled as an else-part with a new control structure

As the meaning of the final modifier is different for a class, a method, and an attribute, we give corresponding names for these changes:

- **Class**. The final modifier of a class declares the class as not derivable. We call the insert operation *removing class derivability* and the delete operation *adding class derivability*.
- **Method**. The final modifier of a method declares the method as not overridable. We call the insert operation *removing method overridability* and the delete operation *adding method overridability*.
- **Attribute**. The final modifier declares an attribute as unmodifiable. We call the insert operation *removing attribute modifiability* and the delete operation *adding attribute modifiability*.

Inserting the final modifier to one of the three entities is functionality-modifying and induces many changes, *e.g.*, deleting all write accesses to an attribute. We define the significance level of this change as *crucial*.

Conversely, deleting the final modifier has no impact on other changes and is functionality-preserving. We define the significance level for this change as *low*.

Attribute Declaration Changes

An attribute declaration A contains an access $\mu_A(A)$ and a final modifier $\mu_F(A)$, a type $T(A)$, and a name $n(A)$. The attribute initializer is not considered in this taxonomy.

ATTRIBUTE TYPE CHANGE *denotes the update operation* $\text{UPD}(T(A), v(T_{new}(A)))$.

An attribute type change is functionality-modifying and implies changes on all accesses of the attribute—even worse when the attribute is public. We define the significance level of this change as *crucial*.

ATTRIBUTE RENAMING *denotes an update operation of the name in an attribute declaration* $\text{UPD}(n(A), v(n_{new}(A)))$.

As attribute renaming also implies changes on all accesses of the attribute, but is functionality-preserving, its significance level is *high*.

Method Declaration Changes

Besides the access and final modifier, a method declaration M contains an optional return type $T(M)$, a name $n(M)$, and a parameter sequence $P(M)$, $\langle \rho_1, .., \rho_m \rangle$. A parameter $\rho \in P(M)$ has a type $T(\rho)$ and a name $n(\rho)$. The combination of the name and the parameter sequence is called the method signature $\sigma(M)$.

RETURN TYPE INSERT *denotes the insert operation* $\text{INS}((l(T(M)), v(T(M))), M, 3)$ *with* $T_{old}(M) = \{\}$.

RETURN TYPE DELETE *denotes the delete operation* $\text{DEL}(T(M))$.

RETURN TYPE UPDATE *denotes the update operation* $\text{UPD}(T(M), v(T_{new}(M)))$.

All three changes of the return type are strongly impacting source code changes, *i.e.*, functionality-modifying and in most cases all method invocations on the changed method declaration have to be adjusted. We define the significance level of this change as *crucial*.

METHOD RENAMING *denotes the update operation* $\text{UPD}(M, v(n_{new}(M)))$.

Method renaming is functionality-preserving, but all method invocations on the changed method declaration have to be adjusted. We define the significance level of this change as *high*.

PARAMETER INSERT *denotes two insert operations* $\langle \text{INS}((l(\rho), v(n(\rho))), P(M), k), \text{INS}((l(T(\rho)), v(T(\rho))), \rho, 1) \rangle$.

PARAMETER DELETE *denotes the delete operation* $\text{DEL}(\rho)$.

PARAMETER ORDERING CHANGE *denotes the move operation* $\text{MOV}(\rho, P(M), k)$.

PARAMETER TYPE CHANGE *denotes the update operation* $\text{UPD}(T(\rho), v(T_{new}(\rho)))$.

PARAMETER RENAMING *denotes the update operation* $\text{UPD}(\rho, v(n_{new}(\rho)))$.

All parameter changes, except for parameter renaming, are functionality-modifying and induce changes on method invocations. We define the significance leve of these changes as *crucial*. Parameter renaming is functionality-preserving and all accesses in the method body have to be adjusted. We define its significance level as *medium*.

Class Declaration Changes

The class declaration C contains access $\mu_A(C)$ and final modifiers $\mu_F(C)$, a name $n(C)$, and class $p_C(C)$ and/or interface parents $p_I(C)$.

CLASS RENAMING *denotes the update operation* $\text{UPD}(C, v(n_{new}(C)))$.

As with other renaming changes discussed so far, the class renaming change is also functionality-preserving, but may induces a lot of changes. We define the significance level of this change as *high*.

The inheritance concept in object-oriented programming languages is variably implemented. Some languages support multiple-inheritance, such C++ or Eiffel, other use interfaces instead, such as Java.

PARENT CLASS INSERT *denotes the insert operation* $\text{INS}((l(T), v(T)), p_C(C), k)$.

PARENT CLASS DELETE *denotes the delete operation* $\text{DEL}(T)$ *with* $T \in p_{Cold}(C)$.

PARENT CLASS UPDATE *denotes the update operation* $\text{UPD}(T, v(T_{new}))$ *with* $T \in p_{Cold}(C)$.

The parent interface changes are defined accordingly. All of the defined parent class/interface changes are functionality-modifying and normally induce many other changes. We define their significance level as *crucial*.

Because an interface declaration is a special kind of a class declaration, it is not treated separately.

3.3. Limitations of the Taxonomy

When a renaming of a method parameter happens and the statements bound to this parameter are updated, the change on this statement must not be classified as also proposed by Neamtiu *et al.* in [17]. Such a statement change is functionality-preserving and has no impact on other changes. To improve the current situation, updated statements can be represented in more detail, *i.e.*, generating an entity tree of the statement, and calculate differences upon the entity trees. Additionally, slicing methods and program dependence graphs [9] can be used to improve the functionality-modifyability of a statement update.

In the current classification, changes on exception handlings are not yet considered as well. An interesting discussion on exception handling changes can be found in [1].

4. Current Implementation

We built the Eclipse[2] plugin CHANGEDISTILLER that implements the source code change extraction algorithm. Our current implementation relies on the CVS capabilities and the Java Development Tools (JDT)[3] of Eclipse. The extracted source code changes are stored in a Hibernate[4] mapped database. The classification of the source code changes uses the data from this database.

CHANGEDISTILLER is able to report what source code changes occurred between two version of a particular Java class using the change definitions of Section 3. That means, in contrast to a textual diff, we have the possibility to state clearly which source code entities have changed, of which change type they are, and what level of significance they have.

4.1. Source Code Change Extraction

We have implemented the algorithm of Chawathe *et al.* to extract the basic tree edit operations. As CHANGEDISTILLER is an Eclipse plugin, the AST of a Java class is available through the JDT API. An AST generated by JDT is not a regular tree, *i.e.*, the tree structures of AST nodes are not always realized in the same way. Since the change detection algorithm expects labeled, valued tree nodes, the plugin construct an intermediate AST suitable for the algorithm. Leafs in such an intermediate AST are statements valued with their string representation.

CHANGEDISTILLER makes use of the CVS plugin shipped with Eclipse to check-out subsequent revisions of a Java source file. Changes of subsequent revision are extracted using the compare plugin of Eclipse giving the set of changed body- and declaration-parts. These parts are transformed into intermediate ASTs and fed into our implementation of the change detection algorithm. Starting the algorithm with two intermediate AST T_1 and T_2, it produces a so called *edit script*, meaning a set of basic tree edit operations transforming T_1 into T_2. The changes are stored in the database and are input for our classification.

Update operations on nodes are detected using the Levenshtein similarity measure for strings [14].

4.2. Classification

The changes stored in the database are used to classify the changes between two subsequent revision of a Java source file.

In some cases update changes must be reconstructed, because a sufficient similarity between two strings is not always guaranteed, *e.g.*, needed to detect type changes or renaming. Consider an attribute type change from "Object" to "Figure." The two strings do not share any character. Therefore, the implemented algorithm reports a type insert and delete change. Since the changed entity is an attribute and since an attribute has by declaration always an unambiguous type, the two operations can be combined to an update operation. Other reconstructions, such as increasing or decreasing accessibility changes, are done similarly.

4.3. Limitations

The implemented algorithm detects all but two changes. First, we do not yet consider class renaming, because we concentrate on Java source files which normally contain one class, except for inner classes. Second, decreasing statement changes detection requires a second parsing of the

[2]http://www.eclipse.org
[3]http://www.eclipse.org/jdt
[4]http://www.hibernate.org

method body to decide whether the deleted/moved structure statement is responsible for the overall method depth. This will be implemented for the next release of our tool.

Chawathe *et al.* assume that two input trees to compare are mostly similar. This heuristic is suitable for source code changes, because subsequent versions of classes seldom exhibit big differences. That means that the edit script may not always be minimal. We are currently investigating how strong the impact of large source changes on the edit script is and how the algorithm can be improved to keep the edit script minimal.

We also plan to provide benchmarks to validate the execution efficiency as well as the accuracy of the implemented algorithm.

5. Case Study

We pursued a case study to demonstrate how the change significance analysis can help in understanding the evolution of source code. The intention of the case study is rather to highlight the applicability of our taxonomy of source code changes than its complete validation. In particular we address the following two questions:

- To what extent are lines added/removed taken from the CVS log indicators for the significance of the applied changes?
- Do the significance levels of change coupled files behave similarly?

Since the current implementation of CHANGEDISTILLER works with CVS and Java, we chose the Java project ArgoUML,[5] an open source UML modelling tool consisting of about 1,400 Java classes.

To answer the above questions, we take a group of classes having a high number of revisions as well as a high change coupling. The four classes *FigClass* with 201 revisions, *FigComment* with 65 revisions, *FigInterface* with 149 revisions, and *FigPackage* with 136 revisions in package *org.argouml.uml.diagram.static_structure.ui* were 19 times commonly committed. We use these classes in the remainder of the case study.

5.1. Lines Added/Removed as Significance Indicator

The change information provided by CVS are a textual diff (`cvs diff`) between two subsequent revisions f_{n-1} and f_n of a file f, and accordingly the overall lines added and removed, *e.g.*, `lines: +15 -6`. For instance, this change information was used to compute code ownership in [6]. The number of lines added or removed may be an indicator for the significance of the changes between two subsequent revisions, meaning that the more lines were added, and removed respectively, the more was changed in the source code. One of the drawbacks of this hypothesis is the high rating of text structure changes, such as indentation changes or the rearrangement of methods and attributes. To investigate this hypothesis, we discuss Figures 1 and 2.

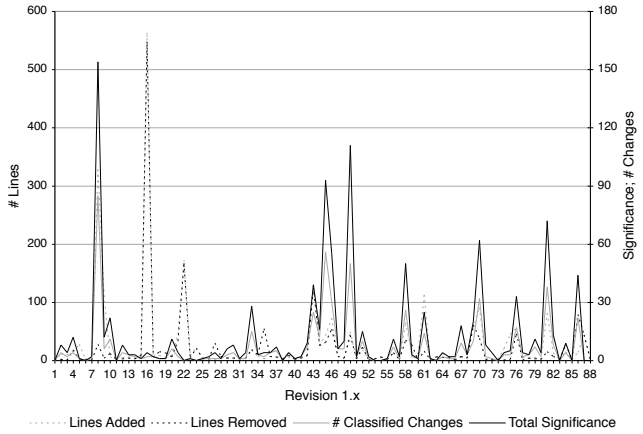

Figure 1. Change significance history of FigPackage

The class (file) FigPackage has 88 (1.1–1.88) revisions. In Figure 1 for each revision (x-axis) the number of classified changes, the total significance, and lines added/removed are plotted. The total significance denotes the sum of all significance levels in this and the following figures. The scale on the left hand side corresponds to lines added/removed; the one on the right hand side to the number of classified changes and the total significance. Examining the trend of the total significance, three major changes stand out: revisions 1.8, 1.45, and 1.49. We see, that the absolute sum of lines added and removed can be an indicator for the significance. This observation is definitely false for the revisions 1.8, 1.16, 1.22, and 1.43. Particularly for revision 1.16, where the total significance is vanishing small compared to the number of lines added and removed. The diff between the revisions 1.15 and 1.16 shows that many text indentation changes were applied, explaining the huge gap between the total significance and the lines added/removed.

For the change history of the class FigPackage, we conclude that the number of lines added and removed do not indicate the significance of the changes.

In Figure 2 the change history of class FigComment with 65 revisions (1.1–1.65) is depicted. The diagram in Figure 2 has the same structure as Figure 1. The change history of FigComment has an interesting development towards the end of the observed revision period. The total significance

[5] http://argouml.tigris.org/

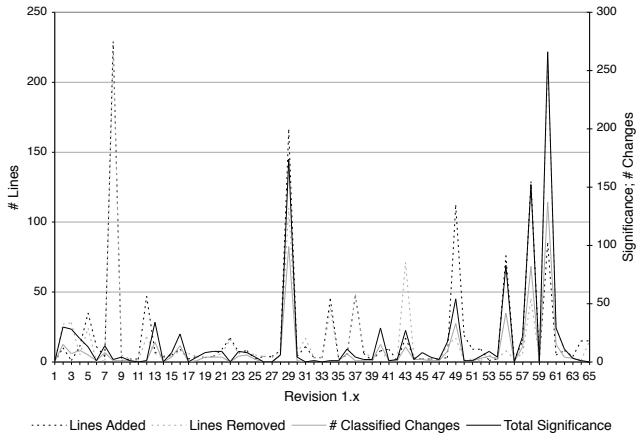

Figure 2. Change significance history of FigComment

from revision 1.49 to 1.60 are increasing (with interrupts). The trend of the absolute sum of lines added and removed has a similar developing in this revision period—at least after 1.49. For this small period we observe, that a relation between the total significance and lines added/removed occurs again. On the other hand, we can also find the situation where no relation at all appears. Noteworthy are revision 1.8 and the shift in directions between 1.12 and 1.13 (decreasing lines added/removed; increasing total significance).

We made the same observation for the other two classes FigClass and FigInterface. We therefore conclude that lines added and removed are insufficient indicators for the significance of changes.

5.2. Significance of Change Couplings

Change coupled files were committed together, *i.e.*, build a transaction, and build a change coupling group [4]. Our release history database (RHDB) approach [3], calculates the change coupling groups with the data provided by CVS logs. Change coupling groups are weighted with the number of transaction in which they occur. To what extent they were changed—or whether they were changed at all—is not considered in this calculation.

Investigating change couplings can help detecting hot spots in the design of a software system [5]. Filtering uninteresting change coupling groups, as we have presented in [4], reduces the number of change couplings to investigate. In this case study we additionally investigate if change coupled files have similar total significances, meaning if they were similarly changed. With the rating of change coupling groups according to the total significance, we are able to further reduce this set—change coupled files having a high total significance may imply a stronger relationship; either dependencies or similar code structure (*e.g.*, code clones).

The four Java classes FigClass, FigComment, FigInterface, and FigPackage, are a change coupling group with the weight 19, *i.e.*, they were 19 times committed together. For each transaction we show in Figure 3 the total significance of each file. The x-axis gives the transactions of the group with the corresponding revision numbers for each file. For instance, the second common commit concerns the revision (after the commit) 1.49 of FigClass, 1.5 of FigComment, 1.35 of FigInterface, and 1.16 of FigPackage.

The shape of the curves of FigClass and FigInterface are similar. Particularly the total significance values between the eighth and twelfth transaction relate more than the values of FigComment and FigInterface. Taking the total significance for the change coupling, only the common change behavior of FigClass and FigInterface is further investigated for similar functionality which has to be adopted together.

In 10 out of 19 transactions only small changes occurred, such that the actual change coupling reduces to 9 (instead of 19). These results show that the total significance of change coupled files do not behave similarly. We therefore conclude that computing the change coupling based on transactions is insufficient. The presented approach emphasizes actually change coupled source code entities.

6. Related Work

Syntactic differencing. Yang describes an algorithm based on a dynamic programming implementation of the largest common subtree problem [27]. The output of the algorithm are sets of matching and modified abstract syntax tree nodes, but it is not reported what operations transform the first into the second tree. In [15], Maletic and Collard present a language independent approach to detect syntactic differences between source files using an intermediate representation of the source code in XML. The output provided by GNU diff is mapped to the XML representation to locate changed entities. Our approach does not use textual differences and is able to detect move changes. Raghavan *et al.* implemented Dex [20] a tool to extract changes between C source files. They use change information provided by patch files, to locate the changed parts in a source files. These parts are fed into their tree differencing algorithm reporting edit operations. Dex can be used with our taxonomy to classify source code changes in C programs. Recently, Xing and Stroulia presented their UMLDiff tool in [26]. UMLDiff tracks changes on the interface (logical design) of classes. In contrast to our work, they are able to track entity moves among different classes. However, their approach focuses on the interface level, whereas our approach additionally considers changes inside the method

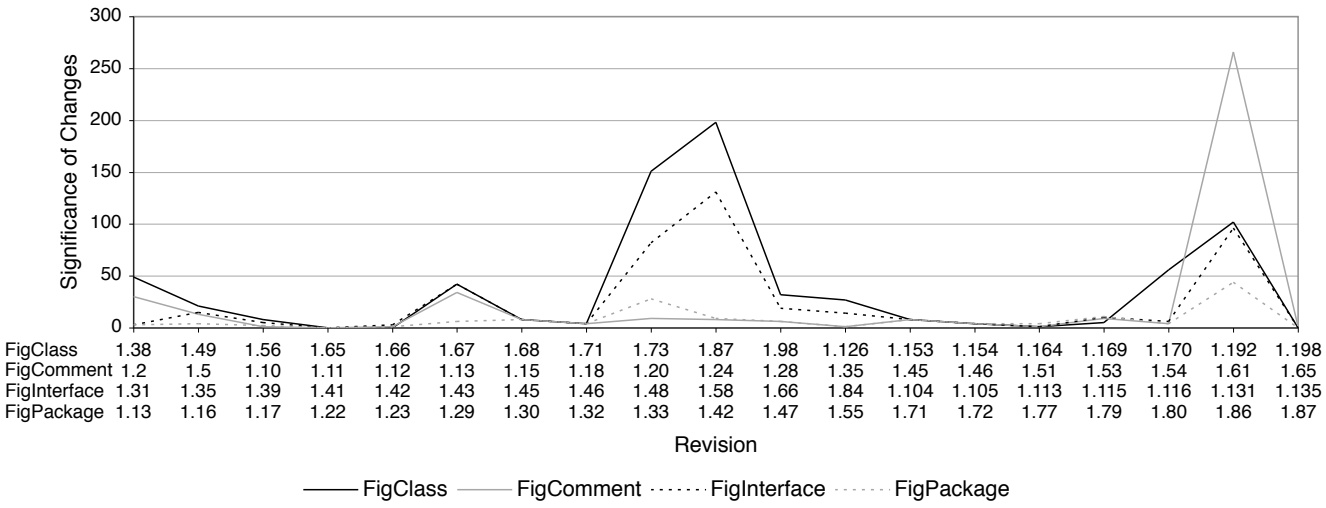

Figure 3. Change significance of each file in the change coupling group per transaction

body. A similar work was done by Tu and Godfrey [23]. They used their BEAGLE tool to detect structural evolution of software systems. With origin analysis BEAGLE detects old functions as the "origin" of new ones based on software metrics and code clone detection. Origin analysis was also used to detect merging and splitting [7] and method renaming [13].

We implemented the change detection algorithm for hierarchically structured data presented by Chawathe *et al.* [2]. In addition, we used the signature change description provided by Kim *et al.* [12] to guide the corresponding definitions in this paper. In contrast to our work, they neither give the corresponding source code change, nor classify the signature changes.

In [8] Hassan and Holt propose evolutionary code extractors. They discuss the need of such tools as well as guide how to choose the source code extraction granularity.

Semantic differencing. The algorithm presented by Jackson and Ladd reports semantic changes in procedural programs [10]. They analyse the input-output behavior of two procedures to detect changed behavior. Apiwattanapong *et al.* [1] use enhanced control-flow graphs to model semantic behavior of methods of object-oriented programs. Identifying modified and unmodified methods is based on graph isomorphism. Their discussion of the impact of path changes caused by exception handling can be used to extend our classification.

Change analysis and classification. Xing and Stroulia [25] use their UMLDiff to classify interface changes. For each class version they assign a volatility level, *e.g.*, "intense evolution" or "rapidly developing," according to the number of changes occurred. In contrast to their work, we classify individual changes. Śliwerski *et al.* classify changes according to whether they induced a fix [22], *i.e.*, changes that lead to problems. Their Eclipse plugin Hatari [21] extracts and visualizes such changes. With our classification, we may detect frequent fix-inducing change types. Small changes are also investigated by Purushothaman *et al.* in [19]. In a large case study, they found that there is less than a four percent probability that a one-line change will introduce a fault. This result implies that the significance level of a one line change is low, as reflected in our classification.

A taxonomy of approaches to analyse source code repositories for understanding software evolution is given by Kagdi *et al.* in [11].

7. Conclusions and Future Work

Current change history analysis approaches rely on information provided by versioning systems such as CVS that hardly provide any data other than lines added and/or removed. Beyond that, change types and their significance in terms of change impact are required to effectively qualify source code changes over several releases.

We presented a *taxonomy of source code changes* to be used for change coupling analysis. Source code changes are defined according to tree edit operations in the AST to classify the change types with a *significance level* that expresses how strong a change may impact other source code entities and whether a change may be functionality-modifying or functionality-preserving. This classification allows one to assess error-proneness of source code entities, qualify change couplings, or identify programming patterns.

The selected examples from the ArgoUML case study revealed that high numbers of lines added/removed do not

correspond with high significance of changes. Furthermore, our results qualify change couplings such that originally high couplings can be better assessed and in many cases the intensity of the change couplings observed from pure CVS analysis can be dramatically reduced.

Future work will concentrate on fine-tuning our Eclipse plugin CHANGEDISTILLER and investigate the change behavior of large software systems.

Acknowledgment This work was partially supported by the Swiss National Science Foundation (SNF) and the Hasler Stiftung Switzerland. We thank the anonymous reviewers for their extremely careful reading of the paper, which clarified several issues and improved the presentation. We thank Martin Pinzger for his valuable feedbacks.

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. 19th Int'l Conf. Automated Software Eng.*, pages 2–13, Nov. 2004.

[2] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pages 493–504, June 1996.

[3] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. 19th Int'l Conf. Software Maintenance*, pages 23–32, Sept. 2003.

[4] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *Proc. 5th Int'l Workshop Source Code Analysis and Manipulation*, pages 66–74, Sept. 2005.

[5] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Proc. 6th Int'l Workshop Principles of Software Evolution*, pages 13–23, Sept. 2003.

[6] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proc. 8th Int'l Workshop Principles of Software Evolution*, pages 113–122, Sept. 2005.

[7] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Software Eng.*, 31(2):166–181, Feb. 2005.

[8] A. E. Hassan and R. C. Holt. Studying the evolution of software systems using evolutionary code extractors. In *Proc. 7th Int'l Workshop Principles of Software Evolution*, pages 76–81, Sept. 2004.

[9] S. B. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Programming Languages and Systems*, 12(1):35–46, Jan. 1990.

[10] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proc. Int'l Conf. Software Maintenance*, pages 243–252, Sept. 1994.

[11] H. Kagdi, M. L. Collard, and J. I. Maletic. Towards a taxonomy of approaches for mining of source code repositories. In *Proc. Int'l Workshop Mining Software Repositories*, pages 1–5, May 2005.

[12] S. Kim, J. E. James Whitehead, and J. Bevan. Analysis of signature change patterns. In *Proc. Int'l Workshop Mining Software Repositories*, pages 1–5, May 2005.

[13] S. Kim, K. Pan, and J. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *Proc. 12th Working Conf. Reverse Eng.*, pages 143–152, Nov. 2005.

[14] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[15] J. I. Maletic and M. L. Collard. Supporting source code difference analysis. In *Proc. 20th Int'l Conf. Software Maintenance*, pages 210–219, Sept. 2004.

[16] T. J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, Dec. 1976.

[17] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proc. Int'l Workshop Mining Software Repositories*, pages 1–5, May 2005.

[18] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proc. ACM Symposium Software Visualization*, pages 67–75, May, 2005.

[19] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Software Eng.*, 31(6):511–526, June 2005.

[20] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code base. In *Proc. 20th Int'l Conf. Software Maintenance*, pages 188–197, Sept. 2004.

[21] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: Raising risk awareness. In *Proc. 10th European Software Eng. Conf. and 13th ACM SIGSOFT Symposium Foundations Software Eng.*, pages 107–110, Sept. 2005.

[22] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int'l Workshop Mining Software Repositories*, pages 24–28, May 2005.

[23] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *Proc. 10th Int'l Workshop Program Comprehension*, pages 127–136, June 2002.

[24] G. Valiente. *Algorithms on Trees and Graphs*. Springer, Berlin, Germany, 2002.

[25] Z. Xing and E. Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Trans. Software Eng.*, 31(10):850–868, Oct. 2005.

[26] Z. Xing and E. Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proc. 20th Int'l Conf. Automated Software Eng.*, pages 54–65, Nov. 2005.

[27] W. Yang. Identifying syntactic differences between two programs. *Journal Software–Practice and Experience*, 21(7):739–755, July 1991.

[28] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng.*, 31(6):429–445, June 2005.

A. Overview of Change Type Classification

Change Type	*Operation*	*Significance Level*
Body-Part		
Additional Functionality*	$\langle$ INS$((l(M), n(M)), C, k)$, INS$((l(P(M)), \{\}), M, 4)\rangle$	low
Additional Object State*	$\langle$ INS$((l(A), n(A)), C, k)$ + INS$((l(T(A)), v(T(A))), A, 1)\rangle$	low
Condition Expression Change	UPD$(L, v(CE_{new}(L)))$; UPD$(CS, v(CE_{new}(CS)))$	medium
Decreasing Statement Delete	DEL(s), $\delta_{old}(M) > \delta_{new}(M)$	high
Decreasing Statement Parent Change	MOV(s, y, k), $p_{old}(s) \neq p_{new}(s)$, $\delta_{old}(M) > \delta_{new}(M)$	high
Else-Part Insert	INS$((l(EP), \{\}), CS, 1)$	medium
Else-Part Delete	DEL(EP)	medium
Increasing Statement Insert	INS$((l(s), v(s)), y, k)$, $\delta_{old}(M) < \delta_{new}(M)$	high
Increasing Statement Parent Change	MOV(s, y, k), $p_{old}(s) \neq p_{new}(s)$, $\delta_{old}(M) < \delta_{new}(M)$	high
Removed Functionality	DEL(M)	crucial
Removed Object State	DEL(A)	crucial
Statement Delete	DEL(s)	medium
Statement Insert	INS$((l(s), v(s)), y, k)$	medium
Statement Ordering Change	MOV$(s, p(s), k)$	low
Statement Parent Change	MOV(s, y, k), $p_{old}(s) \neq p_{new}(s)$	medium
Statement Update*	UPD(s, val)	low
Declaration-Part		
Class Renaming*	UPD$(C, v(n_{new}(C)))$	high
Decreasing Accessibility Change[1]	INS$((l(\mu_A), v(\mu_A)), y, 1)$; DEL$(\mu_A)$; UPD$(\mu_A, val)$	crucial
Attribute Type Change	UPD$(T(A), v(T_{new}(A)))$	crucial
Attribute Renaming*	UPD$(n(A), v(n_{new}(A)))$	high
Final Modifier Insert	INS$((l(\mu_F), v(\mu_F)), y, 2)$, $y \in \{C, M, A\}$	crucial
Final Modifier Delete	DEL$(\mu_F(x))$, $x \in \{C, M, A\}$	low
Increasing Accessibility Change[1]	INS$((l(\mu_A), v(\mu_A)), y, 1)$; DEL$(\mu_A)$; UPD$(\mu_A, val)$	medium
Method Renaming*	UPD$(M, v(n_{new}(M)))$	high
Parameter Delete	DEL(ρ)	crucial
Parameter Insert	$\langle$ INS$((l(\rho), v(n(\rho))), P(M), k)$,	crucial
	INS$((l(T(\rho)), v(T(\rho))), \rho, 1)\rangle$	crucial
Parameter Ordering Change	MOV$(\rho, P(M), k)$	crucial
Parameter Type Change	UPD$(T(\rho), v(T_{new}(\rho)))$	crucial
Parameter Renaming*	UPD$(\rho, v(n_{new}(\rho)))$	medium
Parent Class Delete	DEL(T), $T \in p_{Cold}(C)$	crucial
Parent Class Insert	INS$((l(T), v(T)), p_C(C), k)$	crucial
Parent Class Update	UPD$(T, v(T_{new}))$, $T \in p_{Cold}(C)$	crucial
Return Type Delete	DEL$(T(M))$	crucial
Return Type Insert	INS$((l(T(M)), v(T(M))), M, 3)$, $T_{old}(M) = \{\}$	crucial
Return Type Update	UPD$(T(M), v(T_{new}(M)))$	crucial

*Functionality-Preserving, all others -Modifying

[1]possible values for $v(\mu_A)$, val, and $v_{old}(\mu_A)$ see Section 3.2