Discovering patterns of change types

Fluri, B; Giger, E; Gall, H C
Discovering patterns of change types

Abstract

The reasons why software is changed are manyfold; new features are added, bugs have to be fixed, or the consistency of coding rules has to be re-established. Since there are many types of source code changes we want to explore whether they appear frequently together in time and whether they describe specific development activities. We describe a semi-automated approach to discover patterns of such change types using agglomerative hierarchical clustering. We extracted source code changes of one commercial and two open-source software systems and applied the clustering. We found that change type patterns do describe development activities and affect the control flow, the exception flow, or change the API.
Discovering Patterns of Change Types

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

Abstract

The reasons why software is changed are manyfold; new features are added, bugs have to be fixed, or the consistency of coding rules has to be re-established. Since there are many types of source code changes we want to explore whether they appear frequently together in time and whether they describe specific development activities. We describe a semi-automated approach to discover patterns of such change types using agglomerative hierarchical clustering. We extracted source code changes of one commercial and two open-source software systems and applied the clustering. We found that change type patterns do describe development activities and affect the control flow, the exception flow, or change the API.

1. Introduction

Source code changes are rarely applied separately. In most cases a change induces other changes. For instance, a parameter renaming impacts all statements that access the parameter inside the method body. The statements have to be adapted to the preceding change. Kim et al. extracted such refactorings and their corresponding changes in the method body to represent change patterns as rules [5, 6]. Other change pattern investigations were conducted to reveal error patterns [7, 8] or aspect patterns [1]. Because of their interesting findings we explore whether change types appear frequently together, and whether they describe special development activities. Extracting such patterns allows us then to show that coding guidelines are not followed by developers, that newcomers might not be appropriately trained, or that coding guidelines are frequently changed. This knowledge enables a consistency check of changes as well as a feedback during development whether coding guidelines have been followed correctly.

We leverage data provided by our software evolution analysis tools EVOLIZER and CHANGEDISTILLER [3]. Both tools are plugins for the Eclipse IDE.1 EVOLIZER offers facilities to extract historical information from version control repositories of software projects and to store them in a database. CHANGEDISTILLER uses this information to extract fine-grained source code changes between subsequent revisions of Java classes. The abstraction used by CHANGEDISTILLER focuses on changes in body and declaration parts of attributes, classes, as well as methods, and stops at the statement level. For each of the possible changes, we defined and classified a change type [2]. For instance, with CHANGEDISTILLER we can extract that a method invocation statement was moved inside the else-part of an if-statement or that a parameter was added to a method declaration. The corresponding change type is then a statement parent change.

In this paper we present a semi-automated approach to discover patterns of change types in the evolution of a software system using agglomerative hierarchical clustering. The approach is evaluated by an experimental study with one commercial and two open-source software systems. The results show that change patterns either affect the control flow, the exception flow, or the API.

2. Extraction of Change Type Patterns

In this section we describe the data that we extract and the process to discover patterns of change types.

2.1. Clustering for Pattern Extraction

We explain the clustering technique with a sample change history. We imitate the way clustering is explained from [10]. Assume a small program comprises four methods, \( m_a, ..., m_d \). Each method was changed twice, so that each method has two versions, e.g., \( m_a^1, m_a^2 \). Between the two subsequent versions of a method, \( m_a^1 \) and \( m_a^2 \), we extract the change types with CHANGEDISTILLER and attach

1http://www.eclipse.org/
the change types to $m^3_a$, i.e., the change types that transform $m^2_a$ to $m^3_a$ are attached to $m^3_a$. We build a matrix where change types denote the rows and method versions the columns. A matrix entry is then the number of occurrences of a change type in a method version. The result of this operation for method versions $m^2_a, ..., m^3_a$ is depicted in Table 1. For instance, method version $m^3_b$ had one return type delete and two statement deletes. That means that method $m_b$ no longer returns an object and that two statements were deleted from its body.

$$
\begin{array}{c|cccc}
\text{change type} & m^1_a & m^2_a & m^3_a & m^4_a \\
\hline
\text{parameter delete} & 2 & 0 & 1 & 0 \\
\text{return type delete} & 0 & 1 & 0 & 0 \\
\text{statement delete} & 1 & 2 & 0 & 1 \\
\text{statement insert} & 0 & 0 & 3 & 1 \\
\text{statement update} & 4 & 0 & 1 & 2 \\
\end{array}
$$

Table 1. Example matrix used for clustering.

The rows of the matrix are used as feature vectors for clustering. We apply agglomerative hierarchical clustering on these feature vectors. The distance between the vectors is calculated with the Cosine distance measure. Agglomerative hierarchical clustering starts with each item in a single cluster and then iteratively links clusters together to a new cluster until only one cluster remains. For the linking we use the average linkage function. The result is a dendrogram; a tree like structure that depicts the nested clusters [4].

![Dendrogram](image)

**Figure 1. Example dendrogram with cutoff line**

Having a dendrogram as shown in Figure 1, we can specify a cutoff value, meaning that we cut the dendrogram by a horizontal line at the height of the cutoff value. The dendrogram is then read from bottom to the line and the yielding complete branches are then the resulting subclusters, i.e., in our case change type clusters.

Assume we specify the cutoff value to 0.6 in Figure 1, the resulting change type clusters are: (1) \{return type delete, statement delete\}, (2) \{parameter delete, statement update\}, and (3) \{statement insert\}. The third cluster is comprised of only one change type because the connection from statement insert to the \{parameter delete, statement update\} cluster is above the 0.6 cutoff.

**2.2. Analysis of Change Type Patterns**

For the analysis of change type patterns we perform several clustering passes. The first pass takes the change types during the entire history of a software system into account. We take all method versions that are extracted by CHANGEDISTILLER, create the matrix and perform the clustering. For the second and all further passes we divide the change history of a software system into yearly quarters and build a matrix for each of these quarters. For the remainder of this chapter we distinguish these two kinds of cluster passes as full cluster and quarter cluster.

After building the full and the quarter clusters, we make a two step analysis. First, we analyze the change type patterns of the full cluster. We define the patterns that we reveal from the full cluster as global change type patterns as they can be found when analyzing the entire history. Second, we analyze and compare the change type patterns of the quarter clusters among each other and with those of the full cluster. As the data extracted for calculating the full cluster contains a large amount of method versions, deviations of global change type patterns applied during the history of a software system disappear. Because of that we aim at finding local change type patterns within the quarter clusters—patterns that deviate from the global patterns.

**3. Experimental Results**

We have chosen two open-source and one commercial system for our experiment: (1) jEdit (Java text editor: 6,754 revisions & 88,932 changes), (2) JFreeChart (Java chart library: 4,675 revisions & 23,678 changes), and (3) Webframework (a commercial Java framework for web applications: 19,501 revisions & 116,994 changes).

For all three case studies, we created the full change type clusters and the quarter change type clusters. Based on the evaluation with a test set we define that the cutoff value is set to 0.6. Section 3.1, Section 3.2, and Section 3.3 describe the patterns we found.²

**3.1. Global patterns in the Webframework**

Constructor invocation changes. \{super constructor invocation statement delete, constructor invocation statement insert\} Super constructors are invoked when the super class provides constructors that are not overridden by the class.

²The curly brackets after a pattern name denote the change types introduced in [2] that have to be applied for that specific pattern.
This change pattern is applied, when the super constructor is overridden in the class. Then, the super constructor invocation has to be replaced by a constructor invocation.

Return type based method renaming.  \{return type change, method renaming\} In the Webframework, developers tend to change the method name when the return type of the method changes. This is reasonable because the return type carries important semantics.

Introducing prefixed parameter names.  \{assignment statement update, parameter renaming\} The change type update assignment occurs often and in various conjunction with other change types in the histories of software systems. It is, therefore, remarkable that it is clustered with the change type parameter renaming—a rather infrequent change type.

To find the reason for this change type pattern, we compared the full cluster with the quarter clusters. In six out of 13 quarter cluster of the Webframework update assignment and parameter renaming are clustered, but only twice below the cutoff of 0.6—in the first quarter of 2005 and in the third quarter of 2006. We inspected the occurrences of the change types manually and found that in 285 out of 493 parameter renaming the parameter name gets an undefined article as a prefix; for instance root becomes aRoot. This occurred mostly in setters and constructors where assignments initialize field values.

Introducing single exit.  \{return statement insert, return statement delete\} An inspection of the method versions in which these change types were applied revealed that there was a shift from the multiple exits to the single exit principle.

By examining the quarter clusters, we found that the developers heavily introduced this single exit principle in the third and fourth quarter of 2005. We were wondering whether they kept this principle during the years 2006 and 2007. The answer is twofold. First, they kept introducing the single exit principle. Second, although they were already aware of the validity of the principle in the Webframework, they also made changes violating this principle. A further investigation showed that they sometimes corrected these violating changes within 100 days, but hardly for all affected methods.

Change existing exception handling.  \{try statement insert/delete, catch clause insert/delete, throw statement insert/delete\} Inserting and deleting try statements mostly effect inserting and deleting catch clauses as well, but they do not always have to be applied together. For instance, an additional catch clause can be added to a try statement after the try statement was inserted. Therefore, try statement insert/delete are well clustered with catch clause insert/delete, but not perfectly. The change types throw statement delete and insert is clustered to insert/delete of try-catch-statements. The reasons for that are twofold: (1) Exceptions are not only handled with re-throwing them but caught. (2) During the development of the framework, the developers decided to make a shift from or towards rethrowing exceptions. Both have the effect that no try-catch statement is changed along with a corresponding throw statement over the history of the Webframework.

After an inspection of the changes in the source code, we found that there was not any shift in the exception handling, but the developers rather used two different mechanisms to handle exceptions: (1) They throw a new Webframework defined exception or (2) they log the exception. These are common practices to handle exceptions.

3.2. Local patterns in the Webframework

Swap control flow order.  \{control structure condition expression change, parent change of any statement\} In the third and fourth quarter cluster of 2005, the statement parent changes are grouped together with control structure condition expression change. Compared to the full cluster, the parent changes are indeed clustered together but on a high level in the dendrogram but not with control structure condition expression change.

An inspection of the changes in the source code revealed that the parent change pattern mostly denotes swapping the then and else-part of an if-statement. That leads to the following changes: The if-condition must be negated (control structure condition change), the statements in the then-part are moved to the else-part (statement parent change) and vice versa. A reason for this change pattern is the convention that the default control flow goes via the then-part. Although such control flow order changes appear in the second half year of 2005 concentrically, they can be sporadically found over the history of the Webframework.

Merging control flow.  \{if-statement parent change, control structure condition expression change, if-statement delete, parent change of any statement\} In the first and second quarter of 2007, the developers made another kind of control flow change. The change type pattern \{if-statement parent change, control structure condition expression change\} appears in the first and second quarter cluster of 2007. We inspected the changes in the source code and found out that a certain amount of nested if-statements were merged.

Remove superfluous parameter.  \{parameter delete, update of any statement\} In the third quarter cluster of 2005,
the change types parameter delete and return statement update are grouped. A considerable amount of XML handling functionality is provided via delegators in the Webframework. During a short period in the third quarter, the developers removed the session parameter from various methods and adapted the return statements accordingly.

3.3. Change type patterns in jEdit and JFreeChart

We only found two interesting change type patterns in JFreeChart and jEdit. First, in JFreeChart, the developers use exception flow to check method preconditions. Basically, they check the parameter values for certain conditions. As a result, if-statement and throw statement inserts are grouped in the full cluster.

Second, in jEdit we found the change type pattern \{control structure condition expression change, variable declaration statement update\} during the first quarter of 2005. This pattern did not appear in the full cluster. The developers removed direct field accesses with getters.

4. Related Work

Work related to our change distilling algorithm (i.e., other source code change extraction algorithms) already has been discussed extensively in [3].

Maqbool and Babri give a survey over the current state of the art of hierarchical clustering in software engineering and discuss how to evaluate results obtained with clustering [9].

By mining the revision history of a software Breu and Zimmermann extract method call change patterns and identify cross-cutting changes [1]. In contrast, we focus on the type of change solely without giving them domain specific semantics. We can complement their findings by distinguishing the introduction of aspects with further change types.

BugMem [7] a tool developed by Kim et al. mines bug fixes from software repositories to reconstruct pairs of bug and fix patterns. This is similar in spirit to our work. We focus on general change type patterns but do not limit on bug fix patterns.

Kim et al. presented an approach to automatically infer likely changes at or above the method level [5, 6]. The extracted refactoring patterns are described with rules for method body and declaration changes. The change type patterns on the API level are similar to the patterns found by Kim et al.

5. Conclusions and Future Work

In this paper we introduced an approach to explore whether change types appear frequently and commonly, and whether they describe special development activities. The idea was to use agglomerative hierarchical clustering of change types to discover such patterns of change types.

We performed experiments on one commercial and two open-source software systems. The results show that change patterns capture the semantics of special development activities and either affect the control flow, the exception flow, or the API. Furthermore, our approach can discover those control flow changes that are due to particular source code cleanup activities, that exception flow is used differently in system parts, and that API convention changes are spread over many releases. For that we had to distinguish between changes over the complete history and such that happen in certain periods, i.e., global patterns and local patterns.

In future we plan to perform additional case studies and create a catalogue of change type patterns. We also aim at integrating automated search for change type patterns into IDEs to support developers in their daily work.

Acknowledgments. This work was supported by the Hasler Foundation as part of the ProMedServices project.

References