Studying Versioning Information to Understand Inheritance Hierarchy Changes

Filip Van Rysselberghe and Serge Demeyer
Lab On Re-Engineering
University Of Antwerp
Middelheimlaan 1
filip.vanrysselberghe@ua.ac.be

Abstract

With the widespread adoption of object-oriented programming, changing the inheritance hierarchy became an inherent part of today's software maintenance activities. Unfortunately, little is known about the "state-of-the-practice" with respect to changing an application's inheritance hierarchy, and consequently we do not know how the change process can be improved. In this paper, we report on a study of the hierarchy changes stored in a versioning system to explore the answers to three research questions: (1) why are hierarchy changes made? (2) what kind of hierarchy changes are made? (3) what is the impact of these changes? Based on the results of this study, we formulate 7 hypotheses which should be investigated further to make conclusive interpretations on how hierarchy changes fit in the actual change process.

1. Introduction

The central role of inheritance hierarchies in object-oriented software systems, combined with the knowledge that software systems need to change in order to remain successful [4], lead to the introduction of a number of techniques to (automatically) restructure these hierarchies [1, 2, 3]. A subset of these techniques, i.e. refactorings, has even been included in modern integrated development environments like Eclipse. Despite the introduction of these techniques it is unknown how they relate to the (manual) hierarchy changes that are typically made by developers. In fact, except for anecdotal knowledge (e.g. Meyer [5]) very little is known about how and why developers make changes to the inheritance hierarchy.

This paper therefore describes a study of past hierarchy changes. The study focusses on hierarchy changes that replace the parent of a class by another. The corresponding model (Figure 1) shows that a hierarchy change consists of three components: a center class, an old parent and a new parent class. All changes that satisfy the model are considered in the study. Inner classes, anonymous classes or other language specific types of classes are not considered since they may be too specific for one programming language.

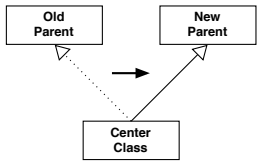

Figure 1. Hierarchy changes replace the parent of a center class with a new parent.

The study is driven by three research questions: (1) why are hierarchy changes made? (2) what kind of hierarchy changes are made? (3) what is the impact of these changes? In order to answer these questions past hierarchy changes are reconstructed from the information stored in a versioning system. As one of the first studies in the field, the primary goal of the study is to explore the possible answers and consequently formulate initial hypotheses.

2. Hierarchy Change Reconstruction

Most hierarchy changes record the history of a software system as a sequence of text differences. These text differences are created each time a developer commits a new version. Identification of hierarchy changes can then be accomplished by locating those differences which alter the parent definition of a class. Since the parent definition in most object-oriented programming languages is made by a designated keyword, e.g. "extends" for Java or "subclass" for Smalltalk, these changes can be identified by means of regular expression matching.

Identification of hierarchy changes is therefore achieved by identifying all text differences stored in the versioning

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

system which contain the inheritance keyword for that language. However this also results in the identification of class introductions, formatting changes etc. Hence an additional test verifies whether the identifier that identifies the parent class, is changed as well. For Java we thus select all changes which (a) contain the ”extends”-keyword and (b) in which the identifier following this keyword differs.

3. Question Verification

RQ1: Why are hierarchy changes made?

Each hierarchy change is manually classified as intended to affect or not affect user-experienced functionality by analyzing the associated log message. When the log message for example states that a new feature is added, the user interface changed or a bug is fixed, then the change is classified as *affecting*. Internal restructuring or performance improvements are classified as *not-affecting*. An additional category contains all changes that can not be classified e.g. because the log message is too abstract.

RQ2: What kind of hierarchy changes are made?

To understand what kind of hierarchy changes that are made, we classify each hierarchy change according to the following properties which characterize hierarchy refactorings.

- *Parent ownership:* For both the old and new parent we determine whether they belong to the current project(=self) or to another (=external) project. Hence four types of changes exist: *self-to-self*, *external-to-external*, *self-to-external* and *external-to-self change*.
- *Parent creation*: Verify whether the new parent did exist before the hierarchy change was carried out.
- *Relation between the replaced parents*: Establish the relation between the old and new parent. Following types of relationships can occur: *new parent is child of old*, *new parent is parent of old*, *new parent is descendant of old*, *new parent is ancestor of old*, *new parent is sibling of old*, *other*, *unknown*.
- *Fields added or removed*: When inheritance is replaced by delegation or vice versa, a delegating field has to be added or removed. Hence we verify whether such a delegating field was added or removed from the center class by the hierarchy change.
- *Code similarity*: Extracting duplicated functionality into a common parent is a typical hierarchy restructuring [1, 2, 5]. Hence we determine the degree of similarity between the code removed from the center class and the code added to the new parent. They are considered *dissimilar* when no textually and/or semantically similar lines exist, *vaguely similar* when they share a number of similar lines, although a reasonable number of lines differ, and *very similar* when both fragments are identical, except for minor differences.

RQ3: What is the impact of hierarchy changes?

A hierarchy change can require additional changes to classes that are coupled with the center class when the change alters the center class's interface. The new parent may for example define less methods than the old parent. Hence we calculate the number of methods that are defined in the old parent, but not in the new parent (*#removed*).

A hierarchy change may also cause the center class to behave differently since method definitions might differ from the old to the new parent class. Hence the number of methods either one of the parents overrides from the common parent (*#overridden*) and the number of methods which have an identical signature although they are not defined in a common parent (*#identical*), is determined.

4. Cases

Two successful, open source target systems with different characteristics (see Table 1), i.e. jEdit and ArgoUML, are selected. The selected development periods are chosen because the system size decreases at least once during these periods which may indicate a restructuring.

Table 1. Characteristics of the cases.

	jEdit[1] v3.2 - v4.2	ArgoUML[2] v0.12 - v0.16
#classes (last version)	544	1234
#methods (last version)	4088	8544
#days of development	1090 days	641
#transactions	1142	3015

5. Results

There is a tendency within jEdit to apply hierarchy changes as (part of) changes to affect user-experienced functionality (Table 2). However, since more than half of the hierarchy changes in ArgoUML could not be classified, it is impossible to generalize this observation.

Most hierarchy changes replace the old parent with an already existing parent. In jEdit 12 hierarchy changes replace the old parent with an existing class, and 3 do not. For ArgoUML the numbers are respectively 52 and 22.

Table 2. Classification of hierarchy changes based on their intention to affect user-experienced functionality.

	affecting	not-affecting	unknown
jEdit	0	11	4
ArgoUML	10	25	39

The outcomes for the *Parent ownership* property are summarized in Table 3. The replacement of an externally-defined parent with a self-defined parent forms in both cases the second largest category of hierarchy changes. Additionally, the number of hierarchy changes that replace a self-defined parent by an externally-defined parent is low. No tendency is observed for the remaining types.

Table 3. The number of hierarchy changes which exist in each category when only parent ownership is considered.

hierarchy change	# for jEdit	# for ArgoUML
self-to-self	3	52
external-to-self	4	12
self-to-external	1	6
external-to-external	7	4

The number of hierarchy changes in each category of similarity is quite different in both projects (Table 4). Hence no generic tendency can be derived.

Table 4. Similarity between the client and new parent for both jEdit and ArgoUML

	dissimilar	vaguely similar	very similar
jEdit	12	0	1
ArgoUML	24	12	28

Classification of the hierarchy changes according to the relation between their parents results in Figure 2. It shows that the largest category of hierarchy changes is the category of changes that replace the old parent with a child of the old parent. The portion of other types of hierarchy changes is largely the same for both cases. Only exceptions are the ”other” and ”new parent is descendant of old” changes.

One hierarchy change in jEdit removes a delegating field, while two other introduce such a field. ArgoUML on the other hand, only contains a hierarchy change that removes a delegating field. Hence few hierarchy change substitute inheritance and delegation.

Few hierarchy changes have a negative impact on the

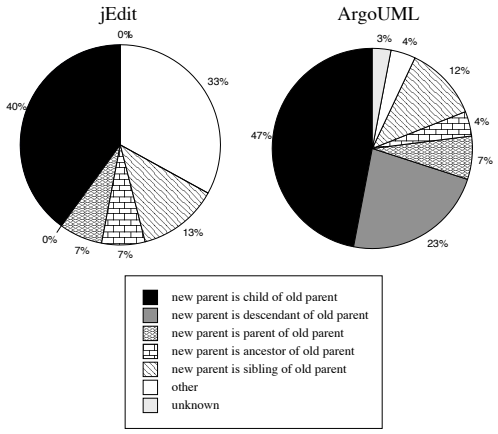

Figure 2. The distribution of hierarchy changes when they are classified based on the relation between the parents they replace.

interface of the center class. Table 5 shows that both ArgoUML and jEdit contain more hierarchy changes that do not remove a method from the interface than changes that do.

Table 5. Number of hierarchy changes which remove or override at least one method.

		Yes	No
jEdit	$\#removed > 0$	7	8
	$\#overridden > 0$	12	3
	$\#identical > 0$	4	11
ArgoUML	$\#removed > 0$	3	34
	$\#overridden > 0$	30	7
	$\#identical > 0$	0	37

On the other hand, hierarchy changes are likely to affect system behavior since the majority of hierarchy changes overrides at least one previously defined method (see Table 5). Few hierarchy changes replace a method with an identical method (identical in signature), which is not declared by a common part.

6. Discussion of the results

Concerning the kind of hierarchy changes, we observed that the number of hierarchy changes which replace an externally-defined parent with a self-defined parent is high. From a refactoring perspective this could mean that developers try to reduce the coupling with external projects by introducing a self-defined wrapping layer. An alternative

explanation is that developers rather redefine functionality than reuse it.

Hypothesis 1 There is a tendency to replace externally-defined parents with self-defined parents to reduce coupling with external projects.

The category of hierarchy changes that replace an old parent by one of its children forms in both cases the largest category of hierarchy changes. Hence hierarchy changes are often used to introduce an additional abstraction between the old parent and the center class. We for example observed how similar functionality, which was spread over a number of classes, was centralized in a new class.

Hypothesis 2 Hierarchy changes are likely to insert an additional abstraction between the old parent and the center class.

The results show that few hierarchy changes introduce a new class. Instead the new parent class is often already part of the system.

Hypothesis 3 Parent classes are replaced by classes which are already present in the system.

This hypothesis is especially important in combination with the second hypothesis since it may indicate that the required abstraction is usually already present in the project. Analysis of ArgoUML's changes for example showed that first an abstraction was introduced for one class. Afterwards, this abstraction was applied in other places as well.

The results show that only in a limited number of hierarchy changes the new parent and the center class are very similar. This could mean that automatic hierarchy refactorings that are based on the extraction of functionality into a new class (e.g. "extract superclass"-refactoring) are used scarcely. Analysis of the hierarchy changes showed that instead of extracting functionality, developers rather seem to create the new functionality manually.

Hypothesis 4 Hierarchy changes which extract functionality from the center class are rarely used.

In both jEdit and ArgoUML few instances are found of changes that replace inheritance by delegation or vice-versa. This may indicate that such replacement is unlikely to happen in a software system, except for some very specific situations.

Hypothesis 5 Inheritance is only rarely replaced by composition.

Given the answers to the different properties, one would conclude that developers are not likely to use behavior preserving refactorings. Especially the results for the similarity property and the replacement of inheritance by delegation, support this conclusion. Analysis supports the observation that few refactorings were applied since few instances of refactoring were identified.

Hypothesis 6 Developers primarily adapt the inheritance hierarchy manually without the support of systematic refactoring steps.

Since few hierarchy changes reduce the interface of the center class, few additional changes are necessary to adapt coupled client classes to the changed interface. Instead, hierarchy changes are likely to affect the behavior of the system since many hierarchy changes cause a method to be overridden differently.

Hypothesis 7 Hierarchy changes primarily influence the behavior of the rest of the system by overriding methods differently.

7. Conclusion

This paper reports on a study to explore (1) why hierarchy changes are made, (2) what kind of hierarchy changes are made, and (3) what their impact is. A reconstruction technique is introduced that, based on the change information stored in a versioning system, selects hierarchy changes in two open-source systems. Afterwards a number of properties and heuristics are analyzed for each reconstructed hierarchy change. The study results in 7 hypotheses which can improve our understanding of how hierarchy changes fit in the actual change process. However, before this knowledge can be validly used, the hypotheses should be empirically validated. The paper therefore provides a clear starting-point for future research which studies the change process. Furthermore, future research can build on the reconstruction technique and the validation properties used in the study.

References

[1] E. Casais. An incremental class reorganization approach. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 114–132, London, UK, 1992. Springer-Verlag.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] R. E. Johnson and W. F. Opdyke. Refactoring and aggregation. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, pages 264–278, London, UK, 1993. Springer-Verlag.

[4] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. Academic Press, 1985.

[5] B. Meyer. Tools for the new culture: Lessons from the design of the eiffel libraries. *Communications of the ACM*, 3:68–88, September 1990.

0-7695-2950-X/07 $20.00 © 2007 IEEE