Reconstruction of Successful Software Evolution Using Clone Detection

Filip Van Rysselberghe
Lab On Re-Engineering
Universiteit Antwerpen
Middelheimlaan 1, B 2020 Antwerpen
filip.vanrysselberghe@ua.ac.be

Serge Demeyer
Lab On Re-Engineering
Universiteit Antwerpen
Middelheimlaan 1, B 2020 Antwerpen
serge.demeyer@ua.ac.be

Abstract

In modern software engineering, researchers regard a software system as an organic life form that must continue to evolve to remain successful. Unfortunately, little is known about how successful software systems have evolved, and consequently little has been learned from previous experience. In this paper, we demonstrate a heuristic to reconstruct evolution processes of existing software systems by exploiting techniques to detect duplication in large amounts of data. A case study, evaluating various versions of Tomcat using this heuristic, revealed that the removal of duplicated code is a much smaller concern than grouping functionality in classes with one clear responsibility.

1. Orientation

Due to the large cost of software maintenance (about 80% of the total development budget [3]) and the increasing strategic value of software, the study of software evolution has grown into a subject of serious study. Most of this research effort is spent on the construction of new methods and means to construct or adapt software. However, based on the idea that a better insight into the phenomenon leads to improved methods, an alternative approach may be taken by identifying how software evolves [11], which is the approach shared by our evolution research.

However, since the focus of software evolution research lies on the first approach, little is known about how changes affect software systems as a result of which little is learnt from previous mistakes and successes. Consider for instance the principle of software entropy [10], a principle well known by experienced software engineers who take appropriate countermeasures. Unfortunately, little of these countermeasures nor their use, are documented. Hence a study on how successful software systems evolved, is necessary to reach the level of self-improving discipline.

2. Software Palaeontology Heuristic

Our heuristic is comparable with the work of palaeontologists who try to deduce the evolution of life on earth by studying collected fossils, hence the heuristics name. By comparing different releases of existing source code (= the fossil remainders of a software system) and analysing the differences, we reconstruct past evolution processes.

For these comparisons, existing techniques, used to detect duplicated code fragments are used. Due to the considerable research effort spent in the last 20 years on this topic, these techniques can be regarded as scalable techniques for the analysis of software systems[1, 5, 2, 8, 9, 12]. Nevertheless these techniques are used in a completely different way in our research: rather than looking for matches which represent duplicated code, we look for mismatches representing changes in the program.

However, searching for mismatches in the large amount of generated data without the aid of visualisations would be a difficult and error–prone task. Therefore we seek the aid of dotplots in locating such mismatches. Dotplots are a visualisation technique originally developed for investigating similarities in DNA-sequences, but later adopted for analysing code duplication[4].

The core of each dotplot visualisation is formed by a matrix in which the horizontal and vertical axis represent the lines of the compared entities. When two lines of both files match, a dot is placed in the matrix position corresponding

1 Clone detection techniques is another name used to denote such techniques.
with these lines. Figure 1 demonstrates the inner–workings of a dotplot by comparing two files (A and B). Both axes of the matrix are annotated with the line numbers they represent. Since line 1 of file A matches line 1 of file B, a dot is placed in the matrix position (1,1). Line 4 of both files doesn’t match, which is shown by the absence of a dot in matrix position (4,4). Figure 2 demonstrates how dotplot visualisations help with locating coding mismatches between two versions.

In some sense, our technique is similar to the work of Pinzger[14] who searches for predefined patterns to recover a systems architecture, where we search for all occurring patterns. Godfrey[7], on the other hand, uses a technique called “origin analysis”. Although this technique is actually built around a clone detection technique, it is used in a much more narrow context of finding the evolutionary predecessor of an entity in order to make a difference between added components which are really new and those that are derived from entities present in a previous version.

3. Tomcat Case Study

With as aim (1) evaluating our heuristic and (2) exploring the use of refactorings during evolution, we set up an introductory case study in which five versions of Tomcat[15], a free and open–source implementation of Java Servlet and JavaServer Pages technologies developed at the Apache Software Foundation, were compared and visualised using the Duploc[5] clone detection tool. During the analysis of these visualisations, the focus lay on the recognition of the “move method” refactoring and studying its effect on the next release.

3.1. Move Method

Moving methods is often described as the bread and butter of refactoring[6]. As one of the most elementary refactor operations, it aids the reduction of a systems complexity by moving functionality from classes suffering too much behaviour or strong coupling. The “pull up method” refactoring, as a special instance of move method, even helps reducing the amount of duplicated code. Due to this central role, we are sure that the move method refactoring deserves special attention.

Moving a method actually consists of two actions: (1) removing the method from the original class and (2) adding the existing method to the new class. Both actions have a well defined effect on the dotplot which visualises the comparison between the pre– and post–refactoring release.

Figure 2 demonstrated how two, identical versions result in a perfect diagonal since each line of the first version matches with the corresponding line of the second. Removing a method thus breaks this perfect diagonal. Since the method is removed, the comparison no longer shows a one–on–one (a diagonal) match but shows a shift where the removed method used to be. To illustrate this with an example, consider a class where the method on the 41st line is removed. In that case, line 41 of the new version will no longer match line 41 of previous version, but match its 51st line when the removed method was 10 lines long. Visually
the removal of a method between two succeeding releases, is represented by a broken diagonal, with the second part shifted downward as shown in area A of figure 3.

Adding the existing method to its new class has the opposite effect of adding a new match. Since the original method is now part of another class, a match is introduced between both the original class and the class of the new version that now contains the method. Visually, a small diagonal, representing the matching function, is introduced in the dotplot which shows the comparison of the original class with the class of the new version, containing the added function (area B in figure 3).

By combining both patterns, a general pattern that indicates the presence of a move method refactoring between two version can be constructed. Figure 3 for example, shows a move method pattern as found in our case study.

- **Reducing duplicated code**: Since duplicated code increases a program's complexity and thus stimulates software entropy, it should be minimised e.g. by moving methods up in the class hierarchy.
- **Encapsulating functionality**: A system where each class has one clear responsibility is easier to understand and maintain, therefore functionality should be encapsulated in small units with well-defined responsibility.

For each of the move method refactorings present in the Tomcat case study, we evaluated the reason behind it.

An evaluation of the code with the clone detection tool Duploc, revealed that each release contained a rather large amount of duplication. Table 1 tries to give an idea of this amount by listing the number of duplicates longer than 10 effective (without white lines and comments) lines of code. What is more, most of this duplication could quite easily be removed by extracting or moving methods. Due to both factors, we were convinced that many pull up refactorings would be present from release to release to remove this duplication.

Even with this considerable amount of duplication, only a maximum of two pull up method refactorings were found between two adjacent releases. It also became clear that the move method functions present were not used to remove duplication but to come to a set of entities encapsulating one specific functionality like stopping Tomcat or handling HTTP-transfer. The only thing that changed by moving methods from one class to another, were the names of the classes containing the duplicated code fragments.

When duplication in between two releases was removed, it was mostly caused because the matching methods were altered independently of each other, making them less obvious duplicates. For example functions were extended, causing the original match to break up into two smaller matches. From time to time, duplication was also removed because a whole class was thrown away, for example because its functionality was deprecated, or replaced.

The number of methods moved between classes, on the other hand, was considerably higher than the number of pull up refactorings. From release 3.2.4 onto 3.3.1, there were about 7 classes which contained at least one method that was moved to another class, 2 of them contained a method that was pulled up. Just by looking at the names of the different classes, it was clear that they were used to group functionality in entities with one responsibility. Error handling functionality was for example extracted from a more general class and added to a separate error handling class. Accordingly, the main reason for moving classes was grouping functionality.

Based on our case study, we thus conclude that during evolution less effort is spent in removing duplication than
into account such non–functional changes, makes it more difficult to reconstruct a program’s evolution.

Although this is a drawback in the recognition of some refactorings, it helps the detection of others. An example is the recognition of the rename method refactoring, which is only detectable by such fine grained clone detection techniques. Furthermore there are a variety of so–called parameterized clone detection techniques which allow the detection of duplicates, even in the presence of non–functional changes [1, 2, 9, 12].

Bearing these problems in mind, we will extend our case study to other case studies such as Mozilla[13]. Through these additional studies we hope to define other dotplot patterns indicating the presence of a certain change to the software. This way we try to establish a set of patterns accompanied with a usage–frequency, which leads to a set of countermeasures suitable for the development and maintenance of future projects.

### 3.3. Conclusions

Another benefit of the Tomcat case study is that it allowed us to get a clear picture of the problems that may accompany our heuristic. For each of these problems, we tried to find a suitable solution or tried to assess its impact.

One possible problem is the scalability of the visualisation. Where the clone detection techniques which form the basis of the visualisation, are scalable, the visualisation itself is not. It is quite easy to see that it is difficult to overview the whole visualisation with such detail that patterns can be recognised, certainly for large programs.

However in our case study we noticed that it is not important to view the whole visualisation at once. Most evolution information is contained in those classes which contain a recognisable diagonal because they are evolved instances of classes from the previous version. In our case for example, we started with locating such classes, after which we checked whether they contained the dotplot pattern characterising the removal of a method. For each of the classes that did contain such pattern, we searched the same horizontal row for a match pointing to the methods new location. Since all the evolution information lies in the same horizontal or vertical region of the classes containing a recognisable diagonal, it is possible to improve the scalability of the visualisation.

One possible improvement is the use of a kind of summary view which summarises the dotplots lying in the same horizontal or vertical region. Another solution is the use of automatic pattern recognition in the dotplot matrix. The disadvantage of this last approach is that it becomes harder to come up with new patterns since all patterns should be thought of in advance instead of being discovered while browsing a programs visualisation.

A second problem that arose, had to do with the clone detection technique at hand. Since an exact line matching technique was used which only returns identical lines as matches, it was sometimes harder to detect duplicates because identifiers were changed between both fragments. The use of a clone detection technique that does not take into account such non–functional changes, makes it more difficult to reconstruct a program’s evolution.

### Table 1. Number of duplicates, longer than 10 effective lines of code, and the length of the longest match in effective lines of code

<table>
<thead>
<tr>
<th>version</th>
<th>3.1.1</th>
<th>3.2.3</th>
<th>3.2.4</th>
<th>3.3.1</th>
<th>3.3.1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>60</td>
<td>66</td>
<td>66</td>
<td>118</td>
<td>117</td>
</tr>
<tr>
<td>max. length</td>
<td>38</td>
<td>33</td>
<td>33</td>
<td>39</td>
<td>39</td>
</tr>
</tbody>
</table>

### References


