Jens Knoop, Uwe Zdun (Hrsg.): Software Engineering 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 61

Copy-Paste Redeemed [1]

Krishna Narasimhan[2] Christoph Reichenbach[3]

Abstract: Software evolves continuously. As software evolves, it's code bases require implementations of new features. These new functionalities are sometimes mere extensions of existing functionalities with minor changes. A commonly used method of extending an existing feature into a similar new feature is to copy the existing feature and modify it. This method of extending feature is called "Copy-paste-modify". Another method of achieving the same goal of extending existing feature into similar feature is abstracting the multiple similar features into one common feature with appropriate selectors that enable choosing between the features. The advantages of the "Copy-paste-modify" technique range from speed of development to reduced possibility of breaking existing feature. The advantages of abstraction vary from user preference to have abstracted code to long term maintenance benefits. In our paper, we describe an informal poll and discuss related work to confirm our beliefs about the advantages of each method of extending features. We observe a potential compromise while developers extend features which are near-clones of existing features. We propose to address this dilemma by coming up with a novel approach that can semi-automatically abstract near-clone features and evaluate our approach by building a prototype in C++ and abstracting near-clone methods in popular open source repositories.

Keywords: Refactoring, Software clones, Static analysis, Software evolution, Abstraction

1 Introduction

Programmers frequently employ copy-paste-modify as a method of implementing extensions to features. Although copy-paste-modify yields quicker results with minimal damage to existing code, it results in bloated code space with redundant code. This is a headache for maintenance as readability is reduced and bug fixing is tedious is a bug in the initial near-clone is propagated to the copy pasted extensions. On the abstraction, provides maintenance friendly code occupying less code space. But, manual abstraction is hard. We propose to resolve this discrepancy with a novel approach that can abstract features from near-clones, thereby allowing developers to quickly extend features by employing copy-paste as a method of extending features and invoking a refactoring which will provide the best possible abstraction.

2 Informal Poll

We conducted an informal poll with five programmers of varying experience with C++ programming ranging from 2 months to 10 years in order to determine which method of

[1] A summary of the publication by the same name in ASE 2015

[2] Goethe Universitït, Informatik, Robert Mayer Strasse 10, 60486 Frankfurt, krishna.nm86@gmail.com
[3] Goethe Universitït, Informatik, Robert Mayer Strasse 10, 60486 Frankfurt, reichenbach@em.uni-frankfurt.de

extending features was easier to develop and which method was preferred for use. For the initial study, we collected 5 near-clone function pairs from popular open source repositories, removed one of the functions and asked the programmers to implement the remaining feature using copy-paste(for one group) and abstraction(for another group). We measured the time taken and observed that **Users find copy paste quicker**. We followed the initial study with a survey on the user preference in the same issue and found out that **Users prefer abstracted versions of code to maintain and use**.

3 Merging Algorithm

The merging algorithm takes as input abstract syntax trees of function definitions and

```
void function1(){
  b(c,d);
  y = f1;
  x(z);
}
```

```
void function2(){
  b(c,e);
  y = f2;
  x(z);
}
```

```
void function3(){
  b2();
  n();
  y = f3;
  x(z);
}
```

```
void fnMerged(int functionId, int fValue, int bParam)
{
  if(functionId == 1 || functionId == 2)
  {
    b(c, bParam);
  }
  if(functionId == 3)
  {
    b2();
    n();
  }
  y = fValue;
  x(z);
}
```

returns a merged function definition. The algorithm identifies the merge points. In the example, the merge points are the two If conditional branches and the position of 'bParam'. After identifying the merge points, the algorithm arrives at the best code transformation pattern to perform the merge. In our example, there are two resolution patterns, one is the extra parameter and the other an if conditional branch. There are many other possibilities depending on the type of nodes in the merge point.

4 Experiments

We evaluated our approach by building a prototype in C++ and using the prototype to merge existing near-clones in popular GitHub open source repositories, including Google's Protobuf and Oracle's Nodedb. Majority of our abstractions were merged into production code, thereby validating the quality of our abstractions.