Abstract

We examine the benefit derived by programmers from using tools to automatically, and interactively complete and adjust their code. In particular, we will evaluate the efficacy of our code repair and synthesis tool, Winston, developed in the course of CPSC 290 in Fall 2014, in a series of two experiments. We will study the productivity gains derived from our tool both in terms of speed (from the programmer’s perspective) and correctness.

1 Overview

1.1 Motivation

Throughout the development process, issues in the design of one’s code often appear. The standard text on the topic, “Refactoring: Improving the Design of Existing Code” [1], lists misplaced methods, overly long methods, and long parameter lists as among the worst so-called “code smells” – indications that a program is poorly-designed – that affect functions.

That same book [1] also describes a set of small, reversible program transformations, called refactorings, that gradually allow a programmer to work these issues out of their programs. No change is made in isolation, however, and most refactorings require other parts of the code to be adapted to the new design.

While excellent tooling already exists for applying the initial transformations, extracting code from a method, for example, there is still much progress to be made in handling the collateral errors of more complicated transformations. Adding parameters to a function, or changing the type of a parameter, necessarily breaks all of its call sites. Current tools allow the programmer to specify a default argument so that the code compiles, but these are never intended to be permanent measures.

1.2 Motivating Example

Suppose you have written the following Java code:

```java
String myFile = "/home/user/file.txt";
```

and you flesh out the containing function using this `myFile` variable. Later on, you might decide to send this file across the network. One possible action the programmer could take would be to change this declaration to
URL myFile = new URL("file:///home/alex/file.txt");

Adding this richer information to the variable increases flexibility (the original name can be recovered from method calls) but also breaks the dependent code. We expect that a tool to automatically correct these errors will boost programmer productivity.

1.3 Problem Description

Our goal is to evaluate the productivity gains (or, possibly, losses) provided by applying Winston to the aforementioned scenarios. To that end, we will conduct the following two experiments.

1.3.1 Experiments

1. **Automated Study.** Self-contained examples drawn from books, tutorial websites, and StackOverflow will be automatically broken and repaired in two independent stages using our tool. We will measure the effectiveness of our tool as a function of how often the original code is restored.

2. **User Study.** We plan to test our tool with the help of Yale Computer Science undergraduates, split into two groups. Both groups will be given the same set of coding tasks. To control for environment familiarity, we will seek students without previous exposure to JetBrains IDEs, such as IntelliJ IDEA, for which Winston is a plugin. One group will try to complete the tasks given while leveraging Winston, while the other group will work unassisted. We will measure the effectiveness of our tool as a function of how quickly each group completes their tasks.

1.3.2 Possible Extensions

Depending on the results of the study, if we find that programmers benefit from code repair in refactoring scenarios, we would extend Winston to automatically propose candidate repairs for code broken after one of the IDE’s refactoring actions.

We also intend to provide the core Winston algorithm as a library and a service (not unlike a SQL database) to ease integration into other IDEs, compilers, and tools. There is still a lot of room for improvement for both the algorithm itself and its implementation. For example, the algorithm is implemented in two stages, one of which is probably unnecessary. Additionally, the algorithm requires as input a depth limit for its graph search. There exist automatic ways of finding good clusters around vertices, and so these techniques will be evaluated as applied to our algorithm.

We also intend to weight Winston’s search by mining weights for functions from a Java corpus. It was shown by Sharma, Nori, and Aiken [4] that applying machine learning to program verification applications provides performance benefits. Since Winston builds off of a similar theoretical foundation, it is likely to benefit from these techniques.
2 Related Work

2.1 InSynth

Gvero, Kuncak, and Piskac [2] describe a tool called InSynth that synthesizes API calls based on the type-inhabitation problem. Winston extends InSynth’s capabilities from simple synthesis to repair, and performs similarly on its benchmarks. We will include these benchmarks in our first experiment.

2.2 Prospector

Mandolin, Xu, and Bodík [3] describe a tool called Prospector that mines for “jungloids”: networks of types related by unary operations, which are used to generate code snippets. As a tool, Prospector is seriously limited compared to Winston, but the paper included a promising user study similar to the second one we’ve proposed.

3 Deliverables

By the end of the semester, we expect to have produced:

1. The results of the two studies described, including a discussion of the statistical significance of our findings.

2. An updated version of Winston, along with any tools developed during the course of our evaluation. This might be an Oracle javac or IDE plugin, or just a standalone tool.

References


