# GI in No Time

David R. White
UCL, London, UK
david.r.white@ucl.ac.uk

## ABSTRACT

We describe a small, simple, and lightweight microframework for the Genetic Improvement of Java code. We call the framework "GI in no time", or "Gin". Gin is designed to be a straightforward, hackable, GI tool for Java. It currently lacks large features found in comparable program repair tools, but nonetheless it is capable of performing optimisation of a Java class via local search. We hope that providing this contribution will encourage researchers to collaborate on GI tool development, whilst lowering the barrier to entry for those interested in experimenting with GI. It is intended to serve both as a toolkit to be extended, and also an example of how GI can be implemented. We discuss some of the design principles behind Gin, and outline observations made during its development.

## CCS CONCEPTS

•**Software and its engineering → Genetic programming;** *Search-based software engineering;*

## KEYWORDS

Genetic Improvement, Optimisation, Automated Programming

**ACM Reference format:**
David R. White. 2017. GI in No Time. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 3 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082515

## 1 INTRODUCTION

Genetic Improvement is still a young field, and the tools available to researchers are limited. The area of program *repair* is quite well-served by a number of frameworks [4, 6], but tools for canonical offline program *optimisation* are comparively scarce, with a few notable exceptions such as Langdon's GISMOE and Schulte's GOA [3, 5]. To encourage collaboration in filling this niche, we have constructed the first iteration of a very simple and minimal GI framework, called "GI in No time" or "Gin". Gin is designed to be used in a white-box manner: using it requires modifying its source code. Gin is implemented in Java, and optimises Java code. It contains less than 400 lines of Java code and four major classes. We achieve this very small footprint by leveraging two popular libraries: we parse code using JavaParser [7] and evaluate patches using JUnit [2]; we delegate most of the hard work to those libraries.


*GECCO '17 Companion, Berlin, Germany*

DOI: http://dx.doi.org/10.1145/3067695.3082515

We release the code under the permissive MIT license and we encourage reuse, forking, extension, issue reports, feature requests, and pull requests. We hope that Gin will be of use in creating simple experiments for teaching and research. Gin is available at https://github.com/drdrwhite/gin.

## 2 DESIGN

To drive the design of Gin, we proposed two use cases: the invocation of a local search from the commandline, and the same invocation in a programmatic manner. The goal was to simplify these invocations as much as possible.

In both cases, we considered only the optimisation of a single class file, and correspondingly a single JUnit test file. We perform optimisation at the statement level, although new edit types could easily be added to Gin. Generalising the system to multiple files should only require a modest amount of work. The main focus of Gin is to minimise the amount of effort needed, in terms of time spent studying the system, or lines of code to perform the above. A further goal, more difficult to measure, was to make the entire framework intuitive so that it can be fully understood by a newcomer in the minimum amount of time possible.

We achieve this minimalism by leveraging two popular libraries: the JavaParser library [7], which presents a very simple interface to Java parsing, and the JUnit unit test library [2], which allows us to test candidates solutions with minimal effort. We also choose to deliberate violate some principles of good practice in design, such as reducing abstraction, repeating code, avoiding unnecessary inheritance, and implementing "heavyweight" constructors that complete several substantial tasks. We avoid caching data where it would reduce readability. In making any design decision, the solution that reduced the complexity of the framework was taken.

The system class diagram is given in Figure 1. There is no central class, rather we anticipate all usage to be through modifying the code, based on the provided example search class as a reference. No parameter file system is used: parameters are hardcoded, and sensible defaults are chosen. A user should tweak these parameters or parameterise the system depending on their intended usage.

### 2.1 Gin Classes

We describe *all* of the functionality of the main Gin classes.

**Program** encapsulates the original source file, and parses it with JavaParser to create a CompilationUnit. This holds the original source, structured by nodes in the Java grammar. `Program` also counts the number of statements in the code, so that the potential points of insertion and change are known.

**Patch** contains a sequence of edits. Separation of concerns is violated: the `Patch` class is aware of the possible types of edit, and is responsible for applying them, improving the readability of the code. It also creates random patches for a program. The edits

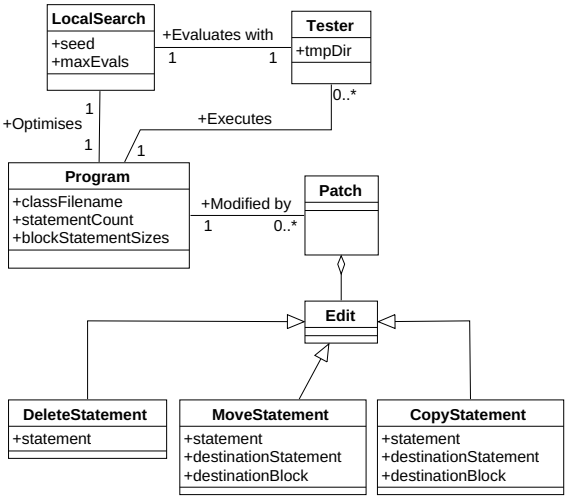


**Figure 1: Complete Class Diagram for the Gin System**

supported by default are those commonly found in GI applications: deletion, removal, and copying of statements.

**Tester** places the modified source and accompanying JUnit test class in a temporary directory and compiles them. It dynamically loads the two classes and executes JUnit, whilst reporting the mean execution time of the test cases over a given number of repetitions.

**LocalSearch** is an example GI algorithm implemented using Gin. It generates random patches of up to length ten as neighbours, and only accepts improving moves that result in a new solution that passes all test cases whilst reducing execution time.

## 3 EXAMPLE USAGE

Gin can be invoked via the commandline:

```
java gin.LocalSearch examples/Example.java
```

To achieve the same effect programatically:

```
localSearch = new LocalSearch("examples/Example.java");
Patch result = localSearch.search();
```

## 4 OBSERVATIONS

During implementation, we came across subtleties that are rarely discussed. We describe three points of interest here.

### 4.1 No "Hello World"

Genetic Improvement, particularly non-functional GI, lacks a canonical "Hello World" example. During development, a contrived example was used to test the system. The lack of such an example, as well as a benchmark suite more generally, make standardisation of GI systems and empricial comparison of different methods difficult. A set of simple optimisation challenges and reference code would be immensely helpful in developing GI tools.

### 4.2 Interactions within Patch Sequences

Most GI work uses a combination of line deletion, copying, and replacement. Gin allows for the deletion of any statement, at multiple levels of abstraction (it may delete an entire method body, for example, as well as a single statement in the `else` branch of an `if` expression), and it allows for insertion of code before any individual statement contained within a block statement, both through copying and moving existing code. We do not implement a "replace" operator, as Java imposes more constraints on patches than C, the traditional language of GI, and a replace operator is of less use here.

Applying a sequence of edits to a program involves design decisions that define the structure of the search space. For example, care must be taken to ensure that deletions occur after all copying of code has taken place; moving code is decomposed into a copy and a deletion. Furthermore, if two different edits require insertion at the same point, an ordering dependency exists. Hence adding a new edit to a patch may impact not only the lines that the edit changes, but also the effect of pre-existing edits. Gin implements the patching process by building two queues of deletions and insertions, before executing those operations. This ensures predictable behaviour, but does not remove ordering dependencies. Additionally, when inserting code at a given location we must consider whether the code should be inserted *before* or *after* a statement.

The nature of the landscape induced by the patch representation needs to be studied carefully. Certainly representations exist that do not suffer from this particular flaw: the direct manipulation of the abstract syntax tree is one (suboptimal) example.

### 4.3 Hidden Language Assumptions

Most work in GI has tended to focus on C code. When implementing GI for the Java language, it became apparent that manipulating lines of code does not translate well to Java. Java files can include multiple type declarations, interfaces, package names, complex method signatures and a hierarchical structure that is appears to be more fragile to change than C. Thus Gin works at the statement level and restricts insertion points to the children of block statements.

Previous work illustrating the robustness of C code may not generalise to other languages, which use complex language constructs and large grammars. It may be necessary to reinvent the component operations that constitute patches to suit the host language.

## 5 CONCLUSION & FUTURE WORK

There are several potential features of immediate interest: handling multiple class files and larger projects; full unit testing of the Gin code; incorporation of profiling to guide the search using a library such as SPF4J [1]. Gin is designed to be an open and ongoing project. Whether it serves as an example of how to quickly exploit JavaParser and JUnit, or develops into a general framework, depends on interest from the research community.

## REFERENCES

[1] Z. Farkas. 2017. SPF4J. http://zolyfarkas.github.io/spf4j/. (2017).
[2] JUnit. 2017. http://junit.org/junit4/. (2017).
[3] W. B Langdon and M. Harman. 2015. Optimizing existing software with genetic programming. *IEEE Trans. on Evo. Computation* 19, 1 (2015), 118–135.
[4] S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE 2016*. 691–701.
[5] E. Schulte. 2017. Genetic Optimization Algorithm (GOA). https://github.com/eschulte/goa. (2017).
[6] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 43–54.
[7] D. van Bruggen. 2017. JavaParser. http://javaparser.org/. (2017).