Java Unit Evolution

Felix H. Dahlke

2010-11-03

Contents

1 Introduction

Java Unit Evolution combines genetic programming and test-driven development to create a new software development process.

Genetic programming (GP) is an evolutionary computation technique where computer programs are evolved based on Darwin's theory of natural selection. A population of programs is created the fittest of which survive to be combined with each other, creating ever fitter programs. A vital part of this is the fitness function; a function that is used to evaluate the programs and to determine their fitness, i.e. chances of survival.

Test-driven development (TDD) is a software development method where code to test a function is written prior to its initial implementation. The programmer writes test cases assessing all requirements so that the resulting implementation will act according to expectations.

Java Unit Evolution combines these two areas by using unit test cases as a fitness measure (i.e. fitness function) to evolve programs using GP techniques. Literally, this is *Test-only development*.

2 Related work

I investigated whether something similar already existed. I found a few articles and conference proposals by people who clearly had the same idea, but I was unable to find anything besides initial thoughts:

There is a system called *OMAPS*, but I was unable to find out more or get in contact with the speakers [Roberts et al., 2009].

There is product called *Neovolve* from a company developing software around what they call *Extreme Genetic Programming* [NeoCoreTechs, Inc., 2003]. However, the website is not up to date and it is not possible to download their software, nor are there any papers detailing their work.

[Oppacher et al., 2009] used JUnit test cases as a fitness measure. They didn't seem to cover test-driven development though, so I didn't regard their work to be the same as my project.

3 Differences between GP and TDD

Although they come from different fields and have different intentions, the similarities are difficult to miss: Both GP and TDD are about developing software by means of specifying the requirements, using fitness functions (sometimes called "fitness cases") in GP and unit test cases in TDD.

However, the approaches do also differ considerably in some respects: Test cases in TDD are for confidence in code whose implementation is known, while confidence in GP can only be achieved if large parts of the input domain are covered. As explained above, TDD is not about test coverage. While a developer

can be trusted up to a point, an algorithm will not employ any kind of common sense and simply create code that passes the fitness function, no matter how.

TDD also introduces an approach to software design that is highly iterative: Developers improve both the implementation and the design by refactoring frequently. In GP, a function has to be carefully designed and specified in advance because refactoring is impossible - a new solution for the altered specification would have to be evolved from scratch. While TDD provides almost immediate feedback about bad design decisions to the developer, such problems can result in the inability to evolve a workable solution in GP, often without a clear indication of the problem.

To build confidence, it will be necessary to use a different set of test cases to validate the evolved result. This can be thought of as the validation set in supervised training of neural networks: A different set of input and output values is used to make sure that the network was not *overtrained* towards solving the problems in the training set and nothing else. This can happen in GP as well.

It is probably because of these differences that few people have tried to combine GP and TDD. I had to depart from the work flow of TDD which I intended to adopt towards one that is more appropriate for GP.

4 The design of Java Unit Evolution

Java Unit Evolution uses JUnit to create and execute unit test cases and JGAP for genetic programming.

The developer's work flow is as follows:

1. Specify the desired method as an abstract Java method.
2. Specify all possible operations as non-abstract methods of the same class.
3. Write JUnit test cases against the abstract method.
4. Invoke the test cases.

One of my key concerns was to hide details of GP from the developer. There is only one special line of code required, which does all the work: Starting the evolutionary process and returning an instance of the desired method used by the test suite:

```
private TrianglePerimeter trianglePerimeter =
        JavaUnitEvolution.evolve(
                TrianglePerimeter.class,
                getClass());
```

The first parameter is an abstract class containing the method that is to be evolved. The method does not have to be specifically specified, I used Java's reflection mechanisms to determine it: I simply pick the first abstract method I can find and log a warning message if I find more. The second parameter is the class of the test case used to determine each implementation's fitness. I don't

like requiring the developer to explicitly pass the test class to this method, since it will only make sense to pass the class invoking this method, but I don't think there's another way in Java.

When the developer executes the test case, e.g. from his integrated development environment (IDE), the first call to the `evolve` method will initiate the evolutionary process, executing the test case which in turn calls `evolve` again. Subsequent calls, however, will merely return an implementation of the abstract class using the program whose fitness is currently assessed. After the process finishes, the initial call to `evolve` will return an implementation using the best solution and the IDE will display the results.

Other than in optimisation problems, problems approached via TDD have a well-defined condition of success: Passing all the unit tests. Thus the evolutionary process returns immediately after producing a solution that can pass all test cases. If a feasible solution can not be found, the process has to be aborted. In GP, it is usually bounded by the maximum number of generations, but since I tried to hide GP details, I had to replace this with something more natural. Hence I made the process run for an unbounded number of generations, but installed a timeout that would abort it after a while, the default value being five minutes. This timeout can be adjusted by the developer, but I believe that he will in most cases be able to solve this problem by writing more or better test cases or operations, not by increasing the timeout.

I decided against having the developer supply a configuration object or file as common in GP frameworks. I wanted to hide these parameters and instead tried to use sensible values applicable for different problems. I used the suggestions made by [Koza, 1992] for this purpose. Nonetheless, complex problems might make it necessary to adjust some parameters, especially the population size.

5 Calculating fitness from executing test cases

The result from executing JUnit test cases contains the total number of test cases run ($runs$) and the number of failed test cases ($failures$). Initially, I used the following approach to calculate the fitness from this data:

$$fitness = \frac{runs}{failures} \quad \text{if} failures > 0$$
$$fitness = 0 \quad \text{otherwise}$$

I used what [Koza, 1992] calls *standarized fitness*, i.e. higher values are less fit, and 0 is the perfect solution, so I soon turned to an even simpler function:

$$fitness = failures$$

However, I was unable to evolve a solution using this approach, probably because the algorithm had no way of knowing how far off it was, i.e. the *delta* (δ). In the

case of numerical problems, this is the same as the euclidean distance between the expected and the actual value:

$$\delta = \sqrt{(expected - actual)^2}$$

Which is equal to the absolute value of the difference:

$$\delta = |expected - actual|$$

But the expected and actual values were not supplied by the test runner. It did, however, supply the exceptions that caused the test cases to fail. I used an assertion to check for the expected value, so the exception raised is an assertion error; an instance of the class `AssertionError` whose messages contain the actual and the expected value. Thus all I had to do was to iterate through all failures, extract the actual and expected values from the messages of the assertion errors, calculate δ and sum it up.

I implemented this for all numerical primitive data types, calculating δ for the other types might be more challenging.

6 Generating operations and operands

The operations and operands were initially hard coded, but I was able to extract the operands from the method's signature using Java's reflection mechanism.

Generating operations proved difficult, and I was discouraged by [Koza, 1992] who stated that every machine learning problem needs the explicit declaration of something equivalent to the function and terminal set (i.e. operations and operands).

I wondered how the developer could specify operations as conveniently as possible, and came to the conclusion that writing a Java method would be the most sensible approach.

That's why the method that is to be evolved is specified as an abstract method of an abstract class, not as an interface method which would be more natural. The evolutionary process will evolve the first abstract method it finds and use all non-abstract methods as operations. These methods also have to be static, I introduced this requirement to emphasise that they should not change the state of the object. Technically, the developer could use them to alter static variables if he really wanted to have state, but I didn't want to completely detain him from doing that, I just wanted to make it obvious that it shouldn't be done. If the objects had state, the test cases might only all pass if executed in a specific order, but according to [Beck, 2002], test cases should never depend on other test cases.

This is a workable solution, but it requires the developer to consider an aspect I didn't want him to consider: The operations required for solving the problem. Depending on the problem, this can require significant consideration and will be a likely source of errors.

7 Experiments

Once I had a working prototype of Java Unit Evolution, I conducted several experiments, with rather disillusioning results.

My first experiment, and the one used throughout development was remarkably trivial: Evolving a method that would add two numbers together. The evolved method would have to pick addition from the available operations and apply it to the operands. As expected, Java Unit Evolution was able to evolve a solution fairly quick. It was encouraging to see that a few test cases were sufficient to evolve a workable solution.

The constraints I imposed made it impossible to use example problems from [Koza, 1992] or [Beck, 2002] because most of them, required the use of objects or data structures, while Java Unit Evolution can only work with standard Java types.

I found a promising problem in the appendix of [Beck, 2002]; a test-driven implementation of a function finding Fibonacci numbers. However, while trying to implement this, I noticed that I would have to provide control structures such as loops and conditionals as operations, something I did not anticipate before. I did not like the thought of requiring the developer to specify e.g. a loop function by means of a Java method, nor did I have an idea how that could look, so I tried to generate some general purpose operations. I failed at combining the generated operations with the provided ones and the operands, so I was unable to evolve a solution to this problem.

I then thought about simple mathematical problems that could be solved and decided to do something just slightly more complicated than the addition sample: Evolving a method that would calculate the perimeter of a triangle. The main difference between this and the addition sample is that it requires to add three numbers together. Just like the addition sample, the method merely had to pick the right operation and apply it to all operands.

When writing the test cases for the triangle perimeter problem, I used an TDD-like approach by taking very small steps and by using the method of triangulation described by [Beck, 2002]. Instead of specifying many arbitrary test cases in advance: I first wrote a single test case and invoked the evolutionary process, which was able to evolve an implementation that passed the test. I then wrote a similar test with different methods to make sure that the solution would generalise. Again, a workable solution could be evolved.

While implementing the triangle perimeter sample, I discovered a disturbing problem: The evolutionary process would always feed all operands to each operation, making it impossible to use operations with a different signature than the method under evolution. I could not think of any more sophisticated problems that could be evolved with this limitation, but because I didn't have enough time left, I decided to leave this issue unsolved for now.

8 Limitations

Java Unit Evolution presently has the following limitations:

- It can only evolve solutions to problems involving the following standard Java types: `Integer`, `Long`, `Float`, `Double` and `Boolean`. It could be extended towards supporting more types such as `String`, but a way to calculate the euclidean distance between two `String` objects would have to be found.
- It can only evolve solutions to problems using operations that accept exactly the same arguments as the method under evolution. This is arguably the most severe issue.
- It can not save the evolved program. If this is not implemented, the evolved solutions, no matter how promising, can not be used in production.
- Evolved programs can not use control structures such as loops and conditions.

Once these limitations are overcome, I see many ways in which this work can be extended:

- Conduct more experiments, e.g. example problems from GP and TDD.
- Extend Java Unit Evolution to make it possible to evolve solutions to problems using arbitrary objects. Considering that all objects should eventually be made up of either other objects or Java standard types, this should be possible. However, calculating the euclidean distance between arbitrary objects will probably be difficult and it might be necessary to make the developer write such a method for each object he designs.
- Analyse more TDD patterns. [Beck, 2002] mentions several interesting testing patterns which I could not investigate because of the limitations of Java Unit Evolution. There is the log string for instance, which means that each operation would append its name to a string and the test cases could use this string to verify that the correct methods where called in a sensible sequence. Another promising pattern is the mock object, which is a hard coded copy of an object that works with an external resource such as a database. It should appear to behave just like the real object. Because there would be no overhead involved with using the object and no reliance on an external resource, Java Unit Evolution might actually become able to work with external resources that way.
- Analyse non-TDD testing patterns. The more sceptical nature of traditional software testing as described by e.g. [Binder, 1999] might lead to more systematic test cases and hence more reliable results. This is certainly worth investigating.

Java Unit Evolution also inherits the problems of TDD, e.g. those identified by Darach Ennis in [Beck, 2002]:

- You can't test GUIs automatically (e.g., Swing, CGI, JSP/Servlets/Struts)

- You can't unit test distributed objects automatically (e.g., RPC and Messaging style, or CORBA/EJB and JMS)
- You can't test-first develop you database schema (e.g., JDBC)
- There is no need to test third party code or code generated by external tools
- You can't test first develop a language compiler / interpreter from BNF to production quality implementation

[Beck, 2002] said that he wasn't sure Ennis was right, but he wasn't sure whether he was wrong either.

9 Conclusion

My plan was to create a framework that would evolve programs, measuring their fitness with a JUnit test suite and to find testing techniques used in TDD and see how they can be applied to GP.

That framework is Java Unit Evolution, and I was able to evolve some simple programs with it. However, I was unable to evolve solutions to anything but very simple problems.

From conducting my experiments and implementing Java Unit Evolution, I noticed that it is remarkably difficult to find problems in this type of development. When no program is evolved after the timeout is reached, the developer has the following options:

- Increase the timeout and try again.
- Write more test cases or try to identify problems with the existing ones.
- Write more operations or try to identify problems with the existing ones.
- Change the GP parameters.

There is practically no way to tell which of these should be tackled, and even if the correct option is chosen, there is no way to tell that it was indeed the problem if not all necessary amendments are made.

The bottom line is that while I consider this to be an interesting experiment possibly pointing towards new ways of software development, Java Unit Evolution has many issues and is clearly not a practical solution.

References

[Beck, 2002] Beck, K. (2002) *Test driven development: by example*, Boston, MA, USA, Addison-Wesley

[Binder, 1999] Binder, R. V. (1999) *Testing object-oriented systems: models, patterns and tools*, Addison-Wesley

[Koza, 1992] Koza, J. R. (1992) *Genetic programming: on the programming of computers by means of natural selection*, Cambridge, MA, USA, MIT Press

[NeoCoreTechs, Inc., 2003] NeoCoreTechs, Inc. (2003) *eXtreme Genetic programming - the official XGP site*, Available from: `http://www.neocoretechs.com/` (Accessed 28 February 2010)

[Oppacher et al., 2009] Oppacher, Y., Oppacher, F., Deugo, D. (2009), 'Creating objects using genetic programming techniques', in *10th ACIS international conference on software engineering, artificial intelligences, networking and parallel/distributed computing*, May 27-29 2009, pp. 455-461

[Roberts et al., 2009] Roberts M., Youell M. (2009) *Test overdriven development with OMAPS*, [online], Open Source Bridge, Available from: `http://opensourcebridge.org/proposals/248` (Accessed 28 February 2010)