Group members: Alia Paddock (akp42), Hans Jorgensen (thehans), Lemei Zhang (lemeiz)
CSE 403 Project 2: Proposal
4/7/18

SAIT: Simultaneous Alternate Implementation Testing

What are we trying to do?

Often, when testing complex software projects, a defect observed in the program or in unit tests might be pretty much anywhere in the codebase. This is especially true for multithreaded programs, when execution order can be complex and nondeterministic and not all of the variables are immediately available to the debugger (one such issue arose for one of the authors when they were trying to debug a liveness issue in an experimental operating system kernel). In these projects, when a defect appears, many debugging plans (hereafter referred to as “probes of inquiry”) are possible. However, the limitations of most programming languages only allow a single probe of inquiry to be conducted at a time. In order to switch to another probe of inquiry, the first must be either archived or abandoned entirely.

We want to offer an extension to Java unit and integration testing that allows the same tests to be run simultaneously on multiple implementations of the same method or class. We will utilize Java’s existing subtyping mechanism to allow multiple unit tests to run on the same module, and provide a ClassLoader for substituting a class for an alternate implementation of that class during integration tests. From this, we hope to create a workflow where developers can quickly set up multiple different debug probes in the same codebase and quickly compare them.

How is this done today, and what are the limits of current practice?

Refinement is an essential part of the software development process. When an implementation does not compile or behave correctly, the developers attempt to refine the design to create the desired behavior. Sometimes, developers can succeed with small, easily testable changes, such as a change in constants or call order, while at other times, the design may have to be heavily edited or completely redone. Developers may also include additional debugging information, such as logging statements or additional temporary variables. All of these are valid forms of refinement, and in all but the simplest codebases, many refinements and fixes are plausible.

However, within most programming languages, limitations in the design of the language or libraries mean that only one refinement may feasibly be attempted at a time. The biggest culprit is simply the module linking mechanism: many programming languages require that modules be referred to by strong, unique names, making it difficult to substitute any alternatives to the same model (to say nothing of even keeping the original version intact in the same copy of the codebase). Systems such as Typemock Isolator for .NET do provide facilities for substituting

modules by name, but are only optimized for creating stub (“mock” or “fake”) versions of another module’s dependency (such as to provide a deterministic result for an HTTP request).

Version control can be used both to maintain the original version of the module as well as to create multiple branches, one for each attempted refinement of a module. This is probably the best solution that currently exists for multiple refinements, especially if each developer assigned to the bug spearheads a different refinement or if the refinement is very widespread over many modules. However, it requires a lot of overhead, since it puts great distance between the possible refinements by placing them in separate copies of the codebase. Most IDEs will not support having both copies open at the same time (at best putting them in separate processes with no automatic comparison options), and some enterprise projects that are strapped for storage space (e.g. those with large codebases in which compilation times are only decent if the code and executables are stored on an SSD) cannot store multiple versions of their code on the same workstation at all.

OOP languages, such as Java, provide an interesting avenue for substitution of modules through their subtyping systems: if the client references the target module through a common interface or base class, the strong name of that module is no longer needed to use it. This can even be used within unit testing suites to write tests for common functionality among multiple objects, such as through JUnit’s parameterized testing. Such code, however, must be written with subtyping in mind, always using the correct interface name instead of the class name - which, in some cases, cannot exist in order to prevent an malicious client from supplying a non-conforming interface or subclass.-In addition, most parameterized testing features in packages like JUnit create tests that only pass if they pass for all parameters - when probing on multiple potential fixes/refinements, we only need one version to pass.

What’s new in our approach, and why do we think it will be successful?

In our approach, we propose allowing a developer to make multiple simultaneous probes of inquiry into refining and testing a Java class or method implementation. The essential strategy is that the class being tested can have multiple potential versions in the code at once, and these alternate versions of the class can either be loaded and tested simultaneously during JUnit testing or loaded in place of the base class during integration testing.

Our solution will present the notion of a *primary class* and an *alternate class*. A class A is an *alternate* to a *primary* class or interface P if:

1. A is marked with the @Alternate annotation, with P.class as an argument, or
2. A extends or implements P.

With this in place, JUnit tests can be marked with the @TestAlternate annotation, which indicates that they should run multiple times on the different class alternates of the same primary class P given in the parameters. The test will then take an instance of P as an

argument, with the final result being that the test is run once with an instance for each alternate class provided.

A distinguishing feature of this annotation is that a test marked with @TestAlternate, by default, will pass if *any one* of the alternate implementations passes. This is key to its intended use in refinement probing - as soon as one alternate is identified to pass the tests, the bug the tests identify has been fixed and the other implementations can be discarded or ignored. Such a test will also report to the user which alternates it passed with.

A few other features may also be included in @TestAlternate:

- Add a @TestClassAlternate annotation, which does the same thing as @TestAlternate for entire test classes. The @TestClassAlternateFill annotation will be added to indicate what fields and method arguments the alternates should fill, in place of where @TestAlternate would be used. Note that in order for a test suite marked with @TestClassAlternate to pass, any one of the alternates must pass *all* the tests in the class.
- Allow @TestAlternate (or a version of it) to take an array of classes that implement Supplier<P> instead of alternate classes for P, to allow more custom alternate creation options such as different initialization parameters or the use of method references for testing individual functions rather than entire classes.
- Add an argument name parameter to @TestAlternate and @TestClassAlternateFill to allow multiple @TestAlternate attributes to be supported on the same test, complete with multiple arguments. The test will be run once with each independent combination of arguments provided by @TestAlternate (e.g. if a test has 3 @TestAlternate annotations with 3 alternates each, the test will be run 27 times), with the whole test passing if any one combination of alternates passes.
- Allow @TestAlternate and @TestClassAlternate to be parameterized so that they pass only if *all* of the alternates pass. This is semantically similar to generating a separate copy of the test for each implementation or using JUnit's parameterized test feature, and can be used if all of the alternates are intended to be permanent.
- Provide more performance comparisons for all alternates that pass the test, such as providing the execution time or RAM usage for each.

For integration testing, a custom ClassLoader will be used. This ClassLoader will take the name of the target primary and the desired alternate (or any list of these pairs) as arguments, and when the runtime attempts to load the target primary, the ClassLoader will instead load the target alternate, changing its name to be that of the primary and thus allowing it to masquerade as the primary. This allows for debug probing and alternate implementation testing without being forced to rewrite all references to the class in the code or to archive the old class for later retrieval. A similar version of this ClassLoader will also be used to generate multiple versions of JUnit tests when alternates marked by the @Alternate tag are used.

Who cares?

Because we are working with Java, and the practice of software refinement is an established standard, our potential stakeholders would potentially include all Java developers, which amounts to approximately 15.8% of all software developers according to the April 2018 TIOBE Index of programming languages (TIOBE). A more target audience would include users of JUnit, which is used in approximately 30.7% of open source GitHub projects according to a survey performed in 2013 (Weiss).

As an interesting use case for our library, many software developers are interested in testing different implementations of the same interface on different merits, such as on their performance, and our library would make these tests easier to express, compare, and maintain. For instance, one study conducted at Auburn University in 2016 measured the energy usages of the different types of collections in the Java Collections Framework, for instance finding that the TreeList consumed about 300% more energy than the ArrayList did when executing the same operations (Hasan et al.). Expressing these tasks, and providing built-in comparison data, would be something that our @TestAlternate feature would be ideal for.

If we're successful, what difference will it make?

With this functionality, Java programmers wouldn't have to rely on version control as heavily, if at all, for keeping alternate implementations separate and testing them independently. By testing multiple implementations simultaneously, SAIT can more efficiently optimize the choice of data structures and algorithms for its users' projects and find potential bugs in each implementation. If we succeed in integrating our proposed functionality with JUnit, we will have improved upon a testing framework used by, as mentioned, about a third of open source GitHub projects.

What are the risks and the payoffs?

Since for this project, we will be focusing on extending the functionality of JUnit, one challenge at the beginning will be to get familiar with the JUnit code base, and there is the risk that we might need to modify our approach as JUnit might not be possible to extend in the way we want. If we do determine that JUnit can be modified to suit our needs, another interesting aspect of working with JUnit in particular is that JUnit is tested with itself. Specifically, a small feature set (the "Jupiter" set) is used to test the rest (the "Platform" set). We would need to ensure that we do not adversely affect this relationship.

However, the payoff for this is that if we can manage to get a good basic understanding of the JUnit code base fairly quickly, then we will be able to efficiently integrate our feature with JUnit's already rich feature set - both in terms of oracles to test and types of tests that can be run - as well as JUnit's large user base.

How much will it cost? How long will it take?

Monetarily, because we are working with open source software (JUnit), we will not incur any costs beyond the typical operating costs of UW CSE student technology or our personal technology. Our aim is to finish implementing our tool (or, at least, the majority of it, minus any bells and whistles) by the end of April. That leaves us some time before the end-of-quarter presentations to test the effectiveness and usability of our product with potential stakeholders (for example, CSE 331 students). Testing our implementation will involve letting our chosen test subjects try out our tool on their code. There will be metrics for testing both inherited/interfaced classes as well as alternate classes. We will quantitatively measure the overall improvement in time and performance of the testing, as well as qualitatively measure the usability and usefulness of testing output. Overall, we want to finish implementing our tool early enough to leave enough time to be able to improve upon it after receiving meaningful feedback.

What are the midterm and final “exams” to check for success?

By the midterm point, we should be able to test alternates using the @TestAlternate attribute on existing subtypes, being able to add a @TestAlternate annotation to a test, and see that test run multiple times with an instance of each alternate class appearing as the argument to the test. We should also be able to use classes that implement Supplier<P> as arguments for @TestAlternate, which should run by creating the Supplier and calling its get() method to supply the argument. The resulting test should pass if any of the alternates pass.

Soon after the midterm point, the @TestClassAlternate annotation should also be implemented.

By the final point, the @Alternate annotation should also be implemented, as well as the custom primary replacement ClassLoader. The user should be able to test @Alternate implementations of P in @TestAlternate tests the same way have been able to test subtypes of P since our midterm checkpoint, and they should be able to replace P with any alternate, either marked with @Alternate or as a subtype, when running the actual application.

Week-by-week schedule

- Week 2 (4/2 to 4/8):
    - Read through the JUnit code base to get a basic understanding of it.
    - Decide whether or not it is a better choice to add our code to JUnit or create our own separate software tool.
- Week 3 (4/9 to 4/15):
    - Finalize the design and specifications of our code.
    - Time permitting, start on writing tests for our code.
- Week 4 (4/16 to 4/22):
    - Finish writing a test suite and documentation for our code base.

    - After finish writing the test suite, start on writing the code itself. Aim to finish the @TestAlternate and @TestClassAlternate functionality (the “midterm” checkpoint mentioned above).
- Week 5 (4/23 to 4/29):
    - Finish the @TestClassAlternate functionality, if not finished already.
    - Allow multiple @TestAlternate and @TestClassAlternate tags on a test, with parameter indicators.
- Week 6 (4/30 to 5/6):
    - Refine any errors in the implementation and submit code to chosen test subjects.
    - Update the project proposal with new implementation info.
- Week 7 (5/7 to 5/13):
    - Gather and evaluate the initial results of our code’s functionality and performance and collect user feedback from our stakeholder test subjects.
- Week 8 (5/14 to 5/20):
    - Aim to finish the implementation of @Alternate annotation and the custom primary replacement ClassLoader (the “final” checkpoint mentioned above)
- Week 9 (5/21 to 5/27):
    - Finish the implementation for @Alternate and the ClassLoader, if not already.
    - Finish the draft of our final report, including details about our code structure, functionality, and performance as well as preliminary user feedback.
- Week 10 (5/28 to 6/3):
    - Create a document that contains a description of the tool, and lists instructions on how to use our tool, including examples.
    - Finalize code refinements
    - Finalize the our report and presentation slides
- Week 11 (finals week):
    - Complete project presentation

Bibliography

TIOBE. “TIOBE Index”. *TIOBE, the software quality company*. April 2018.

Weiss, Tal. “We Analyzed 30,000 GitHub Projects - Here Are The Top 100 Libraries in Java, JS and Ruby.” *OverOps Blog*, 11 Mar. 2018.

Stefan Bechtold, et al. *JUnit 5 User Guide*.

Hasan, Samir, et al. *Energy Profiles of Java Collections Classes*. ACM Press, 2016.

Hours spent on assignment:

This assignment took us about 12 hours collectively to complete.