Group members: Alia Paddock (akp42), Hans Jorgensen (thehans), Lemei Zhang (lemeiz)
CSE 403 Project 2: Proposal
4/5/18

SAIT: Simultaneous Alternate Implementation Testing

What are we trying to do?

We want to offer an extension to Java unit and integration testing that allows the same tests to be run on multiple implementations of the same method or class. This would primarily be used to make multiple probes of inquiry in debugging a method or class, allowing the process of debugging to be faster, more flexible, and less error prone. This could also be used for testing multiple, long term versions of a method or class that are all correct but differ in other ways, such as in dependencies or performance requirements.

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

Bug probing, or the problem of finding the defect(s) in an implementation whose tests are failing, is currently solved by making iterative changes to a function. In a well-disciplined* software environment, this usually consists of reanalyzing the design, noting regions where the design is not confidently defined, determining possible adjustments to those parts of the design, implementing those adjustments, and then running the tests again. These adjustments either consist of adding (or subtracting) logging information or changing the implementation code itself. This process is repeated until either the tests pass or the design is sufficiently verified to determine that either the tests or incorrect or the design itself is misguided.

While the concept of iterative design and redesign does work, its primary limitation is that it can only attempt one line of inquiry at a time. Because software is exponentially complex, changing one aspect of the design can have a large and potentially chaotic effect on the rest of the system, and changing the design in multiple, unrelated ways usually results in an error. However, in debug probing, there are always multiple possible probes of inquiry (otherwise, you would know what the problem was), and if a developer wants to switch from one probe to a different probe, they must either suspend or abandon the probe they are on in order to restore the function to its original state and start the new probe.

Within a centralized version control system such as Subversion, one must almost always abandon probes in order to switch, since expensive, server-side branching typically requires that every commit to the server passes the tests, which disallows publishing incomplete versions of the code. The popularity of decentralized version control systems such as Git and Mercurial help this somewhat, since the separate notions of commit and push allow incomplete versions to be committed, but not pushed, and cheap local branching also allows for multiple chains of these local, unpushed commits. These systems allow users to commit incomplete versions (thus allowing easy rewinding of a probe) and also allow users to suspend incomplete probes of inquiry (using separate branches), but this system still typically requires users to either keep multiple versions of the repository on disk at once (which is not viable for large codebases) or, again, to only work on one probe of inquiry at a time.

As for the somewhat related problem of testing multiple viable implementations, most test suites will simply write separate tests for both classes, copying or testing slight variations on the methods that implement the common interface. While this does work for testing both implementations, and allows for greater flexibility in dealing with each implementation's edge cases, it requires two separate copies of the same test to exist, which is redundant and is vulnerable to changes in the interface. Additionally, the tests do not verify that the client class correctly implements the interface proper (if the interface is a separate entity in the code), nor are the tests semantically linked in any way.

JUnit does support the notion of parameterized tests, which allows a test to be run multiple times with separate arguments. The only supported types for these arguments are primitives, Strings, and Class objects, but those can be and are easily composed into other types of arguments by the annotation processor, such as by instantiating said Class objects and referencing methods through reflection on String arguments. This, however, is only suitable for creating a set of tests that must all pass - it is only useful for maintaining multiple implementations or many test cases, and not for the case of debug probing in which only one implementation has to pass the test.

*In an ill-disciplined software environment, debug probing usually consists of making haphazard changes to a function until the tests pass. Because of the exponential complexity of software, this very rarely works and usually leads to the function's meaning being mangled beyond all recognition, requiring the function to be restored from version control or started from scratch entirely.

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 probes of inquiry into testing a Java class or method implementation. The essential strategy is that the class being tested can have multiple simultaneous 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*. Most classes in a Java program are primary classes, and 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.

An important feature of @TestAlternate, especially in relation to the feature that JUnit already has of parameterized tests, is that @TestAlternate can be parameterized to indicate that the annotated test should pass if it passes for *any one* of the alternate implementations. This is the key feature that allows it to be used for debug probing - as soon as one alternate is identified to pass the tests, the bug the tests identify has been fixed and the alternate implementation can replace the original primary implementation. The user can specify which semantics to use, allowing @TestAlternate to also be used to test multiple viable implementations that should all pass.

A few other features may also be included:

- Allow the @TestAlternate annotation to specify whether the test passes if either *any* or *all* of the alternates pass. The former case would be used to determine which debug probe is successful, while the latter case would be used to test multiple viable implementations. Do the same with the @TestClassAlternate annotation, being sure to distinguish that in "any" mode, the test suite will pass if any one of the alternates passes all of the tests in the class.
- 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. The any/all semantics option will also be provided for @TestClassAlternate, with the important caveat that in order for a test suite class to pass with a @TestClassAlternate with "any" semantics, one of the alternate implementations must pass *all* the tests.
- 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 total set of tests that will be run is the Cartesian product of all the tests generated by each @TestAlternate.

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 do a major restructuring of the project to allow multiple dependencies (since, especially in the case of debug probing, the restructuring will go away once the bug is fixed). 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 probing for bugs is so common, 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 2017). 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 2013). However, since the principles of debug probing are potentially useful for all programming languages with modular elements (virtually any functional or object oriented language), any programmer with an interest in performing multiple debug probes concurrently should be interested in SAIT.

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 individually. 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 during 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 modify 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 quickly 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 and methods as well as alternate classes and methods. We will quantitatively measure the overall improvement in time and performance of the testing, as well as qualitatively measure the usability. 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 (using the “A extends or implements P” part of the primary/alternate definition given above). We should be 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 implementation and calling its get() method to supply the argument. The tests would likely run as independent test methods (thus enforcing the “all alternates” setting), though bonus points will be in order if tests run with “any” semantics or if the user can switch between them.

Soon after the midterm point, the @TestClassAlternate annotation should also be implemented, with the same options and semantics as the @TestAlternate annotation.

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 they tested subtypes of P, 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, and determine 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. Then 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, and finalize the our report and presentation slides

Bibliography

TIOBE. "TIOBE Index". *TIOBE, the software quality company*. April 2018. https://www.tiobe.com/tiobe-index/

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*.

Hours spent on assignment:

9