MIT-CSAIL-TR-2008-002 January 14, 2008

Theories in Practice: Easy-to-Write Speci&#64257;cations that Catch Bugs

David Saff, Marat Boshernitsan, and Michael D. Ernst

Theories in Practice: Easy-to-Write Specifications that Catch Bugs

David Saff
MIT CSAIL
saff@mit.edu

Marat Boshernitsan
Agitar Software Laboratories
marat@agitar.com

Michael D. Ernst
MIT CSAIL
mernst@csail.mit.edu

ABSTRACT

Automated testing during development helps ensure that software works according to the test suite. Traditional test suites verify a few well-picked scenarios or example inputs. However, such example-based testing does not uncover errors in legal inputs that the test writer overlooked. We propose *theory-based testing* as an adjunct to example-based testing. A theory generalizes a (possibly infinite) set of example-based tests. A theory is an assertion that should be true for any data, and it can be exercised by human-chosen data or by automatic data generation. A theory is expressed in an ordinary programming language, it is easy for developers to use (often even easier than example-based testing), and it serves as a lightweight form of specification. Six case studies demonstrate the utility of theories that generalize existing tests to prevent bugs, clarify intentions, and reveal design problems.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*tools*

General Terms

Verification, Human Factors

Keywords

Theories, JUnit, testing, partial specification

1. Introduction

The traditional practice of testing is to execute software on a few specific inputs and to check that the software's output or behavior is as expected for each input; often, this process is automated in a testing framework. For example, a developer could write a JUnit[1] test for a square root routine:

```
@Test sqrRootExamples() {
  assertTrue(sqrRoot(4) == 2);
  assertTrue(sqrRoot(9) == 3);
}
```

Such example-based testing can be very effective at exposing misunderstandings of the code, where a developer believes the code to behave in one way but in fact it does something different. Tests that can be run during software development (due to use of regression testing or test-driven development [10]) can be especially valuable for quickly detecting code errors and clarifying designs.

Because it only uses developer-supplied data, example-based testing falls short in two ways. First, the tester may overlook important inputs, so bugs may remain in unexplored regions of the input space. Second, future maintainers and clients may have insufficient information to correctly generalize the intended behavior. Continuing the square root example, what is the intended behavior when `sqrRoot` receives a negative input? Are inputs that are not perfect squares permitted, and if so, how accurate is `sqrRoot`? May `sqrRoot` ever return a negative square root?

We propose that testing frameworks should support, in addition to example-based tests, the writing and execution of *theories*,[2] which are partial specifications of program behavior. Theories are written much like like test methods, but are universally quantified: all of the theory's assertions must hold for all arguments that satisfy the assumptions. Our implementation of theories appears in JUnit (version 4.4 and later). A developer using JUnit can write:

```
@Theory defnOfSquareRoot(double n) {
  // Assumptions
  assumeTrue(n >= 0);

  double result = sqrRoot(n) * sqrRoot(n);

  // Assertions
  assertEquals(n, result, /* precision: */ 0.01);
  assertTrue(result >= 0);
}
```

The `assumeTrue` clause states the assumption of the theory: `sqrRoot` is intended to work on any non-negative double. If

[1]`http://www.junit.org`

[2]Theories are named by analogy with the scientific method: whereas a test is a single repeatable experiment, a theory is a general statement that may be invalidated by the right future experiment. To paraphrase Karl Popper [17], "Good test [data] kills flawed theories."

`sqrRoot` has defined behavior on negative inputs, this may be expressed in a separate theory.

A theory can be viewed in several ways. It is a universally-quantified ("for-all") assertion, as contrasted to an assertion about a specific datum. It is a generalization of a set of example-based tests. It is a (possibly partial) specification of the method under test.

Likewise, a theory can be used in several ways. It can be executed on a developer-provided set of data points. It can be explored by automated tools to discover new data points. It can be read by maintainers as documentation of the original developer's intent.

The goal of our research is to help developers find more errors without expending any additional effort. We believe that theories can be as easy to write as example-based tests (especially when complemented by automatic data generation tools such as the one we have built: see Section 3.2) Theories permit developers to express as little or as much of the intended software behavior as they wish. Developers need not interrupt their thought process to create a specific example or a long series of examples, unless that is the easiest way to approach the problem at the time. In our experience (see Section 4), theories were the natural mode of expressing a unit's behavior twice as often as example-based tests.

Beyond our implementation of theory execution in JUnit, we have developed Theory Explorer, a tool that identifies overlooked theory inputs. We have collected experiences from a development project using JUnit theories in test-driven development. Finally, we have written theories and used Theory Explorer to investigate released versions of several popular open-source projects, finding bugs that initial development with theories would have prevented with no additional effort.

Section 2 describes how the tests that developers write often relate to the specifications they have in their heads. Section 3 details the features of our tools for writing, running, and exploring theories. Section 4 reports the patterns of development observed from one project using theories, and Section 5 shows how theories would have benefited several open source projects. Section 6 reviews related work. We then wrap up with section 7.

2. Patterns of representing specifications in tests

A developer beginning to write an example-based test for a feature may have in mind only a single example of what should occur ("The absolute value of -3 is 3"), or a specification that applies to many concrete values ("The absolute value of a positive number is itself"), or all possible values ("The absolute value of any number is non-negative", or "The absolute value of x is equal to the absolute value of -x").

If the developer has a specific example in mind, an example-based unit testing framework can capture it well. If, however, the developer has a more general specification in mind, she must translate it to example-based tests. Sometimes, this translation from general specification to specific tests can be enlightening: the developer may find out that the specification in mind was inadequate to fully determine the proper result for a particular input. However, this translation loses information: the individual tests may suggest, but cannot state or prove, how the feature should behave when presented inputs other than those tested.

Reviewing the testing literature and existing JUnit tests, we have found a number of ways that general specifications are translated into JUnit tests. Software projects referenced below are further described in Section 5

1. A *happy path* test provides one example of proper execution, but leaves room for bugs in error handling and processing unexpected input.

2. *Simple triangulation* [1] provides several different input/output pairs that clarify the specification of a procedure or method. The first test below appears to intend that `foo` is the identity function. Adding the second makes it clearer that `foo` may be the absolute value function:

```
@Test public void fooThree() {
  assertEquals(3, foo(3));
}

@Test public void fooNegThree() {
  assertEquals(3, foo(-3));
}
```

   This approach can be seen in Apache Commons Validator's `ByteTest`, which has test methods `testByte`, `testByteMin`, `testByteBeyondMin`, `testByteMax`, `testByte-BeyondMax`, and `testByteFailure`.

3. An *abstract test case* defines a common contract to be upheld by several different instances of a class, in an abstract class such as this:

```
public abstract class AbstractObjectTest {
  public abstract Object getInstance();
  @Test public void equalsItself() {
    Object obj = getInstance();
    assertTrue(obj.equals(obj));
  }
}
```

   Concrete test fixture classes can be extended from this abstract base, overriding `getInstance()` to provide new objects to be checked against the contract. In its most common form, each class that implements the contract will be tested by a single concrete subclass of the abstract base. This pattern is seen in the Apache Commons Collections project, in classes like `AbstractTest-Object`, `AbstractTestBag`, and `AbstractTestSortedBag`.

4. Data-driven testing. Sometimes the desired behavior of a feature can be easily represented as a table of related data in a database, file, or in-memory data structure. This table can be looped through, asserting that the desired relationship between the data (for example, calling a certain function on the inputs leads to the outputs). A test written this way has little duplication, new tests are easily added by augmenting the table, and scanning the table may be useful in understanding the method, if the rows are well chosen and well arranged.

   There are many possible implementations of this pattern, including an explicit loop in a standard test method (such as used in Jetty's `BufferTest.testBuffer`, which loops over three different implementations of `Buffer`), a separate table-driven framework like FIT[3], or explicit support for data-driven tests in the most recent versions of frameworks like JUnit or TestNG.[4]

Both the abstract test case pattern and data-driven testing pattern separate the data being tested from the assertions being made about the data. Each can be used to support one of two different styles of providing test data. A *provided-answer style* depends on the desired output being explicitly provided along with each set of input. This assertion method uses provided-answer-style data:

```
public void assertAdditionWorks(int x, int y, int z) {
   assertEquals(z, x + y);
}
```

Given a provided-answer-style test data set, it is impossible to automatically determine additional input parameters, without referencing an external specification or reference implementation.

A *specification-style* data set is used to verify a self-contained specification, which contains within itself a description of the correct behavior:

```
public void additionIsInverseOfSubtraction(int x, int y) {
   assertEquals(x, (x + y) - y);
}
```

A specification-style data set can be infinitely extended: the assertion should hold over the entire valid input space, and the data chosen is recognized as an interesting subset. This means that specification-style test data could be extended by automatic tools, given a common language for specifications and test data. *Theories* are written in just such a language, and are described in the next section.

3. Theory implementation

We have created two tools that support Java developers who wish to write theories. JUnit 4.4 supports writing and evaluating theories against defined data points. Theory Explorer explores theories to find new violating data points.

3.1 JUnit's theory runner

We have added the ability to write and evaluate theories in the core JUnit library, starting with version 4.4. A theory is written as a method annotated with `@Theory`, and taking one or more parameters. The method must be defined in a class annotated with `@RunWith(Theories.class)`, which can contain both theory methods and traditional test methods intermixed.

The body of a theory method states an assertion that is universally quantified over every possible type-compatible assignment of all the parameters. Theories hold over infinite data sets, but test execution should be deterministic and fast, so JUnit executes theories on a finite set of data points written by the developer. These data points must be defined in the same class, or one of its superclasses, as the values of fields annotated with `@DataPoint`.[5]

[3]http://fit.c2.com

[4]http://www.testng.org

[5]Data points can also be gathered from the return values of methods annotated with `@DataPoint` or the elements of arrays in fields annotated with `@DataPoints`. Finally, parameters can be annotated with a custom value supplier, which can use an arbitrary input-generation strategy (for an example, see Section 4).

For example, this theory says that every `Object` equals itself, and it will be tested on a `String`, an empty `List`, and a `Date`:

```
@RunWith(Theories.class) {
  @DataPoint public static String DIGITS = "0123456789";
  @DataPoint public static List EMPTY = new ArrayList();
  @DataPoint public static Date TODAY = new Date();

  @Theory public void equalsItself(Object object) {
    assertTrue(object.equals(object));
  }
}
```

Assumptions may be added to the beginning of a theory method to further restrict which inputs are valid for evaluating the assertion:

```
@Theory void equalObjectsEqualHashes(Object a, Object b) {
  assumeTrue(a.equals(b));
  assertTrue(a.hashCode() == b.hashCode());
}
```

If an `assume*` call fails, it throws an `AssumptionViolatedException`. JUnit intercepts this exception, and silently moves on to the next datapoint.[6]

With assumptions in place, the specification for every theory method m is simple:

> For any parameters p_i with appropriate types, $m(p_1, p_2, ...)$ either returns normally or throws an `AssumptionViolatedException`.

To verify that this specification holds, JUnit invokes the theory with every possible supplied data point, and takes one of three possible actions:

- If *all* possible parameter assignments violate the method's assumptions, the entire theory is marked as invalid, to prompt the developer to look for valid inputs.
- If *some* parameter assignment causes an assertion to fail, or an exception to be thrown, the failure is indicated to the developer.
- If the theory passes on all assignments, the theory is marked as passing.

We have also enriched JUnit's support for example-based tests with features originally intended to support theories. JUnit tests can use assumptions to state the external conditions on which a test relies, such as implicit inputs to the tested example, which in theories may sometimes be made explicit.

[6]The most common assumption is that the parameters to a theory are not `null`, for example `assumeNotNull(a, b)`. Future versions of the theory tools will allow this assumption to be implicitly applied in appropriate situations. The current versions do require it to be explicitly included, but we have left it out of our example theories for readability.

```
@Test public void filenameIncludesUsername() {
   assumeTrue(File.separatorChar == '/');
   assertEquals("configfiles/optimus.cfg",
                new User("optimus").configFileName());
}
```

Here, the test assumes it is being run on a system that uses a particular file path format. A test may also assume it is being run while connected to the Internet, or against a particular version of the domain code. When the assumption does not hold, the test is "ignored", indicating that the assertions are meaningless in the current context.

Developers using previous JUnit versions have often written their own custom assertion methods, such as `assertValidEmailAddress`. It is cumbersome to require each such assertion method to have several corresponding assumption methods, such as `assumeValidEmailAddress` and `assumeNotValidEmailAddress`. To make this easier, JUnit has adopted the `Matcher` interface from the Hamcrest project[7]. This allows all assertions and assumptions to be made as comparisons between a tested value and a testing Matcher, using either the `assertThat` or `assumeThat` methods:

```
assertThat(output, validEmailAddress());
assumeThat(output, not(validEmailAddress()));
assertThat(output, both(validEmailAddress())
                   .and(containsString(".com")));
```

JUnit is widely used by Java developers — anecdotal evidence from industry research [15] indicates that JUnit is used by about 50% of end-user companies, 75% of open-source projects, and 100% of independent software vendors (ISV's). Adding support for writing theories in JUnit has the possibility of substantially raising the number of mainstream developers accustomed to writing and reading partial specifications, and amplifying the usefulness of new and existing tools for verifying those specifications.

3.2 Theory Exploration

Theory *exploration* is an automated process of discovering new data points that violate a theory. This is accomplished using a new tool, separate from JUnit, called *Theory Explorer*. When invoked on a theory, Theory Explorer uses an input generator to find inputs that will cause the theory to fail. If a failing input is found, the developer can add it to the accepted data points, and determine how to make the theory pass. This may mean fixing a bug in the tested code, or making an implicit assumption from the implementation explicit in the theory. By iterating through the developer's theories until no violating inputs are found, the theories are made more precise, bugs are found, and the developers' confidence increases. Theory Explorer can also optionally report inputs which cause the theory to pass, but which exercise new paths through the tested code.

The input generator for Theory Explorer uses the Agitator engine [2] from Agitar Software through its free web service, JUnit Factory[8]. Traditionally Agitator is used to test implementation code. By contrast, we used its ability to analyze code and create test inputs to search for theory violations.

The Agitator engine generates test inputs to achieve basis path coverage, while attempting to exercise every possible outcome of theory execution (normal and assertion-violating). The Agitator engine uses a combination of static and dynamic analyses to iteratively explore a particular unit of code. The dynamic analysis uses data values and coverage information from the preceding iteration to direct execution along a particular code path. The static analysis uses heuristic-driven path analysis to collect and solve input value constraints that can cause execution to take a different path from the one previously explored.

To achieve scalability and performance, the Agitator engine makes several simplifications to traditional test-input generation algorithms found in literature. These simplifications are supported by profiling and experimentation. For instance, rather than trying to solve constraints for the entire execution path, Agitator may consider only a part of that path. While such an approach may not always direct execution down the expected path, experimentation shows it works well in practice and cuts down analysis time. Also, Agitator unrolls loops and recursive calls until the control-flow graph reaches a the size experimentally determined to give the best trade-off between the completeness of the results and Agitator's performance.

Agitator solves path constraints using a wide array of generic and specialized constraint solvers. For instance, Agitator includes a string solver for `java.lang.String` objects, producing strings that satisfy constraints imposed by the String API (e.g., a string that `.matches(...)` a regular expression). Similarly, Agitator includes several solvers for the Java Collections classes.

In addition to using constraint solving and other sophisticated methods for generating test inputs, Agitator uses a few heuristics that often allow it to explore otherwise unreachable paths. Some of these include:

- For integer values Agitator tries to use 0, 1, -1, any constants found in the source code, the negation of each such constant, and each constant incremented and decremented by 1.
- For string values Agitator generates random text of varying lengths, varying combinations of alpha, numeric, and alphanumeric values, empty strings, null strings, any specific string constants occurring in the source code.
- For composite objects Agitator achieves interesting object states by calling mutator methods with various auto-generated arguments.
- For objects in general, Agitator tries to reuse some of the instances created during execution of the program. Agitator uses heuristics to capture (and discard over time) objects that are created during dynamic analysis.

In some cases, the Agitator engine lacks the information to construct an object, or to put an object in the specific state required to test a class. In those cases, users can define "helper" methods that construct objects to seed test-input generation.

3.3 Process

Developers who already use tests should find that theories fit well into their workflow, with two modifications. First,

[7]`http://code.google.com/p/hamcrest/`
[8]`http://junitfactory.com`

when a developer would have written a test for a feature, it may be written either as an example-based test, a new theory, or a data point for an existing theory, depending on which best captures the developer's intent.

Second, before writing a new test for a feature or checking in the feature's code, all theories for that feature should be explored to make sure that no unexpected data points will violate the existing specifications. If violations are found, just one of the violating data points should be added to the set of accepted data points. The developer should then work on getting the theories to pass: the violating data point may indicate a bug in the implementation that needs to be fixed, or an assumption of the implementation that needs to be explicitly stated in the theories.

Theories can be an especially useful addition to the toolkit of a developer practicing test-driven development. A traditional starting point for test-driven development [1] is a very simple failing example such as this:

```
@Test public void reverseAbc() {
  assertEquals("edcba", reverse("abcde"));
}
```

Although this is an evocative example for a human reader, it can be easily made to pass by a method that always returns `"edcba"`, or a method that sorts the characters in descending ASCII value. While such an implementation is obviously wrong from inspection, it is often the case that well-meaning maintainers can make such mistakes if they do not understand the original intent of a method, or do not read the tests. Another test goes a long way toward preventing such errors.

With previous JUnit versions, the developer's next step may have been to introduce a new example reversing a different string, say `"esuoh"`. It is more expressive, and useful, to instead state a general theory, like this:

```
@Test public void reverseMakesTheFirstLast(String s) {
  assertTrue(reverse(s).endsWith(s.substring(0, 1)));
}
```

Exploring this theory will automatically add three more data points like these:

```
@DataPoint String STRING_2007_09_13_161154 = "testString";
@DataPoint String STRING_2007_09_13_161154_1 = "";
@DataPoint String STRING_2007_09_13_161154_2 = null;
```

This leads the developer to improve the definition of `reverse`, and also add new assumptions that `s` is neither null nor empty, leading to new tests for these corner cases.

4. Test-Driven Development with Theories

To evaluate the use of theories (and to guide development of the tools), we used theories in the development of Glinda, a utility for reading and analyzing a log of developer activity. Glinda displayes reports that can show that a developer is running tests less often than the their own stated goal, or that the number of compile warnings has increased since yesterday.

Glinda is developed using test-driven development [10], including theories (see Section 3.3): before starting any programming task such as adding new functionality or fixing a bug, the Glinda developer first writes a new test or theory. This test or theory must fail at the time of writing, but pass after the task is complete.

After two months of part-time development, Glinda consists of 1700 non-test, non-comment, non-blank lines of code. The test suite contains 51 acceptance tests (which assert the exact textual output after reading an example log file, and are therefore example-based) and 88 unit tests, of which 29 are example-based tests (JUnit `@Test` annotations) and 59 (67%) are theories (JUnit `@Theory` annotations).

We had assumed that a minority of unit tests would be theories, and were surprised to find the opposite. Writing theories was often easier than writing tests.

For example, a `Goal` such as "Run tests every hour" can be *paused*, indicating that the developer has suspended the goal to achieve some other, more urgent task. While this condition holds, any line in a status report about this goal should contain the goal's name, and a `'z'` character that serves as a reminder that the goal should be eventually *resumed*. This is true regardless of the previous state of the goal and the time at which the status is requested. This is expressed in this theory:

```
@Theory pausedStatus(GlindaTimePoint t, Goal goal) {
  goal.pause();
  assertThat(goal.status(t), containsStrings('z', goal));
}
```

Parameterizing over the time of the request and the goal is a convenience for the developer, who does not have to deal with the tedium of choosing a specific time point or goal name. It also makes the theory more expressive, and enables Theory Explorer to find more bugs. Likewise, it's easier to assert a few substrings from the desired result than to construct the entire correct status line. It's also more robust in the face of changes to the status line format that should not concern this theory. Those changes can be reflected in a single example-based test like this:

```
@Test
public void pauseProjects() {
  // input
  read("#startGoal SecurityAudit");
  read("#pause SecurityAudit");

  run("status");

  // output
  assertWrote("status:");
  assertWrote("z SecurityAudit (paused)");
}
```

When testing with theories, designs that make writing theories easier are more attractive. For example, Glinda's log scanning and parsing functionality is encapsulated in less than 300 lines of code, meaning that the remainder of the model classes never deal with any unparsed strings. While this is desirable in a well-designed system, we found that we were much more vigilant in maintaining this division of responsibilities than we have been in similar, earlier systems. When only example-based tests are available, being able to pass literal strings to domain code can actually make the tests easier to write. With theories, one would rather quantify by the parsed data type than make complicated assumptions about the format of a String.

Theory Explorer helped to push important design decisions earlier. For example, we originally wrote one theory that said that any goal in good standing should have a '>' prefix character in its status line. Another theory (shown above) says that any paused goal should have a 'z' in its status line. Theory Explorer quickly suggested a goal that was both paused and in good standing, which must violate one theory or the other. With example-based tests, this conflict in assumptions might have persisted much longer.

We now present some additional observations based on our use of theories.

Example-based tests triangulate to theories. We often started with a few concrete examples, especially when the final specification was not obvious early in development, or when the specification was not easy to fully capture. Then, we generalized from those examples to a theory.

A simple example was this test for the `toString` method on a Matcher for a time interval:

```
@Test atLeastToString() {
  assertThat(atLeast(hours(3)).before(2007jan1()).toString(),
            is("at least 3 hours before 2007-01-01 12:00:00"));
}
```

Providing a second hard-coded example would be tedious, would add nothing to the human understanding of the method, and would be unlikely to cover important corner cases, so we next added this theory:

```
@Theory allDataInAtLeastToString(int h, GlindaTimePoint date) {
  assertThat(atLeast(hours(h)).before(date).toString(),
            containsStrings(hours, date.getYear()));
}
```

We retained or added example-based tests even after writing theories. An example can quickly lead a reader to an intuitive grasp of the desired functionality, but a theory helps make clear which aspects of the example are necessary, and which are arbitrary.

Declarative and constructive theories. Typically, a theory is *declarative*: it states a property that holds for any inputs to a method that satisfy certain assumptions. Alternately, a theory can be *constructive*: it constructs a specific concrete input for one or more of a method's parameters. A constructive theory can be thought of as combining example-based testing (use of specific values for some parameters, with the attendant loss of generality) with theories (use of arbitrary values for other parameters). Constructive theories were most valuable when the given object was complex, when multiple objects needed to be in a certain relation to one another, or when it was not possible to concisely describe the value's state.

Our theories constructed inputs more often for the receiver than for other parameters. Here is an example of a constructive theory that creates a `Goal` and then mutates it into a desired state.

```
@Theory trackedValuesPersist(GlindaTimePoint now,
                             String goalName, double value) {
  Goal goal = new Goal(goalName);
  goal.track(now.minus(DAYLENGTH).minus(hours(1)), value);
  assertThat(goal.yesterdaysValue(now).value(), is(value));
}
```

Using data points in multiple theories. A data point that was useful for one test class (for example, a string with a space in it) often revealed bugs in other classes. JUnit 4.4 allows each test class to have its own set of datapoints to be checked against theories, but most of our test classes inherit from a base class, `GlobalDataPoints`, which contains 14 data points representing several important data types in our system. Some test subclasses add data points to exercise their particular theories. Since JUnit automatically throws out any data point that fails a theory's assumptions, providing extra data points does not produce false failures, although too many data points could conceivably slow down execution of a large test suite.

Stub generation. In some cases, Glinda's analysis depends on input from the developer beyond what is indicated in the log. For example, Glinda may ask the developer whether a particular task in the log benefited a particular stakeholder, in order to determine which stakeholders are benefiting most from the developer's efforts.

When testing features that require user interaction, it is too expensive to require a human tester to provide all the desired answers. Thus, all user questions are submitted to an instance of the `Correspondent` interface, which may ask the user for an answer, or calculate the answer in some other way.

Traditionally, a test of a method that depends on the Correspondent interface would use a hand-written stub implementation. However, understanding such a test requires understanding the stub, which gets in the way of understanding the actually-tested functionality. For example, it is not obvious what the result of this call should be, without learning more about `StubCorrespondent`:

```
new StubCorrespondent("bug137", "true").getAnswer("bug138");
```

Instead, the Glinda test suite registers a custom parameter value supplier (named Diabol) with JUnit to generate values for parameters annotated with `@Stub`. This enables a theory like this:

```
@Theory public void asksToEvaluateBenefit(
               @Stub Correspondent correspondent) {
  assumeThat(correspondent.getAnswer(
               "Does the king benefit from bug137?",
               IdeaList.INTEREST_ANSWERS),
             is("true"));

  book.idea("bug137");
  book.setCorrespondent(correspondent);
  assertThat(book.evaluateBenefit("the king"),
             hasItem(containsString("true")));
}
```

First, Diabol supplies a `Correspondent` to the theory that simply guesses at the right return value to each method call. If the supplied guess fails an assumption, JUnit supplies Diabol with the details of the value supplied and the assumption that failed, available from the parameters to the `assumeThat` method. The custom runner builds a new `Correspondent` based on this feedback, which will make guesses that are more likely to satisfy the theory's assumptions. This iteration continues until either all assumptions are satisfied or the value supplier gives up.

We call this method of generating collaborators for theories "diabolic execution". It is similar to concolic execution, as proposed by Sen et al. [20], but uses run-time dynamic information instead of static analysis, allowing it to be run on-the-fly during test execution.

In future work, we hope to make this automatic stub generator generally available outside of Glinda.

5. Case Studies in Bug Finding

Section 4 discussed our experience using theories during a software development project. This section presents five additional case studies of projects that were developed without theories, but with JUnit tests. We wish to answer whether use of theories could have improved the code quality of these projects without requiring any more effort.

Our methodology is to examine the project's existing JUnit tests, looking for sets of tests that obviously suggest an easily-stated general principle. In other words, we looked for a test or set of tests that easily generalized to a corresponding theory. We rejected any theory that would have taken more time to write than the original example-based tests. We found that theories were easiest to write against code with a loosely-coupled, object-oriented design, and that clear contracts in the code were often reflected in clear specifications translated to tests. Averaged across all code, we found that about about 25% of tests were immediately generalizable to theories. Again, we did not write any new tests or theories, but simply wrote theory versions of already-existing tests.

We then ran Theory Explorer to find violating data points. In some cases, the data points indicated assumptions that were implicit in the test, but needed to be made explicit for the theory. For example, a specification-style data-driven test for implementations of the Buffer class in Jetty (Section 5.3) looped over three Buffer implementations. By construction, each had a capacity of 10 bytes, and the corresponding theory failed if any larger or smaller Buffer was submitted. So the theory needed an assumption stating that the capacity was 10 bytes.

We worked for four hours or until we found a new, undocumented bug. In three of the cases, a bug was found in under three hours, with no more than five theories written; two of these bugs were unknown to the projects' maintainers. In the other two, characteristics of the test suite and the domain code resulted in either no tests that could be made into theories, or theories that our Theory Explorer could not generate input for.

Three hours may seem a long time to find a bug. However, we spent most of the time understanding the code base and test suite, which we were seeing for the very first time. The theories theories were in many cases smaller than the tests they were based on. Our experience suggests the following:

1. Many developers do think in terms of general specifications and contracts, which translate more easily to theories than to specific tests.

2. Writing theories, when appropriate, need take no longer than writing the tests a developer would have written without theories available.

3. In some cases, writing and exploring a theory will catch a bug that would otherwise have been released to users.

5.1 Time and Money

Time and Money[9] is a library implementing time and currency domains in a fluent interface based on Domain-Driven Design.[7]. We began looking at tests for implementations of `Interval`. An `Interval` defines a range of values between two endpoints, which may be inclusive or exclusive. We found a test that asserted that the complement of a particular Interval relative to itself is empty:

```
public void testRelativeComplementEqual() {
  Interval c1_7c = Interval.closed(new Integer(1),
                                   new Integer(7));
  List complement = c1_7c.complementRelativeTo(c1_7c);
  assertEquals(0, complement.size());
}
```

This is true not only of closed intervals between 1 and 7, but of any Interval. The theory version was easier to write than the test:

```
@Theory
public void relativeComplementOfSelfIsEmpty(Interval i) {
  List complement = i.complementRelativeTo(i);
  assertEquals(0, complement.size());
}
```

Theory Explorer found a violating `Interval`, an instance of the subclass `ConcreteCalendarInterval` whose lower limit is higher than its upper limit. The implementation of `Interval` maintains the invariant that the lower limit of Interval is less than the upper limit of Interval, which is assumed throughout its implementation. `ConcreteCalendarInterval` circumvents this protection by calling a protected constructor of `Interval` that is marked as follows:

```
//Only for use by persistence mapping frameworks
//<rant>These methods break encapsulation
//and we put them in here begrudgingly</rant>
```

This bug has been submitted to the maintainers of Time and Money.

5.2 Commons Collections

Apache Commons Collections[10] enhances the JDK collection classes by providing new interfaces, implementations, and utilities. Our study focused on implementations of the Bag interface, for collections that have a number of copies of each object. After writing three theories, we discovered a bug. The following theory throws a `ClassCastException` when `bag` is a `TreeBag`, and `item` does not implement `Comparable`:

```
@Theory
public void bagAddItemToContainIt(Bag bag, Object item) {
  try {
    bag.add(item);
  } catch (Exception e) {
    assumeNoException(e);
  }
  assertTrue(bag.contains(item));
}
```

`TreeBag` is a `SortedBag`, so its elements must implement `Comparable`. According to the `Collection` documentation, `contains` may optionally throw a `ClassCastException` if `item` is not of the right type. However, `add` is *required* to throw an exception under the same conditions, and it does not. Thus, `bag` can be put into an inconsistent state. This bug has been submitted to, and accepted by, the Collections committers.

[9]`http://timeandmoney.domainlanguage.com/`

[10]`http://commons.apache.org/collections/`

5.3 Jetty

Jetty[11] is a HTTP Server and Servlet container. We looked at the tests written for implementations of the `BufferCache` cache and `Buffer` interface for read-write buffers of bytes and the and used those to write five theories. Many of the tests for the Buffer classes were written as specification-style data-driven tests, making theorization easy.

We found a bug in `View`, which implements `Buffer`. `View.compact` has an empty implementation (consisting only of the comment "`// TODO`"), which means that the underlying buffer is uncompacted, leading future calls to the View to break their contract. Since View is not supposed to change the underlying buffer, it should probably throw an `UnsupportedOperationException`. Instead, it silently accepts the method call, leading clients to falsely believe the state has changed. This problem is not indicated in the documentation or bug database, and the uninformative comment does not make it clear if the developer realized that leaving the implementation blank violated the `Buffer` contract. We notified the developers and asked for verification.

We found no bugs in `BufferCache`, but replacing the tests with theories made them much more readable. The existing tests constructed an array and a long string that had nontrivial relationships to each other: all elements of the array were in the string *except* `array[0]`, and the string in `array[i]` could always be found at index `string[i*2-2]`. By generalizing over all String arrays, explicitly constructing a large string containing the array elements, and making separate theories for cache hits vs. cache misses, the intent was made much clearer to future maintainers.

Finally, writing theories for the `Buffer` implementations exposed the fact that it is very easy in Jetty to construct a Buffer with an inconsistent internal state: for example, with its read index larger than the total size. While this is a common design decision made for performance reasons, it can lead to confusing bugs-at-a-distance for clients: developing with theories may have led the Jetty developers to either disallow constructing Buffers with inconsistent states, or at least provide an `isInValidState` method to simplify clients and tests.

5.4 Commons Validator

The Apache Commons Validator[12] project provides a configurable engine for validating user input against a set of validation rules, which may vary by locale.

Validator has few tests that are easily translated into theories. Although Validator uses the abstract test case pattern in several places, (such as an AbstractNumberValidatorTest superclass of all NumberValidator implementations) the tests are written in a provided-answer style, rather than a specification style. This is a reasonable choice given the largely procedural code being tested, but it isn't the only possible choice. If, for example, the e-mail validation tests had been written with theories based on the standard (RFC 822), the following theory could have been written, and read as "an e-mail address can contain a space only if it the address also contains quotes":

```
@Theory addressWithSpaceHasQuotes(String address) {
  assumeThat(address, containsString(' '));
  assumeTrue(validEmailAddress(address));
  assertThat(address, containsString('"'));
}
```

[11]http://www.mortbay.org/

[12]http://commons.apache.org/validator/

5.5 Eclipse JDT

We looked at the QuickAssist functionality of the Eclipse IDE's Java Development Tools[13]. This code has deep dependencies on static state and would require heavy mocking to isolate and test. Although the input generator for Theory Explorer does support creating mock objects for some collaborators, the theories written for the JDT would require many more mocked method calls than are allowed by Agitator's heuristic limits.

There are two lessons learned from this. First, handling code that requires more extensive mocking is an important challenge for future work. However, the untestability of the code is more a function of its structure: to get one of the objects on which this feature depends requires a chain of seven static method calls, which would be even harder for a human developer to find than an automated tool. Thus, an improved design would be easier to use by future programmers, and easier to manually and automatically test.

6. Related work

Design by contract [14] (DbC) and, more recently, contract-driven development [13] attempt to make it easy and profitable for developers to write general specifications for program components, by writing preconditions and postconditions on methods that are enforced at runtime. Java developers can use a suite of tools based on the Java Modeling Language (JML) [12] to write and verify contracts in Java comments. The jmlunit [3] tool is similar to the JUnit theory runner, in that it loops through a set of user-supplied data points to look for contract violations.

However, whereas a contract must describe the behavior of every conceivable invocation of a method, a theory can tackle just the subset the developer wishes to think about now. Theories can construct, rather than describe, some of their inputs, possibly using mutators (see Section 4), an option not available in a contract. Theories can be independently "scoped" by assumptions, whereas general contracts are required (by design) to apply in all cases. Finally, theories are specified in a separate code unit, where they can be added, removed, executed, hypothesized, read, and explored without changing or cluttering the implementation code. These advantages make it easier for a developer's intentions to be written and verified in theories than in contracts.

Theories are closely related to parameterized unit tests [21] (PUT), first investigated by Tillman and Schulte as a technique for unit test generation. A PUT is very similar to what we have called a theory: a true statement parameterized over possible inputs. Tillman and Schulte found that using symbolic execution and an exhaustive set of PUT's for all of a method's collaborators, a minimum set of test cases can be generated that will exercise all code paths in the tested implementation. By writing fresh PUTs based on documentation, they found three bugs in the implementation of collection classes from the .NET Framework. They do not compare the difficulty of writing the complete set of

[13]http://www.eclipse.org/jdt/

required PUTs to creating the original tests for the framework, or whether product groups had the skills to write effective PUTs.

Where Tillman and Schulte generate provably minimal test suites based on complete specifications, we accept heuristics that generate data points designed to exercise as many code paths as possible in a short time. This allows us to drop the requirement for complete specification of a method's collaborators. We are also using JUnit's theory support, which is built into a popular, widely available unit-testing framework, and enables several syntactical shortcuts for convenience, including reusing a single declared datapoint for many different theory parameters. By minimizing the effort required to begin seeing feedback from the technique, it may become more profitable for a wider range of developers and programs.

Specification-based static analysis tools have long been a favorite topic for research. Tools such as Alloy [9], Bogor [18], JPF [11], and ESC/Java [8] traditionally rely on a separate specification language or built-in contracts to detect bugs in the code. Built-in contracts provide a useful but limited form of specification, mostly describing obviously faulty conditions: deadlocks, in-line assertion violations, or uncaught exceptions. Programmer-written specifications can be more revealing in catching errors, but are rarely supplied. Theories offer a mid-point: while not as expressive as most specification languages, they are designed to be lightweight and more palatable to programmers.

Random testing, most recently implemented for Java by Pacheco et al. in a tool called Randoop [16], offers a technique for searching large input spaces for contract violations. The contracts checked by Randoop consist of a handful of built-in invariants, such as checking the reflexivity of the `equals` method. JCrasher [5], another random-testing tool for Java, checks only a single contract that no method is permitted to throw an undeclared runtime exception. Both Randoop and JCrasher can benefit from the existence of theories, functioning essentially as tools for theory exploration. The QuickCheck tool for Haskell [4] applies random inputs to developer-written specifications to find errors. However, it does not incorporate example-based testing as JUnit does.

Generating stubs for an object's collaborators, as our Diabol value generator does (see Section 4), is a recurring theme in unit test generation. Stubs may be generated from observing program executions [19], JML specifications [6], or symbolic execution of the test that uses them [22].[14] One advantage of Diabol is that the assumptions it attempts to use for stub generation can be re-used to verify hand-written stubs, or assumption-passing data points from the data pool can be used as stubs, if the automatic generation fails.

7. Conclusion

We have found bugs that resulted from overlooked valid inputs in open-source projects. Each bug violates a specification that can be easily guessed from examining the examples chosen in unit tests written by the developer. It is very likely that writing a theory would have required no more effort than writing the tests, and using a tool like Theory Explorer would have caused these bugs to be recognized and fixed before release.

[14]Many of these generated test doubles are called mocks, but a true mock object verifies the calls made to it, which is only done in [19]. The others actually generate stubs, as does Diabol.

Many efforts have been proposed to make writing specifications profitable for developers. By building theories into JUnit, we have made writing a specification no harder than, and as immediately useful as, writing a test. An exploration tool, Theory Explorer, uses an intelligent, heuristically-driven input generator to find oversights, amplifying the developer's testing effort.

We enjoy developing with theories, and are continuing to find new ways to use them, such as Diabol, a natural way to generate stubs. We look forward to learning more from other developers who begin using the theory support already in JUnit, and the upcoming Theory Explorer release.

8. Acknowledgement

This work is supported by an Agitar Research Fellowship.

9. References

[1] K. Beck. *Test-Driven Development: By Example.* Addison-Wesley, Boston, 2002.

[2] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA, 2006. ACM Press.

[3] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer, 2002.

[4] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP).* ACM SIGPLAN, 2000.

[5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *SPE 2004*, 34(11):1025–1050, Sept. 2004.

[6] X. Deng, Robby, and J. Hatcliff. Kiasan/kunit: Automatic test case generation and analysis feedback for open object-oriented systems. Technical report, Kansas State University, 2007.

[7] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2004.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[9] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[10] D. Janzen and H. Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*,

38(9):43–50, 2005.

[11] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[12] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[13] A. Leitner, I. Ciupa, O. Manuel, B. Meyer, and A. Fiva. Contract driven development = test driven development: writing test cases. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the european software engineering conference and the 14th ACM SIGSOFT symposium on Foundations of software engineering*, pages 425–434, New York, NY, USA, 2007. ACM Press.

[14] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[15] T. Murphy. personal communication from Research Director with Gartner, 2007.

[16] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[17] K. R. Popper. *Knowledge and the Mind-Body Problem*. Routledge, 1993.

[18] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276, New York, NY, USA, 2003. ACM Press.

[19] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 114–123, New York, NY, USA, 2005. ACM Press.

[20] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, 2005.

[21] N. Tillmann and W. Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, 2005.

[22] N. Tillmann and W. Schulte. Mock-object generation with behavior. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 365–368, Washington, DC, USA, 2006. IEEE Computer Society.