

The Practice of Theories: Adding “For-all” Statements to “There-Exists” Tests

David Saff Marat Boshernitsan

Abstract

Traditional unit tests in test-driven development compare a few concrete example executions against the developer’s definition of correct behavior. However, a developer knows more about how a program should behave than can be expressed through concrete examples. These general insights can be captured as *theories*, which precisely express software properties over potentially infinite sets of values. Combining tests with theories allows developers to say what they mean, and guarantee that their code is intuitively correct, with less effort. The consistent format of theories enables automatic tools to generate or discover values that violate these properties, discovering bugs that developers didn’t think to test for.

The Practice of Theories: Adding “For-all” Statements to “There-Exists” Tests

I. INTRODUCTION

Test-driven development (TDD) works because it allows a developer to specify the intended behavior for their software through simple examples. This has several important benefits. First, it clarifies the task in the developer’s mind and enables thinking about the design prior to committing that design to code. Second, it allows fast automated support for confirming that the implementation works. Third, it provides a reusable set of regression tests to detect future changes to the system’s behavior.

A developer, however, knows more about how a program should behave than can be expressed as a set of a few concrete examples. There are behaviors that can be easily and precisely described, but cannot be captured as a simple example, because they pertain to a large, or infinite, set of concrete executions. For example:

- Converting from a decimal representation of any integer to Roman Numerals and back should preserve the integer’s value.
- Persisting and restoring any object of class A should always yield an object equal to the original.
- Adding a dollar to any currency collection should always produce a new collection not equal to the first.
- If two Java objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result. [7]

At its core, the practice of test-driven development offers a way to organize the development workflow by going from specific examples of the system’s behavior (tests) to general statements about that behavior (implementation). In this article, we introduce an extension to traditional test-driven development that enables developers also to make general statements at the level of tests. We call these statements *theories*. Theories and tests work together to describe class behavior and detect bugs.

In our preliminary experiments with theories, we observed that they can empower developers in several important ways.

- Theories facilitate the development of complete abstract interfaces by allowing the developer to remain at the level of domain abstraction such as numerical systems or currencies.
- Theories enable reuse of the test code, since many traditional specific tests can be recast as instantiations of a specific theory.
- Theories afford the use of automated testing tools to amplify the effects of a theory by testing a theory with multiple data points.

We will introduce Theories by working through a familiar example: the development of a system that performs calculations on monetary values, expressed as amounts of any given national currency. This domain will be familiar to many TDD practitioners from its use both in Kent Beck's *Test-Driven Development* [1] and the JUnit documentation [2].

II. THE FIRST TEST: CHEATING ON THE IMPLEMENTATION

The developer of a multi-currency calculator might start by writing a test of multiplying a dollar amount by an integer coefficient as follows (Here and elsewhere the tests are written in Java and appear without their class context. We use JUnit 4 notation to specify tests by prefixing them with `@Test`):

```
@Test public void multiplyDollarsByInteger() {
    assertEquals(10, new Dollar(5).times(2).getAmount());
}
```

The simplest way to get this test working is to fake the implementations of `times()` and `getAmount()`.

```
class Dollar {
    public Dollar times(int multiplier) {
        return this; // TODO: temporary cheat
    }

    public int getAmount() {
        return 10; // TODO: temporary cheat
    }
}
```

This is a familiar pattern to TDD practitioners: in Beck’s book it is called “fake it.” By writing the obviously fake implementation, and watching it pass, we have learned two things:

- 1) The difference between this test passing and failing must include working definitions of `getAmount()` and `times()`.
- 2) Our current test suite can not distinguish the correct implementations of `getAmount()` and `times()` from our current cheating implementation.

III. THE SECOND TEST: DEFENSE BY TRIANGULATION

There are three common ways of moving from a cheating implementation to the real implementation:

- 1) *The obvious implementation:* For simple methods such as the ones in our example, typing in what appears to be the correct implementation may take very little more time than typing in a fake implementation. A developer can save time by just typing in the obvious implementation first.
- 2) *Look for duplication:* Duplicated data or logic can direct us to the places where the design is lacking in generality. In this case, the number 10 is duplicated between the test and production code, likely indicating that our implementation will fail to give correct answers other than 10.
- 3) *Triangulation:* We can introduce a second test that will fail with the current incorrect implementations.

The first two mechanisms have several disadvantages. There is no good algorithm that can declare an implementation “obviously correct.” Duplication can be hard to spot—clone detection tools exist, but they often miss important examples of duplicated logic. Triangulation, on the other hand, not only gives us a finish line for declaring the cheat removed, but also will catch any further regressions back to the special-case code. It is certainly true that no responsible developer would leave such a degenerate implementation of `getAmount()` in the code. However, a few interacting bugs can work together to cause an almost-correct implementation to have the same behavior for such a limited input set.

Applying triangulation to our example, we might choose to extend our test like this:

```
@Test public void multiplyDollarsByInteger() {
    assertEquals(10, new Dollar(5).times(2).getAmount());
}
```

```

assertEquals(15, new Dollar(5).times(3).getAmount());
}

```

This test demonstrates some disadvantages of triangulation. It takes extra work to write the second test, which does not necessarily add any additional information for the human readers. After writing the new test, we may remain concerned that the implementation still only works correctly for a subset of the intended values. Equivalence partitioning of input values can help to produce confidence that all equivalence classes are tested through at least one member (for example, negative numbers, zero, positive numbers, and `Integer.MAX_VALUE`), but again, there is limited automated support for being sure that the equivalence partitioning guarantees a correct implementation.

IV. THE NEXT THOUSAND TESTS: THEORIES

We know what we want to say: currency multiplication performs the same kind of function on money that integer multiplication does on integers, for *all* integers, not just 2, 3, and 5. This is easy enough to express in code:

```

public void multiplyAnyAmountByInteger(int amount, int multiplier) {
    assertEquals(amount * multiplier,
        new Dollar(amount).times(multiplier).getAmount());
}

```

Once this is written, the existing test can refer to it:

```

@Test public void multiplyDollarsByInteger() {
    multiplyAnyAmountByInteger(5, 2);
    multiplyAnyAmountByInteger(5, 3);
}

```

This simple *Extract Method* refactoring of the test code has given us something very valuable. Beyond just a couple of inputs known to be correctly handled, we can state that if our program is correct, *no* possible invocation of `multiplyAnyAmountByInteger`, with *any* integer parameters, can throw an assertion error. (This is in contrast to data-driven testing, or acceptance testing frameworks like FIT, which often work on a table of example inputs and outputs, but make no guarantees about behavior outside of the enumerated examples.)

The method `multiplyAnyAmountByInteger` is one example of a *theory*, a parameterized method whose only required behavior is that it returns normally for any arguments, never throwing an assertion error or other exception. A theory, by our terms, is a general statement about a set of values; like a scientific theory, it may not be possible to prove a theory correct without trying every possible combination of conditions, but it only takes one failing experiment to prove a Theory incorrect. By contrast, a test is a single statement about a single set of values: like a single scientific experiment, the results should be repeatable and deterministic. Alternatively, a theory can be thought of as a template for generating tests by instantiating with particular values (this is closer to the semantics in [11]). Below, we will use `@Theory` to mark methods that should be read as theories, first for documentation purposes, and then to enable automated support.

Once we have a theory about `times()`, we can generate further tests for it simply by manually adding more invocations to `multiplyAnyAmountByInteger` with interesting parameters. We can also use automated tools to analyze the code and suggest values that can disprove the theory. We will come back to the subject of tools later in this article, but now we need to address a problem with our `multiplyAnyAmountByInteger` theory.

V. FIXING THE THEORY: DUPLICATION AND ABSTRACTION

Experience has taught testers to be concerned about duplication between test logic and tested logic. Consider again our theory and the correct definition of `times()`:

```
public void multiplyAnyAmountByInteger(int amount, int multiplier) {
    assertEquals(amount * multiplier,
                 new Dollar(amount).times(multiplier).getAmount());
}
// ...
public void times(int multiplier) {
    return new Dollar(this.amount * multiplier);
}
```

Here, we have explicit duplication: the logic `amount * multiplier` appears both in the theory and in the definition of `times()`. Essentially, the theory is doing little more than inlining `times()`. At first glance, this may seem unavoidable: How can we define the correct

behavior of `times()` without referring to integer multiplication? But the duplication is actually an indication that we are mixing abstraction layers. The theory should only involve currencies, and operations on currencies. By performing “algebra” on the two sides of the equation in our theory, we can eliminate the old theory, and replace it with a more useful one:

```
@Theory public void multiplyIsInverseOfDivide(int amount, int multiplier) {
    assertEquals(amount,
        new Dollar(amount).times(multiplier)
            .divideBy(multiplier)
            .getAmount());
}
```

Here, we have introduced a new method into the API: `divideBy()`. This allows us to write an identity that is expressed entirely at the level of currency operations. In our experience, good theories are usually identities of some kind. If a method must be added to write the identity, it usually fills an obvious gap in the existing API.

The higher the abstraction layer that a theory talks about, the more generally useful it can be. For example, we can write theories that range over `Dollars`, regardless of how they are constructed (assuming that `Dollar` equality is correctly defined using `equals()`):

```
@Theory public void multiplyIsInverseOfDivide(Dollar amount, int multiplier) {
    assertEquals(amount, amount.times(multiplier).divideBy(multiplier));
}
```

Once a generic `Money` class is written, which encompasses amounts in dollars, Swiss francs, and combinations thereof, it is more powerful again to be able to assert this identity about all kinds of cash collections:

```
@Theory public void multiplyIsInverseOfDivide(Money amount, int multiplier) {
    assertEquals(amount, amount.times(multiplier).divideBy(multiplier));
}
```

If our theories gain power as they grow more general, should we do away with specific tests? No; they are essential for triangulation. Consider our first test:

```
@Test public void multiplyDollarsByInteger() {
    assertEquals(10, new Dollar(5).times(2).getAmount());
}
```

Without this test, we would be unable to catch a regression that reimplemented `times()` and `divideBy()` as addition and subtraction, or any other inverse function pair. The added value of this test is that 10 is a *human-checked* answer to 5 times 2, which is correct no matter what underlying operator is used in the implementation. By making tests very specific, and theories very general, the behavior of the resulting code is constrained to the maximum.

VI. TESTING, EXPLORING, AND ASSUMPTIONS

Since a theory can pertain to a potentially infinite set of values, how can we include evaluation in the workflow of test-driven development, which requires rapid execution of a small, finite number of tests? We recommend that testing frameworks that incorporate theories should allow for the easy specification of a finite set of *accepted data points* that will be used for evaluating theories at testing time.

(As an aside, this separation of test artifacts into theories and data points allows for new kinds of reuse between test suites. For example, a framework author can publish theories indicating the contract that must be met by the implementors of a framework interface. The author can also publish data points that represent interesting states of the framework classes that client code should be prepared to deal with.)

Once a method passes all the currently accepted data points, the developer will want to explore for any further data points that might disprove the theories. In our running example, exploration will reveal that we overlooked that our theory could apply to the multiplier 0. Since we have no intention of defining division by 0, we refine our theory by introducing an *assumption*. Assumptions restrict the domain of inputs to which the theory applies:

```
@Theory public void multiplyAnyAmountByInteger(int amount, int multiplier) {
    assumeNotEquals(0, amount);
    assertEquals(amount,
        new Dollar(amount).times(multiplier)
            .divideBy(multiplier)
            .getAmount());
}
```

When an `assume*` method receives non-matching arguments, it terminates the current theory execution, because the assertions will not provide useful information. Using assumptions,

developers can create theories that apply to sets of values arbitrarily finer than possible with static types alone. Assumptions also help communicate the intent of the theory to the human reader.

For example, assumptions can be very expressive and convenient for describing a class's collaborators. Consider a theory about converting one currency to another:

```
@Theory public void exchangeOneUnitOfCurrency(String fromCurrency,
                                             String toCurrency,
                                             int rate) {

    Bank bank = new Bank();
    bank.addExchangeRate(fromCurrency, toCurrency, rate);
    assertEquals(new Money(rate, toCurrency),
                 new Money(1, fromCurrency).exchangeTo(toCurrency, bank));
}
```

Here, we are constructing a concrete bank, and adding the appropriate exchange rate. Making this test more general, however, has several advantages:

```
@Theory public void exchangeOneUnitOfCurrency(String fromCurrency,
                                             String toCurrency,
                                             Bank bank,
                                             int rate) {

    assumeEquals(rate, bank.getExchangeRate(fromCurrency, toCurrency));
    assertEquals(new Money(rate, toCurrency),
                 new Money(1, fromCurrency).exchangeTo(toCurrency, bank));
}
```

Now, the theory has no dependencies on `addExchangeRate()`, reducing the number of bugs that could cause the theory to fail. It is much clearer to the theory reader that `exchangeTo()` depends on the result of `Bank.getExchangeRate()`. And we are implicitly testing that there are no further dependencies: we can run this theory on many Banks, whose only common behavior is the exchange rate returned for these two currencies.

VII. AUTOMATED SUPPORT FOR THEORY EXPLORATION

We find that the right time to explore for new data points is right before writing a new test or theory: this gives us confidence that all the existing theories are valid before continuing.

We may choose to do this by hand, through inspection, equivalence partitioning, etc. However, this is an excellent opportunity for automated tools, which can incorporate arbitrary levels of sophistication while searching for values that can cause violations in a theory.

For our own experiments we have created a *theory runner*, an extended JUnit 4 runner to simplify the evaluation of new data points at testing time. The theory runner considers all fields on a test object that are annotated with `@DataPoint` as values for any theory that takes an appropriately-typed parameter. For finer control, the developer can use assumptions—the theory runner catches and ignores assumption violations. If none of the supplied data points satisfies the theory’s assumptions, the runner will mark the theory as failing, prompting the developer to explore for acceptable data points.

For non-primitive data types that are constrained using `assume*` methods, the theory runner attempts to automatically generate dynamic stubs using JMock, a popular library for specifying stub objects for testing. As a result, the theory runner creates a data point that will pass that theory’s assumption, but the theory itself need only be concerned that some such implementation exists, not how it was constructed. Automatic stub generation gives developers many of the design benefits of a mocking framework, while entirely hiding the framework’s API.

The theory runner provides automated support for exploring new combinations of human-suggested data points. However, it does not help in finding new data points that might violate a theory. For this purpose we used Agitator [3], a tool for exploratory testing of Java programs created by Agitar Software (www.agitar.com).

Agitator uses a combination of static and dynamic analyses to create method-level test inputs that thoroughly exercise the code, achieving a high degree of data and state coverage. This process is called *software agitation*, and it allows Agitator to record observations about code’s behavior that can be presented to the user for analysis.

Traditionally Agitator is used to test the implementation code. By contrast, we used its ability to analyze code and create test inputs to agitate theory code and search for theory violations. Agitator uses a combination of symbolic execution, constraint solving, and heuristic-and feedback-driven input generation to discover “interesting” data points inputs where a theory might fail. Agitator also includes automated generation of mock objects, though it does not provide support for the `assume*` constructs. If we find a violation while agitating a theory, we record a new data point for the test runner and proceed to fixing the implementation or to

refining our theory. The new data point can become part of the regression testing suite by using Agitator’s JUnit generation engine—all interesting data points can be automatically preserved as generated JUnit tests that invoke our theory.

VIII. EXPERIENCE

The practice of theories was developed as part of the in-house framework for our ongoing test-driven development of a UI-intensive plug-in for Eclipse. We found that there were general properties, especially about input validation and the proper collaboration of asynchronous processes, that were easy to communicate, but hard to test.

One of the authors had recently written and tested an XML persistence layer, which further showed us the value of writing theories as identities. Persisting and then restoring an object should always result in the equivalent object:

```
@Theory public void canSaveAndRestore(IXMLPersistable persistable) {
    assertEquals(persistable, Persistence.fromXML(persistable.toXML()));
}
```

We found that a majority of the tests we had written could be rephrased as instantiations of this single theory, reducing the size of the testing code and making the intent clearer. The persistence code also became simpler, because we now had a single method for retrieving an arbitrary object from an XML stream. This showed that theories could exert the same positive design pressures on the implementation as tests in TDD.

Theories have made it easier to write well-behaved Java classes. For example, we catch a lot of errors in `hashCode()` implementations, one of our frequent downfalls, from just once writing this theory, encoding a feature of the `Object` contract specified in the javadoc for `hashCode()`:

```
@Theory public void hashCodeIsInSyncWithEquals(Object a, Object b) {
    assumeEquals(a, b);
    assertEquals(a.hashCode(), b.hashCode());
}
```

Theory exploration has proved useful in identifying bugs early. In one memorable case, a seemingly correct implementation of conversion between integers and roman numerals turned out to fail only on inputs between 21 and 29, which had not been previously tested.

However, we are still discovering the possibilities and the limitations of theories, and significant questions remain to be answered. How will theories scale if used from the beginning of a large project? Will developers already familiar with TDD find writing theories natural? Will developers not familiar with TDD find theories comprehensible? With enough data points, will the combinatorial possibilities of instantiating a theory bring testing to a crawl, and how could such a slow down be addressed?

IX. THEORY SIGHTINGS IN THE WILD: RELATED WORK

The idea of passing parameters to tests can be traced to data-driven testing, which uses parameterized statements that are similar in form to theories. The inputs for data-driven testing are read from a database or other file, but theories can specify desired behavior for an infinite set of inputs. Also, some data-driven tests can be replaced by theories, which can improve the documentation, by explaining why these particular inputs are valid, and no others.

The Abstract Test pattern [5] encourages reusing test cases for multiple implementations of an interface. The abstract test asserts that the interface's contract is met, and concrete subclasses supply the implementations of the interface that should be validated. Composite Unit Testing [4] anticipated the idea of expressing these contracts as parameterized tests, and dynamically supplying the parameters. We see the same encapsulation represented in theories, but with more flexibility of expression, and the ability to have tests vary over much more than just the runtime type of an interface implementation.

The theories that we produce are similar in intent to the Parameterized Unit Tests proposed by Tillman and Schulte [11]. Their UnitMester tool strives for a provably complete set of data points, where we are content with bounded exploration, which can be useful even when fully specifying an object's dependencies is impossible. In addition, while that work appears to be most concerned with generating minimal test sets for exercising the theories, we are most concerned with how the practice of development with theories works in the overall programming workflow.

The ability to make universal statements about method behavior is a primary feature of Design-by-Contract [8]. However, design by contract is difficult to fit into test-driven development, since it requires that methods be fully specified, rather than incrementally tested into existence, and it's hard to use pre-conditions and post-conditions to assert identities that can only be expressed using more than one method invocation (such as `multiplyIsInverseOfDivide` above.)

If developers choose to create a model of their system in a formal specification language like Alloy [6], they may be able to use a tool like TestEra [9] to generate test cases automatically, exploring the potential state space for violations of representation invariants. Theories can be thought of as lighter-weight partial models, partially bridging the gap between testing and formal methods.

X. CONCLUSION

The benefits of TDD depend on the simplicity of the practice itself and of most popular testing frameworks. Is the increased complexity of theories worth their benefits? We look forward to more empirical evaluations, but our initial experience leads us to believe there's a lot of promise. Developers know general things about their code, and theories allow them to specify these quickly and in just one place, without having to internally translate the general principle into a host of individual test cases.

Theories have helped us to:

- define new useful operations and abstractions that we did not discover through testing alone,
- write isolated tests with clearer intent than mock objects allow,
- simply enforce interface contracts, and
- catch bugs that we did not think to test for.

We continue using theories in our own development, and look forward to feedback from other experimental practitioners of theories.

XI. ACKNOWLEDGMENTS

We would like to thank Kent Beck, Steve Freeman, and Alberto Savoia for their feedback on drafts of this article. Mark Pilgrim's thought-provoking phrasing of the requirements for roman numeral conversion [10] was one early catalyst for this work.

REFERENCES

- [1] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, Boston, 2002.
- [2] K. Beck and E. Gamma. JUnit test infected: Programmers love writing tests. <http://junit.sourceforge.net/doc/testinfected/testing.htm>.
- [3] 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.

- [4] J. de Halleux. Composite unit testing with MbUnit. <http://www.codeproject.com/cs/design/jdhcompositeunittesting.asp>, 2004.
- [5] E. George. Testing interface compliance with abstract test. <http://www.placebosoft.com/abstract-test.html>, 2002.
- [6] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
- [7] Java 2 platform API documentation. <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html>, 2003.
- [8] J.-M. Jézéquel and B. Meyer. Design by contract: The lessons of Ariane. *Computer*, 30(1):129–130, 1997.
- [9] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. *ase*, 00:22, 2001.
- [10] M. Pilgrim. *Dive Into Python*. Apress, 2004.
- [11] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.