Testing Fundamentals

Colin Shaw

Why Test

First, let us briefly touch on the kernel objectives of testing. It isn't for the love of the test, but born of practicality to create more maintainable software that closely follows a known specification. Automated testing helps provide confidence that software does not regress when it is being altered or updated, and helps expedite software releases while never breaking the production code line. From specification to the end of life of a software program, testing is an important aspect of development that augments the specification, provides documentation, assists in development, allows for confidence in functionality, facilitates rapid changes, and reduces cost, adding to the software's proposition of value.

Preliminaries

Testing occurs at all levels of development. Before we get into the types of testing, let's assume a few definitions that will be used later:

Open-box testing: Testing that is performed with the objective of determining internal properties of a system, such as the type of the superclass of an object or whether or not a method is defined on a class, would be called open-box tests. This type of test is most important for testing that interfaces work as expected and generally are low level tests. Open-box tests are not relevant to functional and system testing.

Closed-box testing: Testing that does not assume interest in the internal mechanism is known as a closed box test. Closed-box tests are used to observe how a system reacts to input. As an example, a closed-box test would not be interested in testing the signature of a particular method, that is the domain of an open-box test. Rather, the closed-box test would be interested in what return value the method produces for a known collection of arguments. Closed-box tests articulate how the system behaves and are used for testing the working conditions of the software, including edge cases. Closed-box tests are found in all levels of testing, but are exclusive to functional and system tests.

Bottom-up testing: Testing that is performed starting at the smallest level, unit tests. This type of testing, in general, catches more errors than any other type of testing. It is also the simplest to integrate at the developer level, which allows for more frequent testing to prevent regression. Bottom-up testing is extremely useful in answering questions of specific failure, but cannot answer questions of broader functionality.

Top-down testing: Testing that is performed at the most coarse level. This type of testing is useful in finding gross errors in control. Top-down testing has the weakness that errors are not pinpointed. That is to say, top-down testing may indicate that there is an error, but the level of testing is not specific enough to answer the question where the error came from and why.

End-to-end testing: A test that is designed to run across the entire technology stack used in a given project. That is, simulating the user experience in an application that uses every component that a user would use (e.g. interface, middleware, database, etc.).

Agile Development: A collection of practices that are intended to address the part of human behavior that makes initial specification more difficult than iterated specification. That is, it is simpler to observe what a system does and improve it than to initially specify the entire system. Agile development is an awareness of this and a group of practices, such as short iteration cycle, enhanced team communication and an emphasis on testing, that are essentially best practices for developing evolving software.

Stories: In agile development, customer features are often described by stories. The story is a simple, nearly atomic, operation that is a desire of a customer. Complex stories are broken into multiple simplified stories. The story is both a contract for a deliverable feature as well as a parcel of work that can be estimated and tracked as part of the larger project in order to manage the project.

Continuous Integration: The process of integrating changes with the current production code line, whereupon local changes are tested automatically before committing, and automatically deployed to a separate testing build machine after committing, is termed continuous integration.

Types Of Testing

There are a wide range of definitions for various types of testing. In the below discourse, we will address only those types of tests that are commonly used in ensuring quality of software projects having external specification.

Unit Testing: The finest granularity tests are called unit tests. These tests occur at the class or module level. The level for unit testing is most desirable to be as confined to a basic element of organization as possible. In object oriented software, the natural scope is the class. In other paradigms of programming that are organized differently, we use the term module (in the conversational sense more than the semantic sense) to denote something akin to a class, or at least an interpretable collection of functionality that embodies a single logical premise. The reason for this is to confine the tests as much as possible, as the class (or module), if well designed, should do one and only one thing well (SOLID might be a better thought, at least for object oriented designs). This fine-grained testing organization allows for sufficient edge cases and interface tests to be written simply in order to fully specify the software under test. Though there are many opinions on the matter, it is fairly common to have mocked objects in unit tests (we will get to mocks, stubs and so forth when we talk about testing design patterns; for now we are using the term mock in the more conversational sense, as a stand-in for

an external dependency). One reason for the use of mocks is related to part of the purpose of unit tests, which is to test types and interfaces. Another reason for this is to isolate complex dependent objects to more effectively test the unit in question. That is to say, the mocking essentially is dependency injection with an alternative to that which would be present in the actual integrated system. Reducing dependency produces more robust tests that do not fail for reasons out of the scope of the unit test. Reducing the dependencies also allows for faster tests. Faster tests allow for more, and more frequent, tests. Unit tests are both a convenience for developers that allow for the development of code that is more compliant with specification as well as more resilient to challenges while changing or refactoring later. From a systems perspective, unit tests are also the canary furthest in the mine and are the most important first line assertion that the software is performing properly, at least from a bottom-up approach. They are also generally the least brittle as they have the least amount of dependencies. It is common for people writing unit tests to call these tests specifications, or simply specs. This terminology borrows from Behavior Driven Testing (addressed later), and amounts to a call-out to the purpose of the tests, which is a technical translation of the business specifications for the software under test. Unit tests are a combination of open-box and closed-box tests.

Integration Testing: What is called integration testing is testing that is beyond the scope of unit testing. As the name implies, it is testing of the integration of smaller pieces. As the tests become more integrated, the mocks will slowly start to vanish. This depends, of course, on what is being mocked, but in general the components that were created as stand-in replacements will begin to resemble the system as a whole more. Interfaces representing things such as databases will likely still be mocked, but the connectivity between various modules (units, if you will) will be able to be tested in their composite. Integration tests are tests that are technical in nature, involving internally used representations in addition to customer stories. That is to say, they are a combination open-box and closed-box tests. Integration tests can run locally, depending on the scope of work that programmers are tasked with, or can be run on a separate server. The use of the term integration in Integration Testing and in Continuous Integration are fairly easy to confuse; Integration Testing is a somewhat vague notion of testing integrated software components at a scale above unit testing, whereas Continuous Integration is a process for helping ensure quality throughout the development lifecycle inclusive of system testing (defined below).

Verification and Validation: What is termed verification and validation testing is a type of testing that generally operates at the level of integration testing that has twofold purpose. Verification is a step where humans make sure that the specifications for the software are in accord with the product that is being produced. That is to say, verification is the answer to whether or not the product is being produced correctly. Validation, on the other hand, is a coded step that confirms (or denies) that the product

conforms to the specifications. That is, validation is the answer to whether or not the correct product is being produced.

Functional Testing: Functional testing is testing that is closed-box only. As the name implies, it deals with functions and functionality, not with internal system details. The term is commonly used nearly synonymously with system testing and acceptance testing (see below) because these types of tests are closed-box and deal with system functionality. However, there is no reason not to use the term with unit or integration tests if they are purely closed-box (though this may be rather rare, particularly in object oriented designs).

System Testing: What is termed system testing is an approach to testing the entirety of the system, often with mock data, from beginning to end. System testing is a final stage of testing within the development portion of the project. This generally means using the same interface an end user might use and doing the same sort of operations an end user might perform. However, the testing remains somewhat technical, in that the objectives are not purely user stories, though are closely tied to them and of the same scope. System test environments are generally separate from other systems, though may share common resources such as a database with mock data. It is fairly common for the mock data to be a full or partial backup of production data. System tests are closed-box tests and, due to scope, can be relatively brittle.

Acceptance Testing: What is termed acceptance testing is an old systems engineering term that refers to asserting whether or not a system complies with its specification after completion and before hand-off to the customer. Historically, acceptance testing was a final stage in a contract between the buyer and the engineering firm, and at the interface of the team that built the product and the customer that desired the product. Interestingly, the industries that generally had acceptance testing as part of their contracts, typically large enterprises in conventional engineering such as aerospace, tended to be in an environment that was both rather litigious and demanding of proof of engineering quality, which is somewhat different than in software. In agile software development, acceptance tests are built around the notion of user stories, which are fine-grained articulations about function that a business user would understand. An example of a story might be "a user will be able to change his/her password," and the corresponding acceptance test would log in as a test user, instigate a password change, log out, and attempt to log back in using the new password. Once all of the acceptance tests pass, the software is deemed to be functioning in a fully specified manner as dictated by the user stories. Acceptance testing differs from system testing in that the goal of the particular test is to demonstrate a user story rather than test technical qualities of the system. Acceptance test environments are generally separate from other systems and, like system testing environments, use mock data that is often derived from production data. Acceptance tests are closed-box tests and, due to scope, can be relatively brittle.

Automated Testing: Automated tests are tests that are run automatically. That is, they are not manual. Unit tests can be automated; for example, if the unit tests are run when code is saved or as a check as a pre-commit hook so that code that has unit tests not passing is not a permissible commit. Functional tests can be automated. Generally this is either a scheduled process or occurs as a post-commit hook. The way that functional testing is automated as part of a process for ensuring quality software, particularly as relates to implementation in a process such as asserting proper function as part of a post-commit hook, is called continuous integration. That is, when a developer makes a successful commit (e.g. it passed unit tests as a pre-commit condition), it would then build the project with the new code as a post-commit task and run a battery of functional tests that mimic the acceptance criteria as part of a continuous process that integrates new code into a working production environment.

Regression Testing: Regression tests are tests that are in place to catch failures. All tests are in place in part as regression tests. The notion of regression testing is that a test will indicate whether or not a new feature or refactored existing code breaks existing functionality. A good suite of tests will be able to demonstrate that the functionality either remains the same or that the functionality has suffered in some way. With sufficient test coverage, even small errors will translate into failing tests. It is, of course, imperative that new functionality have relevant new tests added so that later work will enjoy the benefits of regression tests related to the new functionality.

Load Testing: Load tests are tests that are run in an environment to simulate the complete system performing typical operations under heavy load. These are important to determine how the system performs in worst case load situations. Load testing is almost a field unto itself, as it is a different mindset to consider what the most important aspects of the system to test may be. Generally speaking, load testing is mostly performed against parts of systems (for example, database performance) and full systems that require sufficient availability in a high load situation. As well, metrics having to do with loaded performance are generally a goal of the load tests, the result being actionable information about performance and scaling in the loaded environment.

There are numerous other types of testing that generally are intended for more specific purpose applications than the ones discussed here. The ones mentioned above are applicable to most any software project that involves intense development with specified objectives, with testing at typical levels, both from the software and business perspectives. Some examples having fairly obvious scope include: usability testing, accessibility testing, security testing, A/B testing, and many others. While there are certain situations where more specific knowledge of each of these, and other, types of testing are useful, in practice there are tenets of many of these types of tests that are part of the larger path from customer specification to public software release as defined by the above types of tests.

Why Unit Testing Is So Important

Unit testing is important for several reasons. First, it provides documentation of the specification of the software. Unit tests satisfy a broad superset of assertions that not only convey the desired customer requirements, but also how the software is designed to implement the desired system. A customer does not care if a security model does or does not return an object of the right type; however, as prudent software developers we care that it does. At the level of unit testing, the tests involve business logic as well as implementation logic.

Second, unit testing quickly assesses whether or not changes to an existing, functioning system maintain or break that system. Unit tests are constrained to unit both because that is their natural scope, but also to make the test suite fast so that it is run frequently. Some people prefer to set up their environment so that unit tests are run whenever a change is made to the file system. This is an extreme, but it is very clear almost immediately if a change has altered compliance with the specification. The faster this feedback, the faster corrections can be made, the more efficient and free of failure development and refactoring is, and the more confidence we have that our changes do what we want them to and nothing more.

Third, unit testing is a litmus test for how the system itself will be designed. That is to say, when tests are being written, one comes to consider the implementation of other code, particularly object oriented code, as a component of a larger system in a manner that is immediately indicative of good (SOLID) design practices. Good unit tests are very hard to write for sloppy code that does not abstract concepts in a mature manner. Simply attempting to write unit tests after the fact is very good indication of the quality of the code being tested.

Unit tests are the most fundamental bottom-up test. They are the first observation that new code is passing a specification. They are also the first line of defense against regressions when adding to or refactoring code. They are the developer's best friend for ensuring that changes not only meet specification, but do not break specification. They are also handy documentation of what the specification is. Moreover, unit tests provide regression testing against updates to systems that reside below the level of the software, such as language updates or hardware updates. Errors arising in these cases can be very difficult to find without a body of tests that can indicate regression at the lowest level in application code.

Two Philosophies Of Unit Testing

There are two fundamental philosophies of unit testing (and to a degree testing in general). These are writing the software and then tests, and writing the tests and then the software. The latter, often called Test Driven Development, or TDD, was popularized by Kent Beck in the Extreme Programming movement, which aimed at producing higher quality software faster by coercing better adherence to design objectives by coding for the design objectives. This is a fundamentally different notion than

writing tests afterwards, as driving code from the tests amounts to writing the business specification in programmer-speak first and then producing the minimum product that satisfies these tests.

As was mentioned in the prior section about benefits of unit testing, the TDD approach has some interesting advantages. First, it assists in focusing on the work of implementing the features that are desired. This is courtesy of producing formalized specification in the form of tests prior to beginning the actual code development. Before the development begins, there is a complete list of features that need implementation, some notion of the interfaces that these features should use, and the ability to verify that a new feature passes or fails the specification. As noted, the specification for interfaces is well-defined, implying consideration of the design of interfaces and classes at the time of writing the tests. There is advantage in this, as it forces consideration of good design practices, particularly regarding interfaces, which positively influences the design of the code to be tested. Especially in object oriented development, unit tests are much simpler when the design of the underlying code adheres to SOLID principles and separation of concerns. TDD not only specifies the software first as a blueprint of development that is easily referenced while developing software and minimizes feature addition to only those features that are part of the specification embedded in the tests, but it also aids in strengthening the design of the system that is being tested by enforcing good design practices before coding is started.

Comprehensive unit tests enforce specifications on the underlying software. Software that has good unit test coverage is easier to maintain because it is clear if changes offer regression or not. Refactoring such code is simpler as well, leading to an enhanced ability to continually improve the code base without regression. Not only this, but the design considerations for implementing unit tests imply that the underlying software, particularly object oriented software, admits effective design consideration that strengthens the overall system. Test Driven Development helps force better implementation by having a specification written first; this practice alone aids in driving the design of classes that adhere more to SOLID principles and have more distinct separation of concerns.

Unit Testing Example

Just for fun, let's use Ruby since it is concise, expressive, and has interesting constructs that we can use for open-box testing as well as closed-box testing. For our purposes here, we will be testing using Ruby's minitest, which includes functionality in the xUnit family of unit testing frameworks. For the sake of simplicity since we are discussing TDD as well as unit testing in general, we will begin by writing some tests. First, however, we will start by discussing the problem so that we know what we are going to test. We will phrase the problem as a collection of deliverable items, or stories. These stories are essentially our specification guideline. As our example is basically a class as a deliverable, our stories will be a bit closer to the bare metal than a business user's would normally be:

Stories: As a programmer, I want a class that deals in prime numbers.

As a programmer, I want to be able to find primes in a specific, inclusive range.

As a programmer, I want to be able to tell if a particular number is a prime.

A typical business user story would be more like "as a user, I want to be able to log in using a password." The stories that we have written for this example, as mentioned previously, are much closer to the implementation than typical stories. Part of the problem in developing tests is determining what the tests are as relate to the implementation based on the more course requirements given by the user story.

In this example, we can glean three things from the stories as they are written. We will ultimately have a class that provides the functionality in question. For simplicity and demonstration of certain testing features, we will be using the normal object instantiation. We will want to test that our object is the type we were actually intending and we will also like to test the signatures for the two public methods that we know we will have. By writing these stories, both the testing and the ultimate code are considered in the context of design in advance of being written, and we know, aside from specific implementation, what both will look like before we start writing a single line. For example, it is not clear how the arguments would be written or how they would be returned, or what the return types would be; it is the job of the programmer to make good decisions in the context of the larger system as to what the interface is unless it is specified. For a more complex example, it would be prudent to diagram the interface using UML or similar modeling conventions to make sure that class relationships are appropriate and the design follows good object oriented practices. This sentiment might bear repeating; TDD is difficult in large part because of not having a well-defined interface, unknown method signatures and unknown object interactions. While the test is written first, it is foundational to have a fairly solid design with a known specification prior to writing the tests.

Let's start writing a test with what we know so far:

```
require "minitest/autorun"
require "./prime.rb"

class TestPrime < Minitest::Test
  def setup
    @p = Prime.new
  end

  def test_class_and_methods
    assert_equal "Prime", @p.class.to_s
    assert_equal true, @p.public_methods.include?(:in_range)
    assert_equal true, @p.public_methods.include?(:is_prime?)
  end

  def test_in_range
    assert_equal [], @p.in_range([14,16])
    assert_equal [11], @p.in_range([10,12])
    assert_equal [11,13], @p.in_range([10,14])
  end

  def test_is_prime
    assert_equal true, @p.is_prime?(11)
    assert_equal false, @p.is_prime?(10)
  end
end
```

For those not exposed to Ruby, this code simply requires minitest and our prime class code, then extends the appropriate minitest class. Minitest is a fairly encompassing testing framework, hence selecting a specific aspect of it to extend (as opposed to aspects for Behavior Driven Development). We then fill in some methods of our choosing with respect to setup and teardown (in this case simply instantiating our Prime object prior to each test), and create tests based around methods beginning with "test." We have various assertions about the result of our tests. @ is simply an instance variable, syntactic sugar for what some languages require explicit getter and setter methods for. :in_range is a symbol; symbols are used extensively in Ruby for names since they are hashes and are O(1) for lookups. There are plenty of methods that are included in xUnit implementations that allow for before and after each test, before and after all tests, a wealth of variations on the assertions, and many other features. It is useful to consult the documentation of your particular xUnit implementation to find the proper setup that is relevant to the needs of your testing.

In this example, we wrote tests, covering in the exact order the stories making up the open-box and closed-box tests of our software. Indeed, it is quite exhaustive, testing every reasonable case of inclusion in an array for the in_range() method, and both cases of the is_prime?() method. Our first test,

having to do with the class and methods, is not at all that important in this case. It is meant to fulfill the role of an example for open-box testing, but would indeed be useful in cases where the class in question employs inheritance. In this case, it is somewhat irrelevant what the class is (it would have failed the setup() step if we got it wrong), and the existence of the public methods will be tested in the tests focusing on those methods. If there is inheritance, open-box testing becomes more important. This is an important point, as inheritance is basically what object oriented programming is all about (and polymorphism), as these are the kernel of all abstraction and reuse; it should be important to test types in many situations, though the particulars of what to test depend in part on the language and how it treats typing. Weakly typed languages and languages that tend to favor duck typing perhaps require a different degree and style of testing in this regard than do languages that are strongly typed.

Now that we have both come up with a reasonable interface and written our tests, we can start writing the actual code. Since we wrote the tests first and had to come up with a reasonable design, and as such wrote tests for the closed-box aspects, we are well on our way to knowing what the code will look like. Since we defined the method that produces primes in an inclusive range, it simplifies the method that determines if a number is prime or not significantly. Here is a passing example:

```
class Prime
  def in_range(range)
    result=[]
    (range[0]..range[1]).each do |t|
      result << t if check_prime(t)
    end
    result
  end

  def is_prime?(test)
    in_range([test,test]).length==0?false:true
  end

  private
  def check_prime(test)
    prime=true
    (2..Math::sqrt(test).floor).each do |t|
      prime=false if test%t==0
    end
    prime
  end
end
```

For those not exposed to Ruby, [] is an array and the << method (yes, it is a method, not an operator) adds a value to the array. The (a..b) syntax is syntactic sugar for creating a Range object, which has ordering and is iterable. There is no reason for the moment to dwell on the nature of the do blocks, but

suffice it to say these are closures and in this context bind variables in scope with the Range object and the iterated element of the Range object to the block. Everything else should be fairly standard.

As we were writing this code we were testing it against the tests that we wrote. At the very first, we defined the two public methods that we wanted to use, in_range() and is_prime?(), each taking the arguments that we had already determined. For in_range(), at least in a very naive implementation, we simply want to check the primality of each number in our range, inclusive. It seemed practical to have a private method that would perform this, so we wrote it. It simply checks the possible divisors and emits a false if any of them divide the number we are checking. Since we had a specification where in_range() returns an inclusive set of primes, we could make a one-liner method out of is_prime?() based on the existence of array elements. Easy.

Writing a test to go along with the user code appears initially as extra work. However, one would likely be using some sort of console output to test it while it is being developed as an alternative. When you do this, there is not an indelible record of the tests. In fact, the code is altered and then untested when you pull out these console logs, or lingers as commented out clutter. As Martin Fowler observes, "whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead."[1] When you perform unit testing, you have a record of the specification coded in the test. What is more, there is no reason to manipulate the actual code to add testing, which leads to cleaner code. You also have the ability to ensure that the code works as it should any time you like by simply running the unit tests. In the face of updates or refactoring, there is a baseline functionality on record that should not change. This gives you assurance that your changes are internal, and have not changed the external interface or functionality. To gain a quick sense of why unit tests are useful in quickly finding issues with software, testing these eight assertions took slightly under a millisecond on my development machine; granted it is a trivial example, but it did indicate success in under a thousandth of a second. Also, it doesn't depend on external conditions, such as browser or network conditions, that take longer and often impede robust testing.

As an example of using our tests to help ensure proper function while adding to our code, let us suppose we are frustrated with the performance of finding primes. We want to change our algorithm slightly so that we have better performance. While in reality we should simply replace this entire algorithm with a different one, for the sake of this example we will concern ourselves simply with adding some fast pre-checking to the in_range() method. That is to say, there is no need to check, say, multiples of two and three. We won't get big-O improvement, but we will speed it up a bit. Our new in_range() method might look like this:

[1] Fowler, Martin. Refactoring: Improving the Design of Existing Code.

```
def in_range(range)
    result=[]
    (range[0]..range[1]).each do |t|
      if t%2!=0 && t%3!=0
        result << t if check_prime(t)
      end
    end
    result
end
```

All the while we have been altering this method, we have been able to see that we have not broken the tests. Great, no need for ad hoc console output to see what is going on, we already have a much superior mechanism in place for ensuring that it still works. What's more, we haven't spent as much time attempting to ad hoc test it. However, a prudent programmer would note that with this new addition it might be wise to add some test cases covering a full collection of the multiples of two and three to ensure that all of the edge cases in this arithmetic are appropriately covered. This programmer might alter the unit tests as such:

```
def test_in_range
    assert_equal [], @p.in_range([14,16])
    assert_equal [5], @p.in_range([2,6])
    assert_equal [11,13], @p.in_range([10,14])
end
```

In this case we could just replace our existing test that returns one value with a range test that spans a full cycle of multiples of two and three. It is nice to have the assertions concise but covering the edge cases that are important. It may be of merit in some cases to comment the test cases. It is not always clear to the next person working on it why the test cases are what they are. As an example, would you remember a year from now why the second assertion in the above reworked code has the range argument that it does? It would be good to make that explicit, either by splitting that test case into a new test with a reasonable method name, or writing a comment about it in the current method.

Naturally, we will encounter issues with our software that we may have overlooked as we were writing it. In these cases it is prudent to have the tests, they will help us refactor our code so that it passes current tests and any future tests. Without them, not only are we only hopeful that our new code addresses the newly found problem, but we lose confidence that the code satisfies the conditions that it has been in the past. That is to say, we don't have a recorded specification in the form of tests. Can you spot the error in the example code that is easy to write a new test for, and also would be useful to know that existing tests are passing for while being reworked? It is not really a hard one to spot. As an aside, there is another shortcoming of the example test listed above; this will be addressed in the following sections.

Once we get done with the tests and the coding, we can step back and see if there are improvements to be made or refactoring that could be done to improve it. An example in this case

would be the question of whether or not this code might be nicer as static public methods. It is certainly the case that the two public methods we currently have never, from an external perspective, benefit from instantiating a Prime object. That requires an extra line and seems to serve no purpose. However, the private check_prime() method would either have to be integrated into the in_range() method or the calls to it would have to be refactored to reference the static method. Neither of those options sound all that appealing. Perhaps we consider that this is a very simple example and most of the time we really would like to instantiate the object. After all, in this example, we aren't doing object oriented programming, we are simply organizing methods. All good things to consider, and should be considered. Much of the time it all depends on the larger project, which we must know something about in order to make the best possible design decisions. Perhaps the most concise enumeration of best-practice ideas for writing good tests and good corresponding software come from Miško Hevery of the Google testing group[2], a source of excellent testing practice related quick reads.

It is interesting to consider what the scope of unit tests might be under variations in language. Ruby, for example, and other weakly and dynamically typed languages, are often more amicable to duck typing. It is implied to the user based on context what the interface is, and it is (loosely) checked at runtime by the interpreter. Is it good to have tests for interfaces in a context such as this? Maybe. What about if it were in a strongly and statically typed language? In that case we would not even be able to build an executable unless the interfaces were well defined. What would the implications to testing be? Perhaps a bit different. There are many arguments on this debate, but it is worth noting that language makes a difference on some of the aspects of testing, particularly open-box testing.

One last thing to keep in mind is that as software complexity and size increases, there will be more to test that has object oriented feel to it (as our example does), which can give a false confidence in the tests (that they are a good collection of test cases). This is said as a warning; just because it looks like an object oriented design and has some features of an object oriented design does not mean it was the right design or good software, or admits the best tests. Take our example here to heart: It was created as a simple example for demonstration, but one of its more prominent legacies ought be that it is a misuse of the design paradigm since it effectively is only a collection of static methods, and several of the tests (open-box) that were written are essentially irrelevant since they are effectively tested by the closed box tests. Testing, like software development, is hard, and it is important to be attentive to writing and maintaining good tests.

Behavior Driven Development Testing

In recent times there has been a move to simplify the translation from customer specification to testing specification that is called Behavior Driven Development, or BDD. It should be noted that BDD is not strictly a subset of unit testing, but a change in paradigm with respect to the semantics of test specification for any level of testing. Historically, BDD was an initiative to garner interest from the client

[2] http://googletesting.blogspot.com/2008/08/by-miko-hevery-so-you-decided-to.html

in being able to confirm stories by reading test semantics in as near to natural language as possible. This appears in practice to be relatively hard to implement, as business stakeholders are not, despite the good intentions of the behavior driven development movement, as interested in specifying testing using domain specific languages as in simply requesting that developers write good test suites. That said, it is worth noting what tests of this nature look like, as the semantic mechanics for this type of DSL are part of many testing frameworks. Here we will use Ruby's Cucumber test suite since it is concise and a great example of a BDD testing framework.

First, one writes statements that are palatable to the customer. An example might be:

```
Feature: Prime Testing
  In order to work with prime numbers
  Programmers must be able to use a prime object

  Scenario: Generating primes
    * I have entered [2,5] as an argument
    * The result should be [3]
```

These statements are then to be used in the testing framework, the arguments and the results ultimately being lifted out by regular expression (see below). What is rather elegant about this way of doing things is that there is insulation between the specification and the actual tests. The specification is easily read and understood by anyone. Once the specifications are articulated, the actual tests can be written. These would appear somewhat like this:

```
Before do
  @p = Prime.new
end

Given /I have entered [(\d+),(\d+)] as an argument/ do |i,j|
  @r = @p.in_range([i.to_i,j.to_i])
end

Then /The result should be [(\d+)]/ do |result|
  @r.should == [result.to_i]
end
```

Pretty interesting, isn't it. First of all, the code is written without parenthesis, which is a perfectly fine grammar in Ruby and increases the readability in this context. The argument to the Given() method is a regular expression that creates an English-like DSL, which of course pattern matches against the text that is so easily human readable and is related to the previously written English language specification. Given() also takes a block argument in which the test is actually performed. The flow of the coded part of the test is concise and quite similar to what unit tests would appear as. However, the specification reads essentially in English and the mapping from the specification to the test is rather straightforward given the use of the regular expressions and the syntax used. Even the setup, teardown and test steps

have been given thought enough in method naming to make the coded routine a rather natural-reading DSL. The topical headers in the specification are simply written out in the testing process, allowing you to see the same level of detailed verbosity in the output as in the input.

There is nothing remarkable about behavior driven development tests; they are functionally equivalent to other paradigms of testing. The difference is simply the syntax and use of domain specific design to create enhanced readability with the idea of bringing business decision makers closer to the development. That said, it is more common to see BDD tests in languages that facilitate easy DSL creation. Hence Ruby has a more robust BDD culture than many other languages.

There are arguments against the BDD approach, generally taking the form that the only thing that tests of this sort do is add regular expressions that require maintenance, though most concede value if the person writing the specification is a non-technical project manager. The pragmatic answer to the debate is to use whichever tool makes the most sense in your situation. If your projects are mainly overseen by non-technical project managers, perhaps a BDD test is appropriate; if your projects are mainly overseen by other programmers or technical staff, perhaps xUnit-type test are more appropriate. It is worth noting that this treatment of Behavior Driven Development is more an exposition of what it is in practice, as the tools and the ways of accomplishing it have grown since its inception. In the beginning, BDD was more of a methodology for getting the customer closer to the development cycle, not a specific syntactic convention for writing tests. Whether the tests look the way they do in the example or not, the point is that the distance between the customer specification and the actual test is reduced.

Invoking Unit Tests

Unit tests, and for that matter most other tests (automated system, acceptance, etc.), are invoked by what is termed a test runner. It is quite clear from the preceding examples that the test that you write is not some standalone entity. Rather, it is a middleware of the inversion of control sort that allows you to specify methods in a class, as well as a certain degree of boilerplate and setup, and then, when run, the framework takes care of the heavy lifting. Essentially, testing frameworks are testing-specific DSLs that require the code that you are wanting to test. In the xUnit family of testing tools, there are a lot of common features; however, there can be a lot of differences based on what language is being used and details of making the tests read well in practice.

Features that are common to most of the xUnit variants are the assertions and the ability to perform setup and teardown operations. As well, the general notion of the inversion of control DSL that allows you to define the tests. Many of the xUnit implementations have the ability to export results in a standardized format. This allows the test runner to be remotely invoked and return the test results to another program; this is typical for automation of tests. It is fairly common for test frameworks to include helper functionality that facilitates stubbing, mocking, etc. In many cases there are add-ons for testing frameworks that also provide this functionality, either if the base testing framework lacks the

feature or if, in the mind of the add-on author, the testing framework implementation is bloated or could be done better or more consistently. A common feature that is relevant to the example earlier and our subsequent discussion of testing design patterns in testing is that in a given test method, if an assertion fails, the entire test fails. This can lead to an unwanted reduction in resolution of the test results.

Differences that are prevalent include things such as the precise name of the various assertions, as well as the naming convention of the methods for both tests as well as setup and teardown operations. Some test runners make use of meta-programming to dynamically alter conditions for testing. Some languages, particularly compiled languages, offer annotations to indicate what different class methods are for; these are generally unraveled using reflection to aid the test runner in understanding what you want to test. Telerik Test Studio is an example of such a testing framework; test methods are marked with the [CodedStep] annotation to let the framework know that the method is a user-written test. Naturally, different languages and implementations allow for more concise, expressive coding or in other cases more verbose coding.

(Flawed) Arguments Against Unit Testing

Typical arguments against robust unit testing and test driven development includes concern that the extra work will require extra time, and that the return on investment of this time is not warranted. There are not any definitive studies to date, but while this concern superficially appears valid, it is not without counter argument.

First, in the short term, and on the topic of new code development, there is always some form of testing occurring. One does not simply write code and commit it. Various considerations are attempted, generally not systematically or in a focused manner. When some code is in place it is in fact tested, often via console logging or printing internal state to validate it is doing as intended, though there is no formal specification. In normal development, these activities fall under the budget for development. The argument is that testing activities, when unit testing or test driven development is employed, become line items effectively competing against the actual development. As can be easily appreciated, the actual scenario is that there already is a line item for testing, which takes the form of poorly implemented, undocumented ad hoc tests that do not conform to a specification. Unit testing and test driven development simply codifies the specification and makes the operation of the tests formal. The time to develop these tests, which are formalities rather than ad hoc implementations that constitute distractions in the operating code base, should be no more than the alternatives having none of the benefits.

Second, in the long term, and with regard to code maintenance, a robust collection of unit tests removes uncertainty and speeds development by providing protection against regression. This can dramatically decrease the amount of time spent developing ad hoc testing methods in the main body of code, which inevitably requires additional time to remove. Simply adding new unit tests for any added

functionality, and monitoring the status of the full suite of tests, gives higher confidence that the developed code fulfills the new requirements without regression. Since the testing is formalized and not ad hoc, the opportunity for errors is decreased and the time to develop and test is reduced. Moreover, the suite of tests can be run at any time, continuously ensuring freedom from regression.

Design Patterns In Testing

Observations about general methods to solve problems that are not frameworks and are not language-specific (e.g. do not depend on some particular feature of a language) are often termed design patterns. These patterns are more commonly considered in the Object Oriented world in Algol and Smalltalk derived languages. Design patterns are organizations of code to accomplish a specific task wherein the base design is clear to articulate and variations in problem result in minor variations in implementation. There have always been design patterns, but the study of them was popularized in Design Patterns by the Gang of Four. These patterns tend to be somewhat domain specific (and while perhaps not language specific, language family specific), at least in the sense of what patterns are more commonly seen in which applications.

Testing has a collection of similar repeated interests, so it is of no surprise that design patterns for testing is a field of interest. Design patterns in testing was brought about in great part by Meszaros in his treatise on patterns in the xUnit family[3]. Testing patterns are a special case of software design patterns at large, in that they have the additional specificity of testing and testing related constructs, and in the case of Meszaros, within the context of the xUnit testing strategy. That said, the patterns that emerge are contextual variations on the design patterns that one finds in day-to-day software, and are worth briefly investigating.

Much like conventional patterns in software design, testing patterns have mostly to do with standard forms of development that solve common problems. In the context of testing, these are testing problems. As noted in the preceding section, the tests that you write are fairly specialized software in a fairly specialized environment. It should come as no surprise that the patterns that emerge are somewhat like conventional patterns, but with the added features related to the imposition of the testing framework, and also the way the testing framework is implemented.

It is not within the scope of this section to dive into a grand treatment of testing patterns. You can rest assured that on the whole they are similar to traditional design patterns (e.g. Decorator, Visitor, etc.). Rather than become involved in enumerations, we will simply look at a case study. It was mentioned in the previous sections that there was a problem with the test that we considered in the example, and further discussed that a common difficulty, at least in the sense of retrieving the best possible feedback, has to do with multiple assertions in a particular test method. This situation is a very common anti-pattern that requires refactoring. This goes by the name of the Chained Test pattern. Let us consider an excerpt of what we had written earlier in our example test:

[3] The other main line of discourse in this field derives from Roy Osherove's "The Art of Unit Testing"

```
def test_is_prime
    assert_equal true, @p.is_prime?(11)
    assert_equal false, @p.is_prime?(10)
end
```

If the first assertion fails, the second assertion is not even going to run! While it may be of little consequence in this example, if the idea is serially testing a significant collection of assertions, the test does not produce resolution regarding what failed. One of our objectives as test writers is to produce tests that are as detailed and specific as possible. Particularly when testing is derived from lists or other sources of data automation (wherein many assertions could be made in a single test method), it is very important to consider refactoring the test. In the present case it is very easy. We could simply refactor this method like this:

```
def test_is_prime_true
  assert_equal true, @p.is_prime?(11)
end

def test_is_prime_false
  assert_equal false, @p.is_prime?(10)
end
```

Now, of course, we only have one assertion per test method. This simple change gives us significantly more information about what is working and what is not working when the test is run. That is, we have higher resolving power with respect to the output of the tests. Again, this situation is very common; indeed, it is possible to find examples of tests in the manuals of various xUnit frameworks that ought to be refactored for this very reason! It is important to the quality of information that the test suite provides to be attentive to this, particularly, as noted before, if the assertions are iterated from a list or other data source.

There are a few terms that Meszaros defined that are in common use. We should take a look at them so that when we use them we are using the proper terminology. One of the reasons to be clear on these terms is that they are used regularly in an incorrect context. Another reason is that the colloquial meaning of the words is fairly vague and misleading when applied to the testing context. Here are the terms we need to be most aware of:

Test Double: A generic replacement for an object that has limited function or surveillance function. A stub may be thought of as a double with limited function, a spy may be thought of as a double with surveillance function.

Test Stub: A replacement for a real object that returns pre-specified indirect inputs to the system under test. That is, an alternative object that mimics the behavior or another (more complex) object having the same types and returning known, constant values. This is to provide consistency in testing the response from an object. Part of the purpose in using stubs is to be able to test edge cases that may not be commonly accessible by other means. Stubs are used in closed-box tests.

Test Spy: A spy is a component that is installed to record method calls made to an object. It is easy to remember since it is spying on the object. Though they could, spies often do not actually perform a true function in their methods, as that is not the point of their use. Spies are used in open-box tests and are intended to observe that the code under test is invoking external methods correctly.

Mock Object: A mock object is, like a spy, another object that has added functionality for testing. Like a spy, the objective is not the function but rather the interface. You use a mock object to test that the software under test uses the object properly. The essential test using a mock object is for types. In some languages, such as those that are duck typed, the extent may only be in testing that the method names are correct. In strongly-typed languages the extent may be the arguments and the return values as well. Mocks are used in open-box tests.

Fake Object: Fake objects are lightweight implementations of more complex objects. Like stubs, they are used in closed-box tests. The difference is that the fake object attempts to mimic the actual object rather than provide a potentially incomplete implementation. The fake object pattern is used when the actual object is slow or has dependencies that may not always be present. An example of a fake object might be a stand-in for a database. The interface would be identical to that of an actual database, but the data might come from a file, for example, so that it is simpler to maintain and update, and delivers significantly better performance without the overhead of having to maintain an actual database. Fake objects are used in closed-box tests.

Dummy Object: A dummy object is a simple object that is used as a fill-in when there is no dependency on the function of the object. For example, if a certain object used by the software under test typically passed another object vis-a-vis dependency injection to an object that we are using in testing, it is likely that this second object will not be used (since we are contriving the function and data of the test object anyway). To remain faithful to the interface, we would want to pass an object; since this object is not used and is essentially a throw-away, we would make it as simple as possible. We would call this throw-away object a dummy object.

As can be appreciated from these definitions, and knowing that these are involved in testing patterns, we can see that these testing patterns have a significantly further reach than the simple example we first looked at. As with traditional software design and the design patterns associated with it, testing patterns and their nuance are something that become more clear with experience.

Tests, like other software, have a collection of smells. These smells tend to be related to the readability and long-term maintainability of the tests. Smells could be rephrased as very general ideas about what leads to problems. That said, they are a bit general, and in many cases a bit obvious. They are also fairly well related to patterns. Meszaros breaks testing smells into categories. We are not going to worry about the more obvious categories of behavioral smells (tests are erratic, fragile, slow, etc.) or

project smells (not writing tests, high maintenance costs, production bugs, etc.) because these things are both rather obvious problems and they are the results of not attending to testing in a mature manner. We also will not examine hard to test code; obviously some things are hard to test, and we are generally making an attempt to consider how to write code and tests so that this is not the case by employing good practices throughout this document. We will simply focus on code smells. Below are some relevant types:

Obscure Tests: As has been mentioned, it is very easy to write tests that are very difficult to determine what was most important about the data (in the case of closed-box tests particularly) and even what the point of the test was at all. This, of course, is not the only type of obscurity in tests, as tests can have data embedded in the test that is hard to notice and all manner of other implementation flaws that make servicing it difficult. The critical thought with obscure tests is that they should be designed and documented so that the intent is as clear as possible to the observer the next time the test requires review.

Tests With Conditional Logic: Much as the case with obscure tests, conditional logic in tests makes the test very difficult to follow. Tests ideally should be rather blunt. There should be a specific implementation concept that is being tested or there is a specific functional condition that is being tested. Depending on the specific circumstance it may not be possible to remove all conditional logic from a test, but generally speaking it would be better to remove unnecessary conditional logic to multiple tests than to leave it. The purpose of testing is to be resolute, easily maintainable and easy to read as documentation. Conditional logic in tests often negatively impacts these ideas. If there is a reason that conditional logic must stay, it likely is wise to comment the purpose of it.

Duplicated Test Code: Like all software development, it is generally a good idea to remove duplicate implementation as much as possible. That is, when there is an opportunity for abstraction, use it. Like any code, the maintainability of tests hinge on being able to make sweeping effects from one place rather than many places. Any time there is duplication it is easy to miss and the system is at risk of being out of sync when it is updated. One of the reasons this is a challenge for testing is that the testing frameworks are very susceptible to the boilerplate pattern of development. Copy-paste, copy-paste, copy-paste is quite effective when creating new tests, and everyone is going to be tempted and probably will do it. However, it makes the tests less maintainable and it makes them more difficult to understand. It is best to create test code with as little duplication as possible by creating the appropriate abstractions.

Test Logic In Production Code: This is one of those obvious smells, but it seems to be a fairly prevalent practice to embed some testing logic in code that reaches production. This is likely well-intended, but it is not the best way to test. Testing is a discipline that places a relevant framework around code that has no built-in testing fixture and allows

for systematic testing of this code. Well-intended tests embedded in production code could inadvertently be executed, clutter up the code, many times are not updated, and generally are a liability. Much like the maxim regarding printing out logging, if there is a reason one might want to have a test embedded in production code, the proper thing to do is move it to a real test. This will open the testing up to a much wider range of testing methods and automation, it will make the testing more likely to be maintained, and it will render the production code lighter weight without the distraction and liability.

Test Coverage

A very important concern with respect to testing is the test coverage. The problem, of course, is determining reliably whether or not we have tests that fully describe both the implementation features that are of concern as well as the larger scale business features that are of concern, without over-specifying the system. Additionally, these tests need to be reliable when everything is working properly, but be sensitive to change such that they fail appropriately in order to be an early indicator of regression. Test coverage is immensely important and quite challenging to ensure. While there is some science to it vis-a-vis coverage metrics, there is also some art to it.

Test coverage metrics take many forms, most of which can be determined automatically using more advanced testing tools. It is for the tester to determine what makes the most sense as a metric of coverage. Many of the following ideas are derived from the notion that the greater the surface of coverage, the more sensitive the tests will be to regression. There is, of course, truth to that, but it is also the case that the more tests there are, the slower the testing, the more errant tests distract from the essential goals of codifying specification and providing assurance against regression, and the harder the testing suite is to develop and maintain. Typically, small scope, mission-critical software (such as control systems for nuclear plants) will exhibit intense coverage, whereas larger scope, less critical software (such as a login page on a medium traffic website) generally is more selective about what is tested.

Unfortunately, even if one desires intensive coverage, one finds a somewhat opaque answer to what coverage really means. It is common to ask what percentage of functions in the code base have tests, or what percentage of lines of code are covered by a test. Reducing the granularity might suggest answering what percentage of conditional logic has been tested, which can vastly increase the number of tests. One might ask what percentage of possible machine codes could be tested, as different conditions may yield different machine codes in modern processors or with dynamic, interpreted languages. The goal, of course, is to fully specify and test the system in the face of as few unknowns as possible, though at the expense of exhaustive articulation of the possible states of the system. Toward these goals, it is fairly common to perform analysis on the cyclomatic complexity of the code base and use this to steer the code toward simplicity, as the simpler it is the better coverage can be achieved for a lower testing cost. Going down this path, while extremely good for test coverage, can be a long and

slippery path indeed; this is one reason why these methods are used more on simple systems where the cost of failure is extremely high.

The pragmatic approach is that of considering what is important to test; that is, what is an objective deliverable, what is a fundamental aspect of the way the system is implemented, what are the possible causes of failure under duress, what are the edge cases that define the operating cases of the system, where is the highest risk of system failure. This can be a greatly reduced collection of tests, though bearing the vast bulk of risk mitigation that testing provides. Unlike the notion of complete conditional testing, the pragmatic approach would suggest that we should test what we know is important both from a software development standpoint and a user story perspective. To this end, testing is often performed this way rather than by computationally comprehensive means.

There are fundamentally two types of things that are generally tested: interfaces and function. It depends on the organization of the software under test as to specifically what should be tested, and what blend of interface and function makes the most sense. Generally speaking, if a module has more to do with interface (for example, if it is a module that exposes a public API), then it will have a focus on testing of interface. By testing interface, we are referring to tests having to do with types, exposed methods and signatures, etc. If a module has more to do with business logic, say, then the tests are less about interface and more about function. In this case, we would expect testing to focus on business logic and edge cases. Of course, the vast majority of testing will have a blend, some favoring the interface and some favoring the function. There is little software of interest that has either no interface or no function.

The reason software developers write tests rather than customers is because software developers translate customer stories into programming logic. Part of this translation is the design of the software, which adds abstraction and logical reasoning to solve a concrete customer problem. The abstraction adds interfaces that require testing to ensure that they are correct, whereas the logical reasoning to solve the problem adds complexity like conditional logic that needs to be tested to ensure it complies under the proper stimulus in the way it is expected to in solving the customer problem. The bulk of what testing accomplishes in this regard is related to ensuring the practical solution to the problem is what we planned it to be (validation); the rest is that the problem that was trying to be solved in the first place (the customer's problem) performs as expected (verification).

End-to-End Test Automation

In this section we mainly speak of automated testing in the colloquial sense as end-to-end system and acceptance test automation rather than, say, automation of running unit tests. In truth what will be discussed are challenges faced with end-to-end testing, as the portion of the system one must be aware of and design controls for is significantly expanded, and the interactions of the different portions of the composite system presents new challenges. Understanding of practical customer use must be known in

order to effectively create and maintain tests that reflect appropriate tests for use cases related to known user stories.

Curiously, while there is greater complexity in scope associated with the this type of automated test, there is also lower complexity insofar as test writing skills are concerned. What is meant by this is that while the scope of components that one must manipulate in order to set up and execute a test is larger (e.g. database, interface, etc.), the actual code behind the test is generally more mundane and does not require as much knowledge of the implementation of the smaller parts of the system (e.g. types) or how to apply testing design patterns to them. Of course, much of the reason for this is that the tests we are speaking of are closed-box tests. The code that is written to perform this type of automated test in many ways is much simpler than its unit test counterpart, and often more direct.

Though the frameworks in use for running these type of tests are generally similar to lower-level testing frameworks inasmuch as they maintain the xUnit family of naming conventions for assertions, the style of implementation of these tests is often much different. It is more common to see very direct testing design, particularly when dealing with controlling interface components, and it is very often the case that tests are iterated over test data variations coming from some form of preset data store. The setup components of the tests are generally much more significant, and the steps that are required to even be able to make an assertion are generally much greater in number. These types of test are also more apt to suffer from certain anti-patterns, such as those remedied by the Chained Test pattern, and are generally quite susceptible to the full range of testing code smells, not the least of which is obfuscation of how the test actually works for testing user stories and use cases. It is prudent to comment these tests, as the logic can be rather unapparent.

The general idea of end-to-end automated testing is that the framework grants the ability to interact with the browser and other complex mechanisms such as databases. The ability to work with databases is nothing especially unusual, though these interactions are generally transformed into test doubles (an object with the same interface as the real object, but with limited or surveillance functionality, as we will discuss later). In some ways, end-to-end automated tests are more familiar than tests that use test doubles. However, the often arduous process of setup and teardown of data used with end-to-end testing in a database having a schema that in sync with production can be an awakening experience. The code under test perhaps is modifying the data slightly, though often simply presenting the data to the user interface, but the test is responsible for ensuring the starting point is stable under changes from other users and other tests. It can be challenging to ensure the state of a dynamic system at the beginning of each run of a test.

There are various interfaces to the browser that are available. Unlike the nearly ubiquitous xUnit open source frameworks, some are proprietary, such as the offerings from Telerik (Test Studio), whereas others are open source, such as Selenium. The essence of all of them is that it is possible to write plug-ins for most major browsers at this point in time, and by doing so it is possible to access the user functions, such as mouse movement and keyboard entry, as well as the DOM and events, programmatically. In general, the low level features offered by the various browser control plug-ins are fairly similar, as all browser controllers essentially implement the same rudimentary APIs. However,

differences emerge as one looks at what is built on top of these basic features. What is resting on top generally amounts to more advanced DOM manipulation, greater attention to a consistent interface that works across all supported browsers, the facilitation of more advanced features such as drag and drop, and helpful features that have more to do with data than with browser control. You can think of the these features as being more akin to convenience libraries built on top of a basic web browser control interface than a more complex control interface. As with most examples of subscription versus open source, the real issues are long term support and increased productivity based on prewritten conveniences, as well as attention to deployment for running suites of tests across multiple computers. It simply depends on what problems you are trying to solve as to which is the best choice.

The increased complexity of the testing environment, spanning interface, database, network, etc., coupled with the significantly increased quantity of software being tested (in comparison to, say, a unit test) yields much more brittle tests. Changes to the database will break tests; changes to the network will break tests; updates to browsers and other software will break tests; even the weather and its effects on network latency can break tests. It is hard to write good automated tests for system and acceptance testing. The key is to make the environment where the tests run as consistent as possible. Otherwise, the value of the tests is reduced by the number of false positive failures that are reported. Like the boy who cries wolf, tests that routinely fail when there is no real failure of the software tend to not receive the attention they might when there is a real failure.

Critical to effective end-to-end automated testing is risk assessment and test prioritization. For various reasons, it is recommended[4] to minimize the number of end-to-end tests that assess the functionality of low risk or relatively stable software features. This is because end-to-end tests tend, as we have noted, to be slow and brittle. Testing low risk and stable features amounts to adding noise to the test suite without creating value. There is more expense developing and maintaining a larger testing code base, and a reduced sensitivity to the high risk or unstable features being detected, by tests with the presence of interference from a cacophony of failures coming from slow, brittle, low risk and stable test cases.

Continuous Integration

As the name alludes, continuous integration is a process for streamlining new features (or fixes, refactorings, etc.) into the main production code base. The goal is to do so as quickly as possible while maintaining passing tests in a system test environment. By doing this, the current main code line is always able to be deployed, as it is always in a state of passing all tests. This is in stark contrast to deployment strategies that involve monolithic, delayed releases, as there is not necessarily a main code line that is always fit for release, and there is not rapid feedback regarding individual changes against the production code base.

[4] Telerik. 10 Tips On How To Dramatically Reduce Test Maintenance.

It was important in the previous section to defend the advantage of unit tests, as they are a fundamental component of continuous integration. Continuous integration is a philosophy for producing faster turnaround code releases with less bugs. An integral part of this philosophy is facilitating verification of proper function with as much automation as improves integration outcomes. Therefore, before a commit is made, it is the responsibility of the developer to make sure that the code passes local automated tests, and that the code being tested is the current working copy merged with the most up-to-date production main code line. Tests should be run locally before committing the code. To prevent code that is not working as expected ending up in the repository, it is common practice to put pre-commit hooks on the repository that run unit tests on the committed code and reject the commit if they do not pass (or exist).

Upon the completion of the commit, a continuous integration system builds the new system in a system testing environment and runs all relevant tests. These tests are presumed to be passing reliably before the new commit. The tests are also presumed to run with reasonable expedience, typically in less than ten minutes. If the tests are not passing reliably or are not sufficiently quick in execution, their utility is diminished as they are not providing timely and useful feedback about the operating condition of the new build in the system testing environment. The developer who committed the code is alerted to the success or failure of the build in the system testing environment. Depending on what works best for a project, the system test environment may remain with the change or revert to the commit prior that is known to pass tests. The developer can then debug his or her code while the rest of the team can continue working and committing changes that are (if passing) integrated into the main code line. In this way, the main code line evolves incrementally and quickly, and never is in a state where it cannot be applied to production. It is commonly the case that there is an expectation that developers would fix any commits that are not passing the system integration tests before leaving for the day.

It is fairly common practice to have scheduled or nightly builds being managed by a continuous integration server. This is not continuous integration, it is verification of nightly builds. Scheduled nightly builds only serve the purpose of ensuring that previously verified software remains verified, and is intended as a first-line indication of software failure before it is noticed by users. This is much different than continuous integration, as the nightly builds do not provide rapid feedback to developers regarding system integration and they do not maintain a fluid main code line that remains deployable to production in the face of rapid change. In other words, continuous integration is more a philosophy having to do with maintaining a flexible main code line than it is about verification (though verification is a central aspect of continuous integration), whereas nightly builds are purely verification.

Opinions On Best Practices

There are many people who are excellent practitioners of software development, and nearly as many opinions about best practices for testing. These opinions can essentially be broken down into two groups, which we will look at briefly with some excerpted words from members of each. We will also look at a small group that does not perform testing.

On the one hand there is a group that is firmly esconced in testing everything. Members of this group would not feel that software is complete if it is not fully tested. Indeed, many in this group feel that it is the duty of software developers to have good unit tests, and a violation of their ethical code to produce quality software if they do not test it thoroughly. Those in this group consider unit testing fundamental to the project. Robert Martin is an excellent example of this group, writing in response to a growing number of failed large scale software projects where users have deep dependency on the software (as an example, the scandalous health exchange rollout):

> *"If I am right... If TDD is as significant to software as hand-washing was to medicine and is instrumental in pulling us back from the brink of that looming catastrophe, then Kent Beck will be hailed a hero, and TDD will carry the full weight of professionalism. After that, those who refuse to practice TDD will be excused from the ranks of professional programmers. It would not surprise me if, one day, TDD had the force of law behind it."*[5]

Now that's an idea, isn't it? Verification of software design with repercussions akin to other fields in engineering. One might hope software does not become that litigious. But the point remains valid, testing can be viewed as an act of professionalism, particularly when the system is of consequence to others. But it isn't the whole story, there are certainly other ways of testing than TDD. The more pragmatic have differing opinions, as David Heinemeier Hanson expounds:

> *"When I first discovered TDD, it was like a courteous invitation to a better world of writing software. A mind hack to get you going with the practice of testing where no testing had happened before. It opened my eyes to the tranquility of a well-tested code base, and the bliss of confidence it grants those making changes to software.*
> *...*
> *Over the years, the test-first rhetoric got louder and angrier, though. More mean-spirited. And at times I got sucked into that fundamentalist vortex, feeling bad about not following the true gospel. Then I'd try test-first for a few weeks, only to drop it again when it started hurting my designs.*
> *...*
> *Maybe it was necessary to use test-first as the counterintuitive ram for breaking down the industry's sorry lack of automated, regression testing. Maybe it was a parable that just wasn't intended to be a literal description of the day-to-day workings of software writing. But whatever it started out as, it was soon since corrupted. Used as a hammer to beat down the nonbelievers, declare them unprofessional and unfit for writing software. A litmus test."*[6]

On the far end of the spectrum are those who do not test at all. Those in that camp generally are highly academic, use strongly-typed functional languages, and view testing as an admission of using tools that are unnecessarily error prone. Freedom of errors in these applications typically is first-principals proof

[5] http://blog.8thlight.com/uncle-bob/2014/05/02/ProfessionalismAndTDD.html
[6] http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

based on language constructs rather than the type of testing we are generally describing in this document. The reason, of course, is that strongly-typed languages have excellent compile and runtime type checking qualities, and purely functional languages have no side effects, the combination yielding systems that tend to only fail when the algorithm is wrong, something that is better remedied in code review than testing. Contrary to what one might think, there are many critical systems created this way, particularly in finance and telecommunications.

These differing views come from different people who take on different types of projects with different requirements in different languages. It is, of course, the case that some software, particularly when used in critical environments and situations, has a greater need for the assurance of testing than do other types of software. Some software is more relevant to unit tests and TDD, other software is more relevant to system testing. Some software undergoes verification that is not testing based. While it may seem there is little to be learned from the group that does not test, that is where the academics in the industry are going and it is fairly easy to concede that most instances of failing tests in imperative languages that are not algorithmic errors are, in fact, due to side effects. The more possible side effects, the more brittle the tests; this is why functional testing is generally more brittle than unit testing.

Somewhere between the various views is where the best solution for most projects lay. Testing isn't just about the system, it is about the maintenance of the code and ensuring that at that level there is freedom from regression. It is about deliverables to customers and it is about ensuring interfaces. It is about testing expectations and it is about reducing risk. There ought be no doubt why there is such a wide-ranging view of testing when testing covers such a wide range of needs for a wide range of audiences.

Curiously, there is much less contention surrounding practices such as continuous integration, aside from the typical anxiety regarding implementing it. Martin Fowler sums it up best:

> *"When I've described this practice to people, I commonly find two reactions: 'it can't work (here)' and 'doing it won't make much difference'. What people find out as they try it is that it's much easier than it sounds, and that it makes a huge difference to development. Thus the third common reaction is 'yes we do that - how could you live without it?'*
> *...*
> *Projects with Continuous Integration tend to have dramatically less bugs, both in production and in process. However I should stress that the degree of this benefit is directly tied to how good your test suite is. You should find that it's not too difficult to build a test suite that makes a noticeable difference. Usually, however, it takes a while before a team really gets to the low level of bugs that they have the potential to reach. Getting there means constantly working on and improving your tests."*[7]

What we can glean from all of these thoughts in concert is that testing is good, perhaps moving toward necessity as the complexity of software increases, test driven development is something a lot of people

[7] http://www.martinfowler.com/articles/continuousIntegration.html

support but perhaps is not the best practice for all projects and teams, and that continuous integration is very well adopted when it is adopted. The essence of it all is that software is generally better with more, faster and more frequent testing, and as much automation to create rapidly evolving production code that is verified by tests as is possible.

Economics Of Software Testing

There are not many available quantitative studies regarding the cost and cost-benefit of software testing. This is likely for several reasons. First, the ability to compare projects is often difficult because there are so many variables that contribute to the end result (different languages, environments, deployment mechanism, customers, etc.). Second, the actual net present value of maintenance is hard to determine, and since most software projects are different, it is difficult to estimate based on comparables, as there generally are none. Third, it is somewhat unclear and fairly difficult to relate any findings regarding value on a cost basis with objective measures regarding testing. Fourth, it is difficult to determine what the proper objective measure of testing is in the first place. Though there are a few quantitative resources specifically about software[8] and some with reference to software[9], likely inspired by Barry Boehm's "Software Engineering Economics," this relegates most of the work in the field to qualitative project management studies and rule of thumb guidelines, the progenitor of which likely is Fred Brooks' seminal "The Mythical Man Month."

Empirical evidence suggests that there is a powers of ten relationship between the cost of fixing bugs in software[10], which is actually just a rephrasing of a total quality management rule of thumb. That is, given a unit cost for fixing a bug in the design phase, there is a ten unit cost of repair in the testing phase and a 100 unit cost of repair after the code has shipped. Since unit tests are an aspect of the design phase, put another way, there is a small cost in repair when detected from a failing unit test, a higher cost when triggered by a failing system test or acceptance test, and an even greater cost when detected by a user. The reason for the extraordinary escalation in cost is the number of people involved and the aggregate opportunity cost of the defect at each stage in the product lifecycle. It clearly pays to detect defects early.

It has been found empirically that unit test writing accounts for about 10% of the time in a project and 40% of the total number of lines of code[11]. The amount of time is reflected by the fact that the tests are straight forward, and while occupying a significant share of the volume of code, do not require the same level of design and thought as the underlying code being tested. The benefits for this level of time commitment show dividends through reduced bugs ending up being found by system and acceptance tests and by users of released code. Similarly, it has be found typical that quality assurance

[8] NIST (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing.
[9] NASA (2008). Cost Estimating Handbook.
[10] http://www.riceconsulting.com/articles/achieving-software-quality-using-defect-filters.htm
[11] http://googletesting.blogspot.com/2009/10/cost-of-testing.html

on the whole, inclusive of manual testing and other non-programmatic activities, typically accounts for between 5 and 10 percent of an IT project budget.[12]

It has been observed that the number of errors in imperative code is proportional to the number of lines of code, and that this is reasonably constant across different programming languages and paradigms of programming, with the industry average being in the range of 15 to 50 bugs per 1000 lines of initially delivered code (there are some mission critical projects, of course, that have far less, such as the space shuttle code, which has zero known defects in 500,000 lines).[13,14] Curiously, there are few examples of decisions being made regarding language choice for new projects knowing this information. In light of this observation, it is of no surprise that the more verbose a language is, not only the more time it takes to write software (and tests), but the more bugs will be in the software (and the tests). This leads to a super-linear problem with defects in software as a function of the verbosity of the language. The more concise a language, the shorter the development time, and the less bugs will ship. As well, the simpler the code, the less bugs will ship, which has been the leverage applied to promoting lightweight and micro-service architectures for some time.

Take Home Recap

Unit testing is fundamental, even in an environment with extensive system testing. Not only does it provide testing that relates to what users experience and relieves the need of more extensive (and more brittle) integration and system testing, it provides a first line of defense indicating whether or not an addition, deletion or refactoring of existing code alters the software in a regressive manner. Not only does a developer not have to worry so much about breaking functionality, but he or she does not have to spend time performing ad hoc tests. Furthermore, writing good unit tests implies writing good software, as good unit tests are hard to write if the underlying software does not have a well-designed interface. The cost of adding unit tests is relatively low, the benefit generally being far higher.

Test Driven Development can be even better in the right circumstances, as it provides documentation of function and interface before the code is actually written. Superfluous functionality that is not part of the contract (and often will create additional difficulties) are not developed, saving time. The thought process of development is more focused on implementing what tests exist, and implementing the code in an extensible, abstracted and minimalist way. Despite these positive aspects, it can be very hard to do well and may not be right for every project or team.

Testing in the Behavior Driven Development paradigm is roughly equivalent to TDD, but the tests are two-part and they are designed to be readable by a larger audience than programmers. The specification is always developed first, and the actual test assertions are generally written before the

[12] Dabney, J.B., Barber, G. and Ohi, D. (2006). Estimating Direct Return on Investment of Independent Verification and Validation using COCOMO-II.
[13] McConnell, Steve (2004). Code Complete.
[14] Cobb, R.H., Mills, Harlan (1990). Engineering Software Under Statistical Quality-Control.

code being tested, in a TDD-like manner, though they could be written after. In a sense, BDD testing could be viewed as something of a compromise, separating the concerns of the specification and the specific assertions in order to add development flexibility while maintaining the specification from the onset.

Automated testing, taken in the sense of end-to-end testing, has its own set of challenges. These challenges stem from both a heightened level of dependency that adds complexity and brittleness to the tests, as well as challenges both in undertaking engagement of the dependent aspects of the system (database, etc.) and the development of logic that is consistent with practical use cases that are aligned with user stories. These tests tend to be more prone to erratic failure than simpler tests, and require the developer of the tests to be alert to all aspects of the composite system (interface, database, systems, etc.) as well as be knowledgeable about the practical use of the software by end users.

There are nuances to writing good tests just as there are nuances to writing any good software. Testing comes with its own set of challenges and its own set of common practices. That is, there are smells and patterns germane to testing, just as there are to any other software. Writing good tests takes effort and it takes experience to become proficient at avoiding the pitfalls, achieving good testing resolution, making tests run expediently enough to be useful (and used), and be maintainable. In order for tests to maximize impact and deliver results that best help identify errors early and reduce costs over the lifetime of the software under test, all developers should be aware of best testing practices and practice it.

Continuous integration forms an infrastructure that allows for rapid deployment of code in a manner that does not result in times when the main code line will not pass tests. The philosophy of continuous integration is posited on having a robust set of tests that can be verified quickly and reliably, and that are built into an automation system that ties testing system builds with repository commits. With this in place, developers can make updates and merge them into the main production code line quickly, resulting in much shorter and more reliable development cycles for releasing additions, enhancements and updates.

The further from the initial development bugs are found, the more costly the bugs are. Quick bug detection at the unit test level can dramatically help reduce the number and cost of bugs that are found in system and acceptance testing, and reduce the number and cost of bugs that are found in production by users. End-to-end testing should focus on integration testing that is not possible to test with unit tests, that carries high risk to the business and is not stable. Implementation of unit and end-to-end testing positioned around continuous integration can result in faster deployment of changes with significantly greater confidence and a lower defect rate.

Selected General References

There are a large number of references that prove useful to testing. These not only include the obvious ones that examine specific tools, but those references expounding on the philosophical nature of development with testing in mind, those addressing design patterns that appear in testing, and of course all resources relevant to programming practices that make for better tests. Below is a brief list of resources that prove good reads for developing testing skills. These are entirely full books, and do not include most of the footnote references. Some of the books listed are not specifically about testing, but about good programming practices in general, mainly related to object oriented programming and refactoring.

Beck, Kent (1991). Extreme Programming. Presents a new paradigm for the software development process that was born from the observation of problems in effective completion of projects. Concerned with process philosophy.

Feathers, Michael (2004). Working Effectively with Legacy Code. Explores how to improve and refactoring poorly written (legacy) code.

Fowler, Martin (1999). Refactoring: Improving the Design of Exiting Code. The classic refactoring text.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). Design Patterns. Classic gang of four treatment of patterns in software development.

Martin, Robert (2008). Clean Code. The well-known book on producing maintainable code. Much of the content is germane to creating code that yields maintainable tests.

McConnell, Steve (2004). Code Complete: A Practical Handbook of Software Construction, Second Edition. It is, as the title states, a handbook of software construction. Classic.

Meszaros, Gerard (2007). xUnit Test Patterns. An overview of observed design patterns as applied to testing. The beginning chapters are excellent as a review of the cost-benefit of tests and the fragility of different testing strategies.

Metz, Sandi (2012). Practical Object-Oriented Design in Ruby: An Agile Primer. Practical refactoring, taking into account as many language features as possible, with a focus on organizational cost as well as test integration.