Designing for Testability

John A. Deters

9/26/2017

Designing for Testability 3

Summary 3

The current state of testing 3

Fragile tests 5

Testing taxonomy 6

Test Frameworks 8

Test framework architecture 8

Test framework structure example 9

The problem is always dependencies 10

Test Doubles 12

Mocking frameworks 13

Objects 14

Verifications 14

Test refactorings 14

Good test attributes 15

SOLID Design Principles 16

Test problems 17

Code structure 17

Issues that make code hard to test 18

Design patterns to help break dependencies 19

Recommended Reading 20

Designing for Testability

Summary

This paper has three objectives. The student will learn what automated testing is, and the various types of automated testing. They will learn the requirements of a developer who is being asked to write automated unit tests. And they will learn how design impacts the developer's efforts.

The current state of testing

The testing gamut runs from manual ad hoc testing to fully automated. Software can be tested for many different qualities, such as correctness, fitness for purpose, performance, security, and compatibility. It can be regulated by formal standards such as those used for aircraft or health systems. And testing can be performed at different phases of software development. This paper will focus on "developer-created automated testing", which are tests created by and run by the software developers.

Many software companies have learned that fully automated testing is the only way they can deliver reliable code on a continual basis. Tests start with requirements, and developers are responsible for writing the tests that test their own code. Some companies have been very open with their development process, and we can learn a lot from them. The Etsy developers host a blog called "Code As Craft[1]" that details their processes.

Etsy developers deploy code about 25 times per day, and they run a suite of automated tests with every build and deployment. They have over 7,000 automated tests, which would take about 30 minutes to execute them all on their Jenkins servers, but because there are so many developers delivering so often throughout the day, they need to deploy faster than that. The developers select tests based on attribute tags that are most relevant to their changes. These tags include things like database, network, sleep, flaky, and slow. Flaky tests are those that fail intermittently, which interfere with development efforts. Sleep tests are those that include a sleep command, or those that depend on the system clock. And they've found that a few tests take up most of their time, so they're tagged as slow. They generally exclude the kinds of tests that don't add specific value.

Etsy defines a unit test as a test for one and only one class. It can have no database interaction at all, not even fixtures. Their unit test servers do not have SQL or PostgreSQL available, so they find out immediately if a database dependency was accidentally included in the code. And running unit tests is not optional. There is no provision to deselect unit testing from their automated environment. All 4,500 of their required unit tests run in about a minute.

Their integration tests are any tests involving fixtures or external services, such as memcache or gearman. They avoid network tests because they're fragile. They run smoke tests with `curl` to prove that their site still returns expected headers and other

[1] https://codeascraft.com/

data. Finally, they require all of their changes to be protected by a “config flag”, meaning that a new feature can be turned on or off at the flick of a switch.

That is how a mature software development team operates today.

Not all development teams follow a methodology like this. Many shops are still dealing with a waterfall mentality, which begins with a big analysis and design phase up front, a coding phase, and a large testing effort prior to shipment. But when code isn’t working and developers are rushing to ship on a deadline, testing gets pushed down the priority stack in favor of delivering the product on time. The code that gets shipped may work along the happy path, but edge cases, corner cases, and exceptions receive limited testing, if they are tested at all. The result is code that contains an unknown number of bugs that range from incorrect operation in some cases to fully exploitable vulnerabilities in others.

Developers need to see testing code as their responsibility, although they don't always see the value in creating automated tests. But the value exists, and it’s huge. First, developers get immediate feedback while writing code that proves the code works today. It also proves the code that was written six months ago still works. Bug reports should be recreated in tests; the tests remain in place to prove that the problems are not reintroduced by future changes. Unit tests serve as documented examples of how to call functions, as well as the behavior expected of them. Automated tests also reduce very expensive external testing. Not only does it cost time and money to manually execute tests, but it costs the project in terms of time when the testers discover a bug. The testers have to write it up in a bug tracking system; send it back to the developers; the developers have to categorize and prioritize the bugs for fixing among all the rest of the work they’re responsible for. This cycle means that bugs caught after release to testing can cost orders of magnitude more than bugs caught during development.[2] Code that developers have tested have fewer bugs, and need many fewer trips through this loop.

But those are only side benefits of automated testing. When you look at code as an asset, the primary benefit comes from the impact unit testing has on the design of code.

When the testing of a module is an immediate responsibility of a developer, he or she has incentive to create an interface that is easy to test. The attributes of code that make it easy to test are the same attributes that make code easy to reuse – it’s cohesive, decoupled, and idempotent. Creating the unit test serves as a test of the usability of the interface. Interface design that is driven as a result of unit testing yields a clean code base, which is critical to owning a long-term maintainable product.

[2] http://www.agilemodeling.com/essays/costOfChange.htm

Fragile tests

To provide the most value, tests need permanence. We want to run them today, and again and again in the future. The quickest solution seems to be to buy a testing "robot". This is a tool that simply records a test session and plays it back on demand, then ensures the actual output matches the expected output. These tools are certainly valuable, and have their place in the testing department. But they don't work well for developer testing or automated unit testing because the tests are "fragile". As the name implies, fragility means they break easily, and a broken test causes a failure of the testing process.

What causes tests to break?

Fragility type	Cause and effect	How Unit Testing addresses this
Behavior fragility	The old tests may (should) break if you implement a functional change. You may change code because of changing requirements, or when you fix a bug. This is to be expected. The bigger problems come when the behavior is pervasive: change the requirement from 4-digit PIN to 5-digit PIN, and suddenly every test using a 4-digit PIN is broken until you fix it.	By isolating a test to just the item logic, it is not needing to test items by signing on with a 4 or 5 digit PIN. It's testing only items. If the business requirements include a change in PIN length, simply change the PIN length tests, and nothing else.
	This can make changes expensive in terms of the number of tests that need to be fixed.	
Interface sensitivity	A user running a test might click on a text box and type some input. The robot records that the user clicked on point (123,45), and pressed the X, Y and Z characters. Now, your requirements aren't to "make a change", you're just supposed to make the screen pretty, so you slide the input box over 200 pixels. When the robot plays back that series of canned inputs, it will click on something that's not the input box, and fail.	By not testing through the GUI, the GUI can't interfere with the tests. It doesn't matter where the robot was trained to click, because there is no need to test the robot's ability to click. UI dependence is eliminated. The test only examines the software's ability to process the rules built into the code.
	A human can cope if you resize the input box, or change the font, but an unsophisticated robot will break.	
Data sensitivity	This is a huge problem for testing teams. A test to sell three bananas for $0.75 requires bananas in the database to have a price of $0.25 each. It also requires the bananas to be in the produce department, and to have the correct velocity code, and have the correct taxability attribute, and to have the hazardous waste flag turned off, and no bottle deposit, and a positive on-hand count, and a hundred other bits of data that have to be exactly right.	By not testing the data in the database, data independence is achieved. When a test is needed where a banana item costs exactly 25 cents, define a test item object that already contains a 25 cent banana, and pass it into the routine.
	If the test environment gets a production data update, and the merchants have raised the price of bananas to 30¢ each, every test that relies on a 25¢ banana is going to fail.	
Environmental sensitivity	The test environment has to be constant. Did the original user ID expire? Did the user have to change their password after 90 days? Is the test box Windows XP service pack 3? Does the test platform have the authority to access the databases it needs, or the services? Is the hard drive full? Is the network up? Hundreds of hidden environmental attributes have to remain constant in order to run the tests.	By not requiring a file system or a network or a user ID, it doesn't matter if the file system is full, or if the network is down, or if the user ID has expired. The more moving parts that aren't tested, and the more isolated the test system is from the environment, the more likely the test will execute and check the only part that truly matters: does the module do what it is supposed to do?
	The first dependency may be that each developer has to have a licensed copy of the expensive robot software installed.	

Testing taxonomy

It's important that when we're using a common name that we're all talking about the same concepts. This is not an "official" list, nor is it all-inclusive, but rather a framework for this paper.

- Approaches
    - White box testing. Testing with full knowledge of how the system is internally designed.
    - Black box testing. Testing with knowledge only of the public interface (GUI, API, requirements, specifications, instructions.)
- Focus
    - Unit test. Examining the smallest possible piece of functionality from the point of view of the code, isolated from all external dependencies.
    - Functional or use-case test. Examining one piece of functionality from the point of view of the user.
- Levels
    - *Ad hoc* testing. Run the system; manually feed it "appropriate" inputs, manually check for expected outputs.
    - Scripted testing. Run the system; feed it documented inputs chosen to exercise various functions, check for documented outputs.
    - Automated scripted testing. Run the system; have an automated tool feed it the chosen inputs, have an automated tool examine the outputs.
    - Automated unit testing. A framework that executes suites of developer written tests that each test one small aspect (unit) of code.
    - Integration testing. Running the system in a production-like environment, testing how the system integrates with other systems.
    - System testing. Running the system in a production-like environment, testing all aspects of a system to verify they meet requirements.
- Objectives
    - Health check (aka smoke test). A minimal amount of testing to determine if the system is ready for more extensive testing or deployment.
    - Regression testing. Re-running a suite of tests after a software upgrade to ensure existing functionality is not broken by the software changes.
    - Performance testing. Running the system with a maximum load generating tool to ensure it meets load and timing specifications.
    - User acceptance testing. A tollgate step where the user signs off that the software is of an acceptable quality to deploy to a production environment.
    - Alpha testing. Having a small, select set of users execute the code in a production environment; one form of user acceptance testing.
    - Beta testing. Having a limited self-selected set of users execute the new code in a production environment; typically follows alpha testing, can be considered the final stage of user acceptance testing.
    - International testing. Having people of different cultures test the code to ensure it is acceptable in their language and environment.
    - Certification testing. Tests to prove conformance to a regulatory requirement.

- Learning
    - Exploratory testing. Run the system manually checking for appropriate behavior using experience and feedback to dive into potential problem areas. Related to *ad hoc* testing.
    - Stress testing. Running the system with a variable load generating tool to establish the maximum throughput the system can sustain, and to discover how the system reacts in partially loaded, fully loaded and overloaded conditions. Related to performance testing.
    - Usability testing. Having a user execute the user interface of a system in a monitored environment, watching the user for reactions.
- Security
    - Penetration testing. An expert penetration tester feeds the system data with the intent to cause the system to behave in an exploitable manner.
    - Fuzz testing. An automated testing system that generates random inputs attempts to cause failures in the system. Often used with other security tests.
- Operational
    - Chaos monkey[3]. A task that randomly causes unplanned faults in a production environment, with the intent that the system must continually prove itself robust enough to automatically recover.
    - Latency monkey. Introduces artificial delays in communications to simulate service degradation.
    - Chaos gorilla. A task that simulates an outage of an entire zone or data center.

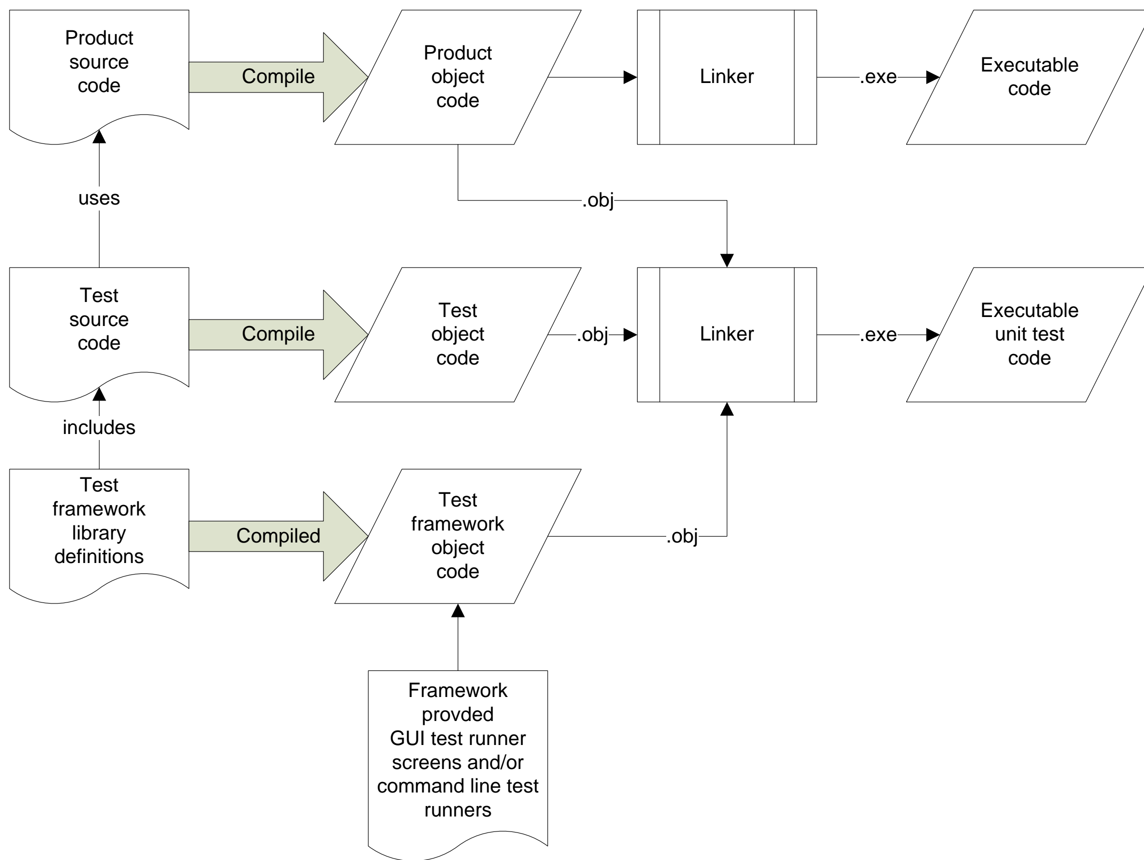
[3] https://github.com/Netflix/SimianArmy

Test Frameworks

Automated unit tests are run in a test framework. This framework is generally a simple structure that provides knows how to discover test code, execute that test code either in a GUI screen or as a command line application suitable for adding to the build chain, and provides a library of commonly used test functions, such as "assert()".

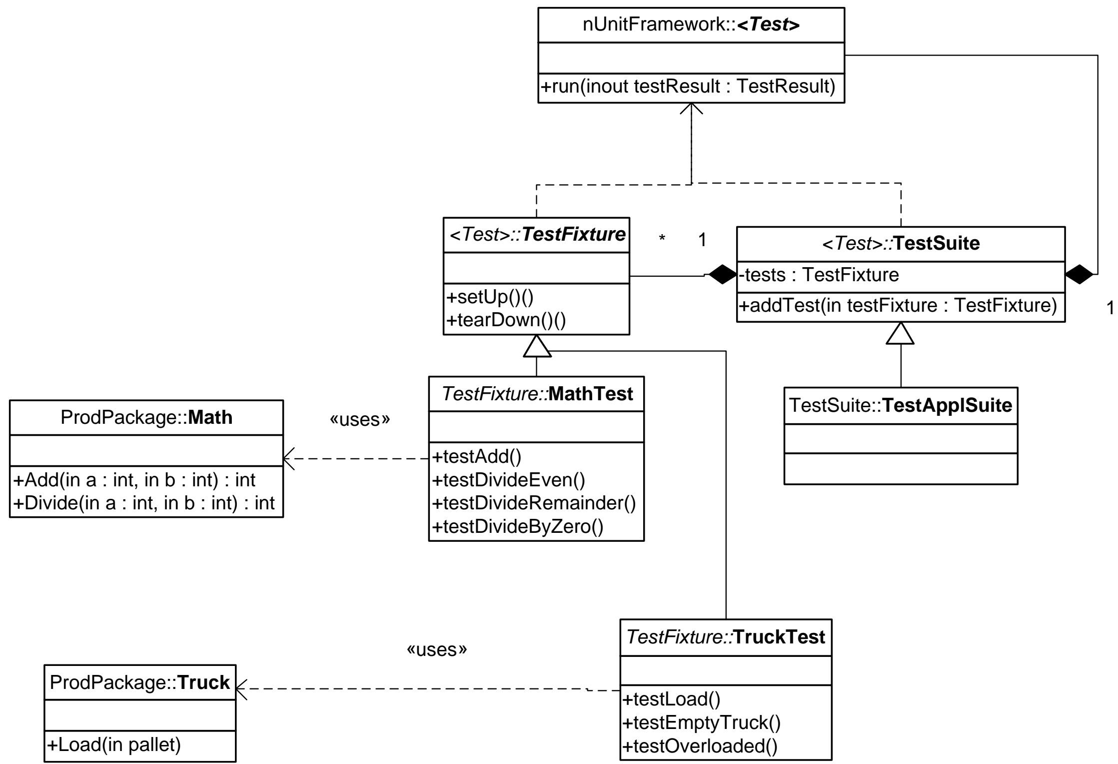
Most unit test frameworks are built on the pattern of SUnit[4], which was developed for Smalltalk, ported to Java as JUnit, and was the first widely used automated unit test framework. Generically, they are referred to as xUnit test frameworks, and there are now over 200 different xUnit frameworks existing for languages from ABAP to XSLT.

Compiled test framework architecture



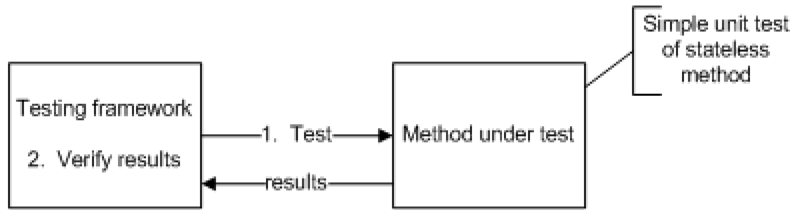
[4] http://junit.org/

Test framework structure example

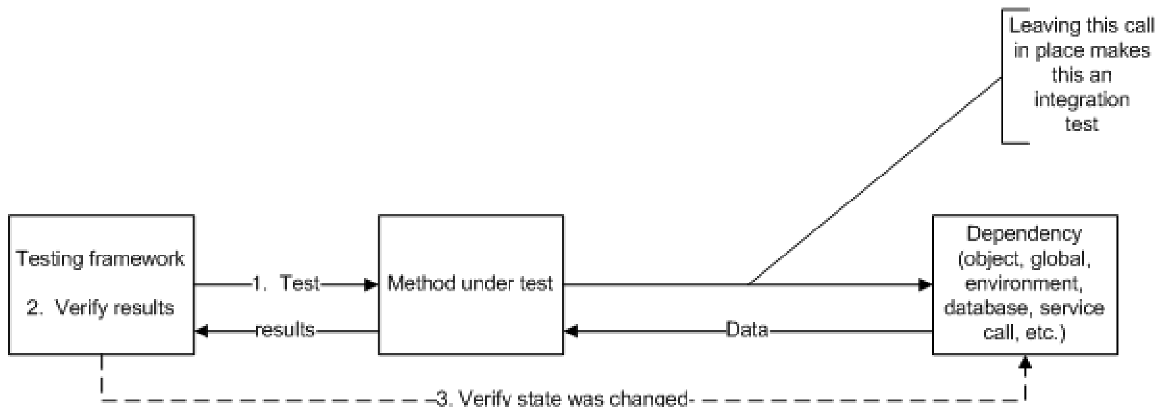


The problem is always dependencies

The simplest scenario is the easiest to test. Call a stateless method, ask it to invoke its logic, then examine the results. There should be no problems writing this kind of test.

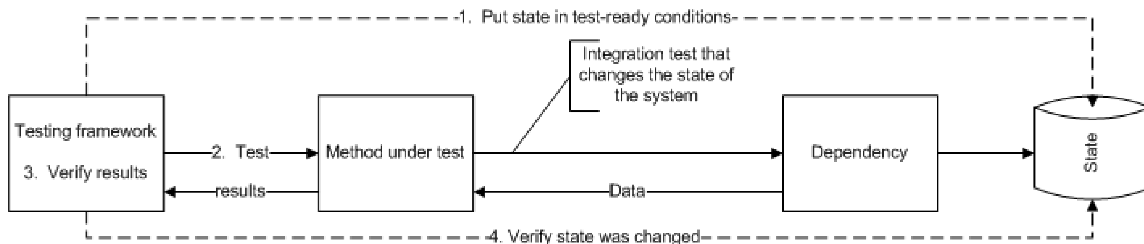


The next common scenario involves a method that uses or changes some external state, such as reading from or writing updates to a database.



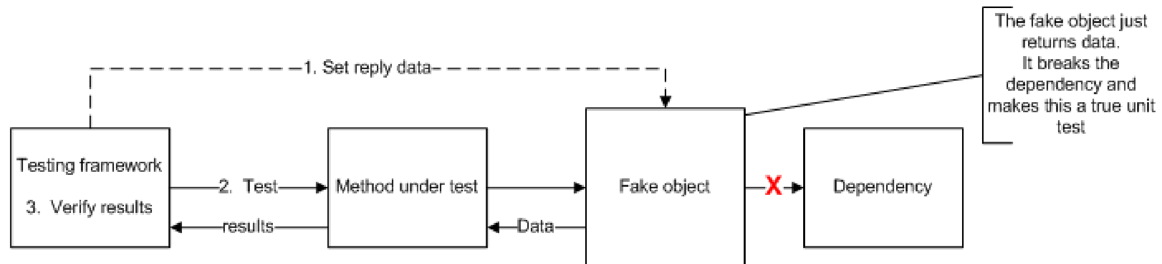
Consider all the problems databases pose to a test. Does the testing framework have access to the database server? Does it have write permission to the database? Is the database server online? Does the query take 30 minutes to run? Will the testing database exist three years from now, or when an external contractor needs to run the tests outside of the firewall?

Even if the database does exist, and is accessible, does the test database have the right schema? Does it have the right rows in it? Do those rows have the right data for the start of the test? If so, we could take this approach: have the test framework set up the database, make the calls, then check the database to make sure the new data is correct. It looks like this:

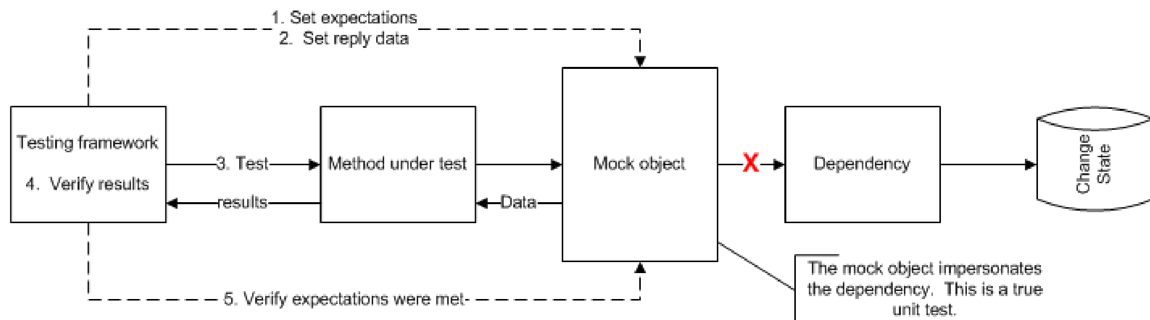


That works well, and is the correct approach for an automated integration test. But the intent of a unit test is to test the method, not the database. Yet any little database issue will keep the test from running successfully. What's the solution?

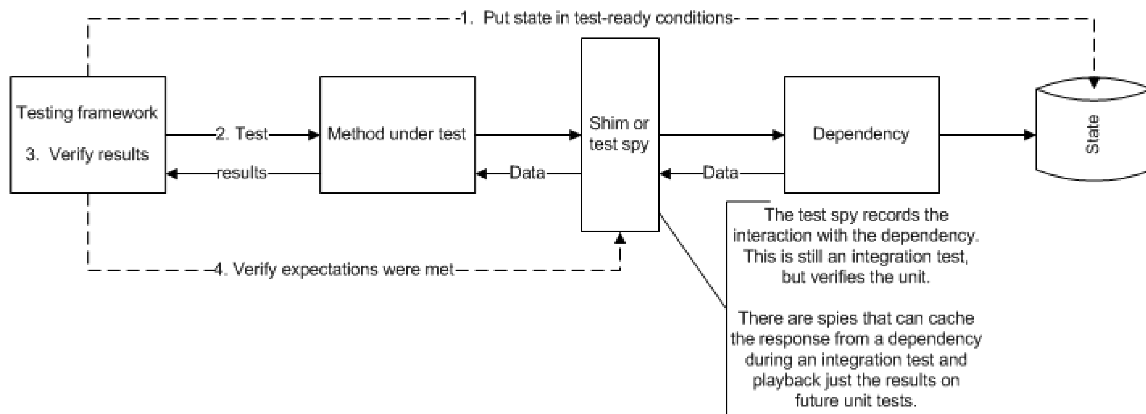
The answer lies in providing a fake source of data. Instead of having the method call a real database object, have it call a fake database object that is specially created to provide the data needed to exercise the test.



What if a method has a complex interaction with a dependency? Perhaps it controls a device that must be properly initialized using a structured sequence of calls, e.g. Open(deviceName), ClaimDevice(bool exclusive), EnableDevice(), and then needs to undo the sequence by calling DisableDevice(), ReleaseClaim(), and finally Close(). How do we get the fake object to be smart enough to report on the correctness of the interactions? The answer is to use a mock object framework, and before running the test to initialize the mock object with the expected calling protocol.



Of course, the real world is often much messier than these pictures. Sometimes the interaction with the dependency is so complex that it's easier to capture or monitor a successful interaction than it is to code it. The test spy is such a tool.



Test Doubles

Test doubles act as stand-in for a dependency. They replace a real component for testing purposes. This may be done to speed up a test, to avoid an external dependency, to stand in for a component that isn't yet ready, or to stand in for a component that's too difficult to instantiate. A double can also check for correct behavior by the module under test. They often return values that cause the system under test to flow down a desired path for testing.

- Test Dummy
  - Very simple object that meets a method signature requirement that isn't important to the test. Null objects or null strings are common dummies.
- Test Fake
  - Simplistic version of a true object. Primary use is to stand in for a dependent component that is not yet built. Often used to replace a database or service.
- Test stub
  - Returns the expected results, and/or calls expected methods. Used to force production code to flow down a desired path.
- Test mock
  - Initialized with a set of expectations that it will be called in a certain way, then observes the calls made and throws an exception in case of failure. Returns values that force production code down a desired path.
- Test spy
  - Observes the calling code, recording what was called for later verification. Returns values that force production code down a desired path.

Doubles can exist in several flavors and have several aspects.

- Responder
    - Injects valid inputs to cause code to flow down a desired path.
- Saboteur
    - Injects invalid inputs to cause testing of exception handlers.
- Temporary stub
    - Stand in for a component that is not yet built.
- Entity Chain Snipping
    - A responder that uses a few objects to act as a large network of objects.
- Hard coded
    - Responses are embedded in the double's code.
- Configurable
    - Responses are configured during test setup.

Mocking frameworks

A mocking framework provides mock objects that can be configured with expectations (a test mock), a recorder for later verification (a test spy), both, or neither. Generally, these construct the needed objects via reflection. In languages that don't support reflection (such as C++), mock objects must be explicitly declared.

Because a mock object is typically used to stand in for a complex dependency, a Test Driven Development approach to constructing software can minimize the number of complex mocks needed. This can reduce or eliminate the need for a mocking framework altogether.

So-called "tunneling mock frameworks" (such as Moles and Stubs, or the Fakes Framework) support replacing any object with a mock. This is useful when dependency injection is unavailable, such as third-party library code that interacts with OS/framework objects. Care must be taken to ensure that the use of these are limited to substituting code that is out of control of the developer, and not used merely to bypass the dependency inversion principle. Tunneled mocks can be avoided by wrapping the problematic library in an abstraction layer, and then using dependency inversion to access the API.

Objects

Different types of objects are common in unit testing

- Data transfer object
  - An object with no behavior that contains only values.
- Null object
  - An object with no values or behavior, often used as a test dummy.
- Explaining variable
  - A literal value assigned to an object, given a name to make its intent clear.

Verifications

After executing the code, the results must be verified to meet expectations.

- Assertions
  - During the inspection of the results, an assertion is a method of the test framework that is called to test for success, and informs the test framework of any failure. Assertions are used in unit tests instead of "if" statements. Any unfinished test should throw an assertion until it is complete.
- Verify state
  - Examine the state of the system under test after execution to make sure it's exactly as expected.
- Delta assertion
  - Examine the state of the system before and after the test to make sure the differences are no greater than expected.
- Behavior verification
  - Examine the behavior of the system during the test. Often done through mock objects.

Test refactorings

- Minimize data
  - Remove anything unneeded or unused in a test.
- Inline resources
  - Move contents of an external resource into the fixture setup method.
- Setup external resource
  - Create the external resource in the fixture setup instead of relying on the environment. (Files, database rows, etc.)
- Make resources unique
  - Making the names of resources unique reduces inter-test dependencies.
- Replace dependency with test double
  - Break dependencies
- Extract testable component
  - Modify the system under test by extracting the code you want to test into a separate component that is specifically designed for testability.

Good test attributes

Good tests are simple!

- Testing one thing at a time vs. testing many things in one test
  - Testing many things in one test will exit at the first failure, leaving the rest of the things untested. But tests with extensive setup lead to excessive duplication of the test code. Use your judgement here.
- Readable test
  - A test should clearly state what it is going to test, what it is passing in, what it is receiving out, what its expectations are, and if they are met. Use explaining variables instead of literal values. Use meaningful names.
- Single path
  - Test code should have a single path, with no branching and no (OK, minimal) looping. Instead of "if (expected)", use the assertions provided by the test framework. Instead of loops, use multiple tests that test the individual scenarios specifically. Complex tests are not readable, and can fail.
- No extraneous code
  - Tests are often copy and paste affairs, passing in slightly different parameters to exercise different paths through the same code. Eliminate copy and pasted code that isn't used in the test. Duplicate code should be refactored out to the test fixture setup method.
- Organize tests with clear names
  - Name the test suite <originalClassName>_Test
  - Name the method test <originalMethodName>_<descriptionOfTest>
  - For parameters passed into a routine, use the method parameter names as the local variable names.
  - Rename liberally.
- Use assertions provided by the test framework
  - Framework assertions do the appropriate logging and reporting
  - Single outcome assertions such as "fail" or "testIncomplete" always behave the same.
  - Stated outcome assertions, such as assertObjectNotNull or assertTrue evaluate a single result.
  - Equality assertions compare a result to an expected value, ensuring the expected and actual results match exactly.
  - Fuzzy equality assertions ensure the difference between an actual and expected result is within an allowable tolerance.
  - Expected assertion outcomes evaluate a method to ensure it raises an expected exception.

SOLID Design Principles[5]

Principle	Description	Effect
Single Responsibility Principle	A class should have only a single responsibility, and that responsibility should be entirely encapsulated within that class.	A class should have one and only one reason to change. An example is a class that compiles data and prints a report. Because it could be changed due either to content or to formatting requirements, those responsibilities should be divided between two classes. This principle improves modularity by increasing cohesion.
Open/Closed Principle	Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.	An interface must remain stable to provide backward compatibility. If additional or changed functionality is needed, simply add another interface, and add a new method on it. This principle improves maintainability by decoupling implementations from each other, strongly enabling asynchronous deployment.
Liskov Substitution Principle	Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.	This makes objects interchangeable while keeping them individually provably correct. This principle promotes reuse of modules by ensuring the compatibility of their interfaces, reducing both coupling and dependencies.
Interface Segregation Principle	Many client-specific interfaces are better than one general-purpose interface.	No client should ever be forced to depend on methods, properties, or parameters that it does not use. The interface should be tightly tailored to the service it provides. Changing one interface does not require changes to the others. This principle increases readability and understandability by reducing coupling.
Dependency Inversion Principle	Depend upon abstractions. Do not depend upon concretions.	If you depend on something not to change, then you are unable to change the thing that is depended upon. Instead, depend only on an interface (the abstraction) to the thing, permitting the thing to be changed independently. This is the overarching principle that ties the other principles together. Dependency Injection is a pattern that enables a way to follow this principle. This principle strongly enables change, and it enables testability by reducing coupling.

[5] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Test problems

Writing tests isn't without problems. Writing tests for existing code can be extremely difficult. Most of the tough testing problems come from having to satisfy dependencies. Here is a list of types of code, and why they are easy or hard to test. Understanding that code should have these different purposes will help you avoid mixing them.

Code structure

Logic type	Stateful	Stateless
Construction	Stateful construction is generally the sign of a factory or pure data object. With no execution logic present, testing this code is limited to simply ensuring it doesn't crash.	This kind of code rarely exists as it doesn’t do any useful work. If it triggers a related change, consider refactoring the code to make the relationship more explicit.
Execution	Impossible to directly examine the values of private members, they can sometimes be checked through getters. If complex objects are used, it may require the use of mock objects to examine changes to state.	It's easy to test the returned values of a stateless function. Code that calls methods on complex objects requires only passed-in fake objects to stand in instead of mock objects.
Mix of construction and execution	Almost impossible to unit test this module as is. Cannot substitute doubles when the objects are constructed in the code. Cannot check the values of private members.	Not able to write true dependency-free unit tests as the code will have dependencies on the constructed objects that can't be substituted with mocks. Can still write tests that examine results, as long as the dependent objects don't add further hidden dependencies on external data, but the tests are fragile.

Issues that make code hard to test

Flaw	Why is this a flaw	Recognizing the flaw	Fixing the flaw
Global state / Singletons / Static methods	Globals enable "spooky action at a distance". Globals / statics cannot be substituted for testing. Code is brittle. Code is very confusing. Highly coupled designs are difficult to test.	Presence of globals (either process or module level). Presence of singletons ("globals in sheep's clothing"). Presence of static fields Code calls static methods Tests fail in suite but work individually. Tests fail if you change order of execution.	Encapsulate global state in a stateful object. Pass in state object that you need. Use dependency injection pattern to help correct the flaws.
Production code contains test flows	The actual production code is not always being tested. The test code may cause defects in the production code.	Production code contains "if test" logic	Remove production code that exists only for unit testing. Replace with actual unit tests.
Constructor does real work	Violates the single responsibility principle. Testing constructors is difficult. It forces collaborators on you.	If the "new" operator in a constructor constructs anything you might want to replace in a test. Static method calls in constructor. Conditional or loop logic in constructor. It's the same flaw if this work occurs in an initialization method.	Pass collaborators into constructor instead of creating them. Isolate dependencies and extract them. Divide stateless function and stateful data into a service class and a value object.
Digging into collaborators	API hides dependencies. Code is brittle. Code is confusing. Hard to test because you don't know all the dependencies. Violates "Law of Demeter[6]". High coupling.	If a call goes deeper than one de-reference. Class primarily dereferences another class instead of accessing its own members.	Pass only the collaborators in that you need. Pass all the needed collaborators in. Divide collaborators into stateful DTOs and stateless service objects. Don't talk to strangers.
Class does too much	Hard to debug. Hard to test. Hard to understand. Hard to extend. Hard to explain. Hard to name. Low cohesion. High coupling.	A good description of the class needs the word "AND". Excessive scrolling to read logic. Hard to name. Class contains unrelated methods. Many methods or fields. "Feels too large".	Split it up. Each newly divided class should have a clear and obvious name. Use a new class to mediate the interactions of the old data. Keep state-affecting code near the top of your call graph.
Mixing object construction with application logic	Can't substitute mock objects. High coupling.	Presence of "new" operator in application logic for anything other than simple data. Creating helper or service objects .	Extract object construction from service methods into factory classes. Pass in helper objects in method invocation.

[6] http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html

Flaw	Why is this a flaw	Recognizing the flaw	Fixing the flaw
Using inheritance instead of composition	Can't replace inherited functionality with mocks or fakes. High coupling.	Inheritance used for anything other than polymorphism. Presence of mix-in classes.	Don't inherit unrelated functionality. Pass in collaborators to reuse their code.
Using conditional logic instead of polymorphism	Makes methods complex. Hides intent.	Presence of switch statements. Presence of repeated "if" clauses.	Extract switch code to strategy pattern. Extract if clauses to strategy pattern. Extract complex logic involving state transitions to finite state machine pattern.
Mixing service objects with value objects	Can't separate data from logic for testing. High complexity. Hard to reuse services.	Public data or getters and setters with business methods in same class.	Extract data into value objects (domain objects or data transfer objects). Extract logic into business services. Pass value objects into new services.

Design patterns to help break dependencies

When writing a unit test for legacy code, you will find methods that are difficult or impossible to test.

- Adapter / Bridge
    - By extracting structural problems that make code hard to test into two or more pieces, you create a backward compatible and still hard-to-test thin outer wrapper, but an easily testable inner core.
- Dependency Injection
    - By extracting depended upon components to make them externally provided, you turn stateful code into stateless code, and give yourself a substitution point for test doubles.
- Strategy / State
    - By extracting ladder logic into state objects, you simplify the code. You can independently test the conditions that the ladder used to evaluate in series.

Recommended Reading

Design Patterns : Elements of Reusable Object-Oriented Software.
Gamma, Helm, Johnson, and Vlissides. 1995, Addison-Wesley. ISBN 0201633612. This is the seminal book on categorizing software design patterns, and it is still highly relevant today. It is a reference book, meant to be used to help make informed architectural choices.

Refactoring : Improving the Design of Existing Code
Martin Fowler. 2000, Addison-Wesley. ISBN 0201485672. This is the groundbreaking textbook on the practice of refactoring, which is the act of rearranging code to improve its qualities. By referring to "code smells", it lists ways to restructure the code that do not impact its correctness, but help make it modular and maintainable.

Working Effectively with Legacy Code
Michael Feathers. 2005 Pearson Education. ISBN 0-13-117705-2. This book introduced me to the exciting idea that an existing out-of-control codebase is always salvageable. It suggests strategies for refactoring and adding unit tests to modernize any codebase.

xUnit Test Patterns : Refactoring Test Code
Gerard Meszaros. 2007, Pearson Education. ISBN 978-013-149505-0. This book is in two parts. The first part is a 181 page section that provides an overview of the goals, principles, philosophies, patterns, smells, and coding idioms related to unit testing. The second part is a reference that lists the various code smells and patterns used to correct them. The first part is the best introduction I've read, but the potential reader may be put off by the overall size of the tome, which weighs in at 883 pages.

The Art of Unit Testing : With Examples in .NET
Roy Osherove. 2009, Manning Publications. ISBN 978-1-933988-27-6. This book is an excellent and practical introduction to automated unit testing, and at 296 pages is fairly short.

Test Driven Development : By Example
Kent Beck. 2003, Pearson Education. ISBN 978-0-321-24653-3. This is historically significant as the first comprehensive book on the topic, and introduced many people to the practice of Test Driven Development. But the advice is showing its age, and more recent work has refined the practices.

Growing Object-Oriented Software, Guided by Tests
Steve Freeman and Nat Pryce. 2010 Pearson Education. ISBN 978-0-321-50362-6. This book explains the Test Driven Development cycle, and demonstrates how TDD is used to develop software one feature at a time, rather than designing an entire project in advance.

Agile Software Engineering with Visual Studio : From Concept to Continuous Feedback
Sam Guckenheimer and Neno Loje. 2012 Pearson Education. ISBN 978-0-321-68585-8. This book covers the Agile software methodology, and how to use Visual Studio to manage a project. Agile is based on Test Driven Development, which is in turn based on Automated Unit Tests.