Designing for Testability

Good design coincides with good testability (usually)

unit tests do not save us from poor design, such as:

- mega-functions within mega-classes
- creating copies of data that might change in the original and need syncing
- making things threaded without making them thread-safe
- etc

Thinking about any class

The public API consists of:

- create/destroy
- public methods — inputs & outputs

But also!

- protected or virtual hooks intended for subclasses to use

- interaction with subordinate objects

- interaction with loosely-coupled objects (message queuing, signals, notifications, delegates, interfaces, etc)

- interaction with overly-familiar objects

- state changes to global data

- operating system API calls

- 3rd-party library calls

When working on the design of a class, you are delivering the functionality along with a test suite — build them in tandem

When designing a class, think it through up front:

- coherent purpose

- crisp public API

- “seams” at all the interaction points

- test suite that: exercises the public API, senses at the seams, and provides faked-up data as necessary from across the seams

For each interaction point, decide:

- how will I sense that the class under test is sending the right message across the boundary?

- how will I provide fake responses/signals coming from the other guy?

- simplest is best; only introduce complexity as necessary. If an instance of the real class is sufficient and does not introduce difficult dependencies or performance hassles, just use it. Otherwise, choose a seam strategy that gets you what you need (input, output, or both) for the least complexity. You're probably also improving the structure of the class even apart from testability! (Sometimes in a "you'll thank me for this later" way.)

- by introducing a seam, you are saying: for the purposes of this test suite, I don't care what's going on on the other side of the seam — just that this class is throwing the right stuff over the fence. (If you're worried about what's going on over there, work it out in the test suite associated with that other guy.)

The test suite for this class should:

- fully exercise the public API (constructors, method calls, and signaling mechanisms initiated from outside he class)

- run through common interaction scenarios where the sequence of calls is important

- for edge cases, test them in the simplest possible scenario

- if you’re having a hard time exercising/sensing a particular line of code, you probably need to bubble off a subordinate class and do the testing as part of this new class’ test suite (back at this guy’s test suite, you can then only care that the right stuff is getting sent over the fence).

Provide a layer of insulation around stuff that’s hard to test, and make the hard stuff as “declarative” or “point-to-point” as possible:

(all branching and computation logic should be in a Model class and tested as part of the Model class’ test suite)

Controller classes should be nothing more than wiring closets (neat ones):

- translate user interactions into requests on the model (with no assumptions about what it will do)

- translate model state changes to representations in the user interface

- navigation

- flair

Thread classes should do nothing more than turn the crank on some other class

- consider abstracting out the thread processing/synchronization to enable reuse, and then don’t worry about unit testing that part every time you simply use one of them

Callbacks, async blocks, notification handlers, etc. should do nothing more than turn around a single call to the object

Troublesome to unit test

- filesystem manipulation

- network resource (e.g., web server; database)

But also!

- globals (including Singletons)

- user interface

- asynchronous work

- hardware values (MAC address, current time, sound I/O, etc)

- system resources: semaphores, shared memory, named pipes, etc

- deeply nested private methods protected by logic gauntlets

Value of unit tests

quickly isolate unwanted behaviour

support refactoring

detect regressions

Value of TDD

“quickly” and “isolate” can speed development (edit/compile/run cycle), rather than be a drag

reduce roundtrips through testers

spend time thinking through a good API for this class

spend time designing good ownership semantics, object graph, dependency graph, etc

Loose coupling

-> good design (usually)

-> provides easy “seams” for unit test sensing and mocking

Strategies for loose coupling and sensing:

interfaces

protocols

message passing

delegate/visitor patterns

virtual methods

getters that report on the internal state of the object

primitive data types as parameters

simple data types as parameters (e.g., dumb data classes/structs)

dependency injection

multiple initializers and/or setters for complex dependencies

multiple constructors

internal getter that can be overridden in a testing-only subclass

function pointers/callbacks

templates

generics

iterators

closures

“variant” data types

preprocessor

technology-specific opportunities: e.g., Qt signals/slots, SQLite in-memory database, STL streams, Key-Value-Observing, data-binding

parameterize time

-> when not to; overdoing it (e.g., making everything virtual; making everything VARIANTs)

Counterproductive unit tests

Post-hoc bolt-on tests

Tests that lock in bad design

Tests that lock in implementation details (e.g., mocks that expect calls in a certain order when order doesn't matter)

Complex mocking/injection such that you can't tell what you're testing (or if you're testing anything real at all)

Testing at the wrong level (reaching deeply into private methods or through delegates)

Repeating all the same checks at a higher level

Didn't do fail-first TDD, and so a unit test always trivially passes

Repetition for the sake of buffing test count metrics

Convoluted gymnastics for the sake of buffing code coverage metrics

Proliferation of low-value tests (e.g., NULL pointer checks — instead, try passing by reference or using asserts)

Blind application of buzzwords, rather than thoughtful application of techniques

Where unit tests may not be appropriate, be creative about other ways to ensure quality (e.g., screen shot comparison regression suite; Selenium integration; scripting language; mock web server)