`deject`: Writing Testable Code in D

Ben Gertzfield <beng@fb.com>

If you take away one thing from this talk, make it this:

Writing tests can be easy, if you separate out your dependencies.

Why write testable code?

Making your code testable lets us make new and interesting mistakes

(instead of those same old boring mistakes)

Don't live in fear of touching code you didn't write

And don't live in fear of touching code you did write…

(6 months ago)

Let's Talk Testing

When describing code, the word unit might bring to mind:

- A single function
- A class
- A module (in D, a single file)

Keeping your personal definition of "unit" small is the key to both comprehending and testing software.

Basics of Unit Testing in D

D is the first major programming language with unit testing built in.

Any code in a `unittest { ... }` block is executed before `main()` when the program is compiled with `dmd -unittest`

Unit tests contain logic and `assert` statements.

What does a basic unit test look like?

Newton Rolling Over In His Grave

```
import std.math;

immutable auto EPSILON = 1e-6;

T sqrt(T)(T val) {
  auto guess = val / 2;
  while (guess > 0 && abs(val - (guess * guess)) >
EPSILON) {
    guess = (val / guess + guess) / 2;
  }
  return guess;
}

unittest {
  assert(abs(sqrt(16) - 4) < EPSILON);
  assert(abs(sqrt(0)) < EPSILON);
  assert(abs(sqrt(100) - 11) > EPSILON);
}
```

Other kinds of tests

Integration tests

These exercise multiple units in concert.

Integration tests ensure that all your units are glued together correctly.

They have larger surface area, so they fail more often and more mysteriously when a unit misbehaves.

Other kinds of tests, cont.

End-to-end tests

These exercise the entire process from start to finish.

These are great for smoke tests to make sure a build isn't dead on arrival.

But… they get flaky even quicker than integration tests.

Writing and maintaining end-to-end tests is expensive and time-consuming.

Isolating dependencies for unit tests

To make code testable, we need to isolate the actual code from its dependencies.

"Constructor injection" requires all the dependencies of a unit to be passed to its constructor.

Let's look at a flaky test to see why this is useful.

```
import std.socket;
import std.stdio;

void checkHost(string host) {
  auto ih = new InternetHost;
  if (!ih.getHostByName(host)) {
    logError("No DNS: " ~ host);
    throw new Exception("Host down: " ~ host);
  }
}

void logError(string err) {
  auto f = File("log.txt", "a");
  f.write(err);
}

unittest {
  assertNotThrown(checkHost("dlang.org"));
  assertThrown(checkHost("qlang.org"));
}
```

This is a flaky test. It will fail:

When the network is down

If someone registers `qlang.org`

When the filesystem is full or read-only

When I hit Control-C after waiting 30 seconds for the test to complete

```
% time rdmd --main -unittest counterexample.d
30.838 secs
```

Let's make this testable

We'll use constructor injection to pass dependencies from above.

```
import std.log; // Wishful thinking..
import std.socket;

class HostChecker {
  private InternetHost ih;
  private Logger logger;

  this(InternetHost ih, Logger logger) {
    this.ih = ih;
    this.logger = logger;
  }

  void checkHost(string host) {
    if (!ih.getHostByName(host)) {
      logger.error("No DNS: " ~ host);
      throw new Exception("Host down: " ~ host);
    }
  }
}
```

What changed here?

- Pass dependencies from above to constructor
- Clearly define seams to stitch together logic and dependencies
- Avoid static methods and globals

Let's Test This Puppy

```
unittest {
  import dmocks.Mocks;
  auto host = "a.b.c";
  auto mocker = new Mocker;
  auto ih = mocker.mock!InternetHost;
  auto logger = mocker.mock!Logger;

  mocker.expect(ih.getHostByName(host));
  mocker.replay();

  auto checker = new HostChecker(ih, logger);
  checker.checkHost(host);

  mocker.verify();
}
```

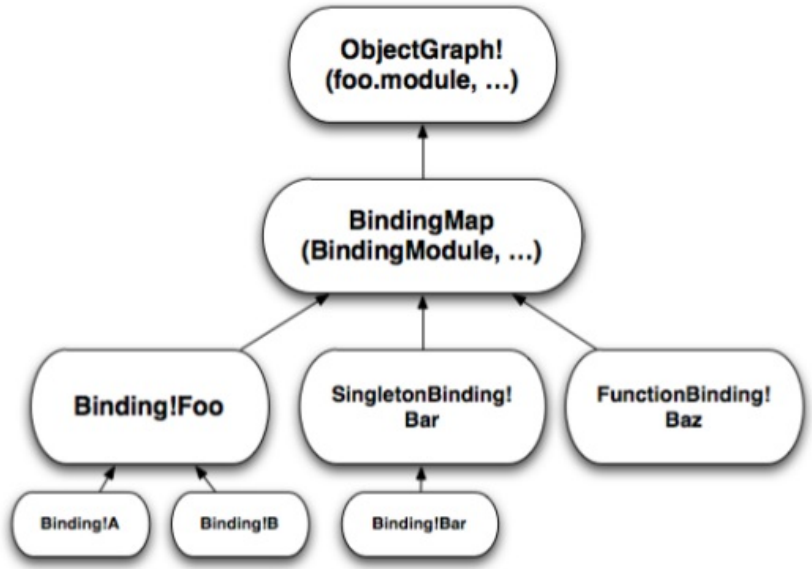
Let's test a failure case

```
unittest {
  import dmocks.Mocks;
  auto host = "a.b.c";
  auto mocker = new Mocker;
  auto ih = mocker.mock!InternetHost;
  auto logger = mocker.mock!Logger;

  mocker
    .expect(ih.getHostByName(host))
    .returns(false);
  mocker
    .expect(logger.error(""))
    .repeatAny()
    .ignoreArgs;
  mocker.replay();

  auto checker = new HostChecker(ih, logger);
  assertThrown(checker.checkHost(host));

  mocker.verify();
}
```

The Good, The Bad, and the Dejected

- The good
  - Writing code this way makes it simple to test
- The bad
  - Calling the constructor with all its dependencies is annoying and requires clients change any time the dependencies change
- The dejected
  - D is a modern programming language: can't it take care of these details?
- Introducing `deject`, dependency injection for D

The library: `deject`



Runtime dependency injection

`BindingModule`s define map of (`TypeInfo` → `Binding`) pairs

`ObjectGraph` analyzes graph of dependencies and uses `Linker` to find and cache dependencies of `Binding`s

Manages object lifetime for singletons and other objects with scoped lifetime

Compile-time dependency injection

D's compile-time introspection lets us build the object graph at compile time

Looks for `@Inject` attribute to denote classes with dependencies managed by object graph

Emits D mixins to define `ObjectGraph.get!T` to construct injected type `T`

One small change to make that code injected

```
import deject.attributes;

@Inject
class HostChecker {
  private InternetHost ih;
  private Logger logger;

  this(InternetHost ih, Logger logger) {
    this.ih = ih;
    this.logger = logger;
  }

  // ...
}
```

Wrapping it up

Use pass from above to separate dependencies from your code

Use mocks to control behavior of dependencies in tests

Grab the source: **github.com/bgertzfield/deject**

Questions?