Testable Code
Testing Testable Code Is Easy

So you decided to finally give this testing thing a try. But somehow you just can't figure out how to write a unit test for your class. Well, there are no tricks to writing tests, **there are only tricks to writing testable code.**

If I gave you testable code you would have no problems writing a test for it. But, somehow you look at your code and you say, "*I understand how to write tests for your code, but my code is different*. Well your code is different because you violated one or more of the following things.

Things That Make Code Hard to Test: Complex Constructors

- Mixing object graph construction with application logic
  - If the code under test creates its own collaborators with `new`, it won't be possible to strategically replace them with test doubles.
- Ask for things, Don't look for things (aka Dependency Injection / Law of Demeter)
  - A class should ask for its direct dependencies as constructor parameters, and not ask for something else which is then used to locate the direct dependencies.
- Doing work in constructor
  - Anything in the constructor must be executed on every test. If it does complex things (like read a config file), it complicates the setup of every test that uses the class.

Things That Make Code Hard to Test: Global Variables

• Global State
  • In production the application is instantiated only once, but in tests each test is a small instantiation of the application. Global mutable state creates dependencies between tests and causes undefined behaviour.

• Singletons (global state in sheep's clothing)
  • The static singleton antipattern [GoF] is the same as global variables. All internal objects of the singleton are global as well, recursively.

Things That Make Code Hard to Test: No Polymorphism

- Static methods (or living in a procedural world)
  - Procedural programs have no seams, or polymorphism (i.e. at compile-time the method you are calling can not be determined). Inserting test doubles to isolate the system under test is not possible. (Simple leaf methods such as `Math.abs()` are good, if they stay simple, but the main method can not be tested in isolation!)

- Favor composition over inheritance
  - Inheritance can not be changed at runtime – can not use test doubles. Using inheritance for code reuse is wrong.

- Favor polymorphism over conditionals
  - If you see a switch statement or a repeated if condition, think about polymorphism. Using polymorphism creates smaller classes, which in turn are easier to test.

Things That Make Code Hard to Test: Badly Structured Classes

- Mixing Service Objects with Value Objects
  - *Value-objects* do not refer to service-objects, can be created with `new`, are never mocked, are often immutable.
    - Examples: LinkedList, EmailAddress, CreditCard.
  - *Service-objects* collaborate with other service-objects, are never created with `new`, are good candidates for mocking.
    - Examples: MailServer, CreditCardProcessor.
  - Mixing the two creates a hybrid which has no advantages of value-objects and all the baggage of service-objects.
- Class has multiple responsibilities
  - If a class does more than one thing (*Single Responsibility Principle*), in reality it has multiple objects hiding in itself and you will not be able to test each of them in isolation.

http://www.infoq.com/minibooks/domain-driven-design-quickly
Global State

insanity  *noun*

- Repeating the same thing and expecting a different result.

```java
int a = new X().doSomething();
int b = new X().doSomething();

- Does: `a == b` or `a != b`
```
a = new X() -->
a.doSomething()

a==b ✓

b = new X() -->
X
b.doSomething()

a = new X() -->
a.doSomething()

a==b ❌

b = new X() -->
X
b.doSomething()
Singletons and Global State

- The static singleton pattern is evil
  - Only one instance per JVM (or ClassLoader), but JVM scope != Application scope. Especially each test needs a new application instance.

- Creating just one instance is good
  - Create one instance of the class when the application starts, and pass it using dependency injection to all who need it.

- Presentation: (54 min)
  - Fight Mr. Untestable Trick #2
    - Avoid Global Mutable State
  - Good Singleton vs. Bad Singleton
    - Encapsulation vs Deceptive API
    - Order of tests should not matter

Dependency Injection

- Do not construct collaborators with `new`. Receive them as constructor parameters.
  - `new` will be used only inside factories or a DI framework.
  - Separates behaviour from dependency resolution.
  - Makes the dependencies of a class explicit.
  - Makes testing easier, enables mocking the dependencies.
- DI is a design pattern – a framework is not needed.
  - In a small application you can write factories manually, or even put everything together in the `main` method.
  - With a medium to large application, using a DI framework removes much boilerplate code, provides scopes, AOP etc.
    - For example: Guice ([http://code.google.com/p/google-guice/](http://code.google.com/p/google-guice/))
Dependency Injection

• Presentation:
  (38 min)

Fighting Mr. Untestable
Trick#1
Ask for Things!
Don’t look for Things!

Cost of Constructor
Always ask for Things
Law of Demeter
Where have all the “new” operators gone?

by Miško Hevery

Course Material