# A Brief and Incomplete History of Testing at IMVU

Andy Friesen

February 17, 2016

# Overview

- Integration
- Functional
- Unit Testing and Mocks
- Dependency Injection
- Automatic Dependency Injection
- Futuristic stuff

# In the beginning...

```
1  def test_my_application_launches(self):
2      p = subprocess.Popen(['my_application.exe'])
3      hwnd = win32ui.FindWindow(None, "My Application")
4      self.assertNotEqual(None, hwnd)
```

# In the beginning...

Integration tests are incredibly frustrating for two reasons:

- Reliability is horrible.
- Sometimes, you really don't have any other choice.

# Functional Testing

Testing everything in terms of UI is error-prone, very slow, and sometimes misses the point.

Rather than drive the entire application, we could write a tiny program that exercises part of the application.

```
1  function test_compute_customers_age_from_birthday($i) {
2      $cid = customer::_test_create(
3          ['birthday' => strtotime('01-01-1977')]
4      );
5      $age = customer::get_age($cid);
6      assertEqual(38, $age);
7  }
```

# Unit Testing

Dropping from an integration test to a functional test is an instance of an important insight:

We can sacrifice some "realness" for performance and reliability by substituting unrelated or unreliable inputs.

```
function test_compute_customers_age_from_birthday($i) {
    $clock = new MockObject;
    $clock->respond('now', '01-06-2015');
    $cid = customer::_test_create([
        'birthday' => strtotime('01-01-1977'),
        'clock' => $clock
    ]);

    $age = customer::get_age($cid, $clock);
    assertEqual(38, $age);
}
```

# Mock Objects

Central to the idea of unit testing systems is a mechanism to substitute unpredictable side effects for "mock" objects that profess to provide the same interface, but provide more consistent behaviour.

A very popular way to do this is with a "replay mock", which is an object that is programmed at the start of the test, for the test.

# A Simple Fake Clock

```
class FakeClock(object):
    def __init__(self, time):
        self.__time = time

    def now(self):
        return self.__time

    def _advance(self, amount):
        self.__time += amount
```

# Fine Tuning: Dependency Injection

Fake objects are what we use to this day, and they work pretty well, but they cause an obnoxious problem: you have a lot of them, and it takes a bunch of effort to plumb them through your software system. (especially if you already have a big software system that needs to be rerofitted!)

Our initial solution was what we called a "service provider". We had a pretty blank kitchen-sink object whose responsibility was to carry a reference to all the services.

# Tiny Example

```
class ChatWindow(object):
    def __init__(self, direct3D, fileSystem, network):
        self.__direct3D = direct3D
        self.__fileSystem = fileSystem
        self.__network = network
```

# Automatic Dependency Injection

This works pretty well, but there's still an annoying problem: Changing the set of services that an object depends on is costly: All the construction sites need to change!

We solved this by adding a feature to the service provider: sp.invoke() takes a data type to be constructed, inspects its dependencies, and passes them into the constructor.

This lets us add or remove dependencies from an object without requiring changes to the code that constructs the object.

# Tiny Example

```
class ChatWindow(object):
    def __init__(self, services):
        self.__direct3D = services.direct3D
        self.__fileSystem = services.fileSystem
        self.__network = services.network
```

# Tiny Example

```
class ChatWindow(object):
    def __init__(self, direct3D, fileSystem, network):
        self.__direct3D = services.direct3D
        self.__fileSystem = services.fileSystem
        self.__network = services.network

# ...
cw = services.create(ChatWindow)
```

# Ultra-futuristic Testing: Haskell

We've started using Haskell as a backend language at IMVU. One of the reasons we really like it is because it lets us write better tests.

But first I need to talk about a bit of syntax.

```haskell
-- double is a function.  It takes an Int, and returns an int
double :: Int -> Int

getCustomer :: CustomerId -> IO Customer
```

# Ultra-futurism Contd.

```haskell
-- Lowercase type names are generics
id :: a -> a

double :: Num a => a -> a -- Generic with a constraint
```

# Ultra-futurism Contd.

```haskell
getCustomer :: World m => CustomerId -> m Customer -- Generic context
```