Evolutionary Anatomy of Test Automation Code

by George Dinwiddie

May 2017 Edition

Evolutionary Anatomy of Test Automation Code

George Dinwiddie

This book is for sale at http://leanpub.com/EvolutionaryAnatomy

This version was published on 2018-06-06

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By George Dinwiddie

Patterns of Agile Journeys

to my lovely wife, Gail, with thanks for putting up with me

Contents

About this edition

This is a sample of the book ***Evolutionary Anatomy of Test Automation Code*** **by George Dinwiddie**. It contains only a little of the content of the full book, but it's free and will give you a taste of the contents.

It also contains detailed setup instructions for the code at https://github.com/gdinwiddie/EquineHoroscope to help you get started.

May 2017

Rather than chase Firefox/Selenium changes, I've switched to using ChromeDriver as a default. You can still use Firefox by setting the environment variable 'EquineHoroscope.browserDriver=firefox' if your version of Firefox and geckodriver play nicely together.

November 2016

This edition has been updated for Firefox 50 and Selenium 3 which changes the drivers needed to talk to the browser.

Minor corrections made December 2, 2016.

Cover Image

from:
Natural history of Victoria
Prodromus of the zoology of Victoria;
or,
Figures and descriptions of the living species of all classes of the Victorian indigenous animals..

by Frederick McCoy, published 1881

https://archive.org/details/naturalhistoryof610mcco
plate 53

via:
https://en.wikipedia.org/wiki/File:Litoria_aurea_development.jpg
by Arthur Bartholemew

Preface

I started using Cucumber back in 2009, when it was pretty new stuff. As is typical of most new toys, I started using it in ways that ended up not being helpful in the long term. Since then, I've helped a number of clients adopt Behavior Driven Development (not always with Cucumber) and experienced many of the problems that organizations face when they do so.

Many people agree that one important outcome of Behavior Driven Development is a set of regression tests to demonstrate the desired behavior and ensure that it is maintained over time. Then they often struggle to do so in a manner that remains maintainable as the system and the test code grows larger. Sometimes they even abandon their tests and start over, repeatedly.

In this book we'll examine the evolutionary history of an application. We'll examine various stages in its life to consider the choices we might make to address growing complexity. We'll highlight patterns that are useful, at least some of the time.

This book is intended to be evolutionary, itself. The codebase is up on GitHub.

```
https://github.com/gdinwiddie/EquineHoroscope
```

Download it and try your own experiments. I'd love to hear your suggestions and comments.

George Dinwiddie
Pasadena, MD
April 2016/November 2016

Preparation

You'll probably get a lot more out of this book if you download the code and play along. Try your own approaches and evaluate them against the ones I show here. Let me know what you find. I'd really like to hear your views.

"Gitting" the Code

Source on GitHub

The code is shared at https://github.com/gdinwiddie/EquineHoroscope in GitHub. I'm using git version 2.10.2 on the command line. Other versions, including GUI shells, should also work.

Cloning repository from GitHub

- Create a workspace directory.
- Within that workspace directory, clone the repository with
  `git clone git@github.com:gdinwiddie/EquineHoroscope.git`

This will create the `EquineHoroscope` project directory and bring down the entire repository to let you explore history. At several points throughout the text, I've specified `git checkout <branch>` commands you can run on the command line to move to historical versions.

To work with the final version of the code as of this writing, run

```
git checkout May2017
```

I also recommend using

```
cd EquineHoroscope
gitk --all
```

to bring up `git`'s GUI browser. This lets you look at the diffs for each commit.

Library jars

I didn't want to check the libraries into version control, and I also didn't want to add a dependency on any particular dependency management system. Therefore I zipped up the jars I'm using and posted them on the web.

Within the `EquineHoroscope` directory, execute

```
wget http://idiacomputing.com/pub/EquineHoroscopeJars.zip
unzip EquineHoroscopeJars.zip
```

If you don't have these commands, download the zip file and unzip it with whatever tools you have. This will create an `EquineHoroscope/lib` directory with all the jars in it.

In Selenium 3, the Firefox driver is no longer built-in. The Gecko (for Firefox) and Chrome drivers for Windows and OSX are included in the `drivers` subdirectory. I've only tested these instructions with the OSX driver.

As Mozilla keeps updating the version of Firefox in ways that require updating the geckodriver, I've switched to defaulting to running web tests with Chrome, instead. To use Firefox, instead, set the environment variable `EquineHoroscope.browserDriver=firefox` when running the tests.

Using Eclipse

You can use another Java IDE, of course. Many people prefer Intellij. I happen to be using Eclipse, and it's free, so that's what I describe here.

I'm using Eclipse "Version: Mars.2 Release (4.5.2) Build id: 20160218-0600" at the moment. Other versions should work, but there may be small differences.

Import the project

Start up Eclipse. Tell it to use the workspace directory we created above when we were getting the source code.

- "File" -> "Switch Workspace" -> Other... -> browse to the workspace directory that contains EquineHoroscope
- Within Eclipse, select "File", "Import...", "General", "Existing Projects into Workspace", "Next"

- "Select root directory" and "Browse" to the `EquineHoroscope` directory that should be in your workspace.
- "Finish" and Eclipse should show the EquineHoroscope project in the Package Explorer.

Use Java 8

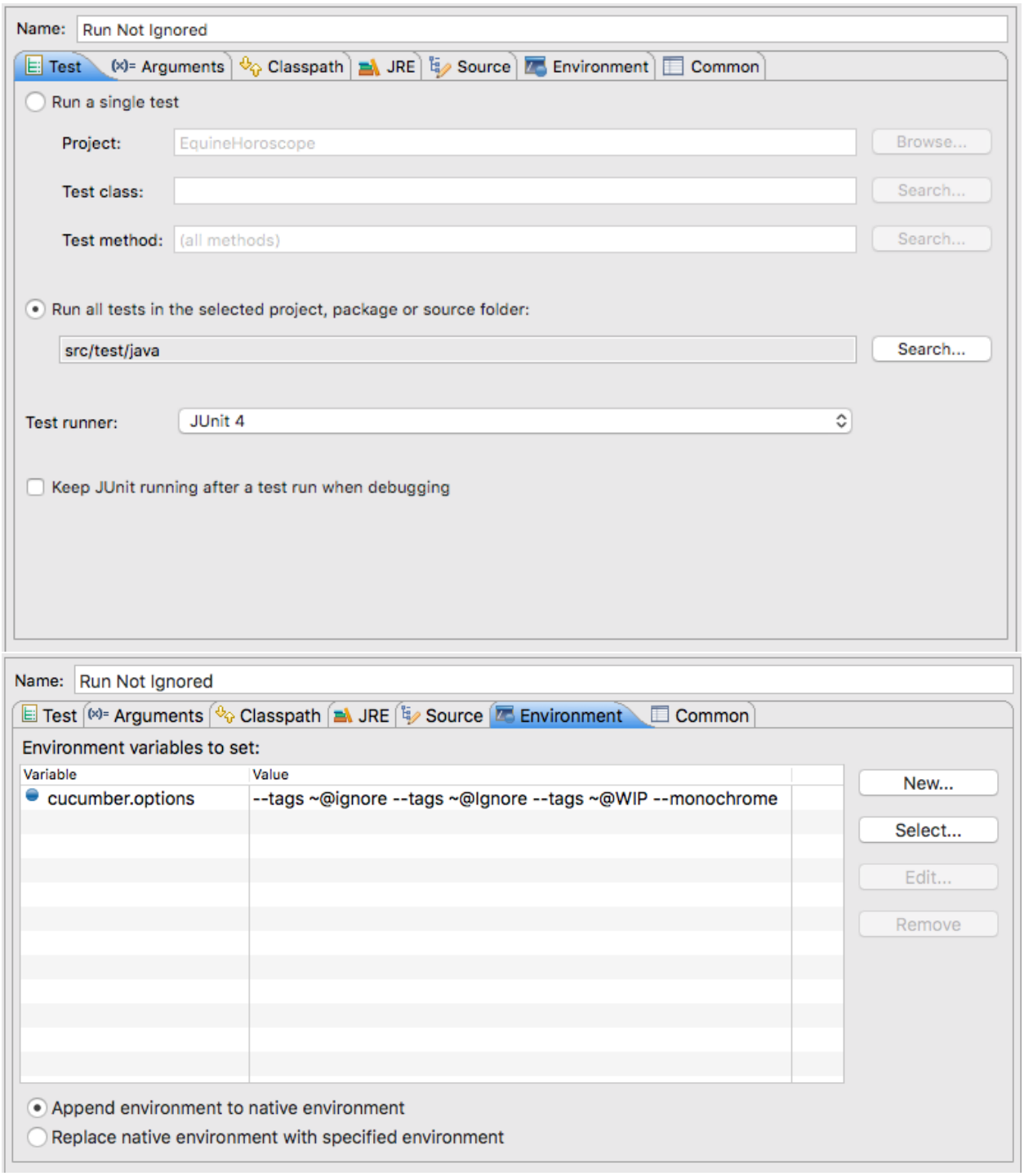
If you haven't used Java 8 before (It's required for some of the dependencies), then you'll need to set that up.

- Right-click on the EquineHoroscope project -> "Build Path" -> "Configure Build Path"
  - -> "Java Compiler"
    * "Enable project specific settings"
    * "Compiler Compliance level 1.8"
    * "OK"
  - -> "Java Build Path"
    * -> "Add Library -> "JRE System Library" -> "Next"
    * -> "Installed JREs..." -> "Search"
    * select "Java SE 8"
  - click "Apply" -> "OK" -> "Finish"

Set Eclipse up to run the tests.

In Eclipse,

- "Run", "Run Configurations..."
- Select "JUnit" on the left-hand column and press the "New" button (the icon of an empty page with a + sign at the top of that column)
- Create a "Run Not Ignored" configuration like this:



- Click “Apply” and “Run”

This should bring up the Eclipse JUnit tab and show 32 tests run and a green bar. At the bottom of the Eclipse Console tab, you should see

6 Scenarios (6 passed)
14 Steps (14 passed)

from the Cucumber-JVM running within JUnit.

If not, it's time to do some trouble-shooting. Look in the Eclipse Console tab and see where the problems lie.

Running the "Merchant Bank" Server

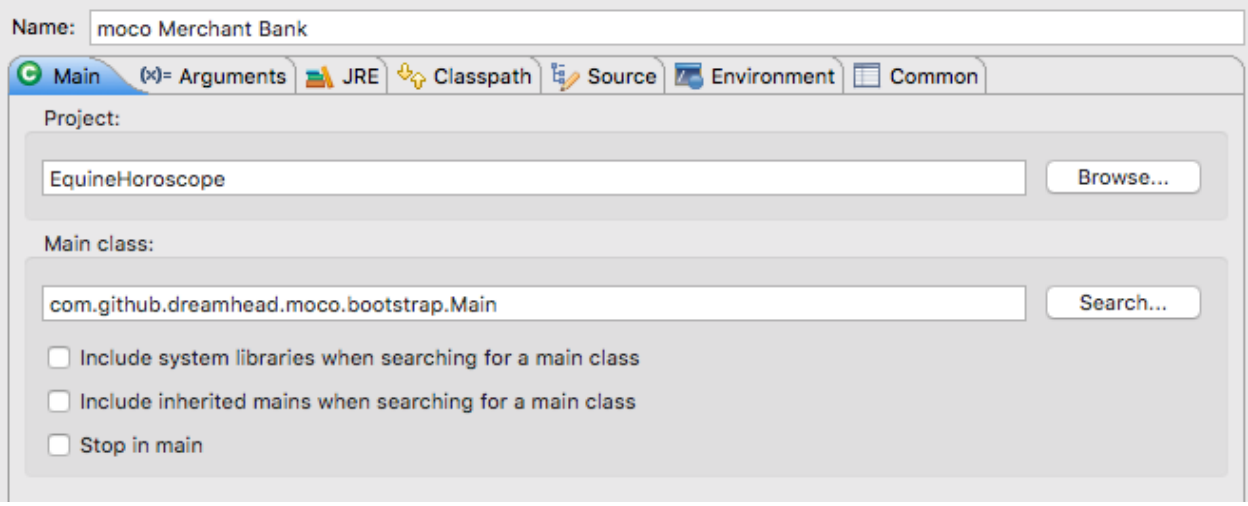
We don't have a real merchant bank account, so we don't have a bank-provided test server to use in development. Instead, we've got a simulated merchant bank test server. In order for our web application to work, we need to have this server running.

You can start this test server from the command line for interactive use.

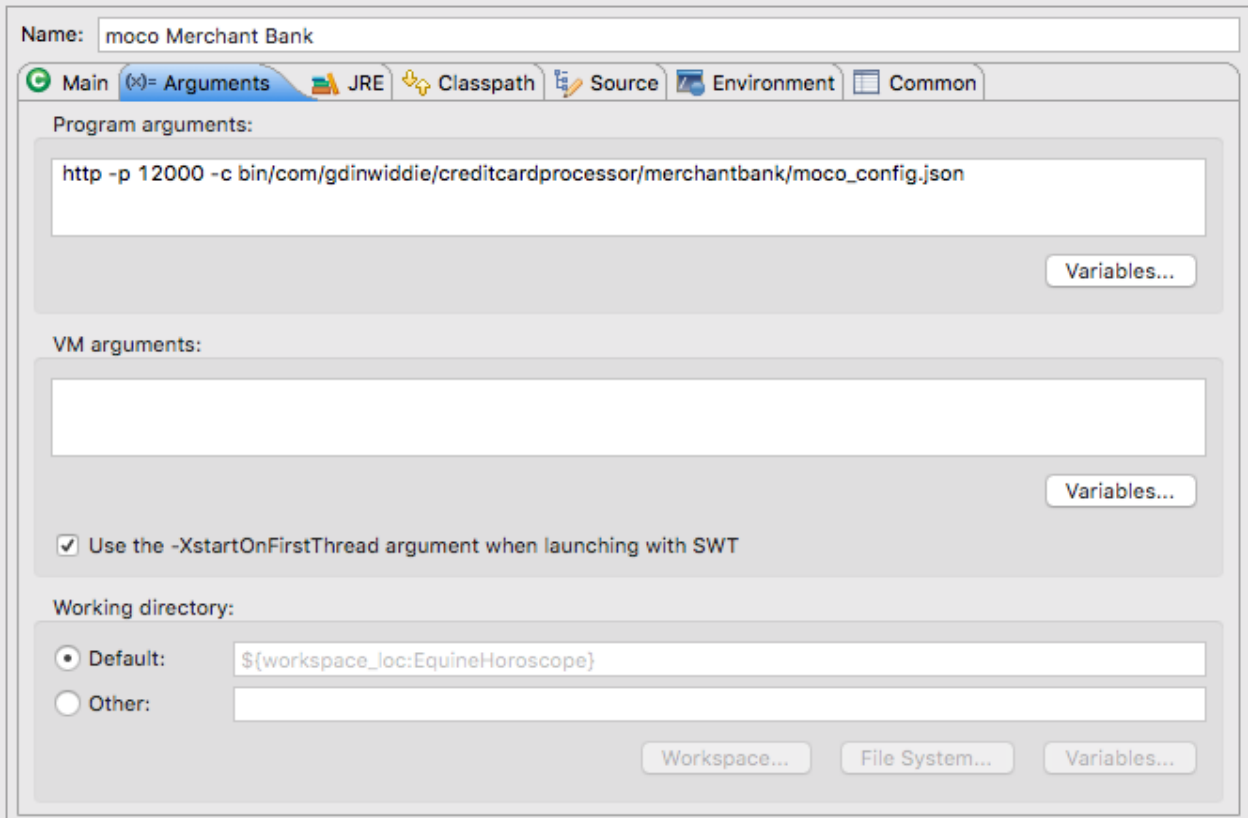
```
java -jar lib/moco-runner-0.11.0-SNAPSHOT-uber.jar http -p 12000 \
-c src/test/resources/com/gdinwiddie/creditcardprocessor/merchantbank/moco_config.json
```

We can setup to run this server from within Eclipse as a java application.

- "Run", "Run Configurations..."
- Select "Java Application" on the left-hand column and press the "New" button (the icon of an empty page with a + sign at the top of that column)
- Create a "Moco Merchant Bank" configuration like this:



Eclipse Run MocoMerchantBank Configuration part 1



Eclipse Run MocoMerchantBank Configuration part 2

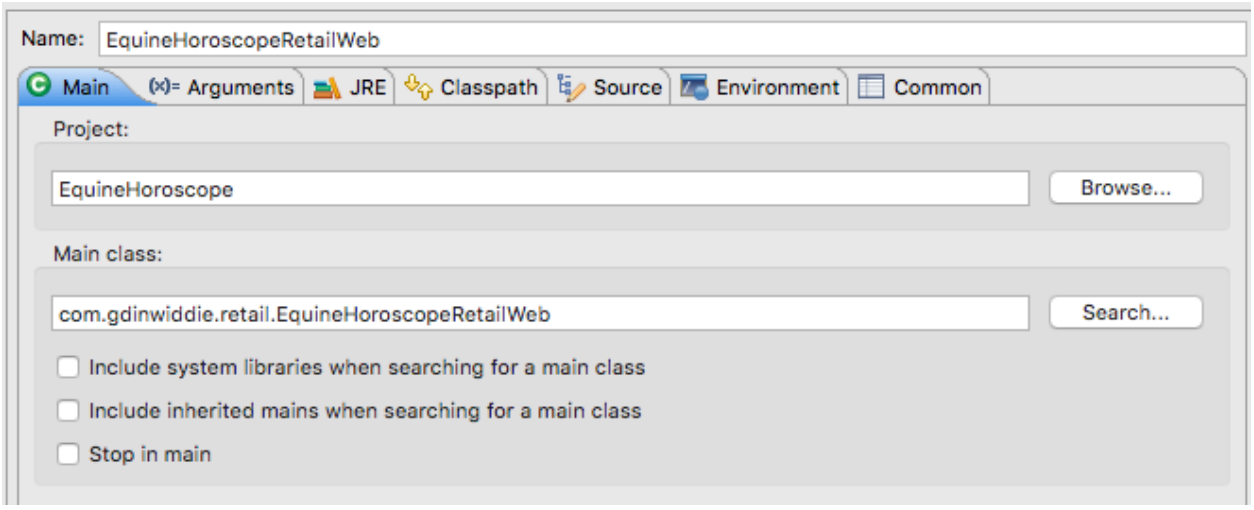
- Click "Apply" and "Run"

Check that it's running by visiting `http://localhost:12000/` You should get a response, "Unrecognized action requested" because this server is expecting JSON requests. This error message lets you know it's running, though.

Running the Equine Horoscope web application:

Running the Equine Horoscope web application is a bit more work, as it requires all the bits to be included on the classpath. Setting it as a java application within eclipse makes this easy for us.

- "Run", "Run Configurations..."
- Select "Java Application" on the left-hand column and press the "New" button (the icon of an empty page with a + sign at the top of that column)
- Create a "EquineHoroscopeRetailWeb" configuration like this:



Eclipse Run EquineHoroscopeApplication Configuration

- Click "Apply" and "Run"

We can visit `http://localhost:20002/buy` manually to see how things work. Use "4111111111111111" (That's a "4" and 15 "1"s) for the credit card number if you want the "purchase" to go through.

Running the Web GUI Tests

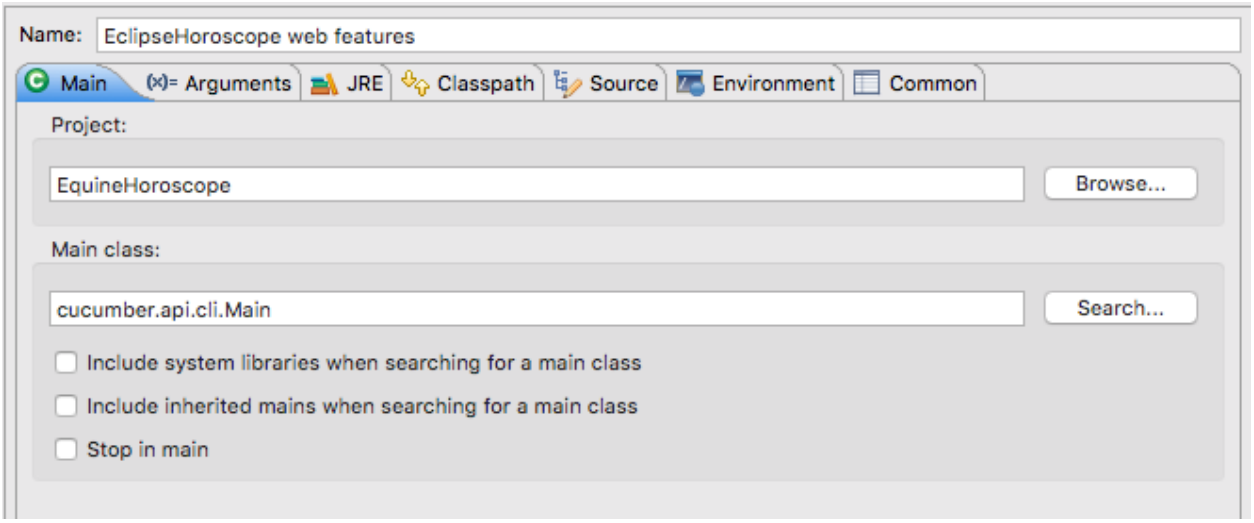
For these tests to succeed, we need both the Equine Horoscope web application and the Moco Merchant Bank server running.

Note that while the web application is running, the non-web unit tests will fail where they try to access the database.

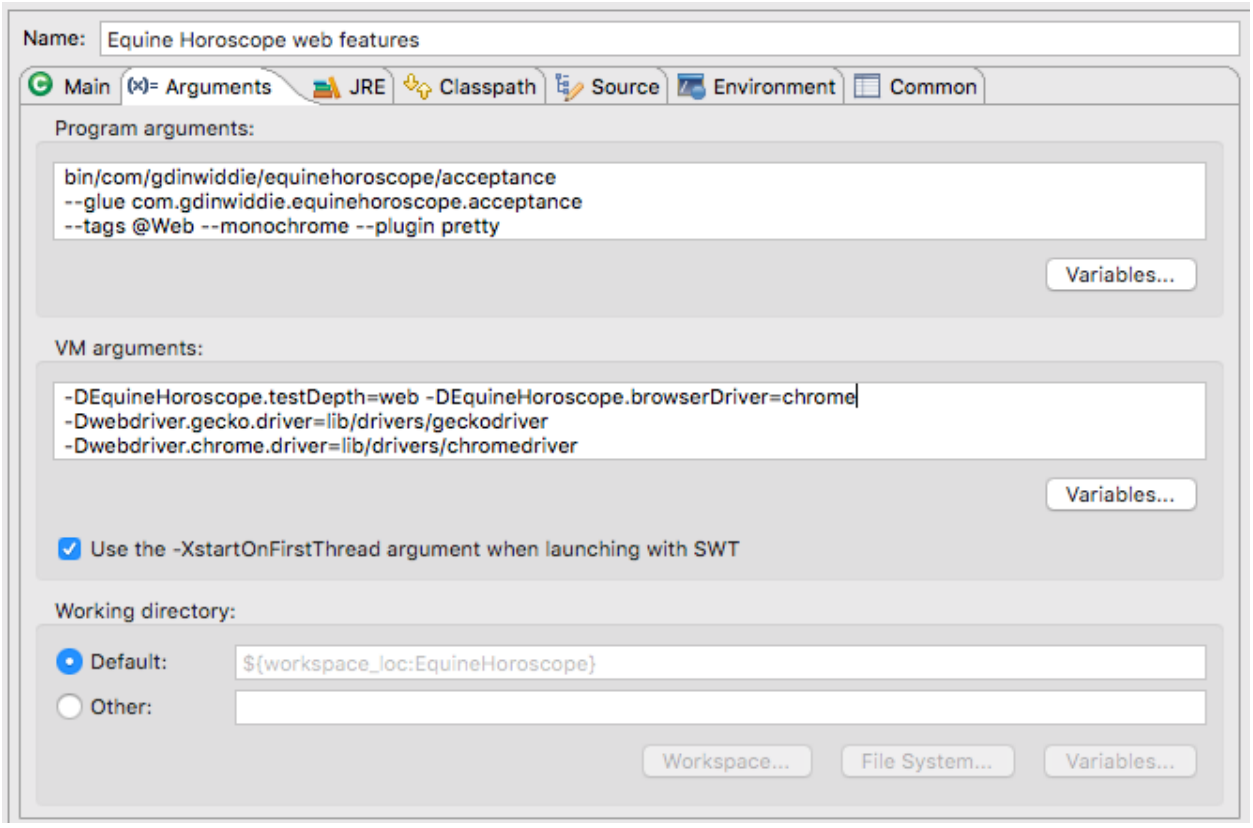
We want to specify the environment variable `EquineHoroscope.testDepth=web` to have the tests execute at the GUI level (running Chrome via Selenium WebDriver) rather than at the API level. We also want to run the appropriate tests which are tagged `@Web`.

We can set this up within Eclipse:

- "Run", "Run Configurations..."
- Select "Java Application" on the left-hand column and press the "New" button (the icon of an empty page with a + sign at the top of that column)
- Create a "Equine Horoscope web features" configuration like this:



Eclipse Run Configuration part 1



Eclipse Run Configuration part 2

- Click “Apply” and “Run”

You should see a Chrome browser window pop up and purchase a horoscope.

Note, you can run the same tests in a Firefox browser by adding

`-DEquineHoroscope.browserDriver=firefox`

to the "VM arguments"

Using Maven

Running Maven is easy, if not simple. I'm not a fan of Maven because of all the hidden magic it does, but I do understand its appeal. I've got Apache Maven 3.3.9 installed on Apple OSX 10.11.6 *El Capitan* by `Homebrew` (http://brew.sh/). Other setups should work the same.

In the `EquineHoroscope` directory, run

```
mvn clean test
```

A whole bunch of stuff should happen as Maven downloads dependencies, compiles the application and tests, and runs the tests. Finally you should see

```
Results :
Tests run: 38, Failures: 0, Errors: 0, Skipped: 1
[INFO] ————————————————————————————
```

This will let you know that things are running. If you don't see this, scroll backwards looking for problems to debug.

Deprecation warning

If you scan back far enough, you'll see

```
[WARNING]
[WARNING] Some problems were encountered while building the effective model for
com.gdinwiddie:equine_horoscope:jar:0.0.1
[WARNING] 'dependencies.dependency.systemPath' for main:moco-runner:jar should
not point at files within the project directory, ${project.basedir}/lib/moco-runner-0.11.0-
SNAPSHOT-uber.jar will be unresolvable by dependent projects @ line 83, column 16
[WARNING] 'dependencies.dependency.systemPath' for main:idiaJdbc:jar should not
point at files within the project directory, ${project.basedir}/lib/idiaJdbc.jar will be
unresolvable by dependent projects @ line 91, column 16
```

> [WARNING] 'dependencies.dependency.systemPath' for main:mumbler:jar should not point at files within the project directory, ${project.basedir}/lib/mumbler.jar will be unresolvable by dependent projects @ line 99, column 16
> [WARNING]
> [WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.
> [WARNING]
> [WARNING] For this reason, future Maven versions might no longer support building such malformed projects.
> [WARNING]

This warning appears because the Maven build depends on three jars that are within the project `lib` directory (where Eclipse expects them), rather than in the Maven repository.

- `moco-runner-0.11.0-SNAPSHOT-uber.jar` is my patched version (https://github.com/gdinwiddie/moco) of Zheng Ye's Moco "Easy Setup Stub Service" (https://github.com/dreamhead/moco). He has accepted a patch for the problem I found, but has not yet released a new version.
- `idiaJdbc:jar` is my own JDBC library, first written around 2002 and then re-written from scratch a couple times before I did it on my own time and open-sourced it. It now lives at https://github.com/gdinwiddie/JdbcLib
- `mumbler:jar` is a grammar-driven language generator. It happens to be written by Shannon Null Code when we were going to work together on Equine Horoscope years ago. That collaboration withered due to both of us having too many other things to do. I use this library to represent the typical situation of a third-party library that we can't change, but depend upon.

Someday I'll figure a better place to put these to make Maven happy.

Thanks to Wouter Lagerweij, you can run the moco simulated Merchant Bank Server with

```
mvn exec:java -Pmoco
```

and the application web server with

```
mvn exec:java -Pweb
```

Using Ant

I prefer Ant to Maven because the workings are more visible. In this case, though, I didn't keep it working. Wouter Lagerweij did some work on this (https://github.com/wouterla/EquineHoroscope) which I incorporated. I don't see the unit tests running successfully on my machine, and haven't investigated.

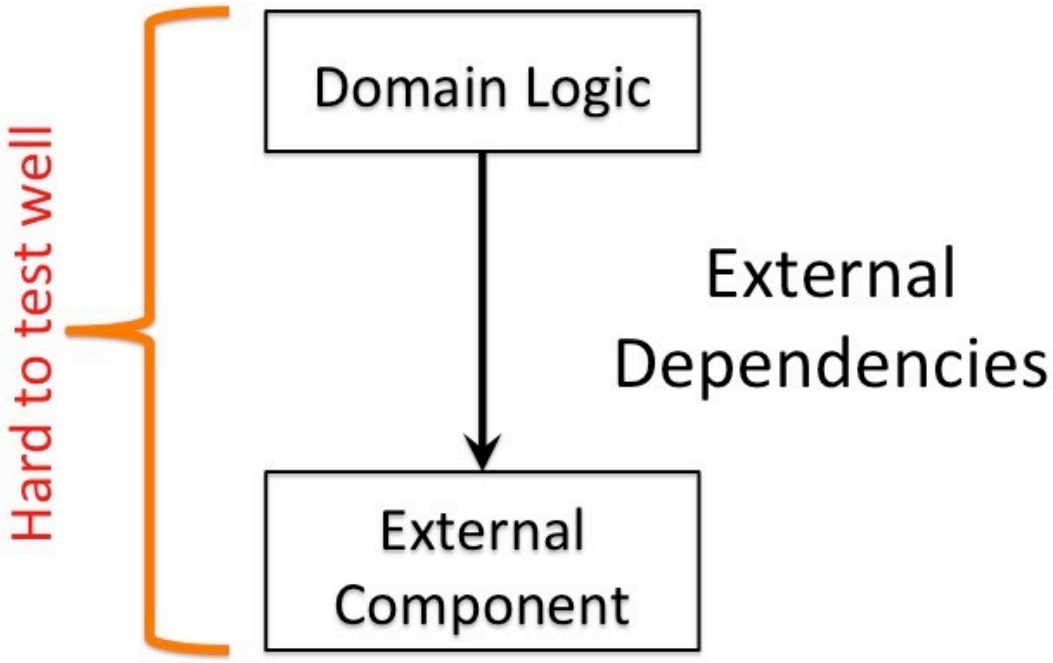
At the moment, you're on your own with Ant. (:-/)

External Dependencies

The Problem with External Dependencies

I once worked with some software developers that owned a bunch of Internet domain names. Why? Because to test the software used to sell those domains, they sometimes had to use real credit cards and make real purchases. The system they had developed was too tightly coupled with the credit card processor.

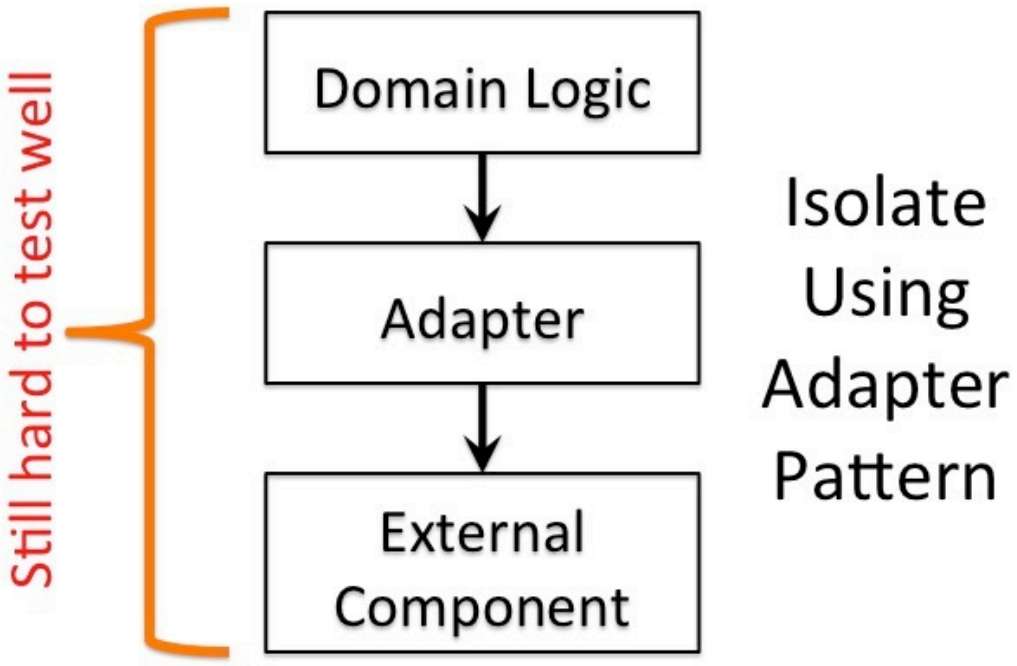
Such coupling makes it hard to know if our code is working correctly, since we need to connect to the dependency in order to run it. And when things aren't working as we expect, we're not sure if the problem is in our code or in the dependency. We can waste a lot of time figuring that out.



External Dependencies

Isolating our Code

The worse case is when dependency on the external system is sprinkled all through our code. The first step to solving the problem is to gather all of that dependency into one place. Usually this one place follows the Gang of Four[1] *Adapter Pattern*. If the data may be driven from either direction, than the *Mediator Pattern* may be more suitable. Either way, we isolate our domain logic code from the external dependency.



Inserting an Adapter

On the inside surface of the adapter, we create an API that is convenient for our domain logic to use. It uses the natural concepts of our domain for both input and output. I generally discover that API by the use by my domain logic as I write the code. This helps me end up with an API that's comfortable to use.

The exterior surface of the adapter does whatever the external dependency requires. It might be

[1]The book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison-Wesley Professional, 1994. http://amzn.to/1RRb9dQ

calling a third-party packaged library, reading and writing data in the database, or communicating with another system over the network. The format of the input and output messages are fixed by the external dependency, and might not be convenient or easy to understand.

In between, we put code to translate from one API to the other, and back again. This code encapsulates any messy stuff required by the external dependency, hiding it from our domain logic. It also prevents any dependency-specific data structures from polluting our domain code. In the event we want to eliminate our dependency and perform its function in another way, we only need switch to a different adapter to make it work.

This approach to domain logic isolation is often called the *Ports and Adapters* pattern, or *Hexagonal Architecture.* [2]

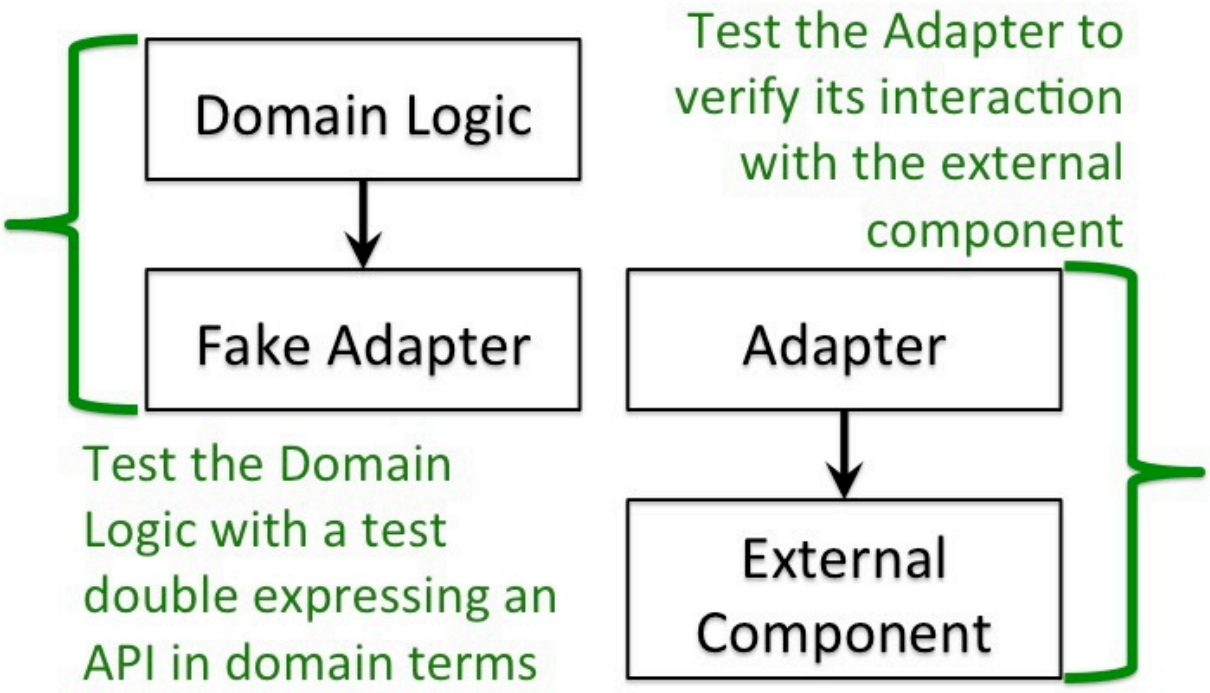
> **Pattern**: Isolate your domain model from dependencies outside of your control using the Adapter Pattern or Mediator Pattern

Breaking the Problem in Half

The adapter is easily exploited as a testing seam. Michael Feathers defines a seam as "a place where you can alter behavior in your program without editing in that place."[3] We can substitute a different adapter using an *Abstract Factory* or *Dependency Injection* and our domain logic will be none the wiser. If the substituted adapter is a fake adapter under control of our test code, we can have it provide any data we'd like and check the details of the domain logic's interactions with it. This is the fundamental purpose of *Test Doubles*.

[2] Alistair Cockburn identified the Ports and Adapters pattern (nee Hexagonal Architecture). http://alistair.cockburn.us/Hexagonal+architecture He describes the discovery on Ward's Wiki. http://c2.com/cgi/wiki?PortsAndAdaptersArchitecture

[3] Working Effectively with Legacy Code by Michael Feathers, Prentice Hall, 2004. http://amzn.to/1Uyo8YN



Exploiting a Testing Seam

Our first step, in `EquineHoroscope`, is for the `CrystalBall` to delegate to a `HoroscopeProvider` rather than provide its own dummy horoscope. We can write a unit test that injects a `HoroscopeProvider` that returns a known horoscope, and then expects `fetchHoroscope()` to return that known horoscope.

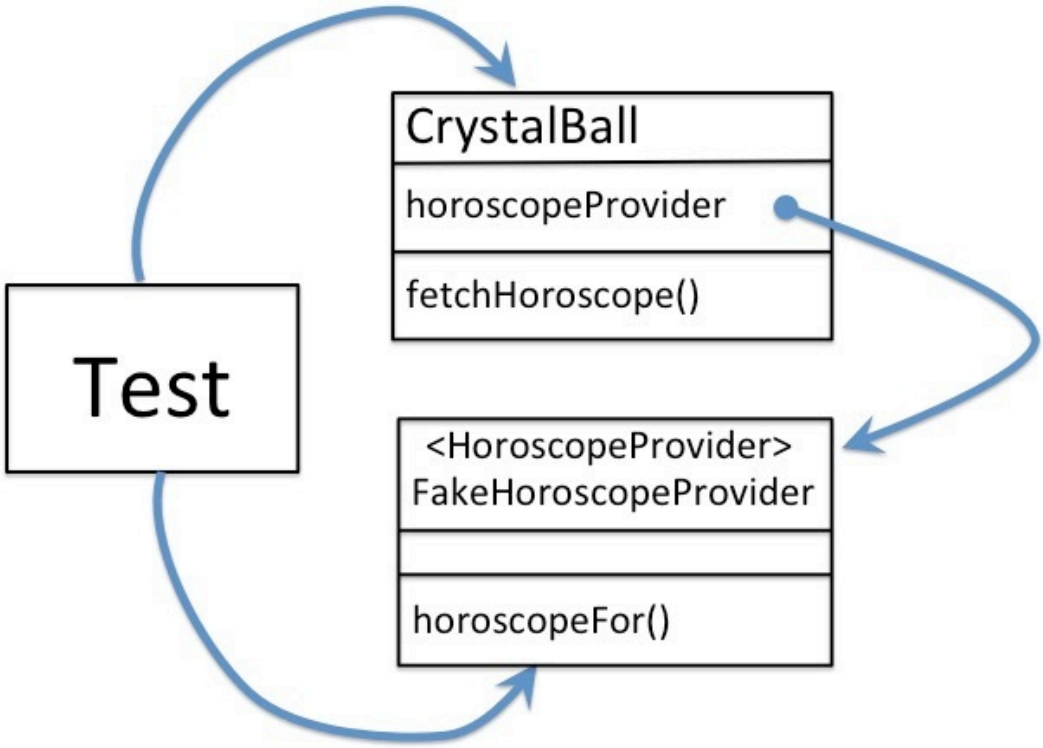
```
1  @Test
2  public void assureWeGetHoroscopeFromProvider() {
3     CrystalBall uut = new CrystalBall(new FakeHoroscopeProvider());
4     String horoscope = uut.fetchHoroscope("Stewball", "today");
5     assertThat(horoscope, equalTo("This is a fake horoscope."));
6  }
7
8  static class FakeHoroscopeProvider implements HoroscopeProvider {
9     @Override
10    public String horoscopeFor(String horse, String date) {
11       return "This is a fake horoscope.";
```

```
12      }
13  }
```

To make this test work, we create the new constructor to allow dependency injections. Then we extract the line that provides "Situation cloudy, try again later" into a method. Our scenario still works. Finally, we embed in an anonymous `HoroscopeProvider` and delegate to it. This code satisfies both our earlier scenario and this unit test.

```
1   private HoroscopeProvider horoscopeProvider;
2
3   CrystalBall(HoroscopeProvider horoscopeProvider) {
4      this.horoscopeProvider = horoscopeProvider;
5   }
6
7   CrystalBall() {
8      this(new HoroscopeProvider() {
9         @Override
10        public String horoscopeFor(String horse, String date) {
11           return "Situation cloudy, try again later";
12        }
13     });
14  }
15
16  public String fetchHoroscope(String horse, String effectiveDate) {
17     return horoscopeProvider.horoscopeFor(horse, effectiveDate);
18  }
```

Our test now controls both the request to the `CrystalBall` and the `HoroscopeProvider` to which it delegates the responsibility for creating a horoscope. This gives us a deterministic result.



Testing with a test-supplied Test Double

> **Pattern**: Test your domain model using *Test Double* adapters or mediators

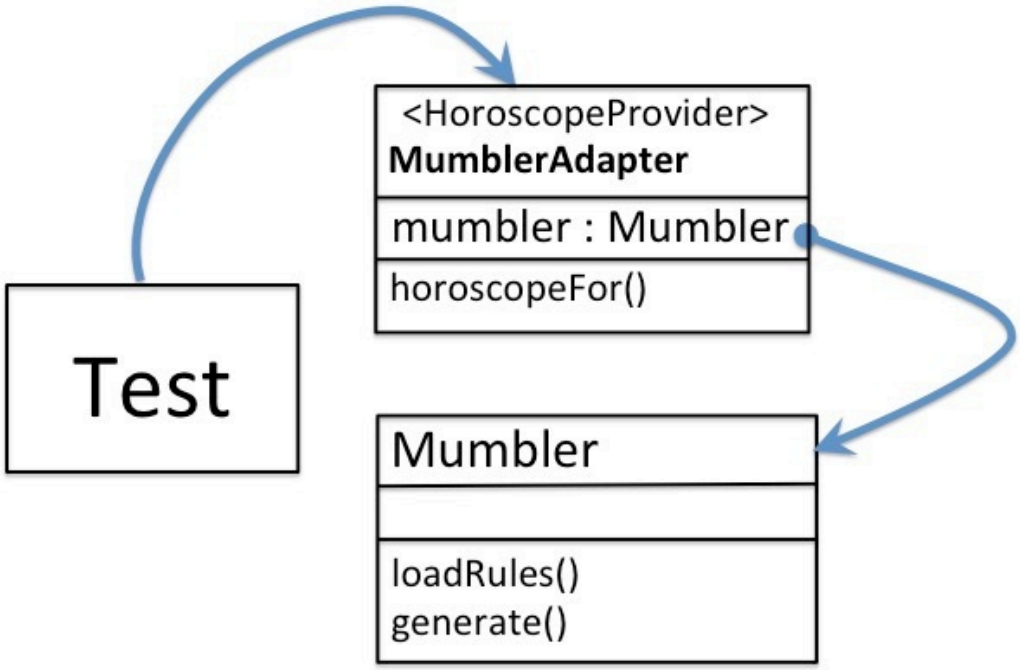
At the same time, we can notice the way our domain logic is using the fake adapter and write tests exercising the real adapter in the same manner. Since the external dependency is outside our control, these tests may not be able to check values received with precision, but it can validate that the adapter is communicating properly.

```
public class MumblerAdapterTest {

   @Test
   public void assureDefaultGeneratesStrings() {
      MumblerAdapter uut = MumblerAdapter.instance();
      assertThat(uut.horoscopeFor("any horse", "any date"),
            not(isEmptyOrNullString()));
   }

}
```

Now we create an implementation that passes:

```
public class MumblerAdapter implements HoroscopeProvider {

   private Mumbler mumbler;
   private static final String DEFAULT_RULES =
         "{<start>Outlook cloudy, try again later.}";

   MumblerAdapter(String grammarRules) {
      this.mumbler = new Mumbler();
      mumbler.loadRules(grammarRules);
   }

   @Override
   public String horoscopeFor(String horse, String date) {
      return mumbler.generate();
   }

   public static MumblerAdapter instance() {
      return new MumblerAdapter(DEFAULT_RULES);
   }
}
```

This implementation doesn't contain a very interesting grammar, but it does check that we're supplying the grammar and retrieving the generated results propery.



Testing the real adapter

> **Pattern**: Test the integration of your adapters with the external dependency

We can write a test to verify that our generator doesn't generate the same result for each call, and even display what it returns.

```
1  @Test
2  public void the_horoscopes_should_be_the_same() throws Throwable {
3     Set<String> horoscopeSet = new HashSet<String>();
4     horoscopeSet.addAll(horoscopes);
5     horoscopeSet.forEach(new Consumer<String>() {
6
7        @Override
8        public void accept(String thisHoroscope) {
9           System.out.println(thisHoroscope);
```

```
10          }});
11      assertThat(horoscopeSet.size(), equalTo(1));
12  }
```

> **Pattern:** It's OK to use test automation for things you want to examine.

Using the test framework to visualize results is often the easiest way to look at values. When you're running tests frequently, anyway, this is much less work than dropping into the debugger, and it keeps you in the test-driven state of mind.

This is a weak unit test, but it's sufficient for us to verify we're feeding a grammar file correctly to our generator.

```
1   public class MumblerAdapter implements HoroscopeProvider {
2
3       private Mumbler mumbler;
4       private static final String DEFAULT_RULES =
5             "{<start>Outlook cloudy, try again later.}";
6
7       MumblerAdapter(String grammarRules) {
8           this.mumbler = new Mumbler();
9           mumbler.loadRules(grammarRules);
10      }
11
12      MumblerAdapter(InputStream stream) {
13          this.mumbler = new Mumbler();
14            try {
15                mumbler.loadStream(stream);
16            } catch (IOException e) {
17                mumbler.loadRules(DEFAULT_RULES);
18            }
19      }
20
21      @Override
22      public String horoscopeFor(String horse, String date) {
23          return mumbler.generate();
24      }
25
26      public static MumblerAdapter instance() {
27          String resourceName =
```

```
28                  "/com/gdinwiddie/equinehoroscope/mumbler/MumblerHoroscopeRules.g";
29        ResourceLoader loader = new ResourceLoader();
30        InputStream stream = loader.loadResourceStream(resourceName);
31        return new MumblerAdapter(stream);
32     }
33
34     static class ResourceLoader {
35        InputStream loadResourceStream(String name) {
36           return getClass().getResourceAsStream(name);
37        }
38     }
39  }
```

Once I'm satisfied, however, then I remove the `println()` call and also `@Ignore` the test.

```
1  @Ignore("Could have spurious failures")
2  @Test
3  public void assureGeneratorHasVariety() {
4     Set<String> horoscopeset = new HashSet<String>();
5     for (int count=0; count<5; count++) {
6        horoscopeset.add(generateHoroscope());
7     }
8     assertThat(horoscopeset.size(), greaterThan(1));
9  }
```

This conforms to two rules I set for myself:

> **Pattern**: Do not leave PRINT statements in your tests.
>
> **Pattern**: Tests that aren't guaranteed to work, but might be worth having around

Just don't include such tests in your acceptance test suite. Flaky tests will waste a lot of energy, and cause people to lose confidence in the test suite.

I find writing these tests contemporaneously with the domain logic tests helps me cover the expected situations, even when I can't run them yet because the 3rd party system isn't available. (That, by the way, is another benefit of this technique. It allows you to proceed when the dependency isn't available.)

Remaining Risks

What risks remain? How can we address those risks?

I can think of a few.

- We could break the loading of the grammar file and not notice.
- The default production version of `CrystalBall` could be wired to a wrong `HoroscopeProvider`.

These are things that are relatively unlikely to change once verified, and can be verified manually. The first one we already verified with a semi-manual junit test. The second will certainly be checked if we look at the application output at all.

Both of these risks have pretty noticeable results. They're not subtle errors that are likely to go unnoticed. This makes me feel more confident.

Can you think of other risks?

Review

For the most part, we've covered testing our dependencies in two parts.

> **Pattern**: Isolate your domain model from dependencies outside of your control using the Adapter Pattern or Mediator Pattern
>
> **Pattern**: Test your domain model using *Test Double* adapters or mediators
>
> **Pattern**: Test the integration of your adapters with the external dependency

We've also discussed a few heuristics about test automation code.

> **Pattern**: It's OK to use test automation for things you want to examine.
>
> **Pattern**: Do not leave PRINT statements in your tests.
>
> **Pattern**: Tests that aren't guaranteed to work, but might be worth having around