Extracted from:

# Modern C++ Programming with Test-Driven Development

## Code Better, Sleep Better



The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

# Modern C++ Programming with Test-Driven Development

## Code Better, Sleep Better

Jeff Langr

*Edited by Michael Swaine*

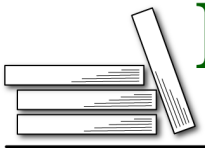
# Modern C++ Programming with Test-Driven Development

## Code Better, Sleep Better

Jeff Langr

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)



Printed in the United States of America.
ISBN-13: 978-1-937785-48-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2013

## 5.1 Setup

In the prior three chapters, you test-drove a stand-alone class and learned all about the fundamentals of TDD. If only life were so simple! The reality is that objects must work with other objects (collaborators) in a production OO system. Sometimes the dependencies on collaborators can cause pesky challenges for test-driving—they can be slow, unstable, or not even around to help you yet.

In this chapter, you'll learn how to brush away those challenges using test doubles. You'll first learn how to break dependencies using handcrafted test doubles. You'll then see how you might simplify the creation of test doubles by using a tool. You'll learn about different ways of setting up your code so that it can use test doubles (also known as *injection* techniques). Finally, you'll read about the design impacts of using test doubles, as well as strategies for their best use.

## 5.2 Dependency Challenges

Objects often must collaborate with other objects in order to get their work done. An object tells another to do something or asks it for information. If an object *A* depends upon a collaborator object *B* in order to accomplish its work, *A* is *dependent upon B*.

> **Story: Place Description Service**
> As a programmer on map-based applications, I want a service that returns a one-line description of the named place nearest a given location (latitude and longitude).

An important part of building the Place Description Service involves calling an external API that takes a location and returns place data. I found an open, free Representational State Transfer (REST) service that returns the place data in JSON format, given a GET URL. This Nominatim Search Service is part of the Open MapQuest API.[1,2]

Test-driving the Place Description Service presents a challenge—the dependency on the REST call is problematic for at least a few reasons.

- Making an actual HTTP call to invoke a REST service is very slow and will bog down your test run. (See Section 4.3, *Fast Tests, Slow Tests, Filters, and Suites*, on page ?.)
- The service might not always be available.

1. You can find details on the API at http://open.mapquestapi.com/nominatim.
2. Wikipedia provides an overview of REST at https://en.wikipedia.org/wiki/Representational_state_transfer.

- You can't guarantee what results the call will return.

Why do these dependency concerns create testing challenges? First, a dependency on slow collaborators results in undesirably slow tests. Second, a dependency on a volatile service (either unavailable or returning different answers each time) results in intermittently failing tests.

The dependency concern of sheer *existence* can exist. What if you have no utility code that supports making an HTTP call? In a team environment, the job of designing and implementing an appropriate HTTP utility class might be on someone else's plate. You don't have the time to sit and wait for someone else to complete their work, and you don't have the time to create an HTTP class yourself.

What if *you* are the person on the hook for building HTTP support? Maybe you'd like to first explore the design of the Place Description Service implementation overall and worry about the implementation details of an HTTP utility class later.

## 5.3 Test Doubles

You can avoid being blocked, in any of these cases, by employing a *test double*. A test double is a stand-in—a doppelgänger (literally: "double walker")—for a production class. HTTP giving you trouble? Create a test double HTTP implementation! The job of the test double will be to support the needs of the test. When a client sends a GET request to the HTTP object, the test double can return a canned response. The test itself determines the response that the test double should return.

Imagine you are on the hook to build the service but you aren't concerned with unit testing it (perhaps you plan on writing an integration test). You have access to some classes that you can readily reuse.

- CurlHttp, which uses cURL[3] to make HTTP requests. It derives from the pure virtual base class Http, which defines two functions, `get()` and `initialize()`. Clients must call `initialize()` before they can make calls to `get()`.
- Address, a struct containing a few fields.
- AddressExtractor, which populates an Address struct from a JSON[4] string using JsonCpp.[5]

You might code the following:

3. http://curl.haxx.se/libcurl/cplusplus/
4. http://www.json.org
5. http://jsoncpp.sourceforge.net

```
CurlHttp http;
http.initialize();
auto jsonResponse = http.get(createGetRequestUrl(latitude, longitude));

AddressExtractor extractor;
auto address = extractor.addressFrom(jsonResponse);

return summaryDescription(address);
```

Now imagine you want to add tests for that small bit of code. It won't be so easy, because the CurlHttp class contains unfortunate dependencies you don't want. Faced with this challenge, many developers would choose to run a few manual tests and move on.

You're better than that. You're test-driving! That means you want to add code to your system *only* in order to make a failing test pass. But how will you write a test that sidesteps the dependency challenges of the CurlHttp class? In the next section, we'll work through a solution.

## 5.4 A Hand-Crafted Test Double

To use a test double, you must be able to supplant the behavior of the CurlHttp class. C++ provides many different ways, but the predominant manner is to take advantage of polymorphism. Let's take a look at the Http interface that the CurlHttp class implements (realizes):

c5/1/Http.h

```
virtual ~Http() {}
virtual void initialize() = 0;
virtual std::string get(const std::string& url) const = 0;
```

Your solution is to override the virtual methods on a derived class, provide special behavior to support testing in the override, and pass the Place Description Service code a base class pointer.

Let's see some code.

c5/1/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService, ReturnsDescriptionForValidLocation) {
   HttpStub httpStub;
   PlaceDescriptionService service{&httpStub};

   auto description = service.summaryDescription(ValidLatitude, ValidLongitude);

   ASSERT_THAT(description, Eq("Drury Ln, Fountain, CO, US"));
}
```

We create an instance of HttpStub in the test. HttpStub is the type of our test double, a class that derives from Http. We define HttpStub directly in the test

file so that we can readily see the test double's behavior along with the tests that use it.

c5/1/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
   void initialize() override {}
   std::string get(const std::string& url) const override {
      return "???";
   }
};
```

Returning a string with question marks is of little use. What do we need to return from get()? Since the external Nominatim Search Service returns a JSON response, we should return an appropriate JSON response that will generate the description expected in our test's assertion.

c5/2/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
   void initialize() override {}
   std::string get(const std::string& url) const override {
      return R"({ "address": {
         "road":"Drury Ln",
         "city":"Fountain",
         "state":"CO",
         "country":"US" }})";
   }
};
```

How did I come up with that JSON? I ran a live GET request using my browser (the Nominatim Search Service API page shows you how) and captured the resulting output.

From the test, we *inject* our HttpStub instance into a PlaceDescriptionService object via its constructor. We're changing our design from what we speculated. Instead of the service constructing its own Http instance, the client of the service will now need to construct the instance and inject it into (pass it to) the service. The service constructor holds on to the instance via a base class pointer.

c5/2/PlaceDescriptionService.cpp

```
PlaceDescriptionService::PlaceDescriptionService(Http* http) : http_(http) {}
```

Simple polymorphism gives us the test double magic we need. A PlaceDescriptionService object knows not whether it holds a production Http instance or an instance designed solely for testing.

Once we get our test to compile and fail, we code summaryDescription().

c5/2/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
   auto getRequestUrl = "";
   auto jsonResponse = http_->get(getRequestUrl);

   AddressExtractor extractor;
   auto address = extractor.addressFrom(jsonResponse);
   return address.road + ", " + address.city + ", " +
          address.state + ", " + address.country;
}
```

(We're fortunate: someone else has already built AddressExtractor for us. It parses a JSON response and populates an Address struct.)

When the test invokes summaryDescription(), the call to the Http method get() is received by the HttpStub instance. The result is that get() returns our hard-coded JSON string. A test double that returns a hard-coded value is a *stub*. You can similarly refer to the get() method as a *stub method*.

We test-drove the relevant code into summaryDescription(). But what about the request URL? When the code you're testing interacts with a collaborator, you want to make sure that you pass the correct elements to it. How do we know that we pass a legitimate URL to the Http instance?

In fact, we passed an empty string to the get() function in order to make incremental progress. We need to drive in the code necessary to populate getRequestUrl correctly. We could triangulate and assert against a second location (see *Triangulation*, on page ?).

Better, we can add an assertion to the get() stub method we defined on HttpStub.

c5/3/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
   void initialize() override {}
   std::string get(const std::string& url) const override {
➤     verify(url);
      return R"({ "address": {
         "road":"Drury Ln",
         "city":"Fountain",
         "state":"CO",
         "country":"US" }})";
   }
   void verify(const string& url) const {
      auto expectedArgs(
         "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
         "lon=" + APlaceDescriptionService::ValidLongitude);
         ASSERT_THAT(url, EndsWith(expectedArgs));
```

```
    }
};
```

(Why did we create a separate method, verify(), for our assertion logic? It's because of a Google Mock limitation: you can use assertions that cause fatal failures only in functions with void return.[6])

Now, when get() gets called, the stub implementation ensures the parameters are as expected. The stub's assertion tests the most important aspect of the URL: does it contain correct latitude/longitude arguments? Currently it fails, since we pass get() an empty string. Let's make it pass.

c5/3/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
   auto getRequestUrl = "lat=" + latitude + "&lon=" + longitude;
   auto jsonResponse = http_->get(getRequestUrl);
   // ...
}
```

Our URL won't quite work, since it specifies no server or document. We bolster our verify() function to supply the full URL before passing it to get().

c5/4/PlaceDescriptionServiceTest.cpp

```
void verify(const string& url) const {
   string urlStart(
      "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&");
   string expected(urlStart +
      "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
      "lon=" + APlaceDescriptionService::ValidLongitude);
   ASSERT_THAT(url, Eq(expected));
}
```

Once we get the test to pass, we undertake a bit of refactoring. Our summaryDescription() method violates cohesion, and the way we construct key-value pairs in both the test and production code exhibits duplication.

c5/4/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
   auto request = createGetRequestUrl(latitude, longitude);
   auto response = get(request);
   return summaryDescription(response);
}
string PlaceDescriptionService::summaryDescription(
      const string& response) const {
   AddressExtractor extractor;
   auto address = extractor.addressFrom(response);
```

6. http://code.google.com/p/googletest/wiki/AdvancedGuide#Assertion_Placement

```
   return address.summaryDescription();
}

string PlaceDescriptionService::get(const string& requestUrl) const {
   return http_->get(requestUrl);
}

string PlaceDescriptionService::createGetRequestUrl(
      const string& latitude, const string& longitude) const {
   string server{"http://open.mapquestapi.com/"};
   string document{"nominatim/v1/reverse"};
   return server + document + "?" +
      keyValue("format", "json") + "&" +
      keyValue("lat", latitude) + "&" +
      keyValue("lon", longitude);
}
string PlaceDescriptionService::keyValue(
      const string& key, const string& value) const {
   return key + "=" + value;
}
```

What about all that other duplication? ("*What* duplication?" you ask.) The text expressed in the test matches the text expressed in the production code. Should we strive to eliminate this duplication? There are several approaches we might take; for further discussion, refer to *Implicit Meaning*, on page ?.

Otherwise, our production code design appears sufficient for the time being. Functions are composed and expressive. As a side effect, we're poised for change. The function keyValue() appears ripe for reuse. We can also sense that generalizing our design to support a second service would be a quick increment, since we'd be able to reuse some of the structure in PlaceDescriptionService.

Our test's design is insufficient, however. For programmers not involved in its creation, it is too difficult to follow. Read on.

## 5.5 Improving Test Abstraction When Using Test Doubles

It's easy to craft tests that are difficult for others to read. When using test doubles, it's even easier to craft tests that obscure information critical to their understanding.

ReturnsDescriptionForValidLocation is difficult to understand because it hides relevant information, violating the concept of test abstraction (see Section 7.4, *Test Abstraction*, on page ?).

c5/4/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService, ReturnsDescriptionForValidLocation) {
   HttpStub httpStub;
   PlaceDescriptionService service{&httpStub};
```

```
   auto description = service.summaryDescription(ValidLatitude, ValidLongitude);

   ASSERT_THAT(description, Eq("Drury Ln, Fountain, CO, US"));
}
```

Why do we expect the description to be an address in Fountain, Colorado? Readers must poke around to discover that the expected address *correlates* to the JSON address in the HttpStub implementation.

We must refactor the test so that stands on its own. We can change the implementation of HttpStub so that the test is responsible for setting up the return value of its get() method.

c5/5/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
public:
   string returnResponse;
   void initialize() override {}
   std::string get(const std::string& url) const override {
      verify(url);
      return returnResponse;
   }

   void verify(const string& url) const {
      // ...
   }
};

TEST_F(APlaceDescriptionService, ReturnsDescriptionForValidLocation) {
   HttpStub httpStub;
   httpStub.returnResponse = R"({"address": {
                                    "road":"Drury Ln",
                                    "city":"Fountain",
                                    "state":"CO",
                                    "country":"US" }})";
   PlaceDescriptionService service{&httpStub};
   auto description = service.summaryDescription(ValidLatitude, ValidLongitude);
   ASSERT_THAT(description, Eq("Drury Ln, Fountain, CO, US"));
}
```

Now the test reader can correlate the summary description to the JSON object returned by HttpStub.

We can similarly move the URL verification to the test.

c5/6/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
public:
   string returnResponse;
   string expectedURL;
```

```
    void initialize() override {}
    std::string get(const std::string& url) const override {
       verify(url);
       return returnResponse;
    }
    void verify(const string& url) const {
       ASSERT_THAT(url, Eq(expectedURL));
    }
};

TEST_F(APlaceDescriptionService, ReturnsDescriptionForValidLocation) {
    HttpStub httpStub;
    httpStub.returnResponse = // ...
    string urlStart{
       "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&"};
    httpStub.expectedURL = urlStart +
       "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
       "lon=" + APlaceDescriptionService::ValidLongitude;
    PlaceDescriptionService service{&httpStub};

    auto description = service.summaryDescription(ValidLatitude, ValidLongitude);

    ASSERT_THAT(description, Eq("Drury Ln, Fountain, CO, US"));
}
```

Our test is now a little longer but expresses its intent clearly. In contrast, we pared down HttpStub to a simple little class that captures expectations and values to return. Since it also verifies those expectations, however, HttpStub has evolved from being a stub to becoming a *mock*. A mock is a test double that captures expectations and self-verifies that those expectations were met.[7] In our example, an HttpStub object verifies that it will be passed an expected URL.

To test-drive a system with dependencies on things such as databases and external service calls, you'll need several mocks. If they're only "simple little classes that manage expectations and values to return," they'll all start looking the same. Mock tools can reduce some of the duplicate effort required to define test doubles.

7. *xUnit Test Patterns [Mes07]*