448 lines (328 sloc) 29.4 KB

(**Note:** If you get compiler errors that you don't understand, be sure to consult Google Mock Doctor.)

What Is Google C++ Mocking Framework?

When you write a prototype or test, often it's not feasible or wise to rely on real objects entirely. A **mock object** implements the same interface as a real object (so it can be used as one), but lets you specify at run time how it will be used and what it should do (which methods will be called? in which order? how many times? with what arguments? what will they return? etc).

Note: It is easy to confuse the term *fake objects* with mock objects. Fakes and mocks actually mean very different things in the Test-Driven Development (TDD) community:

- **Fake** objects have working implementations, but usually take some shortcut (perhaps to make the operations less expensive), which makes them not suitable for production. An in-memory file system would be an example of a fake.
- **Mocks** are objects pre-programmed with *expectations*, which form a specification of the calls they are expected to receive.

If all this seems too abstract for you, don't worry - the most important thing to remember is that a mock allows you to check the *interaction* between itself and code that uses it. The difference between fakes and mocks will become much clearer once you start to use mocks.

Google C++ Mocking Framework (or **Google Mock** for short) is a library (sometimes we also call it a "framework" to make it sound cool) for creating mock classes and using them. It does to C++ what jMock and EasyMock do to Java.

Using Google Mock involves three basic steps:

1. Use some simple macros to describe the interface you want to mock, and they will expand to the implementation of your mock class;
2. Create some mock objects and specify its expectations and behavior using an intuitive syntax;
3. Exercise code that uses the mock objects. Google Mock will catch any violation of the expectations as soon as it arises.

Why Google Mock?

While mock objects help you remove unnecessary dependencies in tests and make them fast and reliable, using mocks manually in C++ is *hard*:

- Someone has to implement the mocks. The job is usually tedious and error-prone. No wonder people go great distances to avoid it.
- The quality of those manually written mocks is a bit, uh, unpredictable. You may see some really polished ones, but you may also see some that were hacked up in a hurry and have all sorts of ad-hoc restrictions.
- The knowledge you gained from using one mock doesn't transfer to the next.

In contrast, Java and Python programmers have some fine mock frameworks, which automate the creation of mocks. As a result, mocking is a proven effective technique and widely adopted practice in those communities. Having the right tool absolutely makes the difference.

Google Mock was built to help C++ programmers. It was inspired by jMock and EasyMock, but designed with C++'s specifics in mind. It is your friend if any of the following problems is bothering you:

- You are stuck with a sub-optimal design and wish you had done more prototyping before it was too late, but prototyping in C++ is by no means "rapid".
- Your tests are slow as they depend on too many libraries or use expensive resources (e.g. a database).
- Your tests are brittle as some resources they use are unreliable (e.g. the network).
- You want to test how your code handles a failure (e.g. a file checksum error), but it's not easy to cause one.
- You need to make sure that your module interacts with other modules in the right way, but it's hard to observe the interaction; therefore you resort to observing the side effects at the end of the action, which is awkward at best.
- You want to "mock out" your dependencies, except that they don't have mock implementations yet; and, frankly, you aren't thrilled by some of those hand-written mocks.

We encourage you to use Google Mock as:

- a *design* tool, for it lets you experiment with your interface design early and often. More iterations lead to better designs!
- a *testing* tool to cut your tests' outbound dependencies and probe the interaction between your module and its collaborators.

Getting Started

Using Google Mock is easy! Inside your C++ source file, just `#include "gtest/gtest.h"` and `"gmock/gmock.h"` , and you are ready to go.

A Case for Mock Turtles

Let's look at an example. Suppose you are developing a graphics program that relies on a LOGO-like API for drawing. How would you test that it does the right thing? Well, you can run it and compare the screen with a golden screen snapshot, but let's admit it: tests like this are expensive to run and fragile (What if you just upgraded to a shiny new graphics card that has better anti-aliasing? Suddenly you have to update all your golden images.). It would be too painful if all your tests are like this. Fortunately, you learned about Dependency Injection and know the right thing to do: instead of having your application talk to the drawing API directly, wrap the API in an interface (say, `Turtle`) and code to that interface:

```
class Turtle {
  ...
  virtual ~Turtle() {}
  virtual void PenUp() = 0;
  virtual void PenDown() = 0;
  virtual void Forward(int distance) = 0;
  virtual void Turn(int degrees) = 0;
  virtual void GoTo(int x, int y) = 0;
  virtual int GetX() const = 0;
  virtual int GetY() const = 0;
};
```

(Note that the destructor of `Turtle` **must** be virtual, as is the case for **all** classes you intend to inherit from - otherwise the destructor of the derived class will not be called when you delete an object through a base pointer, and you'll get corrupted program states like memory leaks.)

You can control whether the turtle's movement will leave a trace using `PenUp()` and `PenDown()`, and control its movement using `Forward()`, `Turn()`, and `GoTo()`. Finally, `GetX()` and `GetY()` tell you the current position of the turtle.

Your program will normally use a real implementation of this interface. In tests, you can use a mock implementation instead. This allows you to easily check what drawing primitives your program is calling, with what arguments, and in which order. Tests written this way are much more robust (they won't break because your new machine does anti-aliasing differently), easier to read and maintain (the intent of a test is expressed in the code, not in some binary images), and run *much, much faster*.

Writing the Mock Class

If you are lucky, the mocks you need to use have already been implemented by some nice people. If, however, you find yourself in the position to write a mock class, relax - Google Mock turns this task into a fun game! (Well, almost.)

How to Define It

Using the `Turtle` interface as example, here are the simple steps you need to follow:

1. Derive a class `MockTurtle` from `Turtle`.
2. Take a *virtual* function of `Turtle` (while it's possible to mock non-virtual methods using templates, it's much more involved). Count how many arguments it has.
3. In the `public:` section of the child class, write `MOCK_METHODn();` (or `MOCK_CONST_METHODn();` if you are mocking a `const` method), where `n` is the number of the arguments; if you counted wrong, shame on you, and a compiler error will tell you so.
4. Now comes the fun part: you take the function signature, cut-and-paste the *function name* as the *first* argument to the macro, and leave what's left as the *second* argument (in case you're curious, this is the *type of the function*).
5. Repeat until all virtual functions you want to mock are done.

After the process, you should have something like:

```
#include "gmock/gmock.h"  // Brings in Google Mock.
class MockTurtle : public Turtle {
 public:
  ...
  MOCK_METHOD0(PenUp, void());
  MOCK_METHOD0(PenDown, void());
  MOCK_METHOD1(Forward, void(int distance));
  MOCK_METHOD1(Turn, void(int degrees));
  MOCK_METHOD2(GoTo, void(int x, int y));
```

```
  MOCK_CONST_METHOD0(GetX, int());
  MOCK_CONST_METHOD0(GetY, int());
};
```

You don't need to define these mock methods somewhere else - the `MOCK_METHOD*` macros will generate the definitions for you. It's that simple! Once you get the hang of it, you can pump out mock classes faster than your source-control system can handle your check-ins.

Tip: If even this is too much work for you, you'll find the `gmock_gen.py` tool in Google Mock's `scripts/generator/` directory (courtesy of the cppclean project) useful. This command-line tool requires that you have Python 2.4 installed. You give it a C++ file and the name of an abstract class defined in it, and it will print the definition of the mock class for you. Due to the complexity of the C++ language, this script may not always work, but it can be quite handy when it does. For more details, read the user documentation.

Where to Put It

When you define a mock class, you need to decide where to put its definition. Some people put it in a `*_test.cc`. This is fine when the interface being mocked (say, `Foo`) is owned by the same person or team. Otherwise, when the owner of `Foo` changes it, your test could break. (You can't really expect `Foo`'s maintainer to fix every test that uses `Foo`, can you?)

So, the rule of thumb is: if you need to mock `Foo` and it's owned by others, define the mock class in `Foo`'s package (better, in a `testing` sub-package such that you can clearly separate production code and testing utilities), and put it in a `mock_foo.h`. Then everyone can reference `mock_foo.h` from their tests. If `Foo` ever changes, there is only one copy of `MockFoo` to change, and only tests that depend on the changed methods need to be fixed.

Another way to do it: you can introduce a thin layer `FooAdaptor` on top of `Foo` and code to this new interface. Since you own `FooAdaptor`, you can absorb changes in `Foo` much more easily. While this is more work initially, carefully choosing the adaptor interface can make your code easier to write and more readable (a net win in the long run), as you can choose `FooAdaptor` to fit your specific domain much better than `Foo` does.

Using Mocks in Tests

Once you have a mock class, using it is easy. The typical work flow is:

1. Import the Google Mock names from the `testing` namespace such that you can use them unqualified (You only have to do it once per file. Remember that namespaces are a good idea and good for your health.).
2. Create some mock objects.
3. Specify your expectations on them (How many times will a method be called? With what arguments? What should it do? etc.).
4. Exercise some code that uses the mocks; optionally, check the result using Google Test assertions. If a mock method is called more than expected or with wrong arguments, you'll get an error immediately.
5. When a mock is destructed, Google Mock will automatically check whether all expectations on it have been satisfied.

Here's an example:

```
#include "path/to/mock-turtle.h"
#include "gmock/gmock.h"
#include "gtest/gtest.h"
using ::testing::AtLeast;                     // #1

TEST(PainterTest, CanDrawSomething) {
  MockTurtle turtle;                          // #2
  EXPECT_CALL(turtle, PenDown())              // #3
      .Times(AtLeast(1));

  Painter painter(&turtle);                   // #4

  EXPECT_TRUE(painter.DrawCircle(0, 0, 10));
}                                             // #5

int main(int argc, char** argv) {
  // The following line must be executed to initialize Google Mock
```

```
  // (and Google Test) before running the tests.
  ::testing::InitGoogleMock(&argc, argv);
  return RUN_ALL_TESTS();
}
```

As you might have guessed, this test checks that `PenDown()` is called at least once. If the `painter` object didn't call this method, your test will fail with a message like this:

```
path/to/my_test.cc:119: Failure
Actual function call count doesn't match this expectation:
Actually: never called;
Expected: called at least once.
```

Tip 1: If you run the test from an Emacs buffer, you can hit `<Enter>` on the line number displayed in the error message to jump right to the failed expectation.

Tip 2: If your mock objects are never deleted, the final verification won't happen. Therefore it's a good idea to use a heap leak checker in your tests when you allocate mocks on the heap.

Important note: Google Mock requires expectations to be set **before** the mock functions are called, otherwise the behavior is **undefined**. In particular, you mustn't interleave `EXPECT_CALL()` s and calls to the mock functions.

This means `EXPECT_CALL()` should be read as expecting that a call will occur *in the future*, not that a call has occurred. Why does Google Mock work like that? Well, specifying the expectation beforehand allows Google Mock to report a violation as soon as it arises, when the context (stack trace, etc) is still available. This makes debugging much easier.

Admittedly, this test is contrived and doesn't do much. You can easily achieve the same effect without using Google Mock. However, as we shall reveal soon, Google Mock allows you to do *much more* with the mocks.

Using Google Mock with Any Testing Framework

If you want to use something other than Google Test (e.g. CppUnit or CxxTest) as your testing framework, just change the `main()` function in the previous section to:

```
int main(int argc, char** argv) {
  // The following line causes Google Mock to throw an exception on failure,
  // which will be interpreted by your testing framework as a test failure.
  ::testing::GTEST_FLAG(throw_on_failure) = true;
  ::testing::InitGoogleMock(&argc, argv);
  ... whatever your testing framework requires ...
}
```

This approach has a catch: it makes Google Mock throw an exception from a mock object's destructor sometimes. With some compilers, this sometimes causes the test program to crash. You'll still be able to notice that the test has failed, but it's not a graceful failure.

A better solution is to use Google Test's event listener API to report a test failure to your testing framework properly. You'll need to implement the `OnTestPartResult()` method of the event listener interface, but it should be straightforward.

If this turns out to be too much work, we suggest that you stick with Google Test, which works with Google Mock seamlessly (in fact, it is technically part of Google Mock.). If there is a reason that you cannot use Google Test, please let us know.

Setting Expectations

The key to using a mock object successfully is to set the *right expectations* on it. If you set the expectations too strict, your test will fail as the result of unrelated changes. If you set them too loose, bugs can slip through. You want to do it just right such that your test can catch exactly the kind of bugs you intend it to catch. Google Mock provides the necessary means for you to do it "just right."

General Syntax

In Google Mock we use the `EXPECT_CALL()` macro to set an expectation on a mock method. The general syntax is:

```
EXPECT_CALL(mock_object, method(matchers))
    .Times(cardinality)
    .WillOnce(action)
    .WillRepeatedly(action);
```

The macro has two arguments: first the mock object, and then the method and its arguments. Note that the two are separated by a comma (`,`), not a period (`.`). (Why using a comma? The answer is that it was necessary for technical reasons.)

The macro can be followed by some optional *clauses* that provide more information about the expectation. We'll discuss how each clause works in the coming sections.

This syntax is designed to make an expectation read like English. For example, you can probably guess that

```
using ::testing::Return;
...
EXPECT_CALL(turtle, GetX())
    .Times(5)
    .WillOnce(Return(100))
    .WillOnce(Return(150))
    .WillRepeatedly(Return(200));
```

says that the `turtle` object's `GetX()` method will be called five times, it will return 100 the first time, 150 the second time, and then 200 every time. Some people like to call this style of syntax a Domain-Specific Language (DSL).

Note: Why do we use a macro to do this? It serves two purposes: first it makes expectations easily identifiable (either by `grep` or by a human reader), and second it allows Google Mock to include the source file location of a failed expectation in messages, making debugging easier.

Matchers: What Arguments Do We Expect?

When a mock function takes arguments, we must specify what arguments we are expecting; for example:

```
// Expects the turtle to move forward by 100 units.
EXPECT_CALL(turtle, Forward(100));
```

Sometimes you may not want to be too specific (Remember that talk about tests being too rigid? Over specification leads to brittle tests and obscures the intent of tests. Therefore we encourage you to specify only what's necessary - no more, no less.). If you care to check that `Forward()` will be called but aren't interested in its actual argument, write `_` as the argument, which means "anything goes":

```
using ::testing::_;
...
// Expects the turtle to move forward.
EXPECT_CALL(turtle, Forward(_));
```

`_` is an instance of what we call **matchers**. A matcher is like a predicate and can test whether an argument is what we'd expect. You can use a matcher inside `EXPECT_CALL()` wherever a function argument is expected.

A list of built-in matchers can be found in the CheatSheet. For example, here's the `Ge` (greater than or equal) matcher:

```
using ::testing::Ge;
...
EXPECT_CALL(turtle, Forward(Ge(100)));
```

This checks that the turtle will be told to go forward by at least 100 units.

Cardinalities: How Many Times Will It Be Called?

The first clause we can specify following an `EXPECT_CALL()` is `Times()`. We call its argument a **cardinality** as it tells *how many times* the call should occur. It allows us to repeat an expectation many times without actually writing it as many times. More importantly, a cardinality can be "fuzzy", just like a matcher can be. This allows a user to express the intent of a test exactly.

An interesting special case is when we say `Times(0)`. You may have guessed - it means that the function shouldn't be called with the given arguments at all, and Google Mock will report a Google Test failure whenever the function is (wrongfully) called.

We've seen `AtLeast(n)` as an example of fuzzy cardinalities earlier. For the list of built-in cardinalities you can use, see the CheatSheet.

The `Times()` clause can be omitted. **If you omit `Times()`, Google Mock will infer the cardinality for you.** The rules are easy to remember:

- If **neither** `WillOnce()` **nor** `WillRepeatedly()` is in the `EXPECT_CALL()`, the inferred cardinality is `Times(1)`.
- If there are `n WillOnce()`'s but **no** `WillRepeatedly()`, where `n` >= 1, the cardinality is `Times(n)`.
- If there are `n WillOnce()`'s and **one** `WillRepeatedly()`, where `n` >= 0, the cardinality is `Times(AtLeast(n))`.

Quick quiz: what do you think will happen if a function is expected to be called twice but actually called four times?

Actions: What Should It Do?

Remember that a mock object doesn't really have a working implementation? We as users have to tell it what to do when a method is invoked. This is easy in Google Mock.

First, if the return type of a mock function is a built-in type or a pointer, the function has a **default action** (a `void` function will just return, a `bool` function will return `false`, and other functions will return 0). In addition, in C++ 11 and above, a mock function whose return type is default-constructible (i.e. has a default constructor) has a default action of returning a default-constructed value. If you don't say anything, this behavior will be used.

Second, if a mock function doesn't have a default action, or the default action doesn't suit you, you can specify the action to be taken each time the expectation matches using a series of `WillOnce()` clauses followed by an optional `WillRepeatedly()`. For example,

```
using ::testing::Return;
...
EXPECT_CALL(turtle, GetX())
    .WillOnce(Return(100))
    .WillOnce(Return(200))
    .WillOnce(Return(300));
```

This says that `turtle.GetX()` will be called *exactly three times* (Google Mock inferred this from how many `WillOnce()` clauses we've written, since we didn't explicitly write `Times()`), and will return 100, 200, and 300 respectively.

```
using ::testing::Return;
...
EXPECT_CALL(turtle, GetY())
    .WillOnce(Return(100))
    .WillOnce(Return(200))
    .WillRepeatedly(Return(300));
```

says that `turtle.GetY()` will be called *at least twice* (Google Mock knows this as we've written two `WillOnce()` clauses and a `WillRepeatedly()` while having no explicit `Times()`), will return 100 the first time, 200 the second time, and 300 from the third time on.

Of course, if you explicitly write a `Times()`, Google Mock will not try to infer the cardinality itself. What if the number you specified is larger than there are `WillOnce()` clauses? Well, after all `WillOnce()`s are used up, Google Mock will do the *default* action for the function every time (unless, of course, you have a `WillRepeatedly()`.).

What can we do inside `WillOnce()` besides `Return()`? You can return a reference using `ReturnRef(variable)`, or invoke a pre-defined function, among others.

Important note: The `EXPECT_CALL()` statement evaluates the action clause only once, even though the action may be performed many times. Therefore you must be careful about side effects. The following may not do what you want:

```
int n = 100;
EXPECT_CALL(turtle, GetX())
.Times(4)
.WillRepeatedly(Return(n++));
```

Instead of returning 100, 101, 102, ..., consecutively, this mock function will always return 100 as `n++` is only evaluated once. Similarly, `Return(new Foo)` will create a new `Foo` object when the `EXPECT_CALL()` is executed, and will return the same pointer every time. If you want the side effect to happen every time, you need to define a custom action, which we'll teach in the CookBook.

Time for another quiz! What do you think the following means?

```
using ::testing::Return;
...
EXPECT_CALL(turtle, GetY())
.Times(4)
.WillOnce(Return(100));
```

Obviously `turtle.GetY()` is expected to be called four times. But if you think it will return 100 every time, think twice! Remember that one `WillOnce()` clause will be consumed each time the function is invoked and the default action will be taken afterwards. So the right answer is that `turtle.GetY()` will return 100 the first time, but **return 0 from the second time on**, as returning 0 is the default action for `int` functions.

Using Multiple Expectations

So far we've only shown examples where you have a single expectation. More realistically, you're going to specify expectations on multiple mock methods, which may be from multiple mock objects.

By default, when a mock method is invoked, Google Mock will search the expectations in the **reverse order** they are defined, and stop when an active expectation that matches the arguments is found (you can think of it as "newer rules override older ones."). If the matching expectation cannot take any more calls, you will get an upper-bound-violated failure. Here's an example:

```
using ::testing::_;
...
EXPECT_CALL(turtle, Forward(_));  // #1
EXPECT_CALL(turtle, Forward(10))  // #2
    .Times(2);
```

If `Forward(10)` is called three times in a row, the third time it will be an error, as the last matching expectation (#2) has been saturated. If, however, the third `Forward(10)` call is replaced by `Forward(20)`, then it would be OK, as now #1 will be the matching expectation.

Side note: Why does Google Mock search for a match in the *reverse* order of the expectations? The reason is that this allows a user to set up the default expectations in a mock object's constructor or the test fixture's set-up phase and then customize the mock by writing more specific expectations in the test body. So, if you have two expectations on the same method, you want to put the one with more specific matchers **after** the other, or the more specific rule would be shadowed by the more general one that comes after it.

Ordered vs Unordered Calls

By default, an expectation can match a call even though an earlier expectation hasn't been satisfied. In other words, the calls don't have to occur in the order the expectations are specified.

Sometimes, you may want all the expected calls to occur in a strict order. To say this in Google Mock is easy:

```
using ::testing::InSequence;
...
```

```
TEST(FooTest, DrawsLineSegment) {
  ...
  {
    InSequence dummy;

    EXPECT_CALL(turtle, PenDown());
    EXPECT_CALL(turtle, Forward(100));
    EXPECT_CALL(turtle, PenUp());
  }
  Foo();
}
```

By creating an object of type `InSequence`, all expectations in its scope are put into a *sequence* and have to occur *sequentially*. Since we are just relying on the constructor and destructor of this object to do the actual work, its name is really irrelevant.

In this example, we test that `Foo()` calls the three expected functions in the order as written. If a call is made out-of-order, it will be an error.

(What if you care about the relative order of some of the calls, but not all of them? Can you specify an arbitrary partial order? The answer is ... yes! If you are impatient, the details can be found in the CookBook.)

All Expectations Are Sticky (Unless Said Otherwise)

Now let's do a quick quiz to see how well you can use this mock stuff already. How would you test that the turtle is asked to go to the origin *exactly twice* (you want to ignore any other instructions it receives)?

After you've come up with your answer, take a look at ours and compare notes (solve it yourself first - don't cheat!):

```
using ::testing::_;
...
EXPECT_CALL(turtle, GoTo(_, _))  // #1
    .Times(AnyNumber());
```

```
EXPECT_CALL(turtle, GoTo(0, 0))  // #2
    .Times(2);
```

Suppose `turtle.GoTo(0, 0)` is called three times. In the third time, Google Mock will see that the arguments match expectation #2 (remember that we always pick the last matching expectation). Now, since we said that there should be only two such calls, Google Mock will report an error immediately. This is basically what we've told you in the "Using Multiple Expectations" section above.

This example shows that **expectations in Google Mock are "sticky" by default**, in the sense that they remain active even after we have reached their invocation upper bounds. This is an important rule to remember, as it affects the meaning of the spec, and is **different** to how it's done in many other mocking frameworks (Why'd we do that? Because we think our rule makes the common cases easier to express and understand.).

Simple? Let's see if you've really understood it: what does the following code say?

```
using ::testing::Return;
...
for (int i = n; i > 0; i--) {
  EXPECT_CALL(turtle, GetX())
      .WillOnce(Return(10*i));
}
```

If you think it says that `turtle.GetX()` will be called `n` times and will return 10, 20, 30, ..., consecutively, think twice! The problem is that, as we said, expectations are sticky. So, the second time `turtle.GetX()` is called, the last (latest) `EXPECT_CALL()` statement will match, and will immediately lead to an "upper bound exceeded" error - this piece of code is not very useful!

One correct way of saying that `turtle.GetX()` will return 10, 20, 30, ..., is to explicitly say that the expectations are *not* sticky. In other words, they should *retire* as soon as they are saturated:

```
using ::testing::Return;
...
for (int i = n; i > 0; i--) {
  EXPECT_CALL(turtle, GetX())
    .WillOnce(Return(10*i))
    .RetiresOnSaturation();
}
```

And, there's a better way to do it: in this case, we expect the calls to occur in a specific order, and we line up the actions to match the order. Since the order is important here, we should make it explicit using a sequence:

```
using ::testing::InSequence;
using ::testing::Return;
...
{
  InSequence s;

  for (int i = 1; i <= n; i++) {
    EXPECT_CALL(turtle, GetX())
        .WillOnce(Return(10*i))
        .RetiresOnSaturation();
  }
}
```

By the way, the other situation where an expectation may *not* be sticky is when it's in a sequence - as soon as another expectation that comes after it in the sequence has been used, it automatically retires (and will never be used to match any call).

Uninteresting Calls

A mock object may have many methods, and not all of them are that interesting. For example, in some tests we may not care about how many times `GetX()` and `GetY()` get called.

In Google Mock, if you are not interested in a method, just don't say anything about it. If a call to this method occurs, you'll see a warning in the test output, but it won't be a failure.

What Now?

Congratulations! You've learned enough about Google Mock to start using it. Now, you might want to join the googlemock discussion group and actually write some tests using Google Mock - it will be fun. Hey, it may even be addictive - you've been warned.

Then, if you feel like increasing your mock quotient, you should move on to the CookBook. You can learn many advanced features of Google Mock there -- and advance your level of enjoyment and testing bliss.