gMock Framework (For Dummies)

★ What is gMock?

⇒ A mock object implements the same interface as a real object.
Lets you specify at run time:

- how it will be used
- what it should do
  - which methods will be called?
  - in which order?
  - how many times?
  - with what arguments?
  - what will they return?
  - etc...

⇒ 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.
- Mocks are objects pre-programmed with expectations, which form a specification of the calls they are expected to receive.

⇒ gMock is a library for creating mock classes and using them.

⇒ When using gMock:

- you use some simple macros to describe the interface you want to mock, and they will expand to the implementation of your mock class
- you create some mock objects and specify its expectations and behavior using an intuitive syntax
- you exercise code that uses the mock objects. gMock will catch any violation to the expectations as soon as it arises.

★ Writing the Mock Class

● How to define it

⇒ Suppose we want to mock Turtle class.

→ Derive a class MockTurtle from Turtle.

→ Take a virtual function of Turtle

→ In the public: section of the child class, write MOCK_METHOD()

→ Now comes the fun part: you take the function signature, cut-and-paste it into the macro, and add two commas
  - one between the return type and the name
  - another between the name and the argument list.

→If you're mocking a const method, add a 4th parameter containing (const)

→Since you're overriding a virtual method, we suggest adding the override keyword. For const methods the 4th parameter becomes (const, override) , for non-const methods just (override) .

→Repeat until all virtual functions you want to mock are done.

```
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;
};
```

```
#include "gmock/gmock.h"  // Brings in gMock.

class MockTurtle : public Turtle {
 public:
  ...
  MOCK_METHOD(void, PenUp, (), (override));
  MOCK_METHOD(void, PenDown, (), (override));
  MOCK_METHOD(void, Forward, (int distance), (override));
  MOCK_METHOD(void, Turn, (int degrees), (override));
  MOCK_METHOD(void, GoTo, (int x, int y), (override));
  MOCK_METHOD(int, GetX, (), (const, override));
  MOCK_METHOD(int, GetY, (), (const, override));
};
```

- 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 .

★ Using Mocks in Tests

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

→Import the gMock names from the testing namespace such that you can use them unqualified.

→Create some mock objects.

→Specify your expectations on them
- How many times will a method be called?
- With what arguments?
- What should it do?
- etc...

→Exercise some code that uses the mocks.optionally, check the result using googletest assertions.

→When a mock is destructed, gMock 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
}
```

⇛ 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.
Stack trace:
...
```

⇛ gMock requires expectations to be set before the mock functions are called, otherwise the behavior is undefined.

★ Setting Expectations

⇛ The key to using a mock object successfully is to set the right expectations on it.

● General Syntax

⇛ In gMock 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);
```

⇛ If the method is not overloaded, the macro can also be called without matchers:

```
EXPECT_CALL(mock_object, non-overloaded-method)
    .Times(cardinality)
    .WillOnce(action)
    .WillRepeatedly(action);
```

⇛ Either form of the macro can be followed by some optional clauses that provide more information about the expectation.

● Matches: What arguments do we expect?

⇛ When a mock function takes arguments, we may specify what arguments we are expecting, for example:

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

⇛ Oftentimes you do not want to be too specific.

⇛ Therefore we encourage you to specify only what's necessary—no more, no less.

⇛ If you aren't interested in the value of an argument, write _ as the argument, which means "anything goes":

```
using ::testing::_;
...
// Expects that the turtle jumps to somewhere on the x=50 line.
EXPECT_CALL(turtle, GoTo(50, _));
```

⇛ _ is an instance of what we call matchers.

⇒ If you don't care about any arguments, rather than specify _ for each of them you may instead omit the parameter list:

```
// Expects the turtle to move forward.
EXPECT_CALL(turtle, Forward);
// Expects the turtle to jump somewhere.
EXPECT_CALL(turtle, GoTo);
```

⇒ This works for all non-overloaded methods; if a method is overloaded, you need to help gMock resolve which overload is expected by specifying the number ofarguments and possibly also the types of the arguments.

• 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.

⇒ An interesting special case is when we say Times(0) .

↳ It means that the function shouldn't be called with the given arguments at all, and gMock will report a googletest failure whenever the function is (wrongfully) called.

⇒ We've seen AtLeast(n) as an example of fuzzy cardinalities earlier.

⇒ If you omit Times() , gMock 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)) .

• Actions: What should it do?

⇒ Mock object doesn't really have a working implementation.

⇒ We as users have to tell it what to do when a method is invoked.

⇒ You can specify the action to be taken each time the expectation matches using a series of WillOnce() clauses followed by anoptional WillRepeatedly().

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

{Example}

⇒ What can we do inside WillOnce() besides Return()?

- You can return a reference using ReturnRef(*variable*)
- Invoke a pre-defined function

Using multiple Expectation

⇒ By default, when a mock method is invoked,
- gMock will search the expectations in the reverse order they are defined
- and stop when an active expectation that matches the arguments is found

⇒ If the matching expectation cannot take any more calls, you will get an upper-bound-violated failure.

Ordered Vs UnOrdered Calls

⇒ You may want all the expected calls to occur in a strict order.

```
using ::testing::InSequence;
...
TEST(FooTest, DrawsLineSegment) {
  ...
  {
    InSequence seq;

    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.

⇒ Expectations in gMock are “sticky” by default, in the sense that they remain active even after we have reached their invocation upper bounds.