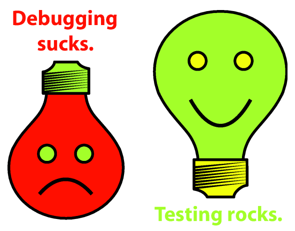

Testing on the Toilet
Don't Overuse Mocks

When writing tests for your code, **it can seem easy to ignore your code's dependencies by mocking them out.**

```
public void testCreditCardIsCharged() {
  paymentProcessor = new PaymentProcessor(mockCreditCardServer);
  when(mockCreditCardServer.isServerAvailable()).thenReturn(true);
  when(mockCreditCardServer.beginTransaction().thenReturn(mockTransactionManager);
  when(mockTransactionManager.getTransaction().thenReturn(transaction);
  when(mockCreditCardServer.pay(transaction, creditCard, 500).thenReturn(mockPayment);
  when(mockPayment.isOverMaxBalance()).thenReturn(false);

  paymentProcessor.processPayment(creditCard, Money.dollars(500));
  verify(mockCreditCardServer).pay(transaction, creditCard, 500);
}
```

However, **not using mocks can sometimes result in tests that are simpler and more useful.**

```
public void testCreditCardIsCharged() {
  paymentProcessor = new PaymentProcessor(creditCardServer);
  paymentProcessor.processPayment(creditCard, Money.dollars(500));
  assertEquals(500, creditCardServer.getMostRecentCharge(creditCard));
}
```

Overusing mocks can cause several problems:

- **Tests can be harder to understand.** Instead of just a straightforward usage of your code (e.g. pass in some values to the method under test and check the return result), you need to include extra code to tell the mocks how to behave. Having this extra code detracts from the actual intent of what you're trying to test, and very often this code is hard to understand if you're not familiar with the implementation of the production code.

- **Tests can be harder to maintain.** When you tell a mock how to behave, you're leaking implementation details of your code into your test. When implementation details in your production code change, you'll need to update your tests to reflect these changes. Tests should typically know little about the code's implementation, and should focus on testing the code's public interface.

- **Tests can provide less assurance that your code is working properly.** When you tell a mock how to behave, the only assurance you get with your tests is that your code will work if your mocks behave exactly like your real implementations. This can be very hard to guarantee, and the problem gets worse as your code changes over time, as the behavior of the real implementations is likely to get out of sync with your mocks.

Some signs that you're overusing mocks are if you're mocking out more than one or two classes, or if one of your mocks specifies how more than one or two methods should behave. **If you're trying to read a test that uses mocks and find yourself mentally stepping through the code being tested in order to understand the test, then you're probably overusing mocks.**

Sometimes you can't use a real dependency in a test (e.g. if it's too slow or talks over the network), **but there may better options than using mocks**, such as a hermetic local server (e.g. a credit card server that you start up on your machine specifically for the test) or a fake implementation (e.g. an in-memory credit card server).

More information, discussion, and archives:
http://googletesting.blogspot.com