Test Isolation and Mocking

Technion – Institute of Technology
236700

Author: Gal Lalouche ©
Unit Testing: Isolation

- Initializing collaborators can include a lot of boiler plate
- There may be bugs in the dependencies
  - Or dependencies may not even be written yet!
- I/O operations (databases/files/sockets) in test subjects may slow down tests and be hard to configure
  - They might have nasty side effects and/or need to be cleaned up after each test
- Using concrete classes couples our tests with those classes
Unit Testing: Isolation

• Solution:
  • Replace dependencies with mock objects
  • simple dependencies that are 100% bug free
    • Although it is possible for bugs to appear in the mock objects configurations
  • Easy to control their behavior
  • Easy to assert their use by the tested class
Unit Testing: Isolation

This is another example where code **testability** improves its design!

- Isolation forces the code to be:
  - Modular
  - **Decoupled** from its dependencies
  - More **coherent**
Unit Testing: Mockito

• Problem:
  • Writing mock objects can be a pain
    • A great deal of time is wasted writing an object that is only used for testing
  • Configuration can be tricky
  • Validation is hard

• Case Study – Shopping cart
  • Shoppers can add/remove items from the shopping cart
  • Each item is a complex object with many attributes
  • The shopping cart is initialized with a ShoppingSuggestions which offer other purchases based on the shopping cart content
Case study – Shopping cart
Case study – Shopping cart

• We want to test the ShoppingCart class

```java
public class ShoppingCart {
    private Map<Item, Integer> items = new HashMap<>();
    private ShoppingSuggestions suggestions;

    public ShoppingCart(ShoppingSuggestions suggestions) {
        this.suggestions = suggestions;
    }

    ...
}
```

• The ShoppingSuggestions is responsible for displaying product suggestions based on the shopping cart contents
Case study – Shopping cart

- We would like to test the following methods

```java
public void addItem(Item item) {
    int amount = 1;
    if (items.containsKey(item)) {
        amount = items.get(item) + 1;
    }
    items.put(item, amount);
    suggestions.displaySuggestionFor(item);
}

public void removeItem(Item item) {
    if (!items.containsKey(item))
        return;
    int amount = items.get(item);
    if (amount == 0) {
        items.remove(item);
        suggestions.removeSuggestionFor(item);
        return;
    }
    items.put(item, amount - 1);
}
```

Bug: Should be 1
Testing our Shopping cart – Take 1

- We want to isolate the subject class and replace arguments and dependencies with controlled objects (Mocks)

- We will start with a naïve mock of ShoppingSuggestions
  - We will soon see this is not enough

```java
public class FakeShoppingSuggestions implements ShoppingSuggestions {
    public boolean invoked = false;

    @Override
    public void removeSuggestionFor(Item item) { invoked = true; }

    @Override
    public void displaySuggestionFor(Item item) { invoked = true; }
}
```
Testing our Shopping cart – Take 1

- Let’s look at our first test:

```java
@Test
public void updateSuggestionWhenAddingItem() {
    FakeShoppingSuggestions ss = new FakeShoppingSuggestions();
    Item someItem = new Item(1, 3000, "laptop", "Lenovo", Category.COMPUTERS);
    ShoppingCart cart = new ShoppingCart(ss);
    cart.addItem(someItem);
    assertTrue(ss.invoked);
}
```

We would like to use the `ShoppingSuggestions` interface, so to be sure we haven’t introduced bugs in the test... but we can’t because of the last assert.

We don’t really care about these details.

This test doesn’t check how many times each method was invoked, and is susceptible to bugs in the mock object.
Testing our Shopping cart – Take 1

- Our second test is even more problematic

```java
@Test
public void removeSuggestionWhenRemovingLastItem() {
    FakeShoppingSuggestions ss = new FakeShoppingSuggestions();
    Item someItem = new Item(1, 3000, "laptop", "Lenovo", Category.Computers);
    ShoppingCart cart = new ShoppingCart(ss);
    cart.addItem(someItem);
    ss.invoked = false;
    cart.removeItem(someItem);
    assertTrue(ss.invoked);
}
```

We must **update** our mocking object throughout the test. What happens if we **forget**?

- In order to deal with some of these problems we must write **smarter** (and possibly **buggier**) mock objects.
Testing our Shopping cart – Mockito

- We will use a Mocking Framework (Mockito) to **create** our mock objects, **define** their behavior and **verify** the results.

```java
@Test
public void updateSuggestionWhenAddingItem() {
    ShoppingSuggestions ss = Mockito.mock(ShoppingSuggestions.class);
    Item someItem = Mockito.mock(Item.class);
    ShoppingCart cart = new ShoppingCart(ss);

    cart.addItem(someItem);

    Mockito.verify(ss, Mockito.only()).displaySuggestionFor(someItem);
}
```

We simply ask for a ShoppingSuggestions and Item mocks.

We can verify that the displaySuggestionFor() method will be called **exactly once** with the someItem argument.
Testing our Shopping cart – Take 2

Let’s look at our second test:

```java
@Test
public void removeSuggestionWhenRemovingLastItem() {
    ShoppingSuggestions ss = Mockito.mock(ShoppingSuggestions.class);
    Item someItem = Mockito.mock(Item.class);
    ShoppingCart cart = new ShoppingCart(ss);

    cart.addItem(someItem);
    Mockito.verify(ss, Mockito.only()).displaySuggestionFor(someItem);

    cart.removeItem(someItem);
    Mockito.verify(ss, Mockito.only()).removeSuggestionFor(someItem);
}
```

Wanted but not invoked:
suggestions.removeSuggestionFor(
    Mock for Item, hashCode: 1992679988
);
-> at
ShoppingCartTestWithMockito.testRemoveItem(ShoppingCartTestWithMockito.java:29)
Testing our Shopping cart – Take 2

- Using a Mocking Framework we do not need to write a complete class to define the mock behavior:

```java
@Test
public void testTotalAmount() {
    ShoppingSuggestions ss = Mockito.mock(ShoppingSuggestions.class);
    Item someItem = Mockito.mock(Item.class);
    ShoppingCart cart = new ShoppingCart(ss);

    Mockito.when(someItem.getPrice()).thenReturn(70);
    Mockito.when(someItem.getShippingCosts()).thenReturn(6);

    cart.addItem(someItem);
    Assert.assertEquals(76, cart.total());
}
```

We only define what we’re interested in. We can also tell the mock to throw an exception.
Mocking – Pros and Cons

• **Pros**
  - Easy to use
  - We don’t have to bother ourselves with configuration if we don’t want to
  - We are truly unit testing. Our tests aren’t dependent on other classes
  - Mocks let us design against an interface, rather than an interface

• **Cons**
  - Mocks are still harder to use than real classes
  - Mocks use reflection, so they are slower than Fakes or Stubs
  - Naturally white-box testing
    - We are deeply coupled with the class’s implementation
    - Dependencies change => tests break => Brittle test
  - **Mocks let us test side-effects, but we should strive to make our code side-effect less**
So What should I mock?

- You **should** mock when:
  - You don’t want to be tied down to a **single** implementation
  - Using real objects is impossible or too expensive
    - Due to I/O, networking, side effects, concurrency, etc.
    - But you should *consider* using **Fakes** in those cases
  - You don’t care about initializing collaborators

- You **don’t** want to mock:
  - Data structures
  - For the sake of mocking; frameworks help, but mocking is still more complicated than using real objects

- You **can’t** mock **final** classes, methods and primitives
  - You can use **property-base testing** to create random primitives and strings
  - **Food for thought:** **File** is **final**, so how do we test code that uses **File** to access the file system?
Mocking – Additional Reading

• **Mocks aren’t Stubs**
• **Quick use guide**
• **Mockito website**
• Alternatives to Mockito: **JMock**, **EasyMock**, **PowerMock** (can mock **final**)
Appendix: Verifying order

- Mockito offers a few gems, such as verifying the order of method invocations

```java
@Test
public void testInOrder() {
    ShoppingSuggestions ss = Mockito.mock(ShoppingSuggestions.class);
    ShoppingCart cart = new ShoppingCart(ss);
    Item item1 = Mockito.mock(Item.class);
    Item item2 = Mockito.mock(Item.class);
    Item item3 = Mockito.mock(Item.class);
    cart.addItem(item1);
    cart.addItem(item2);
    cart.addItem(item3);
    InOrder inOrder = Mockito.inOrder(ss);
    inOrder.verify(ss).displaySuggestionFor(item1);
    inOrder.verify(ss).displaySuggestionFor(item2);
    inOrder.verify(ss).displaySuggestionFor(item3);
    Mockito.verifyNoMoreInteractions(ss);
}
```