Test Isolation and Mocking

Technion – Institute of Technology
236700
Manual mocking: problems

• Problem:
  • Writing mock objects can be a pain
    • A lot 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
Shopping cart: that thing
We want to test the ShoppingCart class

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 two 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
Manual mocking: implementation

- We want to isolate the subject class and replace arguments and dependencies with controlled objects (Mocks)
- We start with a **naive** mock of `ShoppingSuggestions`
  - We will soon see this is **not enough**

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

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

    @Override
    public void displaySuggestionFor(Item item) { invoked = true; }
}
```
Manual mocking (Cont.)

• Let’s look at our first test:

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

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

This test doesn’t check how many times each method was invoked, and is susceptible to bugs in the mock object.

We don’t really care about these details
Our second test is even more problematic

```java
@Test
public void removeSuggestionWhenRemovingLastItem() {
    FakeShoppingSuggestions ss = new FakeShoppingSuggestions();
    ShoppingCart cart = new ShoppingCart(ss);
    Item someItem = new Item(3000, "laptop", "Lenovo", Category.Computers);
    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.
Mockito (finally)

- 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.
Mockito: Error handling

- Let’s look at our second test:

```java
@Test
class ShoppingCartTestWithMockito {
  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)
```
Mockito: Configuration

- Using a *Mocking Framework* means 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.
Mockito supports multiple ways to verify arguments

- Match any argument

Mockito.verify(ss).addItem(ArgumentMatchers.any());

- Assert on argument

Mockito.verify(ss).addItem(ArgumentMatchers.argThat(
    item -> item.getName().equals("foobar"));

- Capture argument

ArgumentCaptor<Item> captor = ArgumentCaptor.forClass(item.class);
Mockito.verify(ss).addItem(captor.capture());
assertEquals(captor.getValue().getName(), "foobar");
Mockito: Advanced answers

Mockito supports even more fine tuned configuration

- Returning multiple answers

```java
Item item = Mockito.mock(Item.class);
Mockito.when(item.getItemId()).thenReturn(1, 2, 3, 4, 5);
```

- Specific arguments

```java
Mockito.when(item.getDiscountForCountry("Israel")) .thenReturn(100);
```

- Full answer (similar to overriding)

```java
Mockito.when(item.getDiscountForAmount(any())).
  .thenAnswer(invocation -> {
    double amount = (double) invocation.getArguments()[0];
    return amount * 0.1;
  });
```

- Spying (behaving like the original class – somewhat discouraged!)

```java
ShoppingSuggestion ssSpy = Mockito.spy(new ShoppingSuggestion());
```
Mockito: Mocking generic classes

- Mocking generic class can be annoying due to the type erasure

  ```java
  Foo<T> foo = mock(Foo.class); // unchecked warning
  ArgumentCaptor< Foo<T>> captor =
      ArgumentCaptor.forClass( Foo.class); // unchecked warning
  ```

- We can use rules and annotation to make our life a bit easier

  ```java
  @Mock private Foo<T> foo;
  @Captor private ArgumentCaptor< Foo<T>> captor;
  @Rule public MockitoRule mockitoRule = MockitoJUnit.rule();
  ```
Mockito: 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);
}
```
Mocking – Additional Reading

- Mocks aren’t Stubs
- Quick use guide
- Stubbing and Mocking with Mockito 2 and JUnit
- Mockito website
- Alternatives to Mockito: JMock, EasyMock, PowerMock