# Test driven development with JMock2

Andy Pols - andy@pols.co.uk
Romilly Cocking - romilly@cocking.co.uk
Nat Pryce - nat.pryce@gmail.com

## SPA 2007

With **lots** of contributions from Joe Walnes, Nat Pryce, Tim Mackinnon and Steve Freeman

# Overview

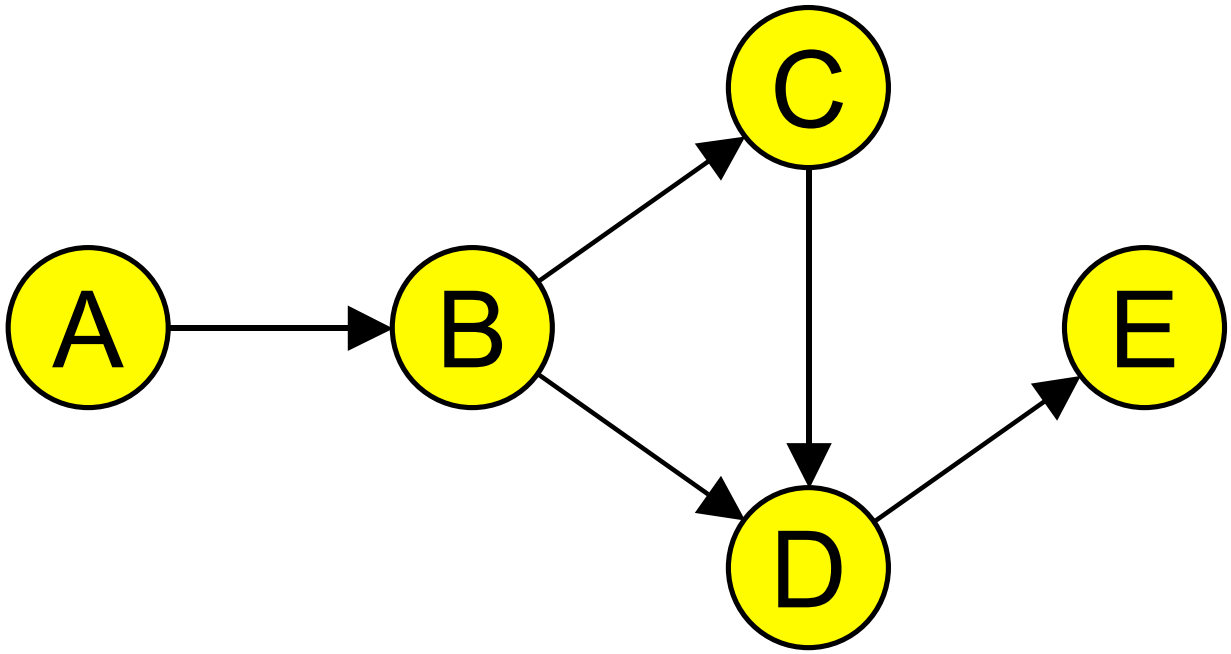
- What are mock objects?
- Using mocks to aid design
- How to use **JMock2**
- Tips for using mocks *effectively*

# *What* are mock objects?

*I can't programe without*
***Google*** *and I can't*
*programe without* ***tests****.*

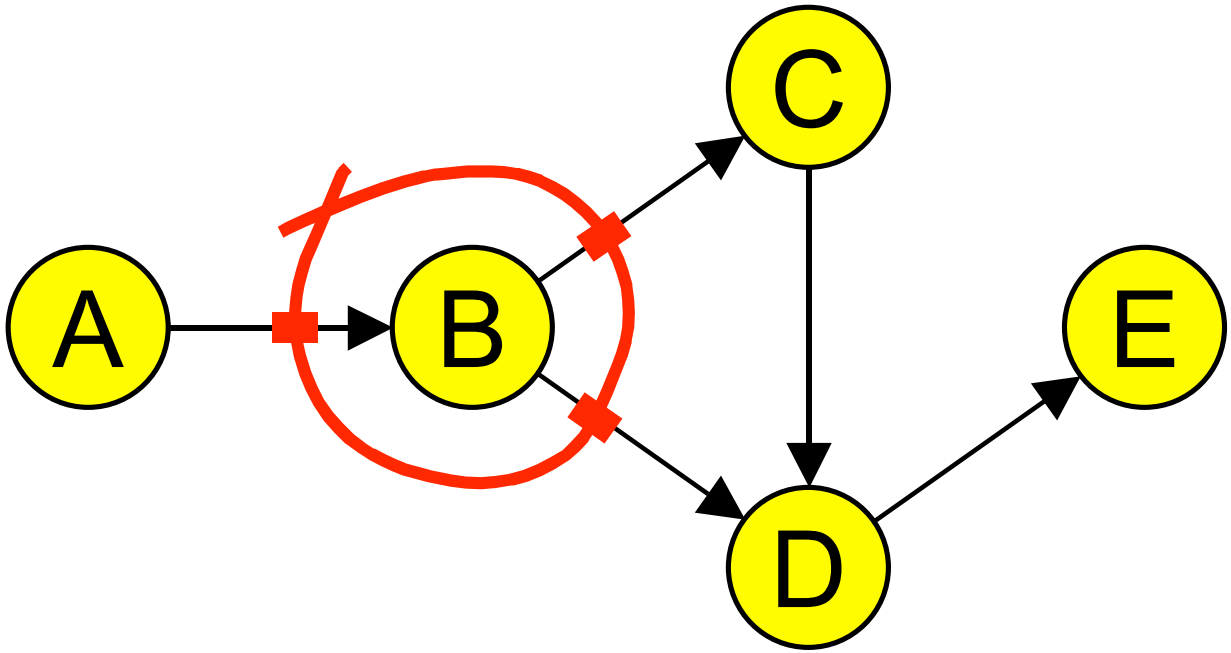
Nat Pryce, XTC

# An OO program is a network of collaborating objects

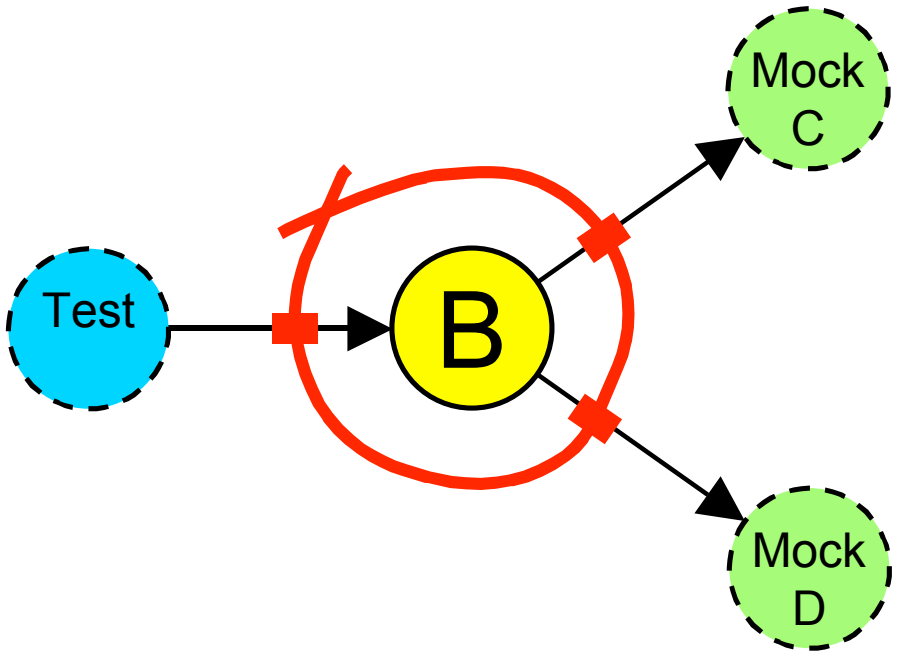



...that needs testing!

We need to be able to test an individual object in isolation without having to know how its collaborators work



To isolate the object under test, we replace its collaborators with a **mock** object



# Mocks != Stubs != Tracers

- **Stubs** provide canned results to enable tests to run predictably.
- **Mocks** provide instrumentation to verify how they have been interacted with (stub with assertions).
- **Tracers** are merely objects that are passed around during tests but are never invoked. Used for tracing
- Use them all but distinguish between them.

# **Mocks** let us test interactions between objects

- Which methods have been called (or not called)
- How many times they are called?
- Which order to the calls happen?
- What values are passed in?

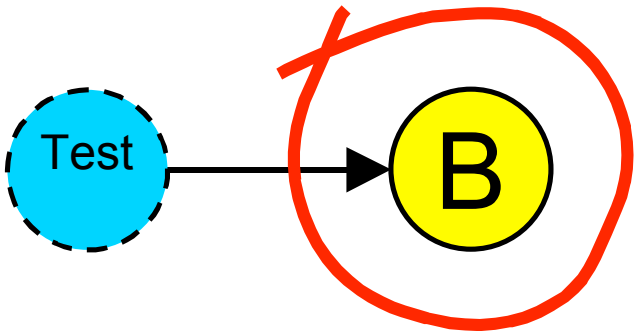
# Using mocks to aid design

# Test Driven Development, is a process of ***design discovery*** and ***validation***.

- It's not just about creating regression test suites.
- Using mocks tends to change the way you design (for the better)
- Let's explore this in more detail…

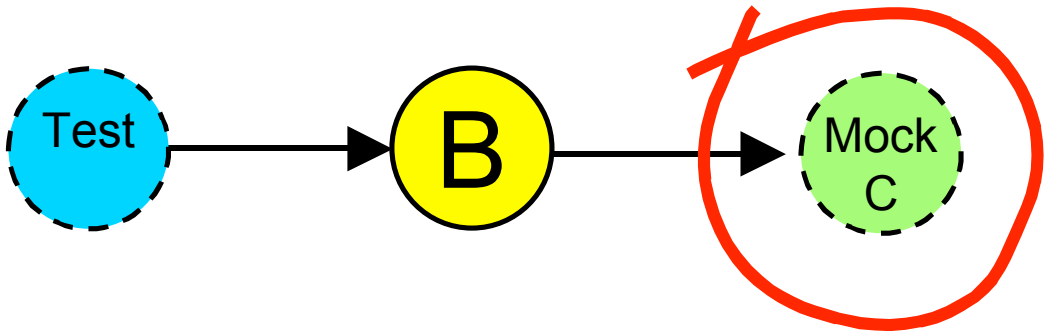
# Mocks can be used to iteratively design interfaces, top-down

TDD allows discovery of the interface for the object under test



# Mocks can be used to iteratively design interfaces, top-down

- Using mocks also allows the discovery of the interfaces for the dependencies.



# Mocks can be used to iteratively design interfaces, top-down

- Top down design allows you to focus on the most important APIs to get your job done. (helps form an application DSL).
- It also makes the tests FAST!

# Dependencies are explicit and externalized

- TDD with mocks forces dependencies to be explicitly injected.
- Avoids hidden dependencies.
- Externalizes configuration.

# Interaction based tests focus on object responsibilities, not structure

**Less** emphasis on internal state of objects.

**More** emphasis on how objects send messages to each other.

Typically results in **fewer** public getters, setters and collections

# *How* to use JMock2 in JUnit 3

(can be used in other frameworks ~ see cheat sheet!)

# You could write the mock objects yourself, but you'd soon get bored of that…

# Simply use the MockObjectTestCase class

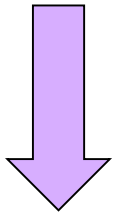
```
public class MessageGeneratorTest extends MockObjectTestCase
{

   …

}
```

# JMock creates the mock objects for you…

```
interface Subscriber {
    void receive(String message);
}
```



```
Subscriber subscriber = mock(Subscriber.class);
```

# JMock provides a DSL for defining expectations

```
import org.jmock.integration.junit3.MockObjectTestCase;
import org.jmock.Expectations;

public class MessageGeneratorTest extends MockObjectTestCase {

    public void notifiesSubscriberWhenSomethingUsefulHasHappend() {
        final Subscriber subscriber = mock(Subscriber.class);
        final MessageGenerator generator = new MessageGenerator(subscriber);

        // expectations
        checking(new Expectations() {{
            one(subscriber).receive("This is doing something useful");
        }});

        generator.doSomethingUseful();
    }
}
```

# You can ensure methods have **NOT** been called

```
public void testWillNotBidPriceGreaterThanMaximum() throws Exception {
    checking(new Expectations() {{
        ignoring (listener);
        never (lot).bid(with(any(Bid.class)));
    }});
    sniper.bidAccepted(lot, unbeatableBid);
}
```

```
public void bidAccepted(Lot lot, Bid amount) throws AuctionException {
    if (lot != lotToBidFor) return;

    if (amount.compareTo(maximumBid) <= 0) {
        lot.bid(Bid.min(maximumBid, amount.add(bidIncrement)));
    }
    else {
        listener.sniperFinished(this);
    }
}
```

# You can ensure the exact number of method calls

```
public void testWillAnnounceItHasFinishedIfPriceGoesAboveMaximum() {
    checking(new Expectations() {{
        exactly(1).of (listener).sniperFinished(sniper);
    }});

    sniper.bidAccepted(lot, unbeatableBid);
}
```

```
public void bidAccepted(Lot lot, Bid amount) throws AuctionException {
    if (lot != lotToBidFor) return;

    if (amount.compareTo(maximumBid) <= 0) {
        lot.bid(Bid.min(maximumBid, amount.add(bidIncrement)));
    }
    else {
        listener.sniperFinished(this);
    }
}
```

# You can simulate exceptions being thrown

```
public void testCatchesExceptionsAndReportsThemToErrorListener() throws Exception {
    final AuctionException exception = new AuctionException("test");

    checking(new Expectations() {{
        allowing (lot).bid(with(anyBid));
            will(throwException(exception));
        exactly(1).of (listener).sniperBidFailed(sniper, exception);
    }});

    sniper.bidAccepted(lot, beatableBid);
}
private final Matcher<Bid> anyBid = IsAnything.anything();
```

```
public void bidAccepted(Lot lot, Bid amount) throws AuctionException {
    if (lot != lotToBidFor) return;

    if (amount.compareTo(maximumBid) <= 0) {
        lot.bid(Bid.min(maximumBid, amount.add(bidIncrement)));
    }
    else {
        listener.sniperFinished(this);
    }
}
```

# *Tips* for using mocks effectively.

# Awkward tests indicate classes taking on too much responsibility and you need to **refactor**

- Big clues:
  - Too much setup in tests.
  - Irrelevant setup in tests.
  - One class having too many dependencies.
  - Brittle tests.

# Only Mock types you own

- Only mock interfaces that you can change.
- Instead:
  - Use mocks to design how your code will use infrastructure services.
  - Wrap a thin layer around the services.
- Stops **infrastructure** abstractions **leaking** into application layer.

# Specify only what's needed

- Over specification leads to brittle tests that fail when you make unrelated changes to the code. It also makes it hard to understand the tests.

- Testing at the wrong level does not describe what is actually being tested.
  - **Be explicit**. Even if it means more typing.

# Role based interfaces reduce coupling

- Use narrow interfaces to break objects down by the roles they play in interactions.

- As opposed to wider interfaces that describe all the features provided by a class.

# Remember that unit tests are not enough!

- Fine grained unit tests require coarse grained end to end tests.
- Just because all the parts of the engine are well designed and fit together, it doesn't mean the car actually drives.
- Mocks help drive out good design and quality of individual components. They do not give end to end confidence.

# Exercise 1

- The checkout can calculate the total price of a set of of scanned items. Write tests that scan some items and check the total.

- One way to implement the checkout:
  - Use a Money class to represent prices. (There's one on the CD).
  - Give the checkout two public methods:
  - *void scan(String barcode),* which finds the item that corresponds to the purchase and
  - *Money total(),* which calculates the total price of the items scanned.
  - Create a *Catalogue* interface that can look up a barcode and return the corresponding item.

# Exercise 2

- The supermarket wants to offer BOGOFs - "buy one, get one free".
- We had two products in our catalogue - bread and butter. We chose bread as the item on which we made a "Buy one get one free" offer.
- This means that a customer can buy two loaves of bread but only pay for one, buy four but only pay for two, and so forth.