SED

Mock Objects

Dr Robert Chatley - rbc@imperial.ac.uk

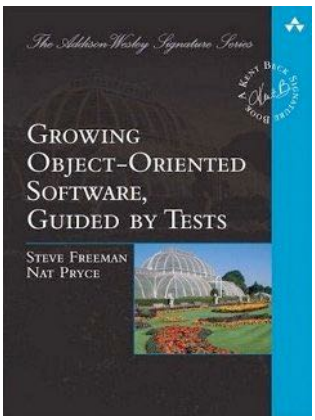
with thanks to Nat Pryce and Steve Freeman

@rchatley

This lecture builds on what we did in the previous week and describes a flavour of Test-Driven Development using a technique known as Mock Objects.

Further reading for this lecture (if you want to know more):

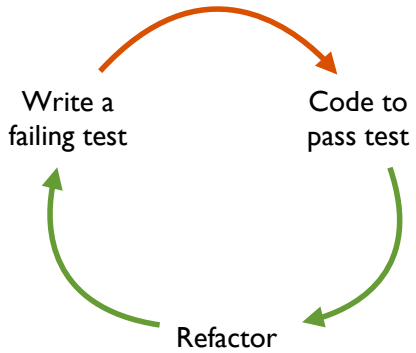
http://www.growing-object-oriented-software.com/
http://www.mockobjects.com/2009/09/brief-history-of-mock-objects.html
http://jmock.org/oopsla2004.pdf



A lot of the original work on Mock Objects was done by Nat Pryce and Steve Freeman, along with Tim Mackinnon, Joe Walnes and other developers based in London. Sometimes this style of TDD is known as "the London school".

Freeman and Pryce authored the book Growing Object Oriented Software Guided by Tests (commonly known as GOOS), which is an excellent book on the topic of TDD, and shows how to apply the techniques to a large project, and how the design of a system evolves over time as it is developed, with the tests helping to guide the design choices.

TDD Cycle



When following the TDD practice, we work in a cycle. We start by writing a test. We write the test first, before writing the implementation. This helps us to specify what we want the code we are about to write to do. How do we expect it will behave when it works properly? Once we have written a test, we watch it fail. This is expected, as we haven't implemented the feature yet. Then we write the simplest possible piece of code we can to make the test pass. After this we refactor our design to clean up, remove any duplication, improve clarity etc etc. Then we being the cycle again. When working with unit tests this cycle should be short, and provide us very rapid feedback about our code.

Alan Kay

The big idea is "messaging" [...] The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

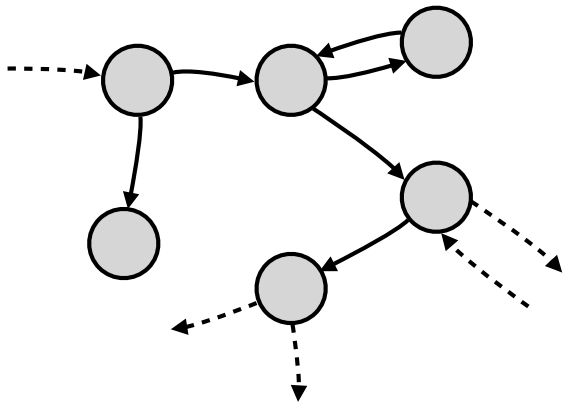
In this lecture we look at how we design (and test) the communication between objects in our systems. The interactions between objects is an important facet of the design in an object-oriented system. Alan Kay, a leading thinker on OOP, gave the quote on this slide, in which he maintains that the messages exchanged between objects are even more important than the internal implementation of any given object. The same objects can be connected together in many different ways to create different systems.

Photo by soapbeard

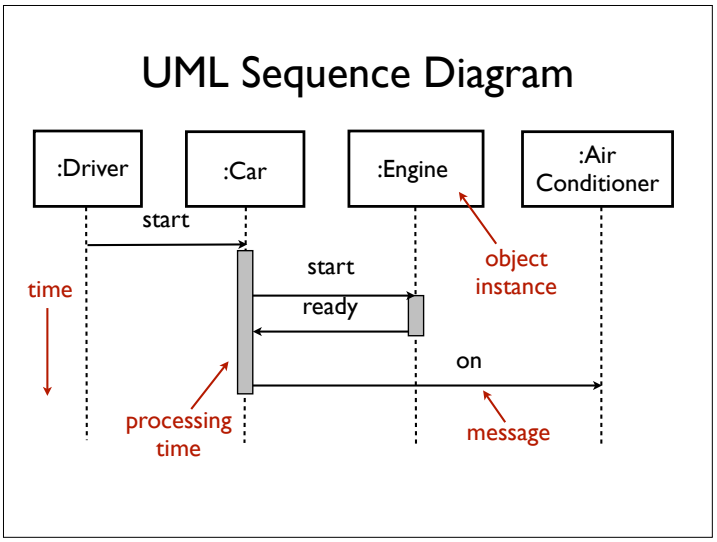
Sending a Message

We often talk of "calling a method", but the way we should really think of this inter-object communication is as "sending a message". We want to give another object a task to do something on our behalf. We send it a *message* informing it of some new state of the world, and expect that it will do something appropriate in response.

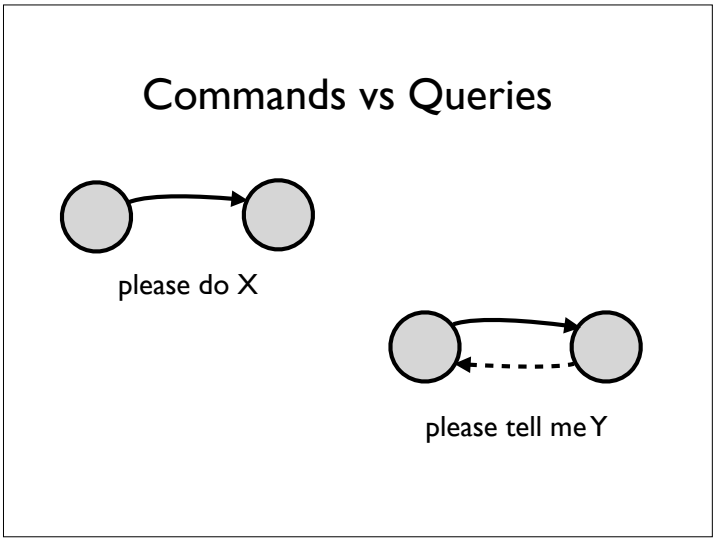
OOP - A Web of Objects



When we build a system using object-oriented programming, we assemble a web of objects with each representing a unit that participates in some way in the system. The structure in which these objects are connected, and the way that they communicate with each other, forms a key part of the design. When we test the system we often want to test the flow of messages, the interactions between objects, rather than the internals of a specific object.



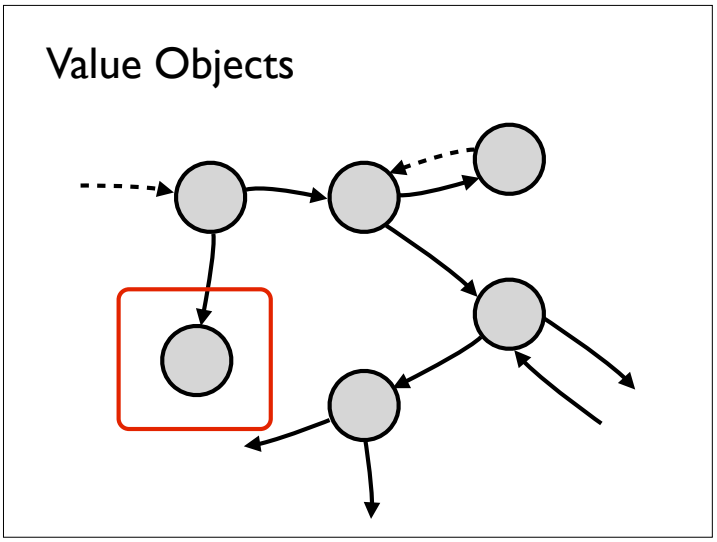
The exchange of messages between objects can be shown in a UML *sequence diagram*, sometimes known as a *Message Sequence Chart*. Where a class diagram shows structural relationships that are always true, and object diagrams show a snapshot of the state of a system at a particular point in time, a sequence diagram expresses a flow of messages exchanged between objects over time, to act out a particular scenario. It is the exchange of messages that makes things happen in the system. The sequence diagram does not reveal much about the state *inside* an object, it focusses on the communication *between* objects.



Messages (or *method calls* if you prefer) fall into two broad categories: commands and queries.

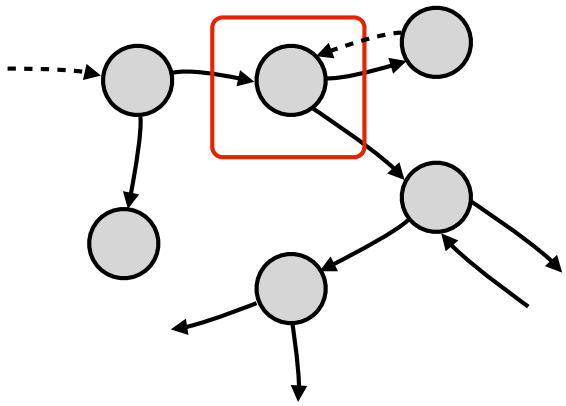
With commands, we tell another object to do something for us. We don't know or care how it does it, we delegate responsibility for the task to the other object. We do not normally get a return value. Commands often change the state of the invoked object (or another part of the program).

With queries, we ask another object to tell us a value so that *we* can use it. For example we may get the value from a textbox on screen, or the current speed of a car. Queries do return a value, but they should have no other side effects on the state of the invoked object.



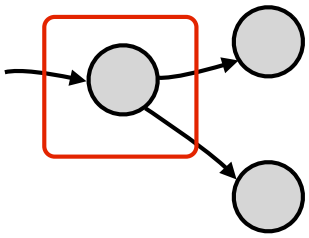
Some objects represent values, pieces of data in our system, and have little interesting behaviour. We may want to test these using a simple state based approach. If we create a weather report object, we might write a test that makes assertions about its celcius() and fahrenheit() methods, to check the calculations are done appropriately. This doesn't involve any interaction with any other objects in the graph. These *value objects* usually sit at the leaves of our object graph.

Tell Don't Ask



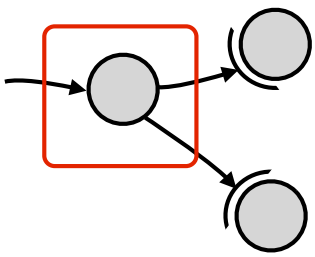
Things become more interesting when an object collaborates with its peers to form a more complex behaviour. A scenario may be carried out by a number of objects interacting, with a combination of queries and commands. With commands, objects send each other messages, requesting certain actions to be carried out, but they do not expect back a return value from those calls. The only calls that should return values are queries, that simply return data and do not cause any other change of state or interaction between objects. Following this style tends to lead to a better object-oriented design, and a design that is easier to test. We refer to this style as "Tell, Don't Ask".

Focus on a Single Object



When we write unit tests, we focus on one object (a single unit) at a time. We want to test its behaviours, and its interactions with its direct neighbours. The collaborators play a key part in the test. We ask ourselves the question, if this object carried out its behaviour correctly, who would know? If we are testing an EmailClient object, and we tell it to send an email, how would we know if that had happened? Well, it might be that the collaborating EmailServer object received a message. We test the object in terms of the messages that it sends and receives, not its internal state.

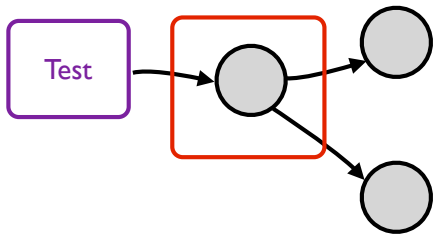
Collaborate Through Roles



In Java we use Interfaces to represent roles.

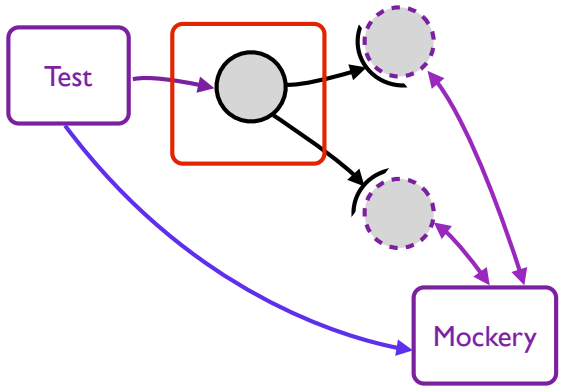
An object's collaborators play a particular role, and have a particular responsibility as seen by the calling object. In a good OO design, this means that we can swap in and out any object (perhaps with differing implementations) that can play this role. We do not have a tight coupling to a particular implementation. In Java (and other similar languages) we can use interfaces to represent these roles. This helps us to test just one object at a time, rather than having to glue together several objects in one test. It may also help us to fully develop and test one object, without having to implement those that it depends on.

Trigger "inward" arrows from test



When we write a test for an object that normally does its work by interacting with its neighbours, we look at two sorts of interactions: messages that go in to the object, triggering behaviour, and messages that come out, as a result. Any inbound messages, we can send from our test code to control the scenario. The test therefore replaces any object that would send a message to the object under test.

Sense using mock objects



We then need to detect any outward messages. We can do this by replacing any of the collaborators that would receive a message with a "test-double". A special implementation that we use just for the test. Mock Objects are one type of test-double, and are used in place of the real collaborators during the test. We could code up a fake implementation by hand, just implementing the interface in a new class, but by using a mock object framework, we can generate mock objects that implement any given interface using the test infrastructure. We will give examples using the jMock2 library. In jMock the component that creates mocks is known as the *mockery*.

```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.junit.Test;
import org.junit.runner.RunWith;

public class TestHeadChef {

    @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

    Order ROAST_CHICKEN = new Order("roast chicken");
    Order APPLE_TART = new Order("apple tart");

    Chef pastryChef = context.mock(Chef.class);
    HeadChef headChef = new HeadChef(pastryChef);

    @Test
    public void delegatesPuddingsToPastryChef() {

        context.checking(new Expectations() {{
            exactly(1).of(pastryChef).order(APPLE_TART);
        }});

        headChef.order(ROAST_CHICKEN, APPLE_TART);
    }
}
```

Test Setup

The Pastry Chef is the collaborator who should receive orders from the Head Chef if the system is working correctly, and we are testing the behaviour of the Head Chef. Therefore, we use the mockery (context) to create a mock Chef to act as the Pastry Chef, and use the real implementation of the Head Chef.

```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.junit.Test;
import org.junit.runner.RunWith;

public class TestHeadChef {

    @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

    Order ROAST_CHICKEN = new Order("roast chicken");
    Order APPLE_TART = new Order("apple tart");

    Chef pastryChef = context.mock(Chef.class);
    HeadChef headChef = new HeadChef(pastryChef);

    @Test
    public void delegatesPuddingsToPastryChef() {

        context.checking(new Expectations() {{
            exactly(1).of(pastryChef).order(APPLE_TART);
        }});

        headChef.order(ROAST_CHICKEN, APPLE_TART);
    }
}
```

Expectation

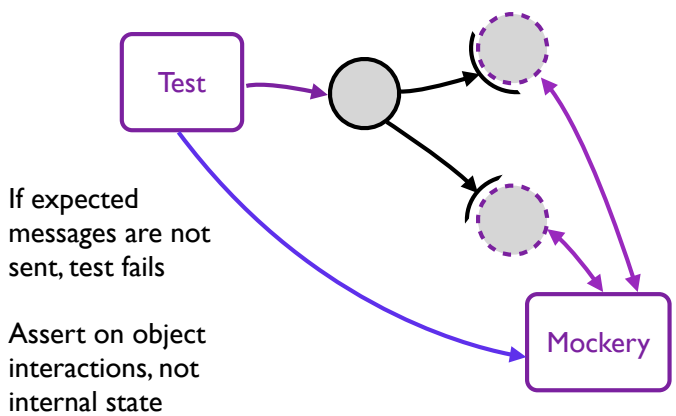
The checking block sets up our expectation of all the messages that should be received by the collaborators if everything goes to plan. So here we expect that the pastry chef will receive a message ordering an apple tart (a pudding).

```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.junit.Test;
import org.junit.runner.RunWith;

public class TestHeadChef {

    @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

    final Order ROAST_CHICKEN = new Order("roast chicken");
    final Order APPLE_TART = new Order("apple tart");

    Chef pastryChef = context.mock(Chef.class);
    HeadChef headChef = new HeadChef(pastryChef);

    @Test
    public void delegatesPuddingsToPastryChef() {

        context.checking(new Expectations() {{
            exactly(1).of(pastryChef).order(APPLE_TART);
        }});

        headChef.order(ROAST_CHICKEN, APPLE_TART);
    }
}
```

Trigger

The situation that we set up where we expect this to happen is where we trigger the Head Chef's behaviour by giving him an order for both a roast chicken (as a main dish) and an apple tart (as dessert).

Mockery verifies Interactions



If expected messages are not sent, test fails

Assert on object interactions, not internal state

The mockery verifies all of the expectations at the end of the test scenario (in JUnit this is done by specifying the mockery with an @Rule annotation). For each mock object, the mockery verifies that all of the messages that were expected to be received were in fact received, and that no other messages that were not expected were received. In this way we use the mock object test to make assertions about the an object's behaviour as expressed through its interaction with other objects, not by peering inside to inspect its internal state.