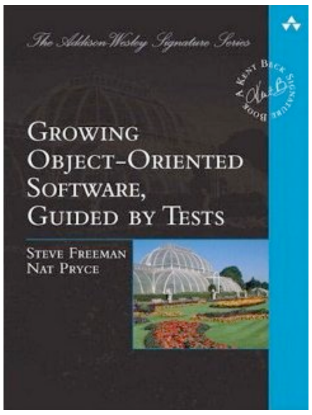
220

Mock Objects

Dr Robert Chatley - rbc@imperial.ac.uk

with thanks to Nat Pryce and Steve Freeman

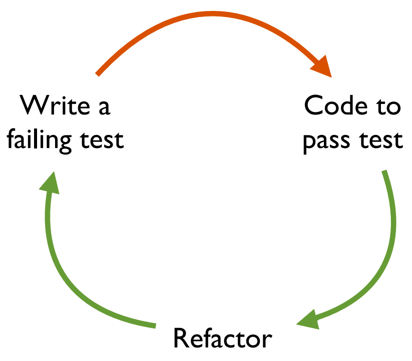
@rchatley **#doc220**



This lecture describes a flavour of TDD using a technique known as Mock Objects. A lot of the original work on Mock Objects was done by Nat Pryce and Steve Freeman.

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.

#doc220

In this lecture we look at how we design 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 above quote, in which he maintains that the messages exchanged between objects are even more important than the internal implementation of any given object.

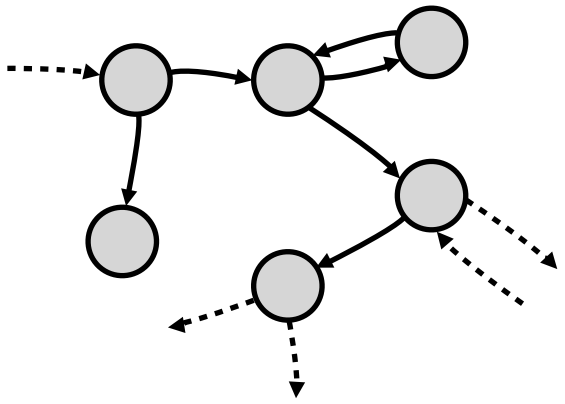
Photo by soapbeard

Sending a Message

#doc220

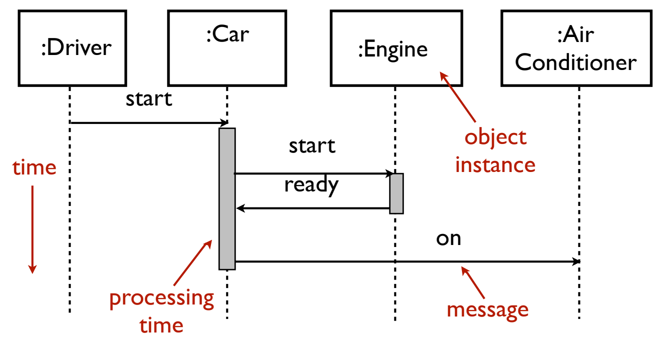
We often talk of "calling a method", but the way we should really think of this inter-object communication is as "sending a message". We want to give another object a task to do something on our behalf. We send it a message asking it to do something for us.

OOP - A Web of Objects



When we build a system using object-oriented programming, we assemble a web of objects with each representing a thing 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.

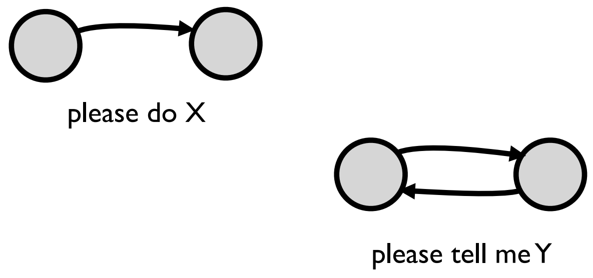
UML: sequence diagram



#doc220

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.

Commands vs Queries



#doc220

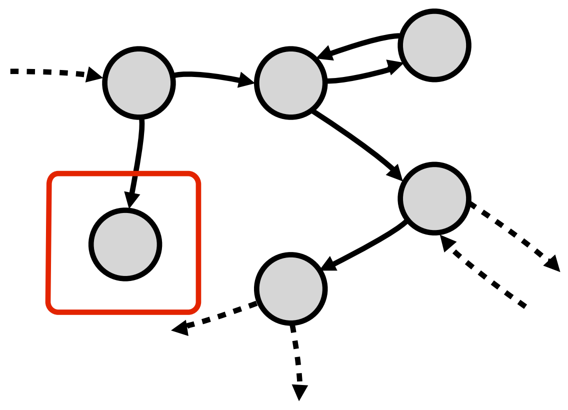
Method invocations fall into two broad categories: commands and queries.

With commands, we ask[1] 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 don't get a return value. Commands 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 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.

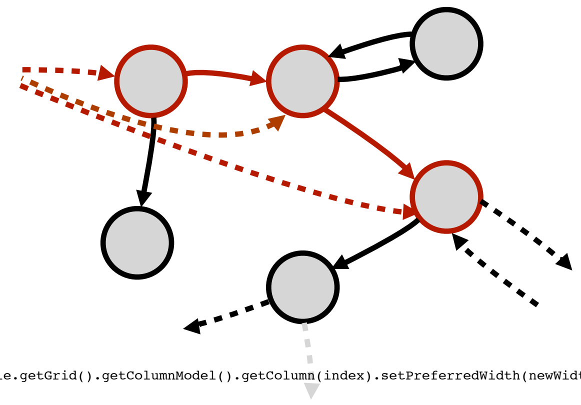
[1] Normally we don't say "please", unless we are programming in INTERCAL: http://en.wikipedia.org/wiki/INTERCAL

Value Objects



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 I construct a weather report object, I might write a test that makes assertions about its celcius() and farenheit() methods, to check the calculations are done appropriately. This doesn't involve any interaction with any other objects in the graph.

Law of Demeter



```
table.getGrid().getColumnModel().getColumn(index).setPreferredWidth(newWidth);
```

Things become more interesting when an object collaborates with its peers to form a more complex behaviour. In a good OO design, an object will only send messages to its direct neighbours in the graph. Having an implementation that depends on knowing the implementation of pieces of the system that are further away results in a tight coupling which can cause problems with testing and with maintenance. This is often revealed by long chains of method calls, asking other objects for a reference to their collaborators, so that the first object can traverse the object graph. This breaks the law of Demeter: "only talk to your immediate friends".

What is **Bad** Design?

Rigidity

Fragility

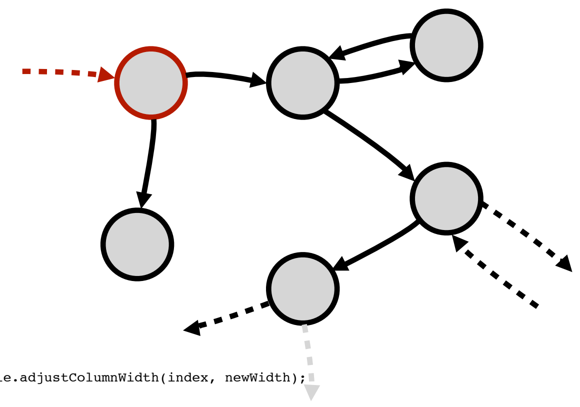
Immobility

#doc220

In this design the caller has in-depth knowledge of how its peers operate and are implemented. It does not delegate responsibility effectively. This code is highly fragile, and difficult to work with. It is also difficult to swap out the implementation of collaborating objects and replace them with different ones, and the caller is tightly coupled to objects deeper in the graph.

Fragility - where when we change one part, other parts break unexpectedly
Immobility - where it is hard to reuse elements of the code in other applications

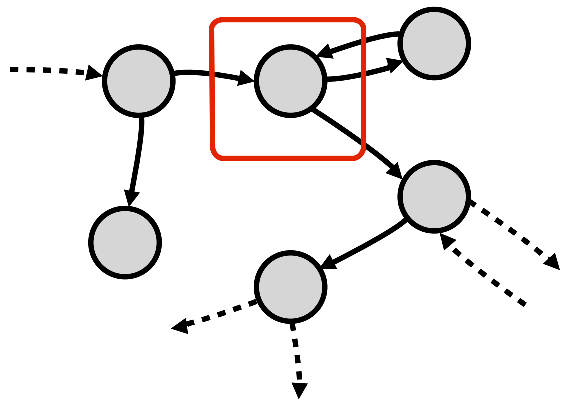
Law of Demeter



```
table.adjustColumnWidth(index, newWidth);
```

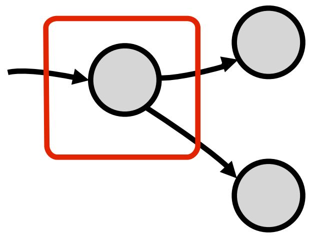
This design can be improved by just sending an intention revealing message to our nearest neighbour, and letting that object take responsibility for coordinating its own collaborators in order to carry out the requested action. This may result in further messages being passed to other objects, but it is not necessary for the first object to know or care about that, the coupling is looser, and the number of interactions that the first object needs to know about is reduced, making the design simpler.

Tell Don't Ask



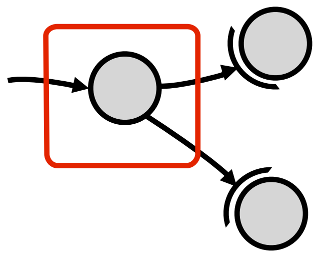
This is distilled in the design principle "tell, don't ask". Following this style, objects send each other messages, requesting certain actions to be carried out, but they do not expect back a return value from those calls. The only calls that should return values are queries, that simply return data 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.

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 ask 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.

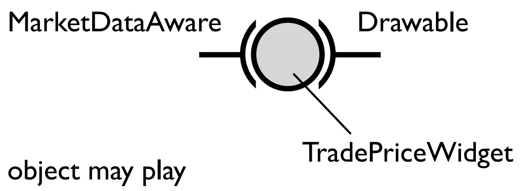
Collaborate Through Roles



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.

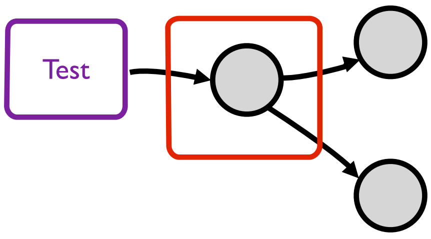
Roles



An object may play
different roles, depending
on its collaborators

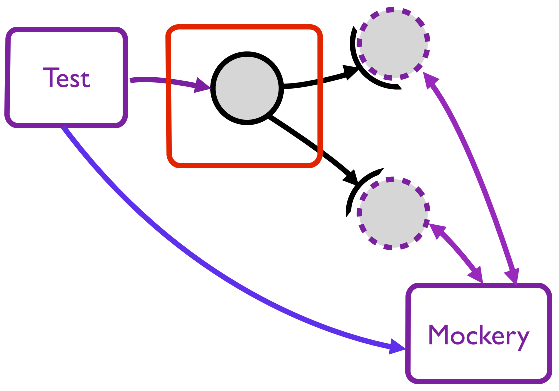
Many different objects may implement the same interface, or one object may implement several interfaces, but this is detail which is unknown to the caller. A GUI screen only needs to know that its components are drawable, but one of those components that is display updating stock prices also plays the role of a consumer of market data in its interaction with the market data provider. The market data provider however, has no interest in what the consumer does with the data, whether it is drawn on a screen or used for batch processing etc.

Trigger "inward" arrows from test

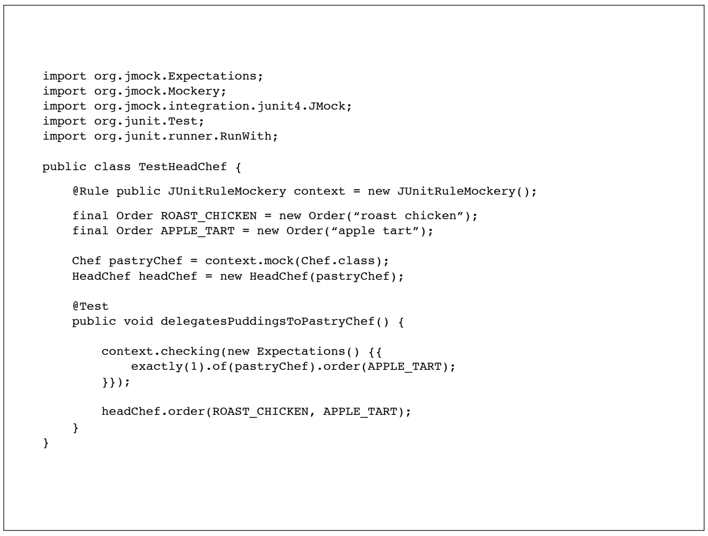


When we write a test for an object that normally exists linked in to a web of objects, interacting with its neighbours, we look at two sorts of interactions: messages that go in to the object, triggering behaviour, and messages that come out, as a result. Any inbound messages, we can send from our test 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

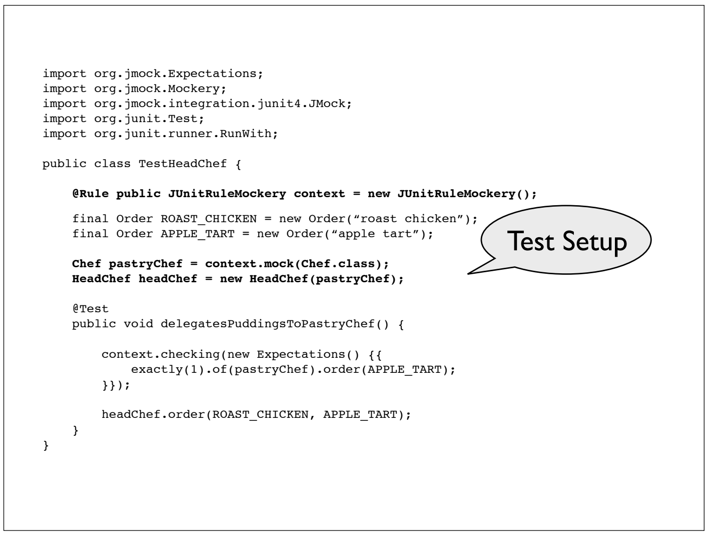


We then need to detect any outward messages. We can do this by replacing any of the collaborators that would receive a message with a test-double. A special implementation that we use just for the test. These are Mock Objects, and are used in place of the real collaborators during the test. By using a mock object framework, we can generate mock objects that implement any given interface using the test infrastructure. In jMock the component that creates mocks is known as the mockery.



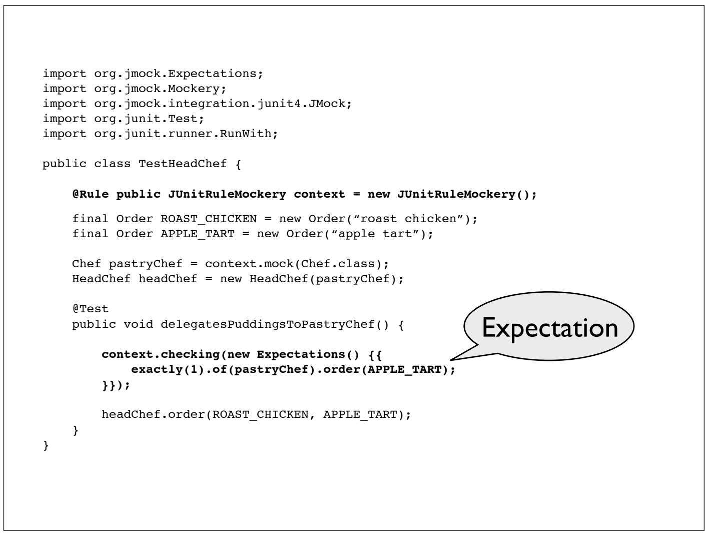
```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.junit.Test;
import org.junit.runner.RunWith;

public class TestHeadChef {

    @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

    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);
    }
}
```

Let’s look at a code example. Here we have a test for an object that represents a Head Chef in a restaurant kitchen. The Head Chef collaborates with the Pastry Chef to produce meals. This test verifies the behaviour of the Head Chef that he delegates all pudding orders to the Pastry Chef.



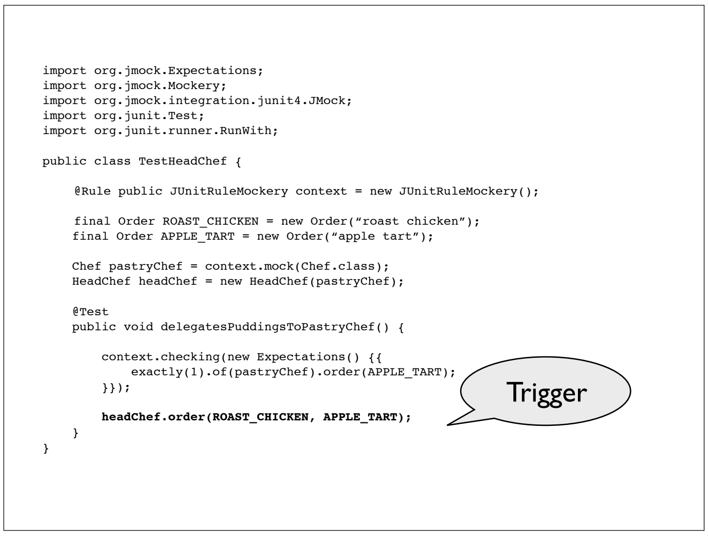
```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.junit.Test;
import org.junit.runner.RunWith;

public class TestHeadChef {

    @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

    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);
    }
}
```

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.



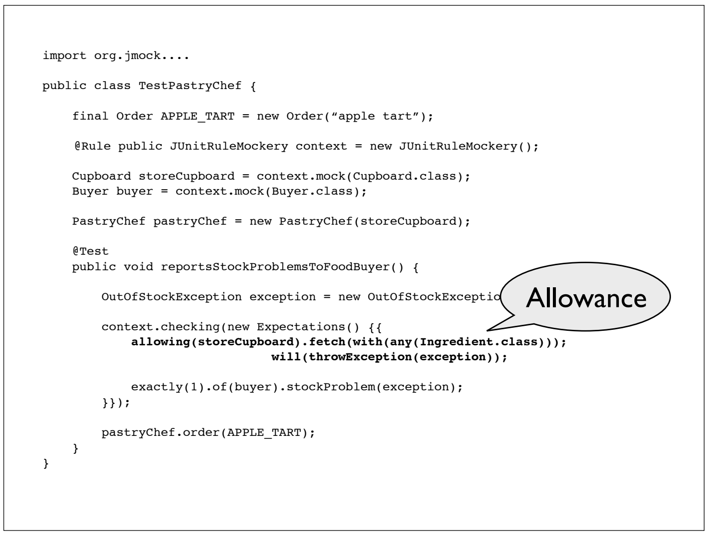
```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.junit.Test;
import org.junit.runner.RunWith;

public class TestHeadChef {

    @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

    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);
    }
}
```

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);
    }
}
```

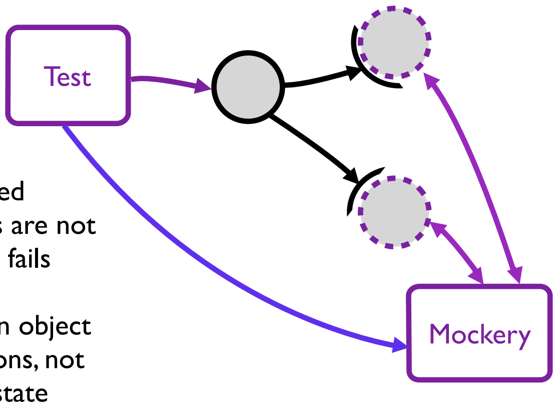
The situation that we set up where we expect this to happen is where we trigger the Head Chef's behaviour by giving him an order for both a roast chicken (not a pudding) and an apple tart.



```
import org.jmock....

public class TestPastryChef {

    final Order APPLE_TART = new Order("apple tart");

    @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

    Cupboard storeCupboard = context.mock(Cupboard.class);
    Buyer buyer = context.mock(Buyer.class);

    PastryChef pastryChef = new PastryChef(storeCupboard);

    @Test
    public void reportsStockProblemsToFoodBuyer() {

        OutOfStockException exception = new OutOfStockExceptio

        context.checking(new Expectations() {{
            allowing(storeCupboard).fetch(with(any(Ingredient.class)));
                                   will(throwException(exception));

            exactly(1).of(buyer).stockProblem(exception);
        }});

        pastryChef.order(APPLE_TART);
    }
}
```

In a different test, another thing we can test for is what if things go wrong. This time we test the pastry chef. What if the pastry chef is asked to produce an apple tart but he doesn't have all the ingredients. We need to set up the collaborating Cupboard object so that it does something particular when he calls it. We control what the mocks do when they interact with the object under test, so that we can act out a particular scenario. Here we say that if the cupboard's fetch() methods is call, with any ingredient as a parameter, it will throw an exception. Then we use an expectation on the buyer to check that the chef deals with this appropriately.

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. For each mock object, it 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.