RefTest

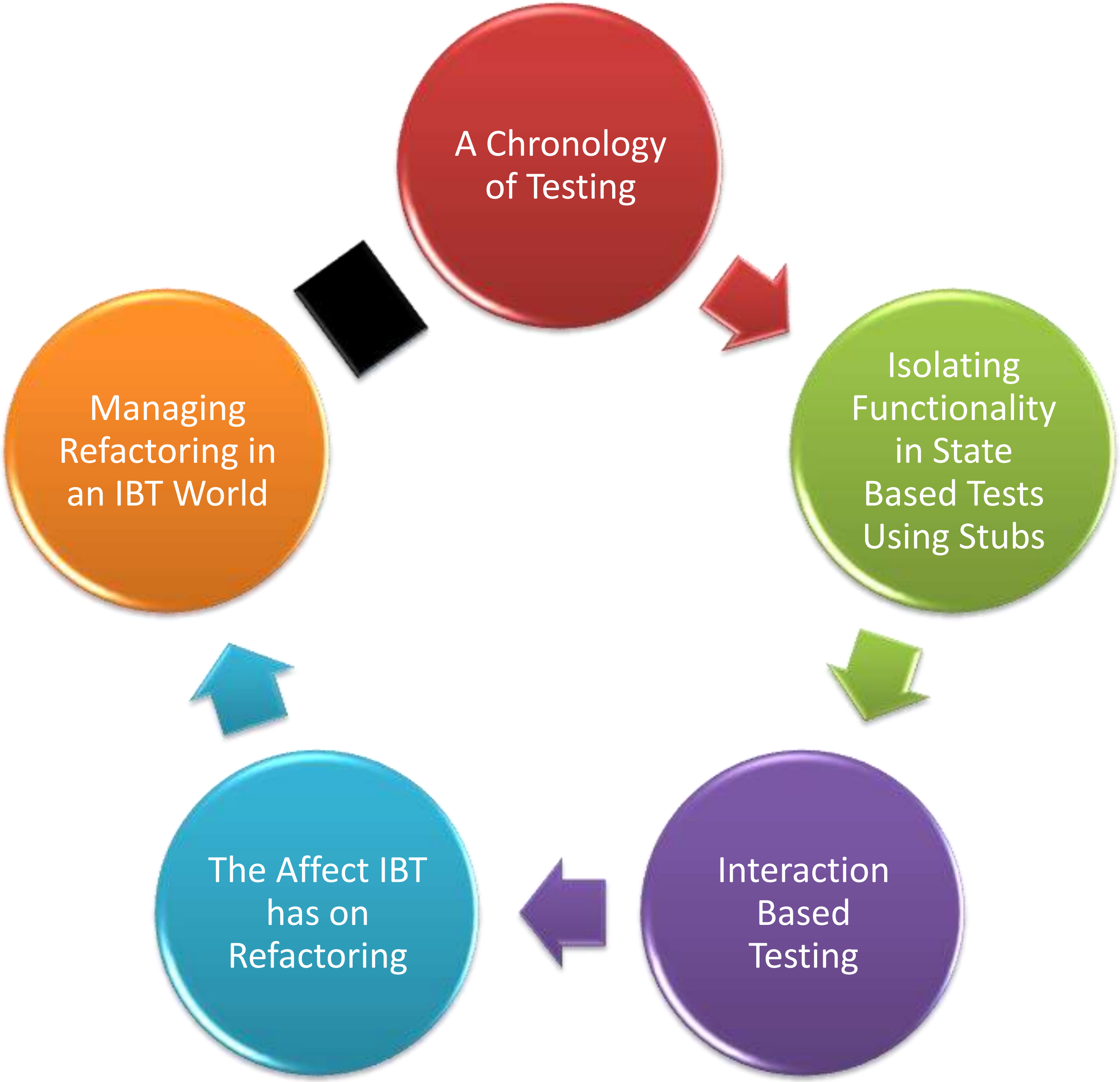
Refactoring Tested Code: Has Mocking Gone Wrong?

Ben Stopford
Royal Bank of Scotland

“I've always been a old fashioned classic TDDer and thus far I don't see any reason to change. I don't see any compelling benefits for mockist TDD, and am concerned about the consequences of coupling tests to implementation.”

This is about as far from the fence as Martin ever gets!!

Mocking, BDD, Interaction based testing. The belief is that they inhibit refactoring because of the coupling they add between test and source. This is what we will be looking at today.


A Chronology of Testing
Isolating Functionality in State Based Tests Using Stubs
Interaction Based Testing
The Affect IBT has on Refactoring
Managing Refactoring in an IBT World

One A Chronology

XP

- Introduced in late 1990's
- Automated testing features heavily in the 12 key practices
- Designed to reduce the risk of change
- Practices that seem somewhat counterintuitive, but work

Test First

- Write tests before writing code
- Change in programming process
- Focuses intention on required functionality

Test Driven

- Evolution of Test First
- Work in a tight loop: Test-Code-Refactor
- Demarcate test areas with stubs: *So you break off no more than you can chew.*

Interaction Based Testing

- Characterised by the use of Mocking Frameworks
- Contrasting technique to State Based testing
- Interactions between collaborating classes are tested not the class' final state.

Nomenclature

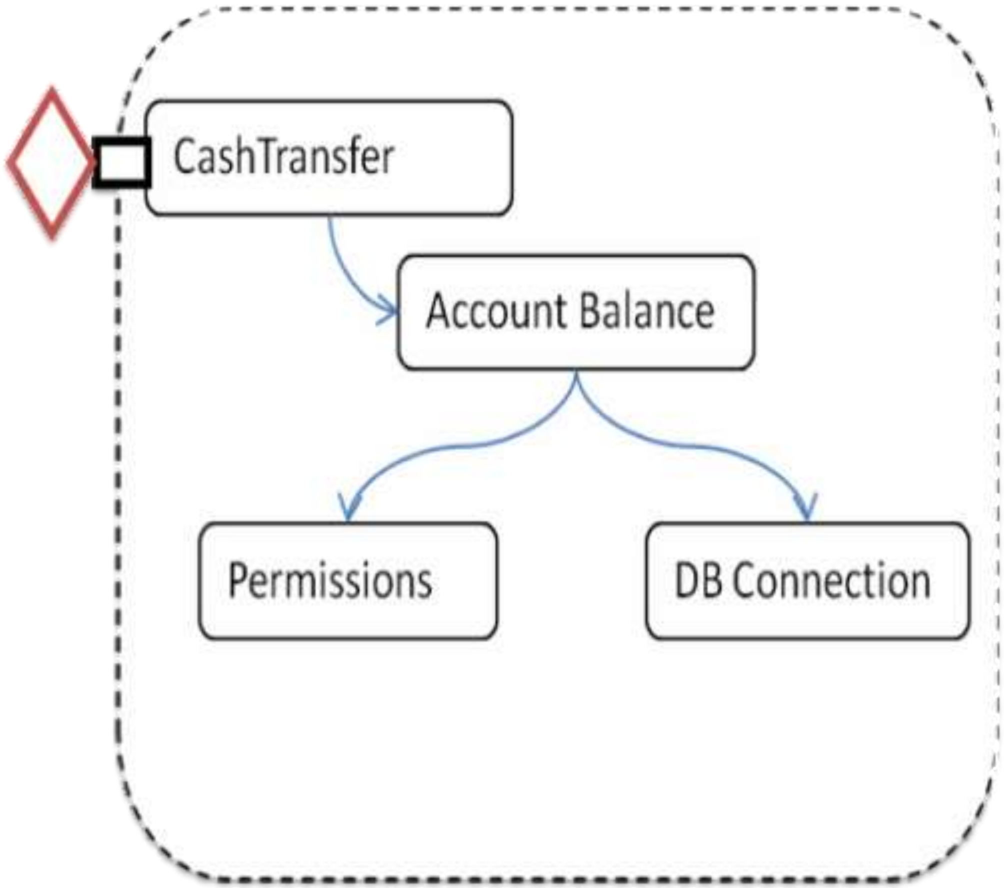
Mock:	Objects pre-programmed with expectations which form a specification of the calls they are expected to receive.
Stubs:	Provide canned answers to calls made during the test.
Dummy:	Objects are passed around but never actually used. Usually they are just used to fill parameter lists.

Two

Isolating the Functionality Under Test

State Based Testing

Testing the CashTransfer object involves testing the whole dependency tree.



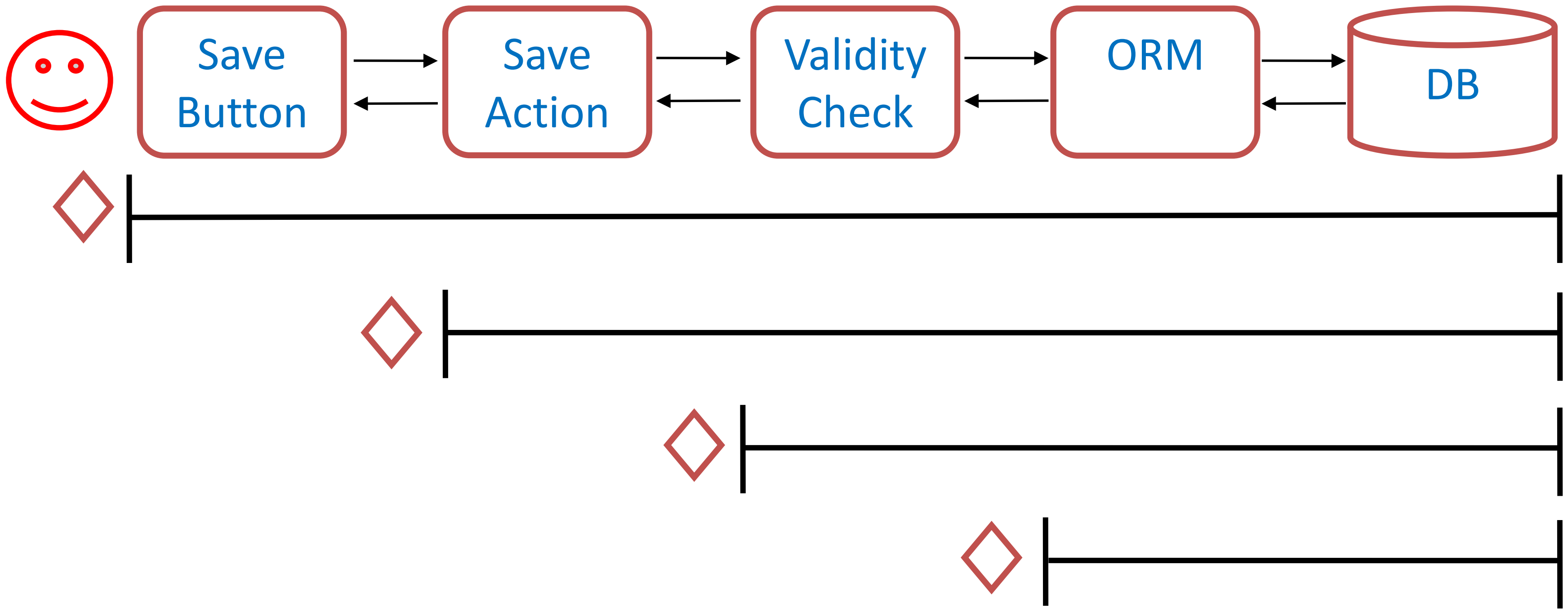
State Based Testing

```
@Test void shouldMoveCashToNewAccount(){
    Transfer tran = new CashTransfer(5,…);
    tran.execute(newAcc);
    assertEqual(expected, newAcc.balance());
}
```

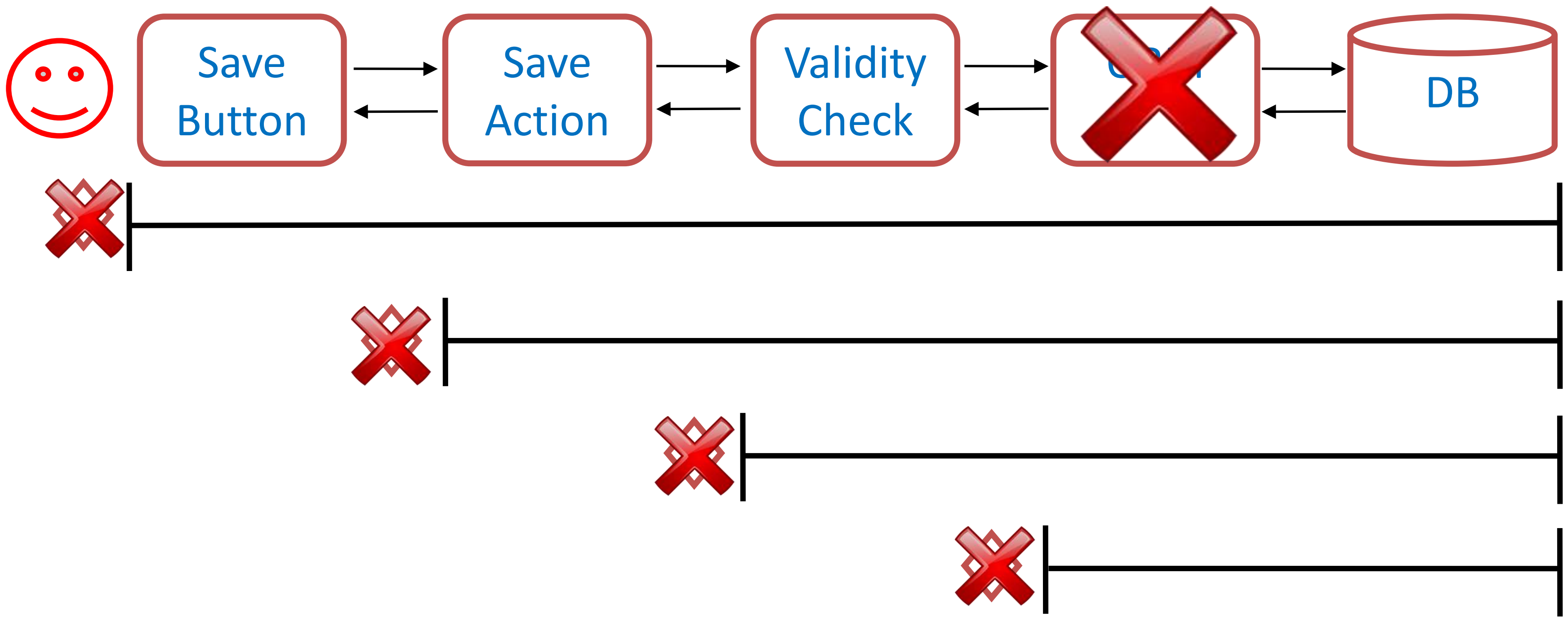
Test body that Exercises the Class under test

State based assertion

State Based Tests Often Overlap



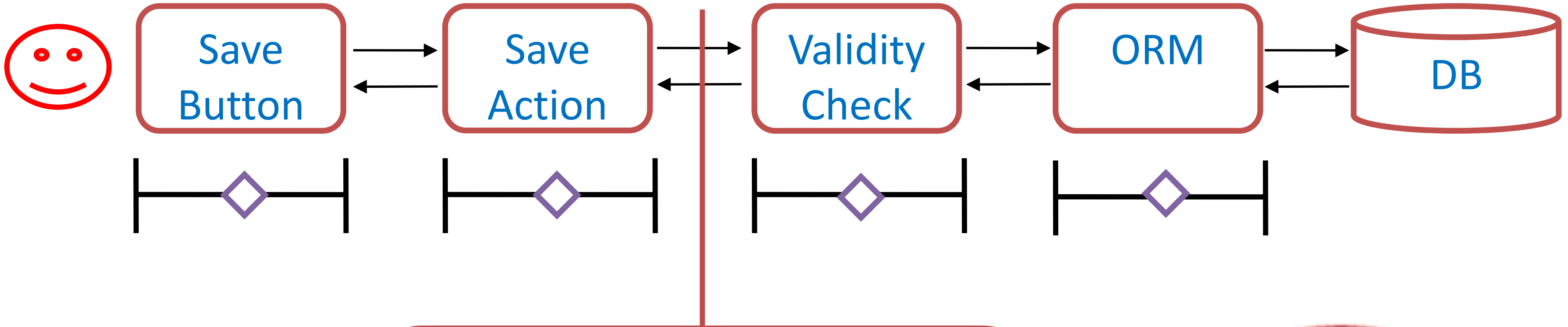
Failure in the ORM causes all tests to fail



Key Point

Tests that do not 'isolate' the code being tested will likely overlap. This makes it hard to diagnose the source of a break.

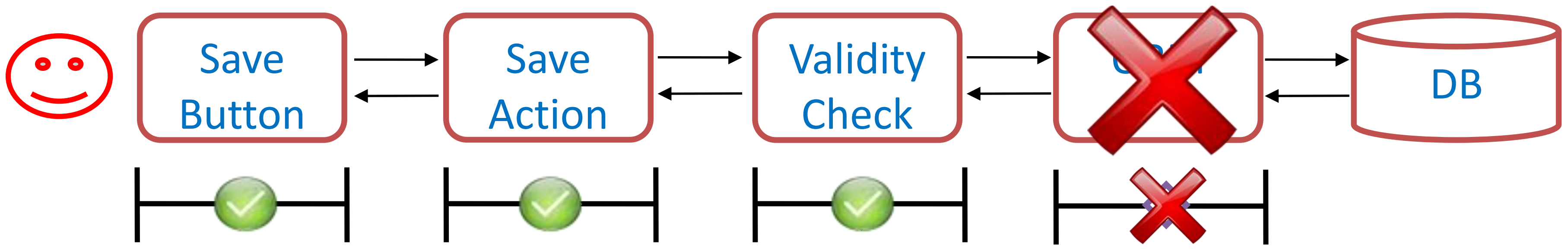
Isolate the area under test with stubbed interfaces that provide fixed behaviour.



```
class StubValididityCheck{
    boolean valid(){
        return true;
    }
}
```

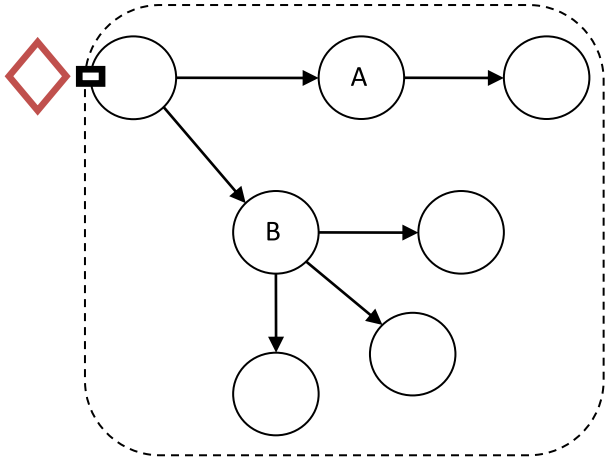
Stub n' State approach

Tests now break in isolation

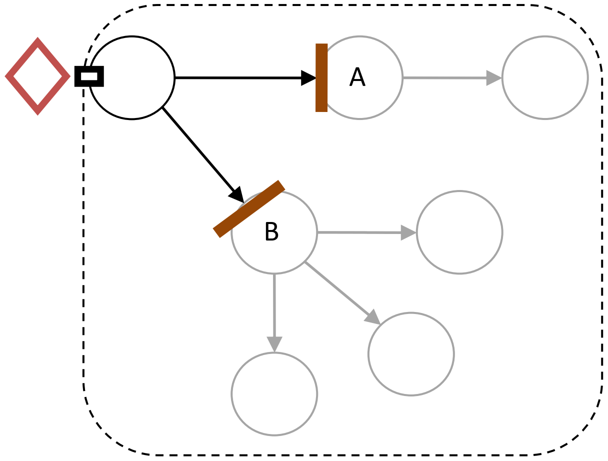


Stubs isolate sections of the Object Graph because they have no real behaviour. They provide pre-canned answers.

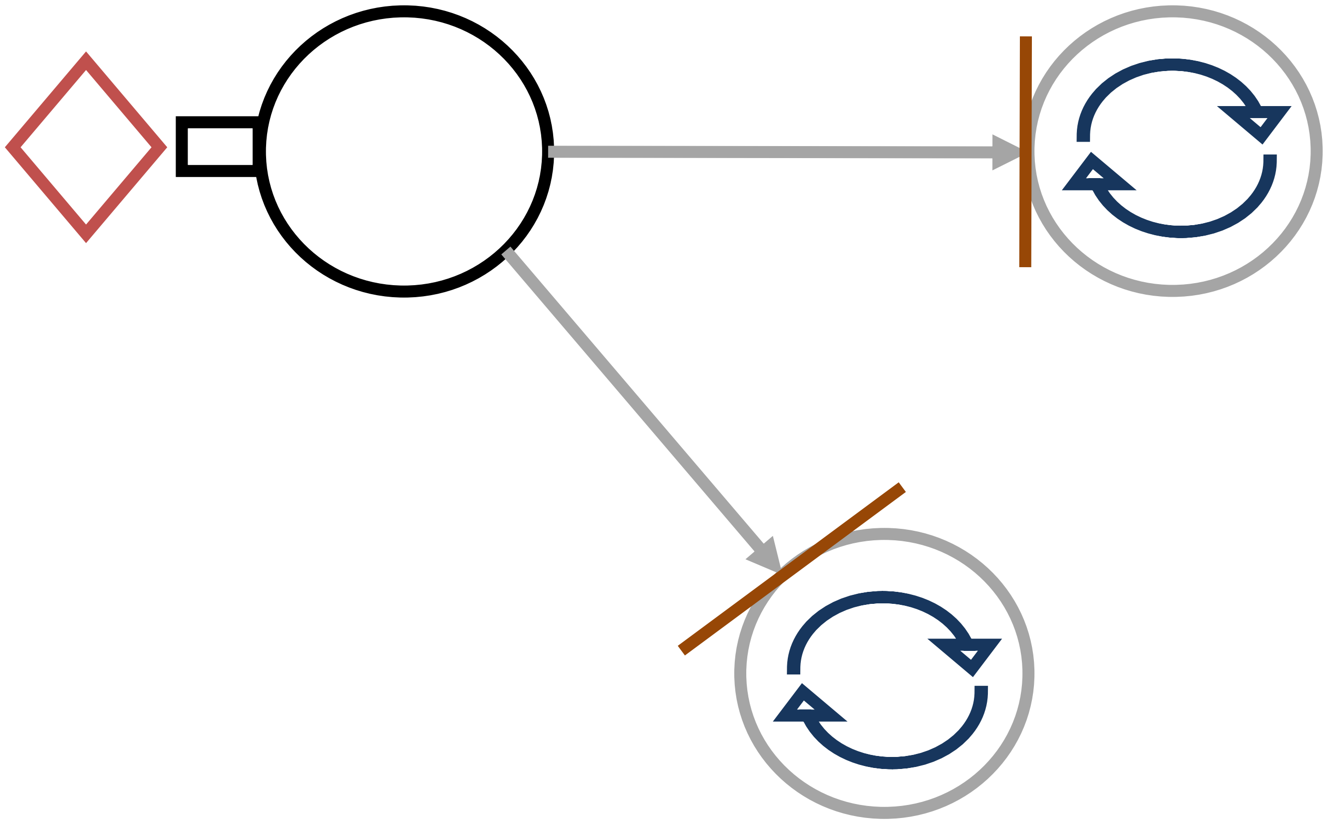
```
o = new ruleComposite(new A(…), new B(…));
result = o.run(user);
assertEqual(Result.VALID, result);
```



```
o = new ObjectUnderTest(stubA, stubB);
result = o.doSomething();
assertEqual(expected, result);
```



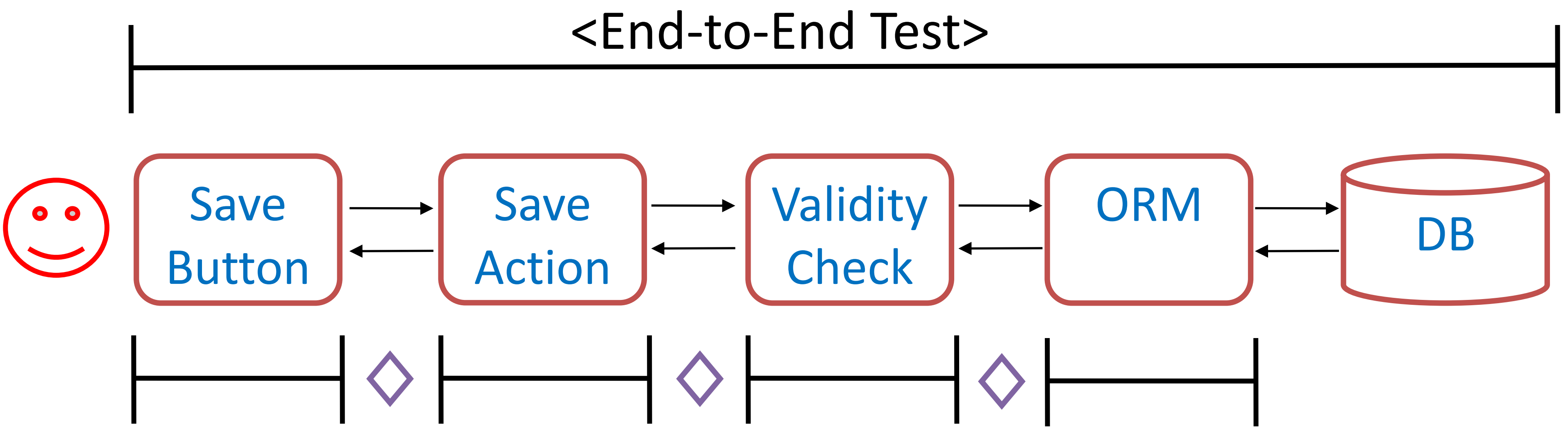
Each test is isolated from change in other classes



Key Point

To unit test properly you need to isolate the area of the code under test.

Conjunction of all tests are also tested End-to-End



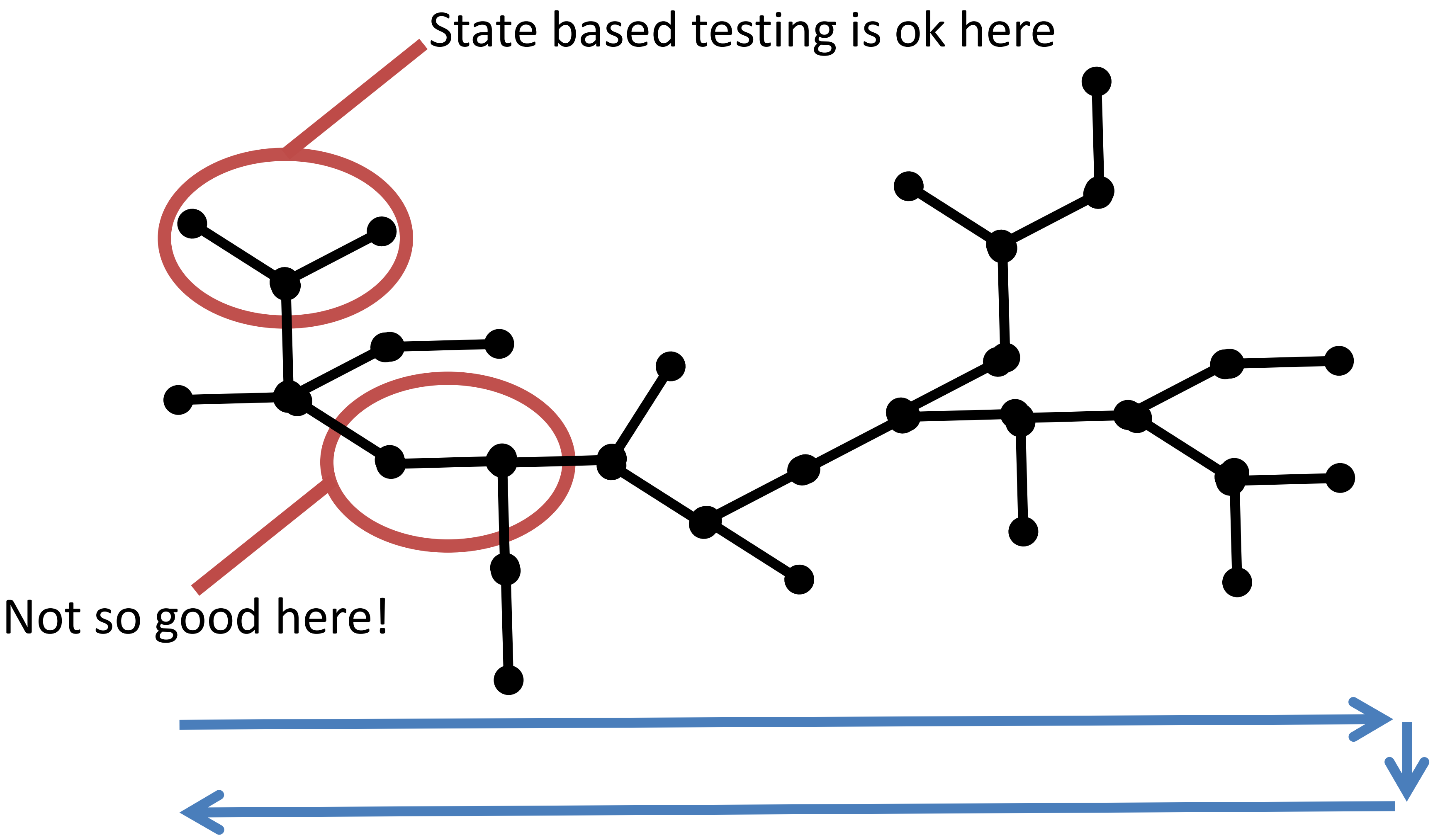
... we need to make sure all the dots tie up!

(else our wrapped up units might start to diverge)

Three

Interaction Based Testing

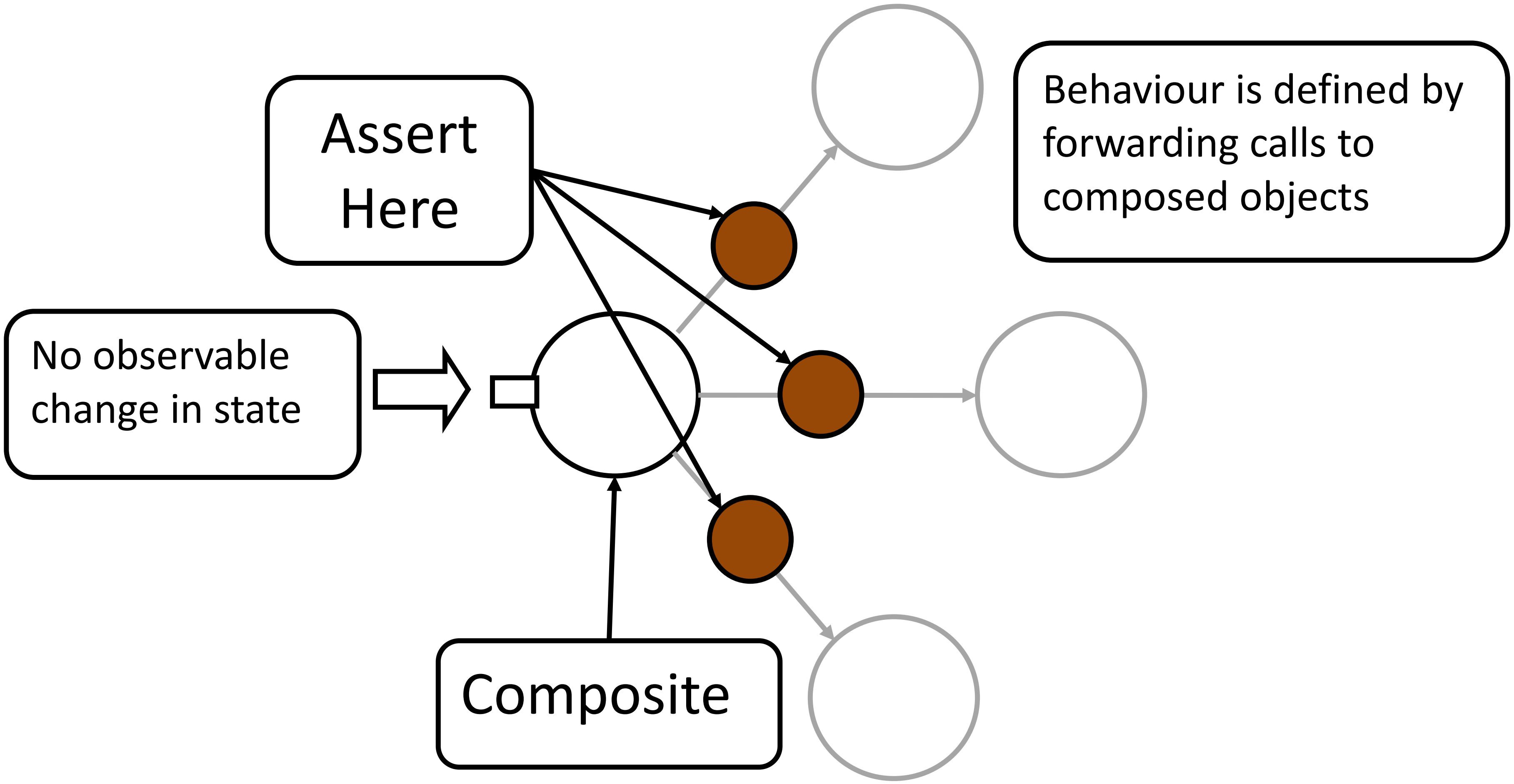
A better model is the 'Budding' model



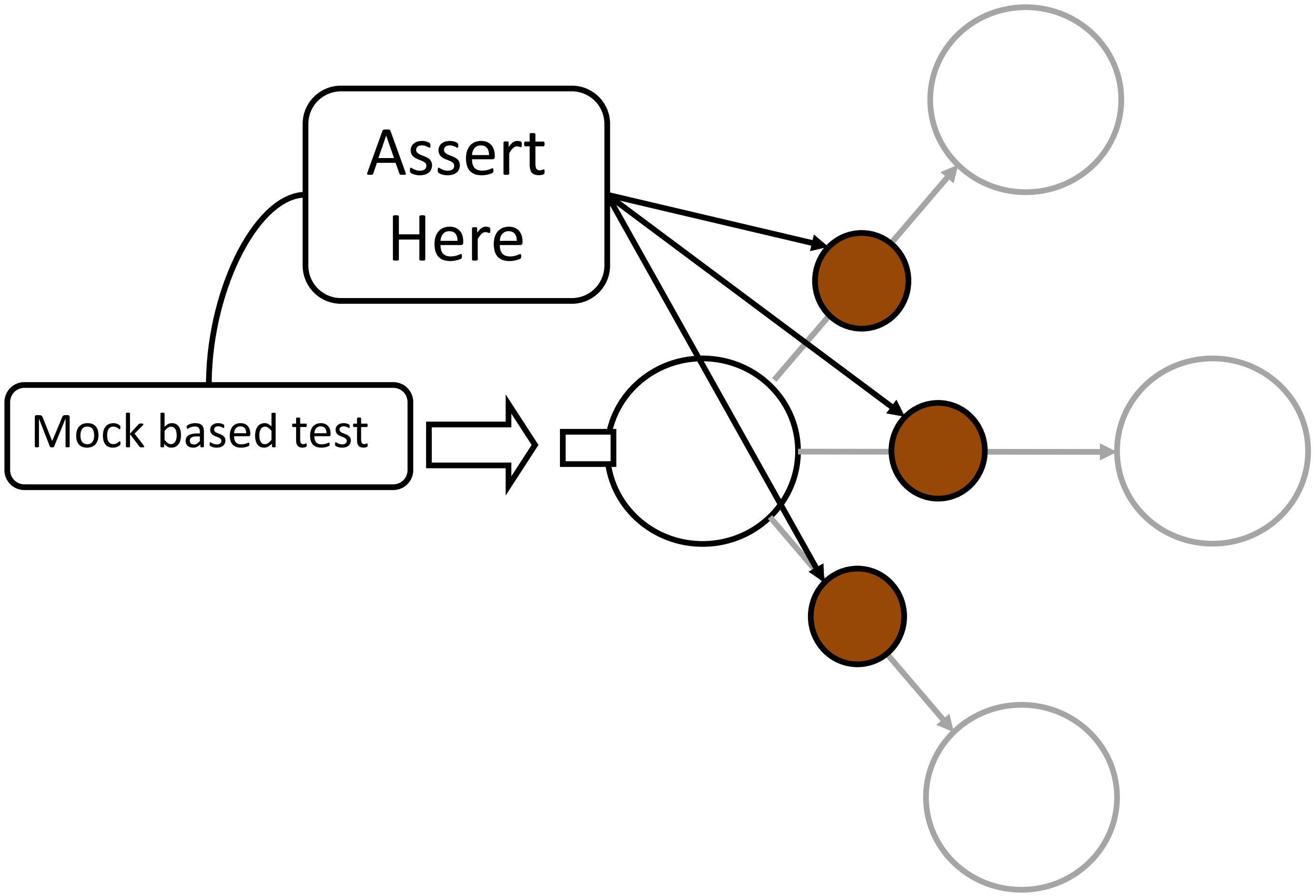
Problem: Classes that don’t change observable state

- Marshellers
- Proxies
- Caches

A Composite Object

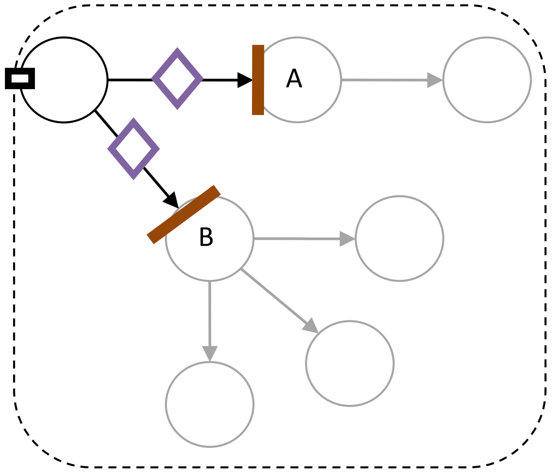

Assert Here
Behaviour is defined by forwarding calls to composed objects
No observable change in state
Composite

Mocking frameworks automate the testing of the interactions between classes



Interaction Based Testing

```
o = new RuleComposite(mockRuleA, mockRuleB);
check(new Expectation(){
        oneof(mockRuleA.run(user));
        oneof(mockRuleB.run(user)).throws(getExep());
};
o.run(obj);
```



Rather than testing changes in state, the interaction between objects are asserted.

Key Point

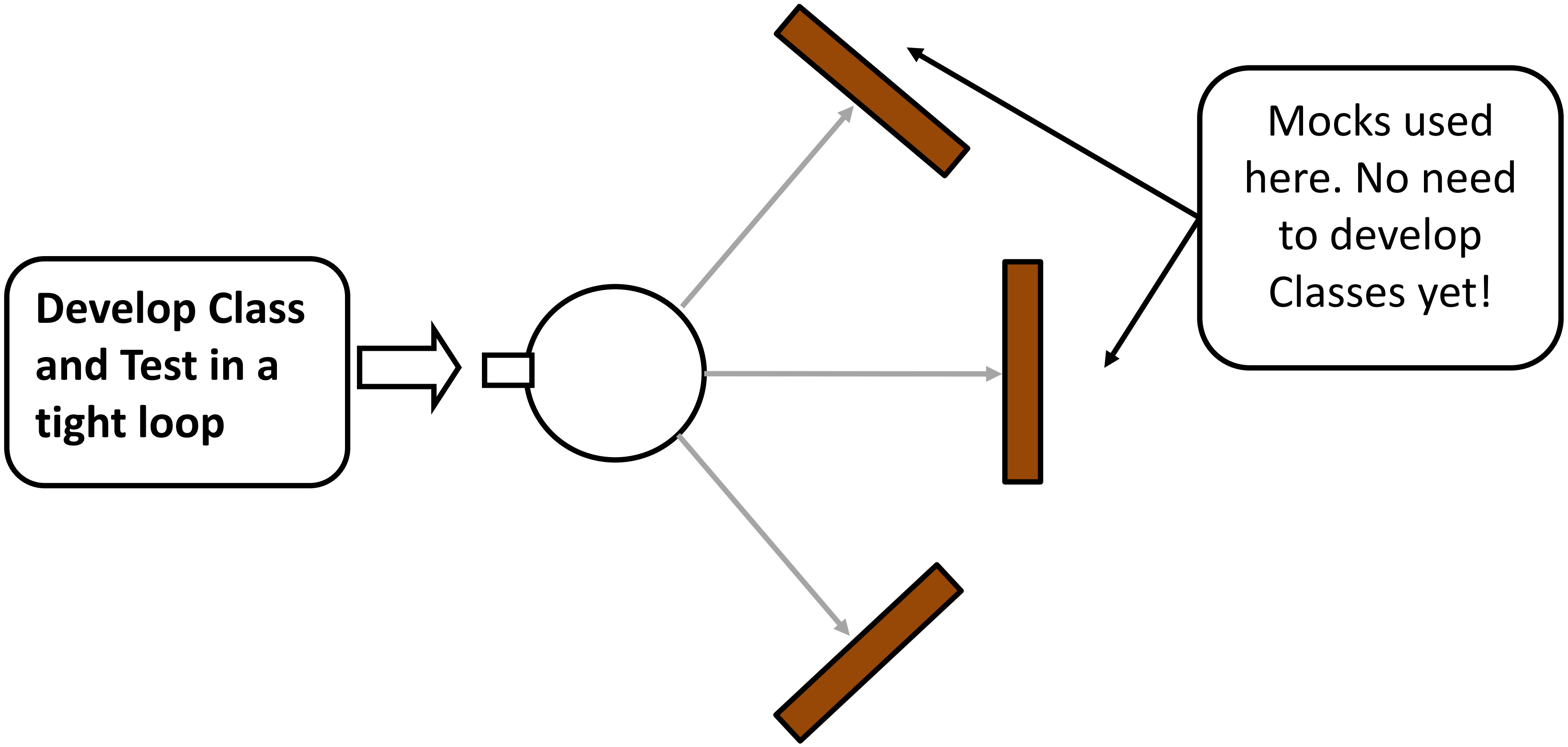
Mocked objects add additional coupling between test and source as mocks assert how an object behaves towards its collaborators, not just how it changes state or what it returns.

How does this relate to the State based testing with Stubs?

Both allow us to isolate the code under test.

In practice Mocking leads to a very different development process largely because you tend to mock at a class by class level, teasing out roles for collaborating classes.

Mocking facilitates a different development process.


Develop Class and Test in a tight loop
Mocks used here. No need to develop Classes yet!

Collaborators do not need to be implemented for the test to pass. They are simply mocked.

The ‘Mockist’ approach is different

- If the class under test needs to collaborate with another class then a mock is used.
- This teases out roles a class requires from its collaborators (similar to Design by Contract).
- All classes are tested in complete isolation, demarcated by mocked objects.
- The interactions between classes form the primary driver for assertions rather than changes in state.

Four

Refactoring Interaction Tested Code

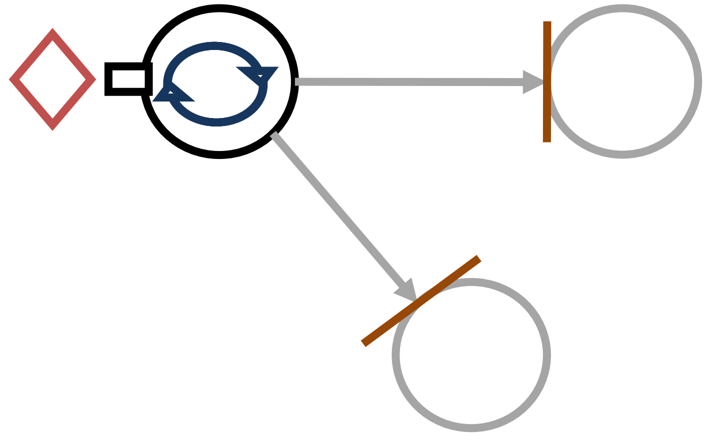
Mocking Increases Coupling

- Using Interaction Based Testing increases the coupling between test and source code.
- Tests assert on whether a method is called, with what arguments and how many times.
- This breaks encapsulation as the internals of how the class interacts with it’s collaborators is exposed.
- Thus, if refactoring changes the way an class interacts with collaborators tests may fail.

Increased Coupling makes refactoring harder

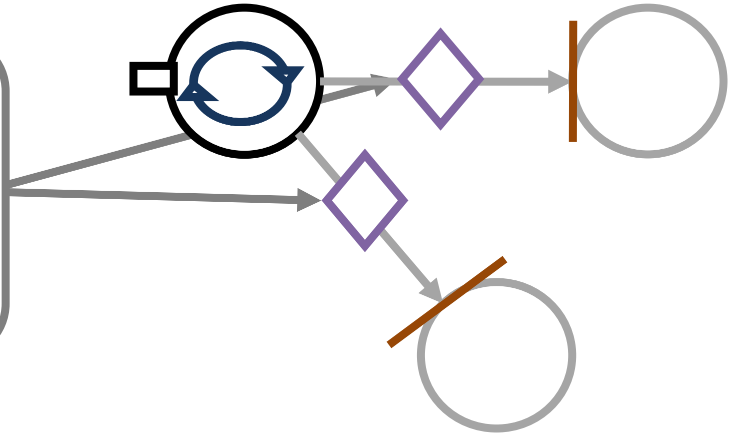
Stub n' State

Refactoring should not change the behavior of a class. Hence state based tests should not break.



Interaction Based Testing

Refactoring a class may change the way it communicates with collaborating classes, breaking interaction based tests.



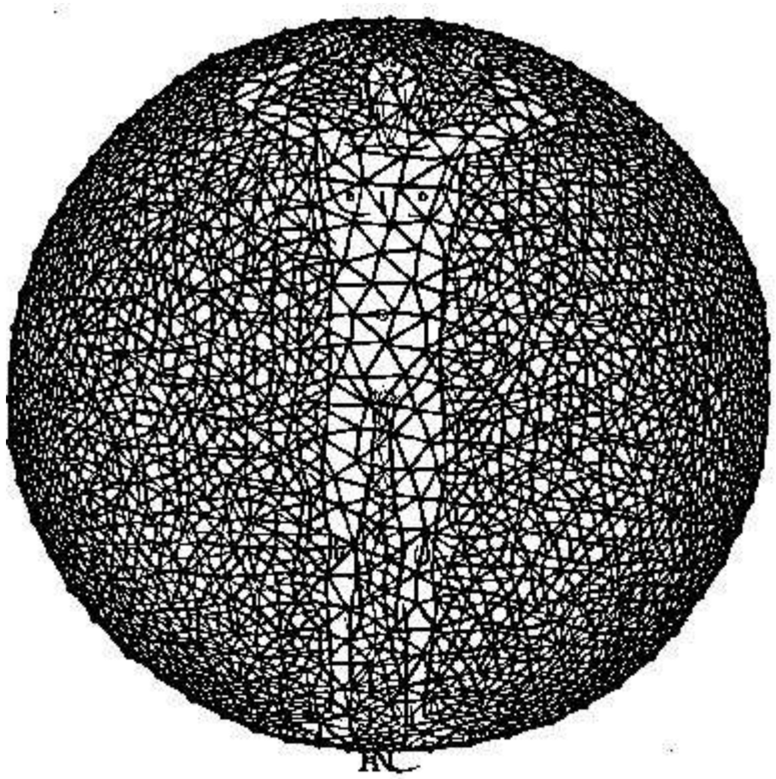
Interaction Based Testing is harder. There is more metaphorical rope.

Most horror stories associated with Interaction Based Tests are a result of excessive coupling produced by poor implementation

How Mocking Can Add Unnecessary Coupling?

Mocking Value Objects: An orange is always an orange

How Mocking Can Add Unnecessary Coupling

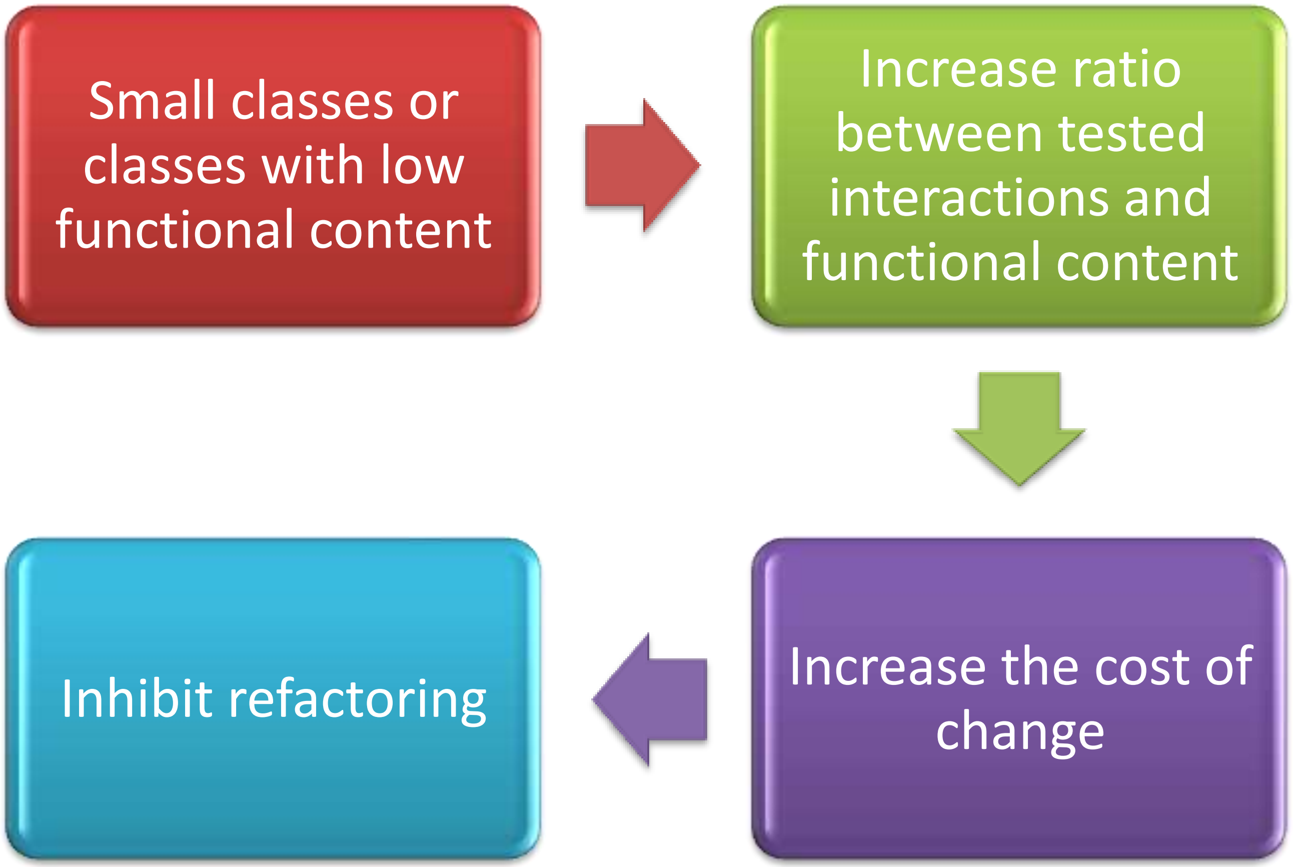


Complex Constructors: There’s a test trying to get out

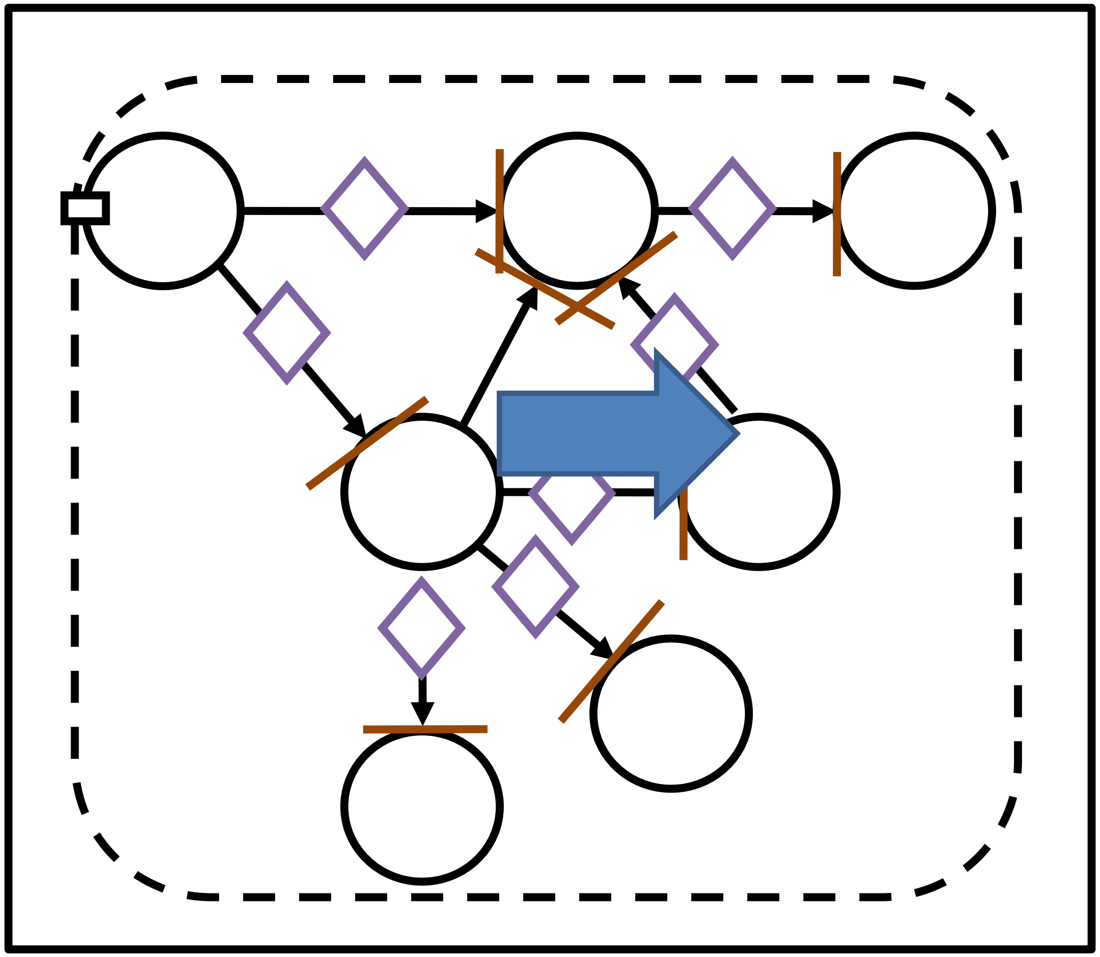
Key Point

If good OO principals are not rigorously applied mock driven tests will may become overly complex. The tests are very sensitive to the design

Smaller classes / classes with little functionality increase the support burden needlessly.


Small classes or classes with low functional content
Increase ratio between tested interactions and functional content
Increase the cost of change
Inhibit refactoring

For Example the Extract Class refactor



Increases the number of interaction points whilst holding the functional content constant

Key Point

The Mockist’s needs to be maintain a balance between the number of interaction based tests and the corresponding functional content.

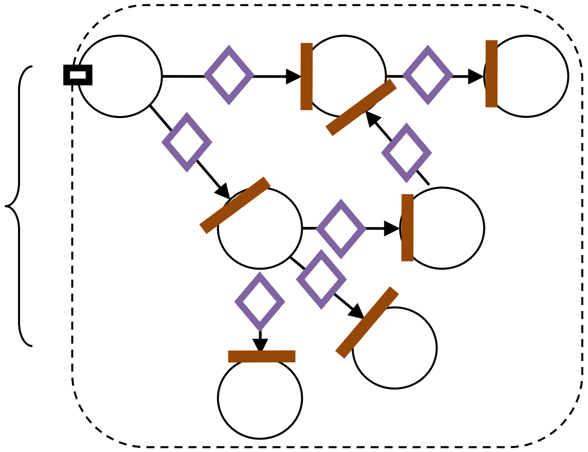
Five

Managing Refactoring in a Test Driven World

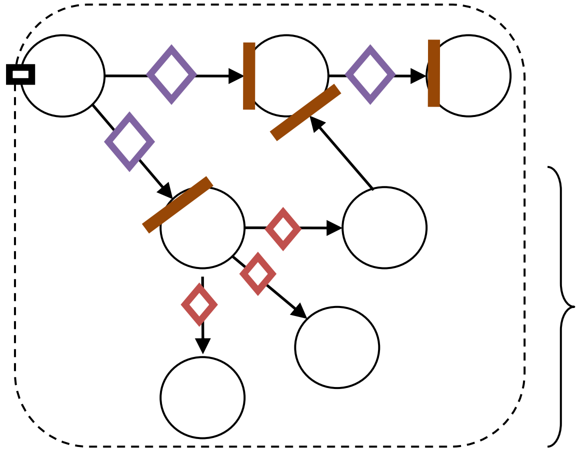
Best Practices for Interaction Based Testing

- Don’t mock behaviours that are not relevant to the test (stub them).
- Avoid complex constructors, static initialisers or other setup code that crosscuts multiple execution paths.
- Only mock classes under your control.
- Don’t mock value objects.
- You don’t need to mock everything.

You don’t need to mock everything!



All interactions are mocked



A group of collaborating objects are isolated with mocks. Internally state based testing is used.

Demarcating groups of objects with mocks and using state based testing internally good practice.

So What Do We Have?

- Unit tests requires isolating the code under test to ensure we get accurate feedback on test failures => Mocks or Stubs.
- Both mocks and stubs add a small maintenance burden to the project as they must be kept up to date.
- IBT facilitates a different method of doing TDD. It allows you to drive out code for a class without developing its collaborators.
- Interaction Based Testing with Mocking Frameworks introduces tighter couplings between test and source. This makes refactoring more difficult.

So What Do We Have?

- ...But most horror stories resulting from the use of IBTs arise from coupling introduced by poor implementation, not an intrinsic property of the process.
- Some of these problems have been highlighted here (complex constructor, mocking value objects etc)
- The Mockist approach of applying IBT at a class by class level magnifies poor design.
- The Mockist approach encourages isolation at a class level but this is not mandatory. Mixing IBT and state based testing provides a balanced approach.

Finally, my personal thoughts…

- Mockist TDD is a pleasant process to follow.
- I like to start with the stub n' state approach (using a mocking framework to create the stubs), then add expectations that relate to the particular test.
- I also tend ***not*** to mock interactions between classes I consider to be closely coupled, I tend to favour state based tests with the demarcation of mocks surrounding the group.
- To me the two approaches are not mutually exclusive, you have to have some demarcation. The trick is to know how much to test in isolation and when to assert expectations over stubs