JidokaQ [ji-do-ka-ku]

Unit testing, Integration testing
TDD (Test Driven Development)
Mocking

by Micael Andersson

TDD
(Exercise Stack)

Presentation provided by
Combitech JidokaQ

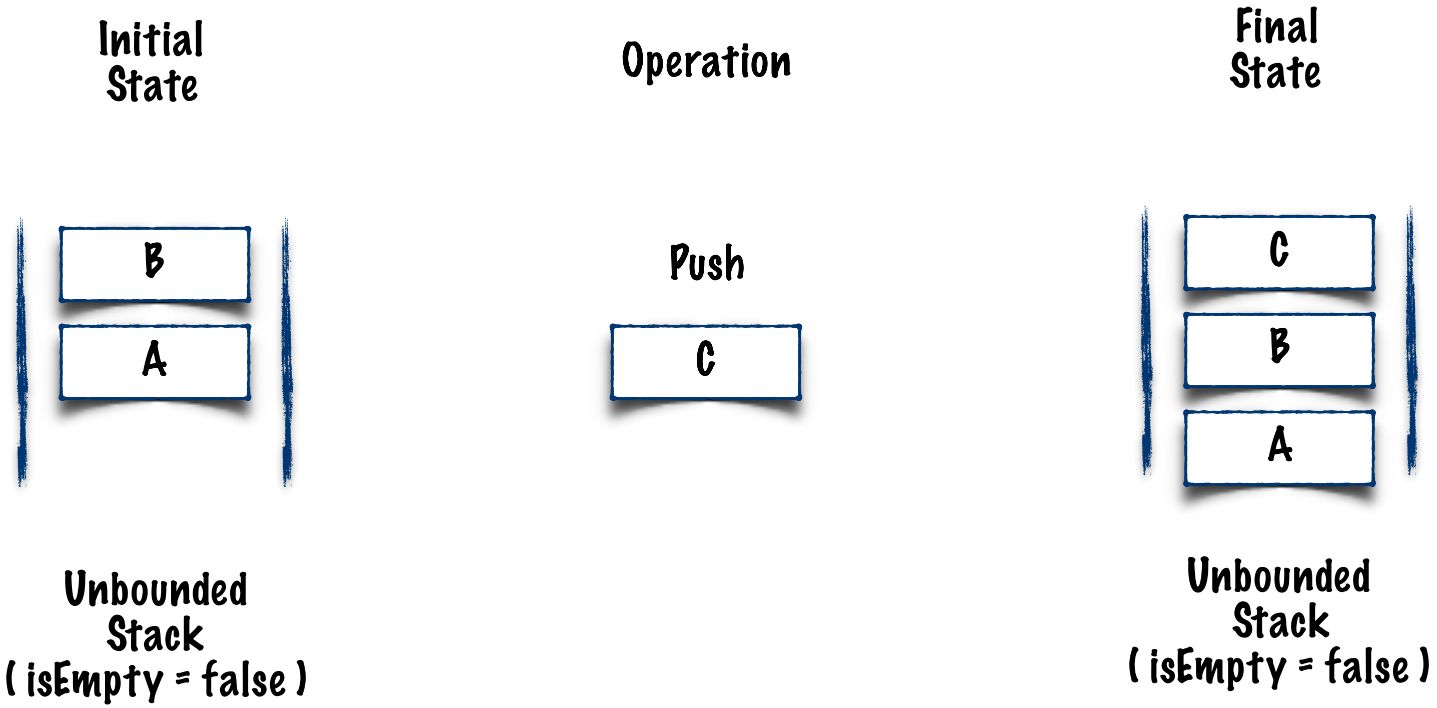
Exercise 5

Use TDD to test, design and implement a Stack class for integers. You are not allowed to use any of the built-in collection classes!

- Specification:
  - "A stack is a data structure in which you can access only the item at the top. With a computer, Stack like a stack of dishes—you add items to the top and remove them from the top."

Remember: Every single line of production code written must be motivated by a failing test!

Push operation



Initial State

Operation

Final State

B

A

Push

C

C

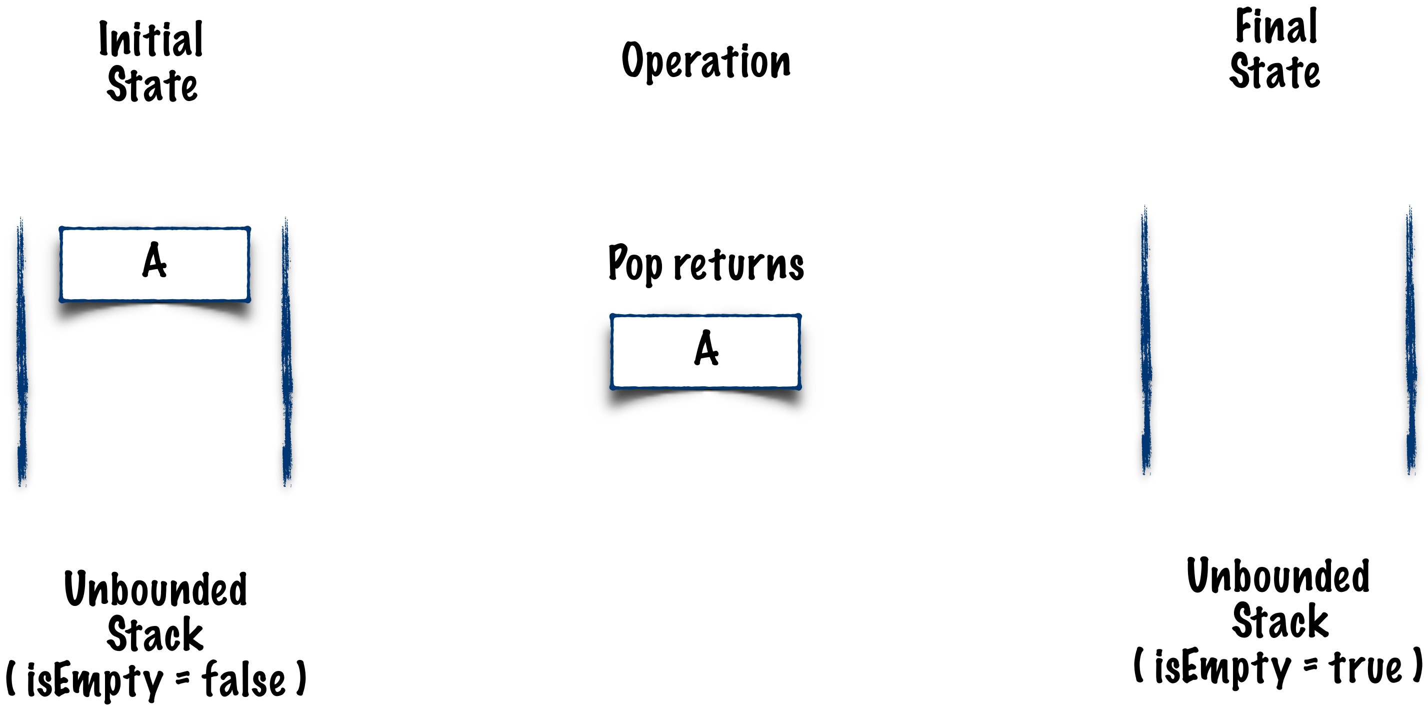
B

A

Unbounded Stack (isEmpty = false)

Unbounded Stack (isEmpty = false)

Pop operation



Initial State

Operation

Final State

A

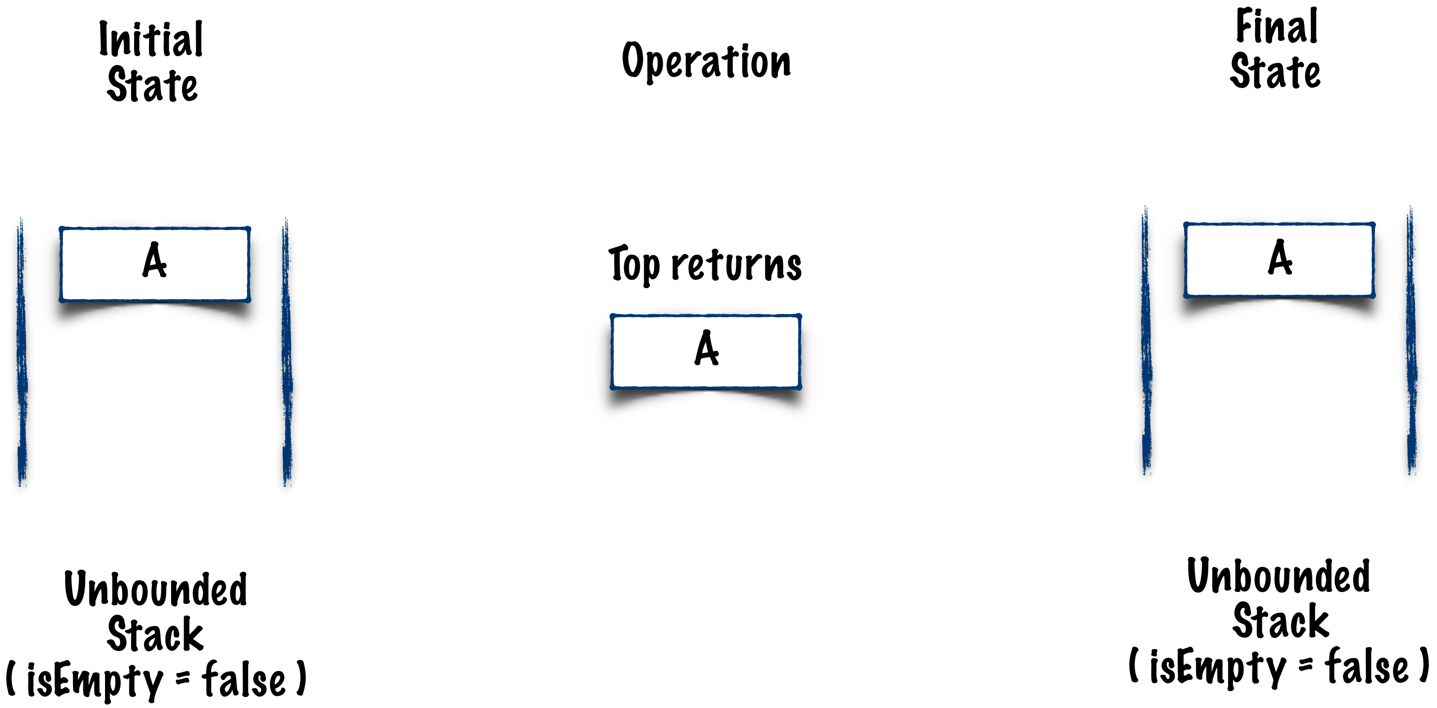
Pop returns

A

Unbounded Stack (isEmpty = false)

Unbounded Stack (isEmpty = true)

Top operation



Initial State

Operation

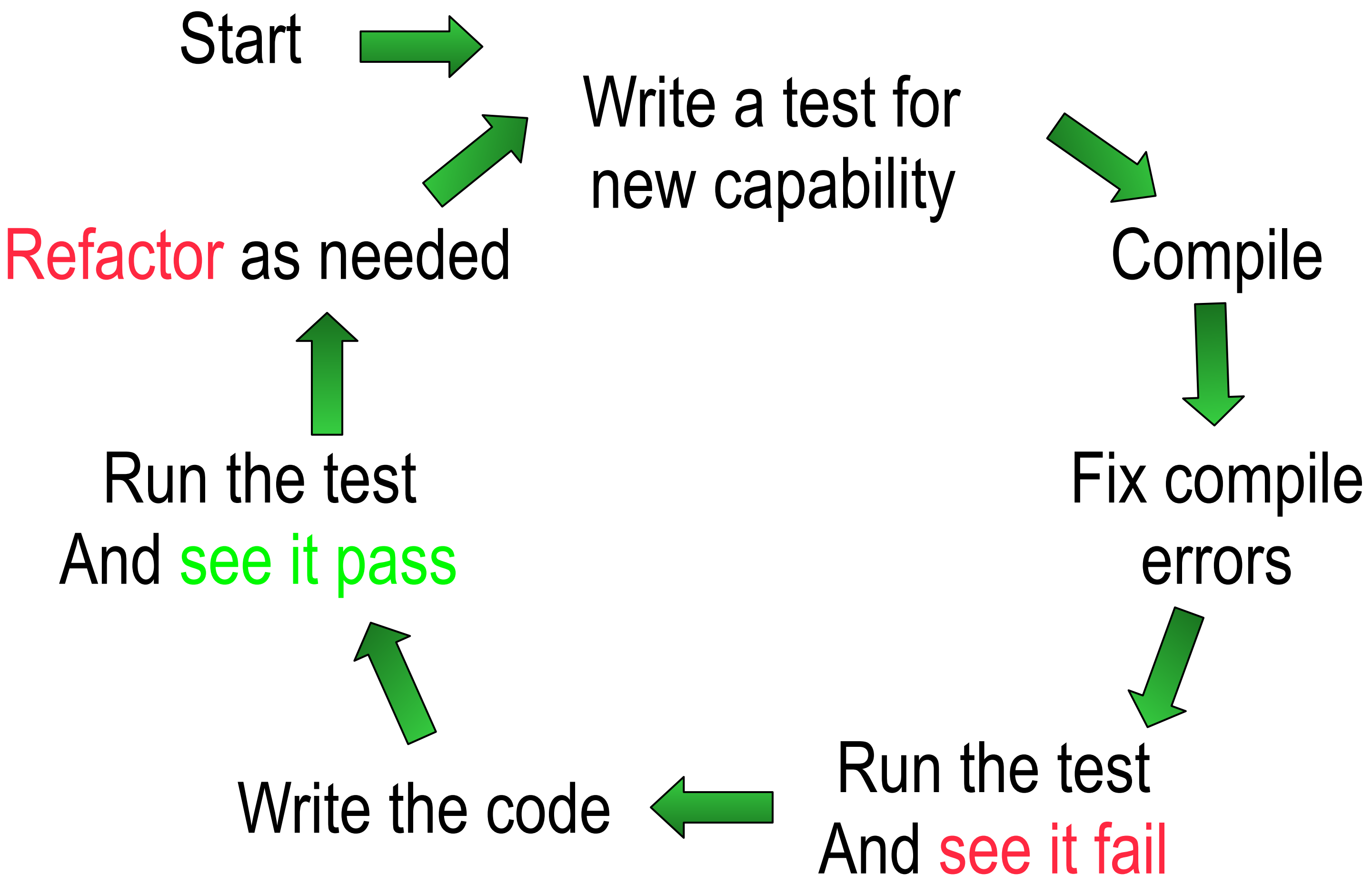
Final State

Top returns

Unbounded Stack (isEmpty = false)

Unbounded Stack (isEmpty = false)

Recap: The TDD process Red/Green/Refactor



Start

Write a test for new capability

Compile

Fix compile errors

Run the test
And see it fail

Write the code

Run the test
And see it pass

Refactor as needed

Exercise 5

- Separate presentation, or demo

Solution 5 - Step 1a

- **Add test of isEmpty() – see it fail**

```
public class SimpleStackTest {

  @Test
  public void testNewStackIsEmpty() {
    SimpleStack stack = new SimpleStack();
    Assert.assertTrue("New stack should be empty!",true == stack.isEmpty());
  }
}

public class SimpleStack {
  public boolean isEmpty() {
    return false;              // See it fail!
  }
}
```

Solution 5 - Step 1b

- **Add test of isEmpty() – make it work**

```
public class SimpleStackTest {

   @Test
   public void testNewStackIsEmpty() {
      SimpleStack stack = new SimpleStack();
      Assert.assertTrue("New stack should be empty!",true == stack.isEmpty());
   }
}

public class SimpleStack {
   public boolean isEmpty() {
      return true;                // See it work!
   }
}
```

Solution 5 - Step 2a

- **Add test of push() – see it fail**

```
public class SimpleStackTest {

   @Test
   public void testNewStackPush() {
      SimpleStack stack = new SimpleStack();
      Assert.assertTrue("New stack should be empty!",true == stack.isEmpty());
      int item = 1;
      stack.push(item);
      Assert.assertFalse("Stack should not be empty after an item has been pushed!”,
      stack.isEmpty());
   }
}

public class SimpleStack {
   public boolean isEmpty() {
      return true;
   }
   public void push(int item) {
      // Pushes to void, but that ok, see it fail.
   }
}
```

Solution 5 - Step 2b

- **Add test of push() – make it work**

```
public class SimpleStackTest {

   @Test
   public void testNewStackPush() {
      SimpleStack stack = new SimpleStack();
      int item = 1;
      stack.push(item);
      Assert.assertFalse("Stack should not be empty after an item has been pushed!”,
      stack.isEmpty());
   }
}

public class SimpleStack {
   boolean empty = true; // Add variable to keep a state
   public boolean isEmpty() {
      return empty; // Return the state
   }
   public void push(int item) {
      empty = false; // Still pushes to void, but that is ok, see it work.
   }
}
```

Solution 5 - Step 3

- **We now have got two tests, refactor (@Before)**

```
public class SimpleStackTest {

    @Test
    public void testNewStackIsEmpty() {
        SimpleStack stack = new SimpleStack();
        Assert.assertTrue("New stack should be empty!",true == stack.isEmpty());
    }

    @Test
    public void testNewStackPush() {
        SimpleStack stack = new SimpleStack();
        int item = 1;
        stack.push(item);
        Assert.assertFalse("Stack should not be empty after an item has been pushed!", stack.isEmpty());
    }
}
```

Solution 5 - Step 4

- **We now have got two tests, make it work**

```
public class SimpleStackTest {

    SimpleStack stack = null;     // Declare for commonalities

    @Before
    public void setUp() {         // Break out commonalities !
        stack = new SimpleStack();
    }

    @Test
    public void testNewStackIsEmpty() {
        Assert.assertTrue("New stack should be empty!",true == stack.isEmpty());
    }

    @Test
    public void testNewStackPush() {
        int item = 1;
        stack.push(item);
        Assert.assertFalse("Stack should not be empty after an item has been pushed!", stack.isEmpty());
     }
}
```

Solution 5 - Step 5

- **Add test of pop() of empty stack, see it fail, make it work**

```
@Test
public void testEmptyStackPop() {
  try { // expect an empty stack to throw exception when pop:ed
    @SuppressWarnings("unused")
    int topItem = stack.pop();
    Assert.fail("IllegalStateException expected");
  } catch (java.lang.IllegalStateException e) {
    // Expected
  }
}
// Production code
public int pop() {
  if (isEmpty()) {
    throw new java.lang.IllegalStateException();
  }
  return 0; // Don’t think ahead, this works for our tests
}
```

Solution 5 - Step 6a (The test)

- **Add test of pop() of stack with content**

```
@Test
public void testPopOfStackWithOneItem() {
   int item = 10;
   stack.push(item);
   int topItem = stack.pop();
   Assert.assertEquals("Popped item was expected to be 10.", item, topItem);
}
```

- **Run test, see it fail**

Solution 5 - Step 6b (The production code)

- **Add implementation of pop() of stack with content, run tests, make it work**

```
public class SimpleStack {
  boolean empty = true;
  int stackValue = 0; // We need a variable to hold the stack

  public int pop() {
    if (isEmpty()) {
      throw new java.lang.IllegalStateException();
    }
    return stackValue;
  }

  public void push(int item) {
    stackValue = item;
    empty = false;
  }
}
```

Solution 5 - Step 7a (The test code)

- **Add test for multiple push() and pop() – Run tests, see it fail**

```
@Ignore
public void testStackPushTwice() {
   int item = 1;
   stack.push(item);
   item = 2;
   stack.push(item);
   Assert.assertFalse("New stack should not be empty after an item has been pushed!",
   stack.isEmpty());
}
@Test
public void testStackPopTwice() {
   int item1 = 1;
   stack.push(item1);
   int item2 = 2;
   stack.push(item2);
   int topItem = stack.pop();
   Assert.assertEquals("Popped item was expected to be 2.", item2, topItem);
   topItem = stack.pop();

   Assert.assertEquals("Popped item was expected to be 1.", item1, topItem);
   Assert.assertTrue("Stack should be empty after all items has been pushed!", stack.isEmpty());
}
```

Solution 5 - Step 7b (The production code)

- **So far so good. It works, but we need to push our solution to be able to take more push. Time for redesign**

```
public class SimpleStack {
   private ListElement stackTop = null;

   public boolean isEmpty() {
      return stackTop == null;
   }

   public int pop() {
      int returnValue = 0;
      if (isEmpty()) {
         throw new java.lang.IllegalStateException();
      } else {
         returnValue = stackTop.value;
         stackTop = stackTop.nextElement;
      }
      return returnValue;
   }

   public void push(int item) {
      ListElement listElement = new ListElement();
      listElement.value = item;
      listElement.nextElement = stackTop;
      stackTop = listElement;
   }
}
```

```
public class ListElement {

      public int value = 0;
      public ListElement nextElement = null;

}
```

- **And, best of all, the tests will be reuse**

Solution 5 - Step 8

- **Add tests for top() – make it work**

```java
@Test
public void testEmptyStackTop() {
    try {
        @SuppressWarnings("unused")
        int top = stack.top();
        Assert.fail("IllegalStateException expected");
    } catch (java.lang.IllegalStateException e) {
        // Expected
    }
}

@Test
public void testStackTopTwice() {
    int item1 = 1;
    stack.push(item1);
    int topItem = stack.top();
    Assert.assertEquals("Top item was expected to be 1.", item1, topItem);
    topItem = stack.top();

    Assert.assertEquals("Top item was expected to be 1.", item1, topItem);
    Assert.assertFalse("Stack should not be empty after stack hass beeb topped!", stack.isEmpty());
}
```

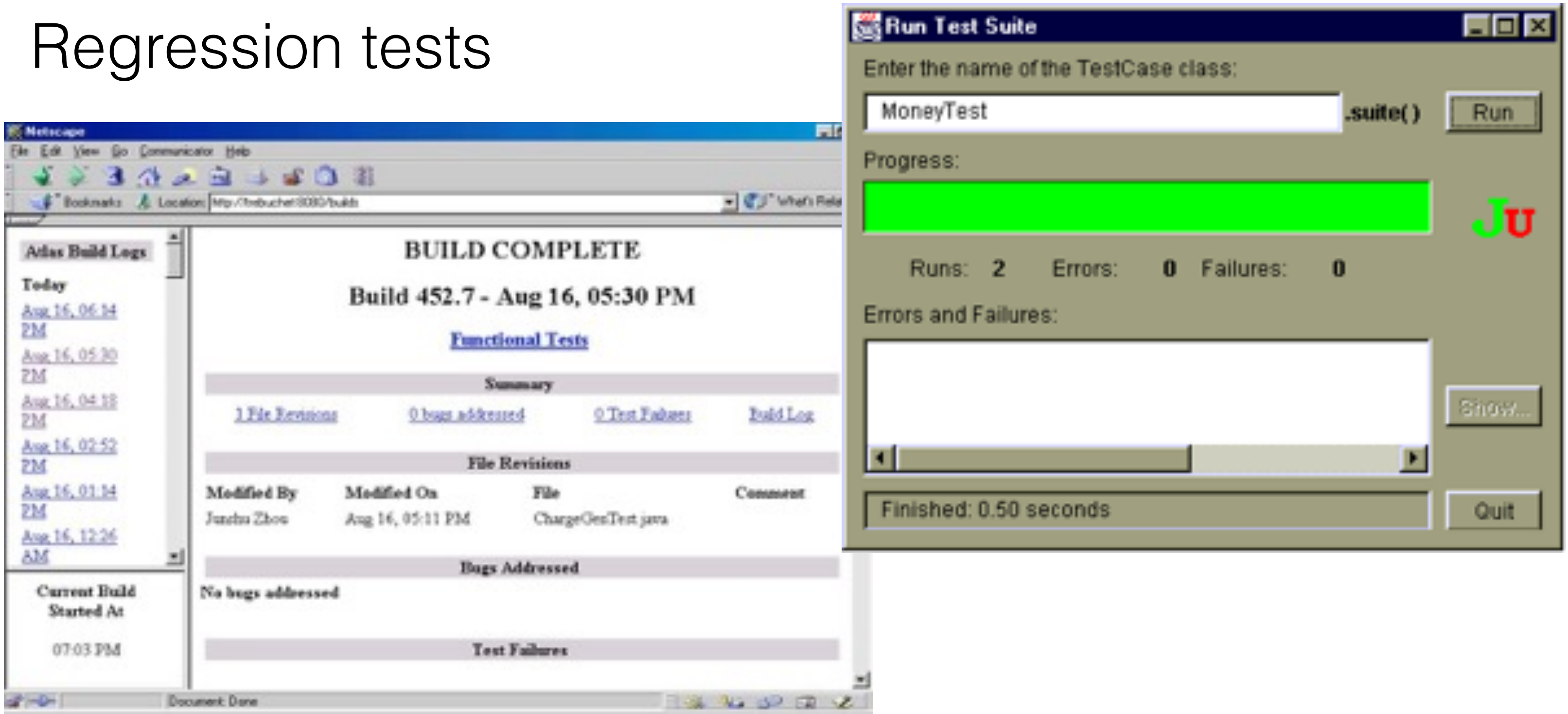
```java
public int top() {
    int returnValue = 0;
    if (isEmpty()) {
        throw new java.lang.IllegalStateException();
    } else {
        returnValue = stackTop.value;
    }
    return returnValue;
}
```

TDD
(Benefits)

Presentation provided by
Combitech JidokaQ

Obvious Effects of Test-Driven Development

- Already automated tests, immediately useful for
  - Integration tests
  - Regression tests



Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://trebuchet:8080/builds
What's Related
Document: Done

Atlas Build Logs

Today

Aug 16, 06:34 PM

Aug 16, 05:30 PM

Aug 16, 04:18 PM

Aug 16, 02:52 PM

Aug 16, 01:34 PM

Aug 16, 12:26 AM

Current Build Started At

07:03 PM

BUILD COMPLETE

Build 452.7 - Aug 16, 05:30 PM

Functional Tests

Summary

1 File Revisions · 0 bugs addressed · 0 Test Failures · Build Log

File Revisions

Modified By	Modified On	File	Comment
Junshu Zhou	Aug 16, 05:11 PM	ChargeGenTest.java	

Bugs Addressed

No bugs addressed

Test Failures

Enter the name of the TestCase class: MoneyTest .suite()

Progress:

Runs: 2 Errors: 0 Failures: 0

Errors and Failures:

Finished: 0.50 seconds

Not so obvious effects of Test-Driven Development

- Testing as we program means we spend less time debugging. We get our programs done faster.
- Testing as we program means that we don't have long tedious testing cycles at the end of our projects.
- Our tests are the first users of our code. We experience what it is like to use our code very quickly. Hence, design turns out better.
- Testing before coding is more interesting than testing after we code. Because it's interesting, we find it easier to maintain what we know is a good practice.

Not-so-obvious effects of TDD (Contd.)

- Intentional Design of Interfaces
  - Since the code in question is not written yet, we are free to choose the interface that is most usable.
- Non-speculative Interfaces
  - Interfaces provide the functionality which is just enough for right now
- Documented requirements and intended usage
  - The tests themselves provide immediately useful documentation of the Interfaces
- Good OO Design: High Cohesion and Low Coupling
  - If you have to write tests first, you'll devise ways of minimizing dependencies in your system in order to write your tests.

Possible weak points of TDD ?

- When test code, very intensively, use production API then it can impact the ability to refactor.
- Buggy tests – tests that is failing because of bugs in the test themselves.
- It will simply not be worth it, when cost for maintenance of the tests will be higher than benefits.

Hey there!

We are developers and should strive to mitigate these weak points, shouldn't we?

Types of tests

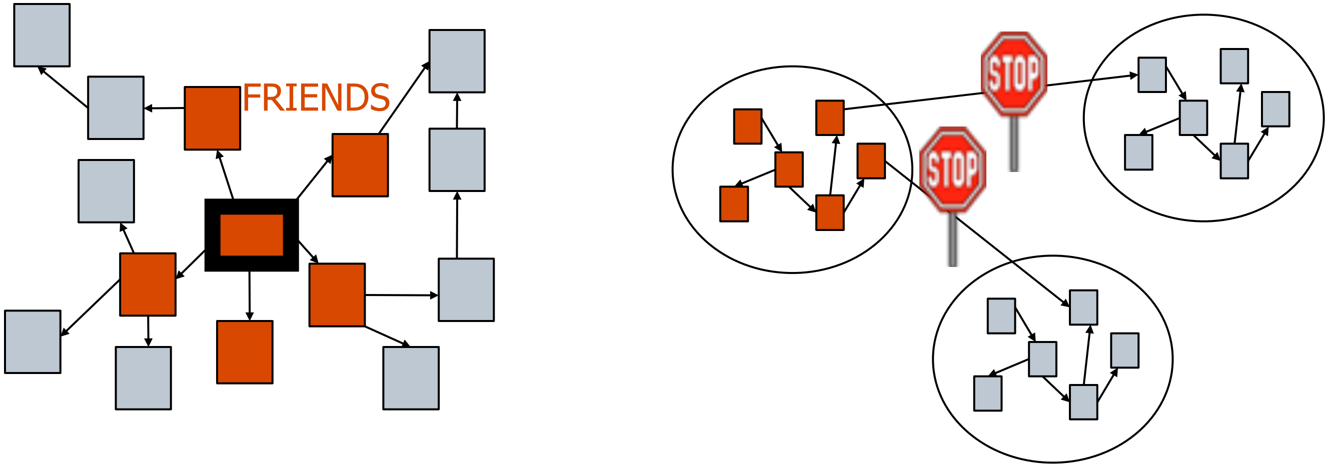
- **Unit test**: Specify and test one point of the contract of single method of a class. This should have a very narrow and well defined scope. Complex dependencies and interactions to the outside world are stubbed or mocked.
- **Integration test**: Test the correct inter-operation of multiple subsystems. There is whole spectrum there, from testing integration between two classes, to testing integration with the production environment.
- **Smoke test**: A simple integration test where we just check that when the system under test is invoked it returns normally and does not blow up. It is an analogy with electronics, where the first test occurs when powering up a circuit: if it smokes, it's bad.
- **Regression test**: A test that was written when a bug was fixed. It ensure that this specific bug will not occur again. The full name is "non-regression test".
- **Acceptance test**: Test that a feature or use case is correctly implemented. It is similar to an integration test, but with a focus on the use case to provide rather than on the components involved.
- A **Canary test** is an automated, non-destructive test that is run on a regular basis in a LIVE environment, such that if it ever fails, something really bad has happened.
  - Examples might be:
    - Has data that should only ever be available in DEV/TEST appeared in LIVE.
    - Has a background process failed to run
    - Can a user logon
    - the concept of a canary in a coal mine.

Canary Test

- In software testing, a canary (also called a canary test) is a push of programming code changes to a small number of end users who have not volunteered to test anything. The goal of a canary test is to make sure code changes are transparent and work in a real world environment.
- Canary tests, which are often automated, are run after testing in a sandbox environment has been completed. Because the canary is only pushed to a small number of users, its impact is relatively small. If the new code prove to be buggy then the changes can be reversed quickly.
- The word canary was selected to describe the code push, because just like canaries that were once used in coal mining to alert miners when toxic gases reached dangerous levels, end users selected for testing are unaware they are being used to provide an early warning.

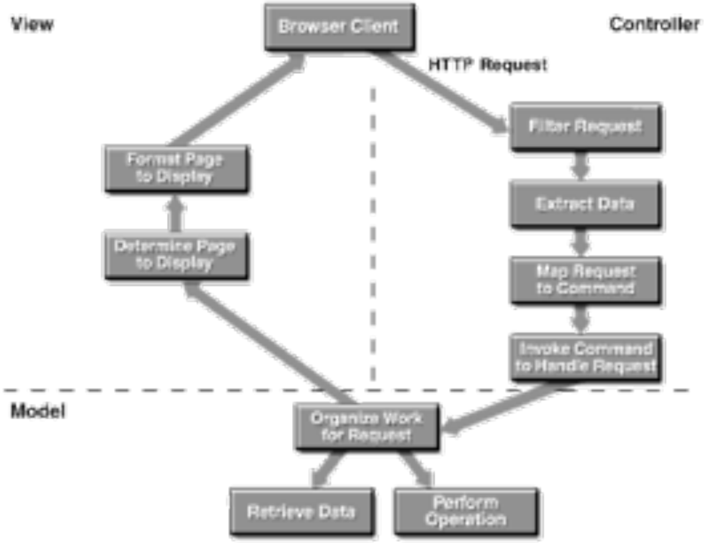
Designing for Testability: Low Coupling

- Minimize dependencies between classes
- Only allow “closely related” classes to interact directly



Designing for Testability: Model-View-Control

- User Interfaces are notoriously difficult to test
- Splitting a complex application into separate, cohesive parts which separates presentation from application logic enables testing of the application logic in isolation

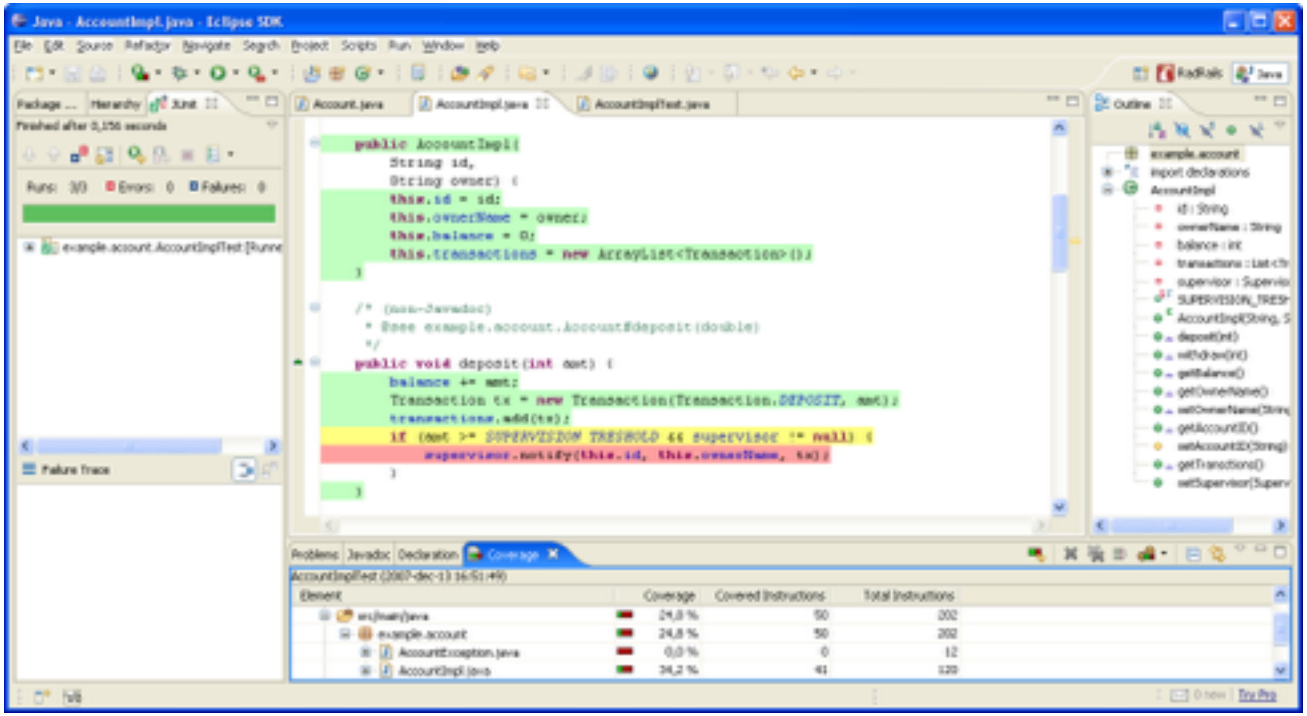


Some quotes on Test-Driven Development

- "*Test-Driven Development is a powerful way to produce well designed code with fewer defects*" – Martin Fowler
- "*The best way that I know to write code is to shape it from the beginning with tests*" – Ron Jeffries
- "*Fewer defects, less debugging, more confidence, better design, and higher productivity in my programming practice*" – Kent Beck

Code Coverage (Java)

- Which statements of my application are being executed?
- Useful to identify incomplete testing (ECLemma plug-In)
  - Option 1: Install from Eclipse Marketplace Client
  - Option 2: Installation from update site
  - Option 3: Manual download and installation



```
    public AccountImpl(
            String id,
            String owner) {
        this.id = id;
        this.ownerName = owner;
        this.balance = 0;
        this.transactions = new ArrayList<Transaction>();
    }

    /* (non-Javadoc)
     * @see example.account.Account#deposit(double)
     */
    public void deposit(int amt) {
        balance += amt;
        Transaction tx = new Transaction(Transaction.DEPOSIT, amt);
        transactions.add(tx);
        if (amt >= SUPERVISION_TRESHOLD && supervisor != null) {
            supervisor.notify(this.id, this.ownerName, tx);
        }
    }
```

AccountImplTest (2007-dec-13 16:51:49)

Element	Coverage	Covered Instructions	Total Instructions
src/main/java	24,8 %	50	202
example.account	24,8 %	50	202
AccountException.java	0,0 %	0	12
AccountImpl.java	34,2 %	41	120

But …

- Focusing only on coverage is not sufficient, you may miss:
    - Missing code
    - Incorrect handling of boundary conditions
    - Timing problems
    - Memory Leaks
- Use coverage sensibly
    - Objective, but incomplete
    - Too often distorts sensible action

TDD enables cheap, early, and frequent tests!

It’s always a bit painful to change your habits, but once you’ve been there, you’re stuck!

- Enables truly iterative projects
- Improves your design
- Doesn’t cost your project a fortune
- It’s even fun!

Bottom Line:

Automated Testing and Test-Driven Development

is infectious!

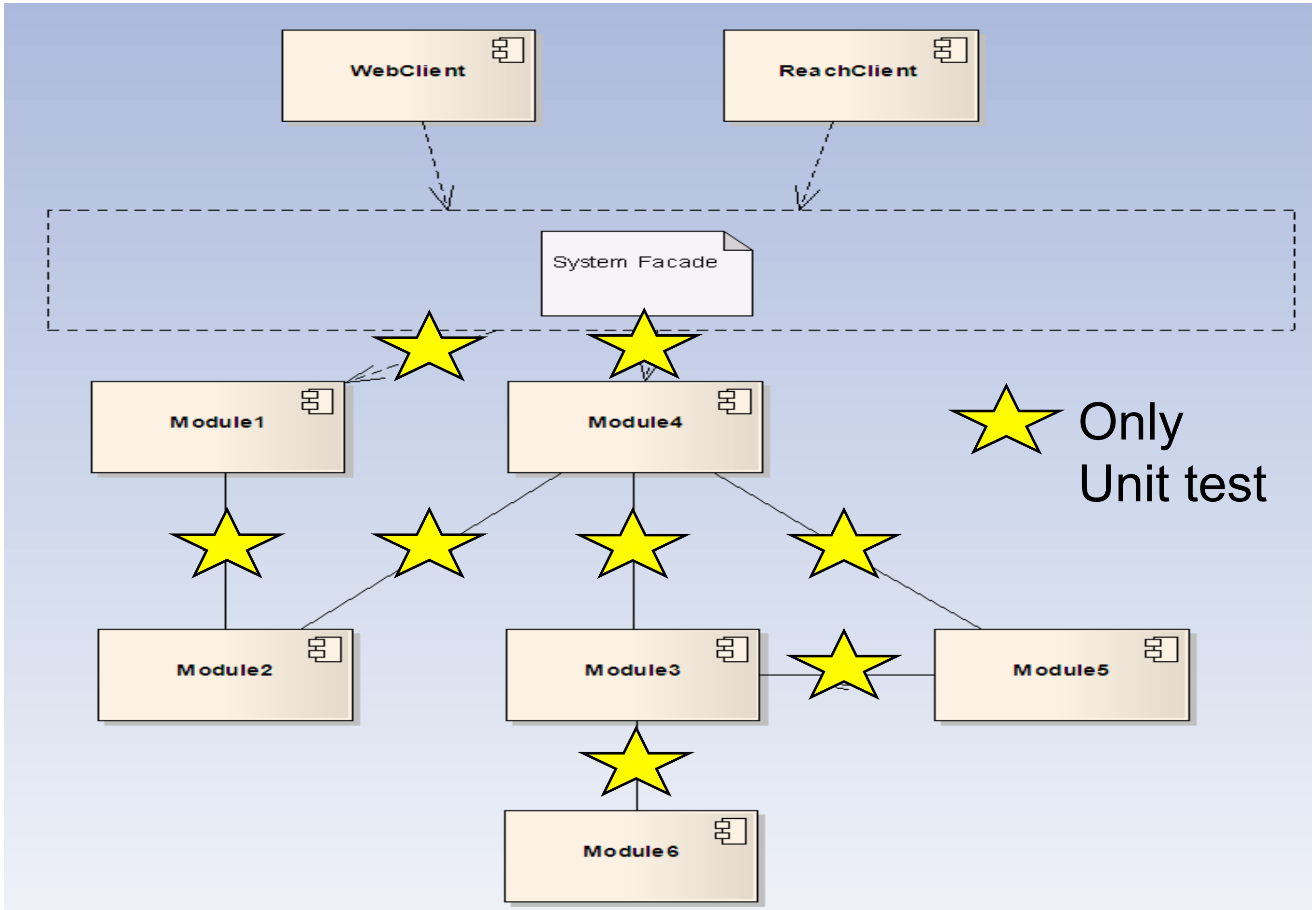
Integration Tests

- An Integration Test is any test which tests a logical unit *together with other units* ***that it depends on***, such as other software units but more frequently external resources such as Databases or Message Queues.
- Thus the integration tests share many of the characteristics of Unit Tests, but the granularity is much bigger.
- Due to the performance costs in accessing external resources, the integration tests usually takes much longer time to execute.

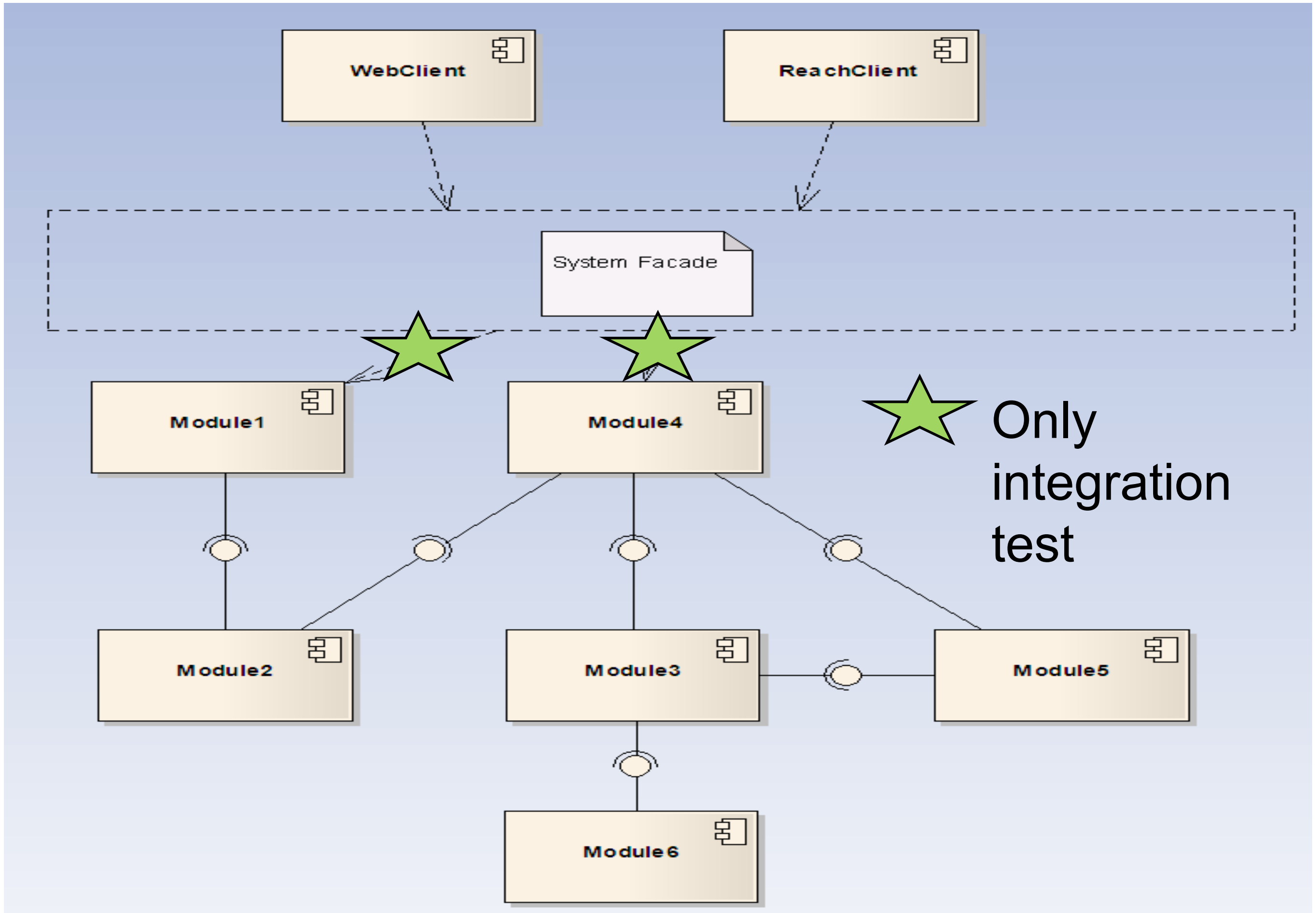
Ratio between Unit and Integration Tests

- Unit tests are naturally the most efficient way of catching defects on the Unit level.
  - If a defect can be caught using a Unit Test, it should therefore be preferred instead of catching it using an Integration test.
  - Hence there will typically be many more Unit tests than integration tests.
- Integration tests should be used for testing interaction between units and between units and external resources.

Ratio between Unit and Integration Tests



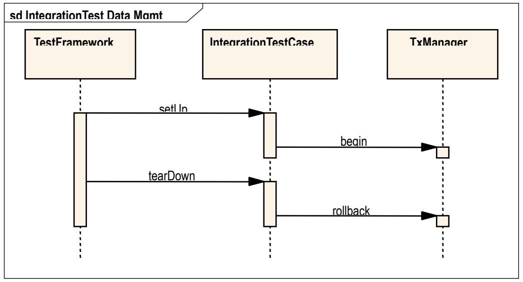
Ratio between Unit and Integration Tests



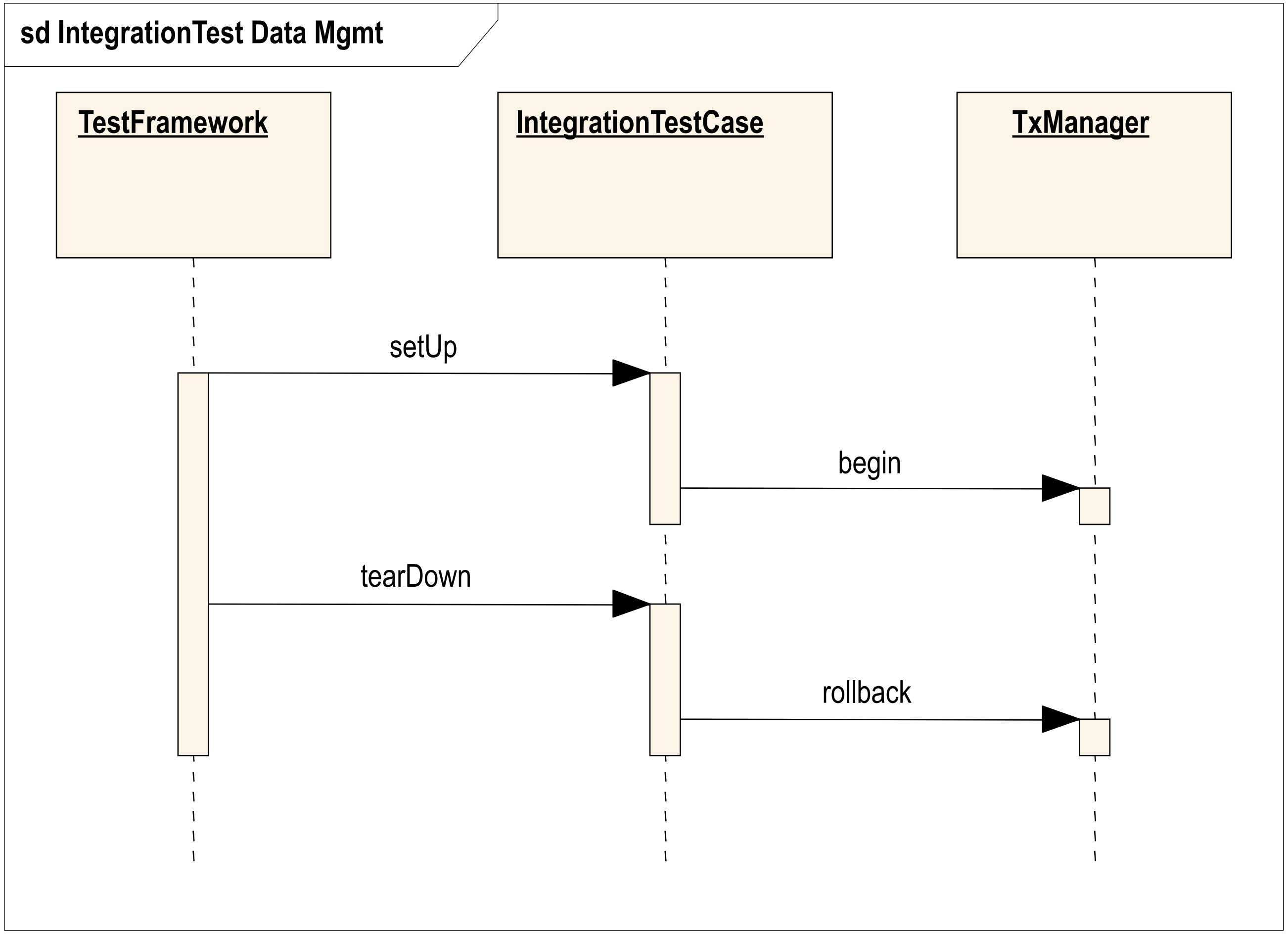
★ Only integration test

Test Data, concurrency and repeatability

- Integration tests which have side effects (i.e. which affects persistent data) are problematic:
  - Modifying data which other tests may depend on, may cause subsequent test failures
  - Several instances of tests which uses the same data may run concurrently, which may cause test failures
- Transaction demarcation is a common idiom to protect test data from modification:
  - Start a transaction in [SetUp]
  - Rollback the transaction in [TearDown]



Test Data, concurrency and repeatability



Test Data strategies

- Integration tests
  - Local test data, owned and managed by test
  - Global, common test data, pre-populated via SQL scripts
- System tests
  - All data owned by test script
- Separate Databases
  - Primary keys/IDS and Test Data

Managing External Test Data Files

- Some test data is most easily kept in a file format (e.g. XML files) which are read and manipulated by the tests. In order to make the tests insulated from how and where they are executed, the tests should not refer to external files via the file system. Both absolute and relative file names may differ depending on the execution environment.
- Instead, the tests may keep data files as “Embedded Resources” within the test assembly itself (src/test/resources).

DbUnit

- DbUnit is a JUnit extension targeted at database-driven applications.
- Puts your database into a known state between test runs.
- DbUnit has the ability to export and import your database data to and from XML datasets.
- DbUnit is used by JVS.

Designing for testability

Presentation provided by
Combitech JidokaQ

Designing for Testability : Law of Demeter

(LoD or principle of least knowledge)

- Any method should have limited knowledge about its surrounding object structure.
- Named in honor of Demeter, “distribution-mother”, Greek goddess of agriculture
- Hence

```
public class SomeUnit {
   private IDependee dependee;
   public SomeUnit() {
      this.dependee = new Dependee();
   }
   ...
}
```

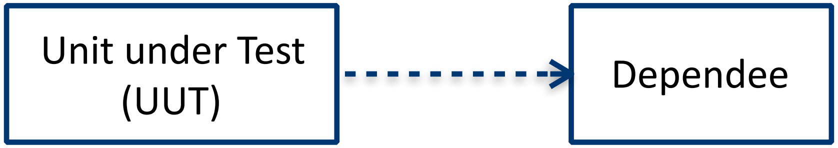
Law of Demeter (Contd.)

- becomes

```
public class SomeUnit {
   private IDependee dependee;
   public SomeUnit() {
   }
   public SetDependee(IDependee dependee) {
      this.dependee = dependee;
   }
   ...
}
```

Designing for Testability: LoD - Don’t Talk To Strangers

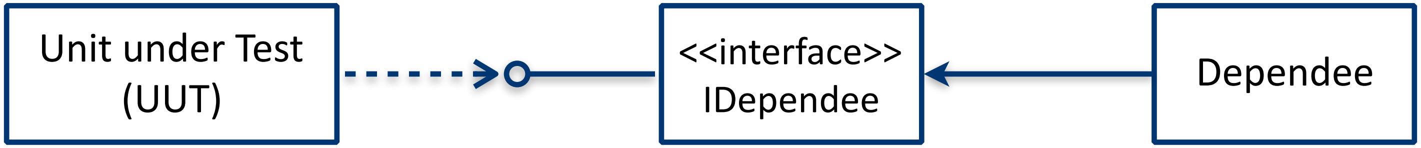
- If there are no strong reasons why two classes should talk to each other directly, *they shouldn’t!*



Unit under Test (UUT)

Dependee

becomes



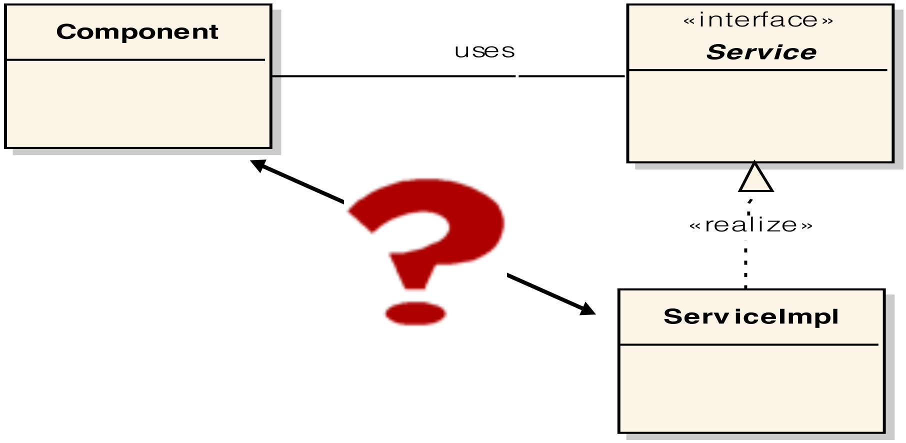
Unit under Test (UUT)

<<interface>> IDependee

Dependee

Designing for Testability: Dependency Injection

- What is it?
- Dependency Management
- Dependency Injection provides a mechanism for managing dependencies between components in a decoupled way
- Makes it easier to unit test components in isolation
- Out of container and with mocked dependencies



Breaking dependencies

Presentation provided by
Combitech JidokaQ

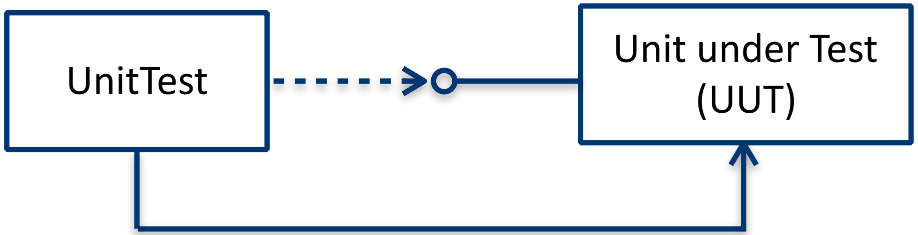
Design properties and Design goals

For Units:

- Modularity
- High cohesion
- Low coupling

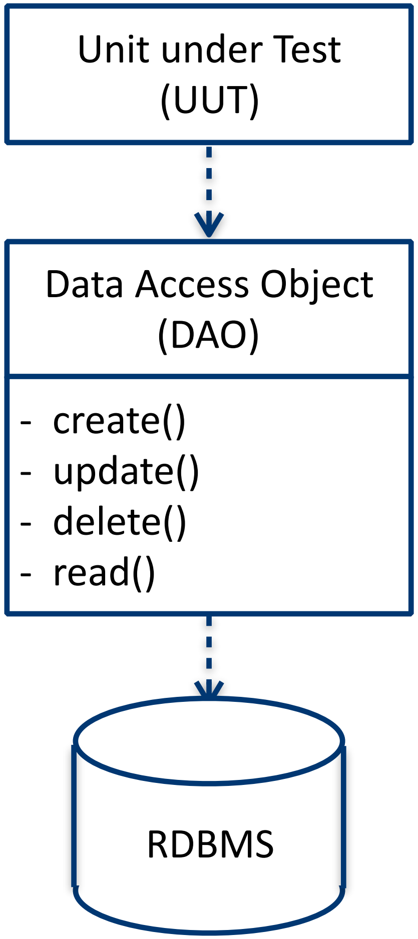
For Tests:

- Modularity
- Locality

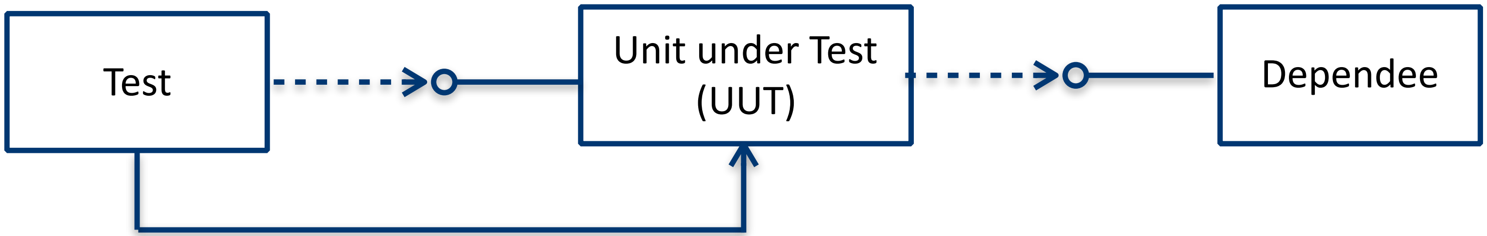


Side effects

But what about units that depend on other units (with potential side effects)?



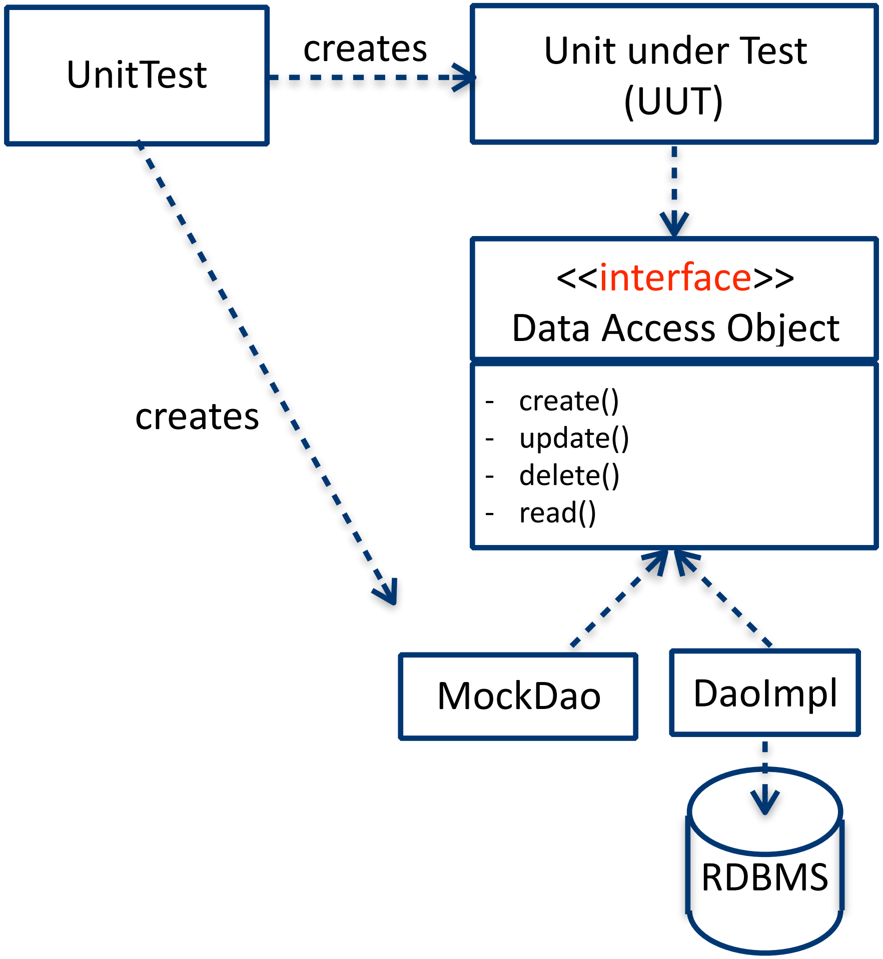
Strategies for testing Units that depend on other units



- Break the dependency: Let the Test create a synthetic ‘Mock’ context
- Run and test the Unit within it’s natural context (In Container in the case of Java EE or .NET)
- Let the Test create the real context

Synthetic context – Mocking

- Implements the same interface as the resource that it represents
- Enables configuration of its behavior from outside (i.e. from the test class, in order to achieve locality)
- Enables registering and verifying *expectations* on how the resource is used



Mocking using plain Java

Presentation provided by
Combitech JidokaQ

Why Mock?

Unit tests should act as a safety net and provide quick feedback. This is the main principle of TDD.

A test may take time to execute due to the following reasons:

- Sometimes a test acquires a connection from the database that fetches/updates data
- It connects to the Internet and downloads files
- It interacts with an SMTP server to send e-mails
- It performs I/O operations

Why Mock?

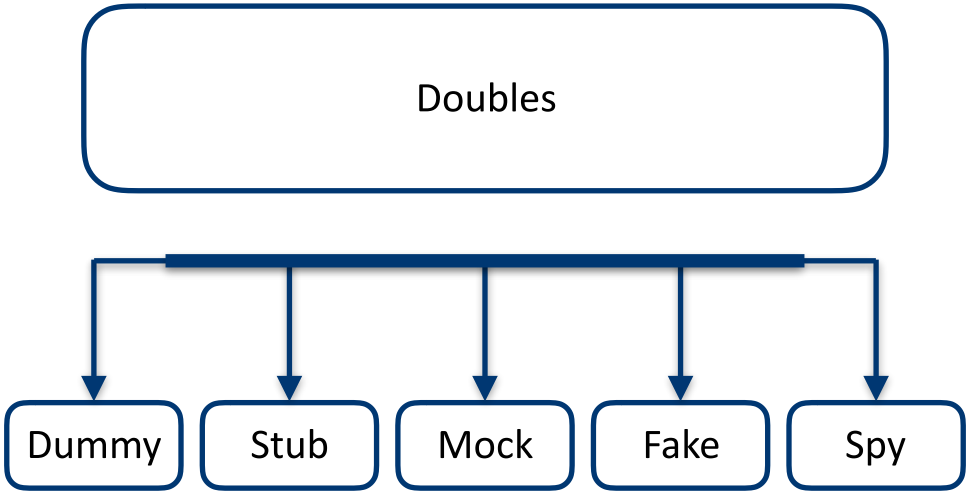
Do we really need to acquire a database connection or download files to unit test code?

The answer is yes.

If it doesn't connect to a database or download the latest stock price, few parts of the system remain untested. So, DB interaction or network connection is mandatory for a few parts of the system, and these are integration tests.

But, to unit test these parts, the external dependencies need to be mocked out.

Test Doubles



Doubles

Dummy

Stub

Mock

Fake

Spy

Dummy

An example of a dummy would be a movie scene where the double doesn't perform anything but is only present on the screen. They are used when the actual actor is not present, but their presence is needed for a scene.

Similarly, dummy objects are passed to avoid `NullPointerException` for mandatory parameter objects as follows:

```
@Test
public void aTestMethod() {
   Book javaBook = new Book("Java 101", "123456");
   Member dummyMember = new DummyMember(); //<-a dummy member
   javaBook.issueTo(dummyMember);
   assertEquals(javaBook.numberOfTimesIssued(),1);
}
```

Stub

A stub delivers indirect inputs to the caller when the stub's methods are called. Stubs are programmed only for the test scope. Stubs may record other information such as the number of times the methods were invoked and so on.

Account transactions should be rolled back if the ATM's money dispenser fails to dispense money. How can we test this when we don't have the ATM machine, or how can we simulate a scenario where the dispenser fails? We can do this using the following code:

```
public interface Dispenser {
    void dispense(BigDecimal amount) throws DispenserFailed;
}

public class AlwaysFailingDispenserStub implements Dispenser {
    public void dispense(BigDecimal amount) throws DispenserFailed {
      throw new DispenserFailed (ErrorType.HARDWARE,
        “Dispenser not  responding");
    }
}
```

Stub example

```
@Test
public void transaction_is_rolledback_when_hardware_fails() {
    // Arrange: set up a failing dispenser
    Account myAccount = new Account("John", 2000.00);
    TransactionManager txMgr = TransactionManager.forAccount(myAccount);
    txMgr.registerMoneyDispenser(new AlwaysFailingDispenserStub());

    // Act:
    WithdrawalResponse response = txMgr.withdraw(500.00);

    // Assert: no success and that no change on the account
    assertEquals(false, response.wasSuccess());
    assertEquals(2000.00, myAccount.remainingAmount());
}
```

Fake

Fake objects are working implementations; mostly, the fake class extends the original class, but it usually hacks the performance, which makes it unsuitable for production. The following example demonstrates the fake object:

```
public class AddressDAO extends SimpleJdbcDaoSupport {

    public void batchInsertOrUpdate(List<AddressDTO> addressList, User user) {
        List<AddressDTO> insertList = buildListWhereLastChangeTimeMissing(addressList);
        List<AddressDTO> updateList = buildListWhereLastChangeTimeValued(addressList);
        int rowCount = 0;

        if (!insertList.isEmpty()) {
            rowCount = getSimpleJdbcTemplate().batchUpdate(INSERT_SQL, "...");
        }
        if (!updateList.isEmpty()) {
            rowCount += getSimpleJdbcTemplate().batchUpdate(UPDATE_SQL, "...");
        }
        if (addressList.size() != rowCount) {
            raiseErrorForDataInconsistency("...");
        }
    }
    public SimpleJdbcTemplate getSimpleJdbcTemplate() { // Fake this method !!!
        // Long complicated implementation…
    }
}
```

Fake example

```
public class FakeAddressDAO extends AddressDAO {
   private SimpleJdbcTemplate jdbcTemplate = new SimpleJdbcTemplate();

   @Override
   public SimpleJdbcTemplate getSimpleJdbcTemplate() {
      return jdbcTemplate ;
   }
}
```

Mock

Mock objects have expectations; a test expects a value from a mock object, and during execution, a mock object returns the expected result.

Also, mock objects can keep track of the invocation count, that is, the number of times a method on a mock object is invoked.

Mock

The following example is a continuation of the ATM example with a mock version. In the previous example, we stubbed the dispense method of the Dispenser interface to throw an exception; here, we'll use a mock object to replicate the same behavior.

```
@RunWith(MockitoJUnitRunner.class)
public class ATMTest {
    @Mock
    Dispenser failingDispenser;
    @Before
    public void setUp() throws Exception {
        MockitoAnnotations.initMocks(this);
    }
    @Test
    public void transaction_is_rolledback_when_hardware_fails() throws DispenserFailed {
        Account myAccount = new Account(2000.00, "John");
        TransactionManager txMgr = TransactionManager.forAccount(myAccount);
        txMgr.registerMoneyDispenser(failingDispenser);
        doThrow(new DispenserFailed()).when(failingDispenser).dispense(isA(BigDecimal.class));

        txMgr.withdraw(500.00f);

        assertEquals(2000.00, myAccount.getRemainingBalance(), 0.01f);
        verify(failingDispenser, new Times(1)).dispense(isA(BigDecimal.class));
    }
}
```

Spy

Spy is a variation of a mock/stub, but instead of only setting expectations, it records the calls made to the collaborator.

```java
public class ResourceAdapter {
    Printer printer;
    SecurityService securityService;

    public ResourceAdapter(SecurityService securityService, Printer printer) {
        this.securityService = securityService;
        this.printer = printer; // <-- Supports dependency injection
    }
    void print(String userId, String document, Object settings) {
        if (securityService.canAccess("lanPrinter1", userId)) {
            printer.print(document, settings);
        }
    }
}

public interface Printer { void print(String document, Object settings); }
public interface SecurityService { public boolean canAccess(String name, String id); }
```

Spy Example cont.

```
public class FakeSecurityService implements SecurityService { // <-- Fake a service
    public boolean canAccess(String name, String id) {
        return true;
    }
}
public class SpyPrinter implements Printer { // <-- Implement a Spy
    private int noOfTimescalled = 0;
    @Override
    public void print(String document, Object settings) {
        noOfTimescalled++;
    }
    public int getInvocationCount() {
        return noOfTimescalled;
    }
}
@Test
public void verify() throws Exception {
    SpyPrinter spyPrinter = new SpyPrinter(); // <-- Arrange
    adapter = new ResourceAdapter(new FakeSecurityService(), spyPrinter);
    adapter.print("john", "helloWorld.txt", "all pages");
    assertEquals(1, spyPrinter.getInvocationCount());
}
```

When to use mocking (and when not to)

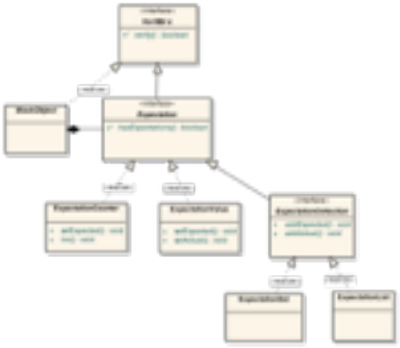
- Mocking is great for
  - Breaking dependencies between well-architected layers or tiers
  - Testing corner cases and exceptional behavior
- Mocking is less ideal for
  - Replacing awkward 3rd party APIs
  - Responsibilities which involves large amounts of state or data, which could be more conveniently expressed in a ”native” format
- This is clearly a judgement call: If breaking a dependency using mock objects cost more effort than living with the dependency, then the mock strategy is probably not a good idea

Mocking with Mockito (Intro)

Presentation provided by
Combitech JidokaQ

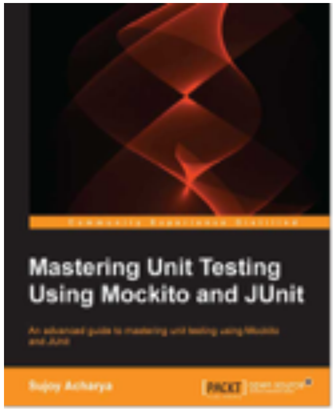
Frameworks and tools for mocking

- code.google.com/p/mockito/ (Active 2015)
  - No expect-run-verify also means that Mockito mocks are often ready without expensive setup upfront
- www.easymock.org (Active 2015)
  - Class library which generates Mock Objects dynamically using the Java Proxy class
- www.mockobjects.org (latest update 2010)
  - Commonly used assertions refactored into a number of Expectation classes, which facilitate writing Mock Objects.
- www.mockmaker.org (latest update 2002)
  - Tool which automatically generates a MockObject from a Class or Interface



Mockito

- Mocks concrete classes as well as interfaces
- Little annotation syntax sugar - @Mock
- Verification errors are clean - click on stack trace to see failed verification in test; click on exception's cause to navigate to actual interaction in code. Stack trace is always clean.
- Allows flexible verification in order (e.g: verify in order what you want, not every single interaction)
- Supports exact-number-of-times and at-least-once verification
- Flexible verification or stubbing using argument matchers (anyObject(), anyString() or refEq() for reflection-based equality matching)
- Allows creating custom argument matchers or using existing hamcrest matchers



Mastering Unit Testing Using Mockito and JUnit

Typical usage scenario for mocking in a test case

1. Instantiate mock objects
2. Set up state in mock objects, which govern their behavior
3. Set up expectations on mock objects
4. Execute the method(s) on the Unit Under Test, using the mock objects as resources
5. Verify the results & expectations

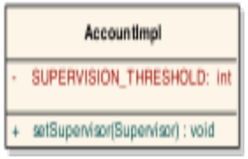
Mockito - example usage

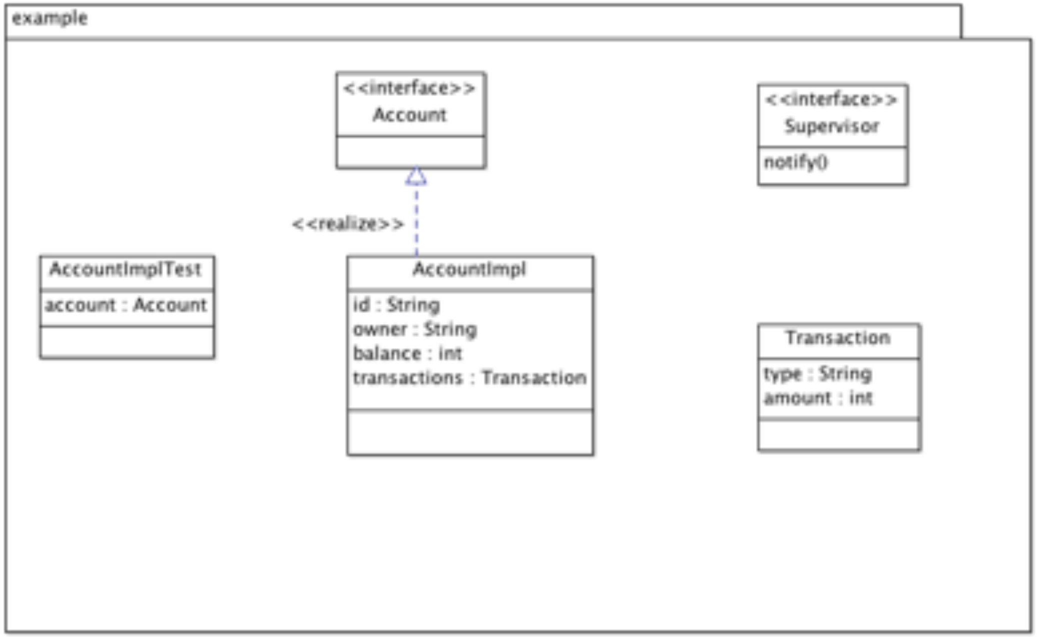
```
@Test
public void testNotificationVetoShouldBeHonoured() {
    int amount = AccountImpl.SUPERVISION_TRESHOLD;

    Supervisor mockSupervisor = Mockito.mock(Supervisor.class);

    Mockito.when(mockSupervisor.notify(Mockito.anyString(),
        Mockito.anyString(), (Transaction) Mockito.anyObject())).thenReturn(false);

    account.setSupervisor(mockSupervisor);

    try {
        account.deposit(amount);
        Assert.fail("SupervisorException expected");
    } catch (SupervisorException expected) {
        // expected
        System.err.println(expected);
    }

    Mockito.verify(mockSupervisor).notify(account.getAccountID(), account.getOwnerName(),
        new Transaction(Transaction.DEPOSIT, amount));
}
```

- Create MockObject
- Let the mock object know how to answer on an expected call
- Inject the MockObject in the class to be tested
- Run the test
- Verify that the mock object received the expected calls and parameters

Exercise 7

- Extend the tests for AccountImpl to use Mockito for validating correct usage of the Supervisor collaborator!


AccountImpl
- SUPERVISION_THRESHOLD: int
+ setSupervisor(Supervisor) : void



Mocking with Mockito

Presentation provided by
Combitech JidokaQ

How to invoke Mockito

Unit tests should act as a safety net and provide quick feedback; the main principle of TDD.

```
import org.junit.runner.RunWith;
import static org.mockito.Mockito.mock;

@RunWith(MockitoJUnitRunner.class)
public class StockBrokerTest {
      // Alternative !: Static mock
      MarketWatcher marketWatcher = mock(MarketWatcher.class);
      // Alternative 2: Annotated mock
      // Needs a call to MockitoAnnotations.initMocks(this);
      @Mock
      Portfolio portfolio;

      @Before
      public void setUp() {
            broker = new StockBroker(marketWatcher);
            MockitoAnnotations.initMocks(this); // <-- Need to call this
      }
      @Test
      public void sanity() throws Exception {
            assertNotNull(marketWatcher); // <-- Needed to check that Mockito initiated properly
            assertNotNull(portfolio);
      }
}
```

Stubbing with Mockito

The stubbing process defines the behavior of a mock method such as the value to be returned or the exception to be thrown when the method is invoked.

The Mockito framework supports stubbing and allows us to return a given value when a specific method is called. This can be done using Mockito.when() along with thenReturn().

```
import static org.mockito.Matchers.anyString;
import static org.mockito.Mockito.when;

@RunWith(MockitoJUnitRunner.class)
public class StockBrokerTest {
   // . . . se previous slide
   @Test
   public void marketWatcher_Returns_current_stock_status() throws Exception {
      BigDecimal aBigDecimal = new BigDecimal(100.00);
      Stock uvsityCorp = new Stock("UV", "UVSITY Corporation “, aHundred);
      when(marketWatcher.getQuote(anyString())).thenReturn(uvsityCorp);
      assertNotNull(marketWatcher.getQuote("UV"));
   }
}
```

Mockito.when(…).thenNnnn()

The when() method represents the trigger, that is, when to stub.
The following methods are used to represent what to do when the trigger is triggered:

- thenReturn(x): This returns the x value.
- thenThrow(x): This throws an x exception.
- thenAnswer(Answer answer): Unlike returning a hardcoded value,
  a dynamic user-defined logic is executed. It's more like for fake test doubles,
  Answer is an interface.
- thenCallRealMethod(): This method calls the real method on the mock object.

Verify() with Mockito

The verify() method has an overloaded version that takes Times as an argument. Times is a Mockito framework class of the org.mockito.internal.verification package, and it takes wantedNumberOfInvocations as an integer argument.

```
@Test
public void when_ten_percent_gain_then_the_stock_is_sold() { // Arrange Mock
      when(portfolio.getAvgPrice(isA(Stock.class))).thenReturn(new BigDecimal("10.00"));
      Stock aCorp = new Stock("A", "A Corp", new BigDecimal(11.20));
      when(marketWatcher.getQuote(anyString())).thenReturn(aCorp);
      broker.perform(portfolio, aCorp); // Act, run business logic
      verify(portfolio).sell(aCorp, 10); // Verifies method invocation
}

@Test
public void argument_matcher() { // This example will be more covered under advanced topics
      when(portfolio.getAvgPrice(isA(Stock.class))).thenReturn(new BigDecimal("10.00"));
      Stock blueChipStock = new Stock("FB", "FB Corp”, new BigDecimal(1000.00));
      Stock otherStock = new Stock("XY", "XY Corp", new BigDecimal(5.00));
      when(marketWatcher.getQuote(argThat(new BlueChipStockMatcher()))).thenReturn(blueChipStock);
      when(marketWatcher.getQuote(argThat(new OtherStockMatcher()))).thenReturn(otherStock);

      broker.perform(portfolio, blueChipStock);
      verify(portfolio).sell(blueChipStock, 10); // verifies invocation

      broker.perform(portfolio, otherStock);
      verify(portfolio, new Times(0)).sell(otherStock, 10); // verifies zero invocation
}
```

Methods used in conjunction with `verify()`

The following methods are used in conjunction with verify:

- `times(int wantedNumberOfInvocations)`: This method is invoked exactly *n* times; if the method is not invoked wantedNumberOfInvocations times, then the test fails.
- `never()`: This method signifies that the stubbed method is never called or you can use times(0) to represent the same scenario. If the stubbed method is invoked at least once, then the test fails.
- `atLeastOnce()`: This method is invoked at least once, and it works fine if it is invoked multiple times. However, the operation fails if the method is not invoked.
- `atLeast(int minNumberOfInvocations)`: This method is called at least *n* times, and it works fine if the method is invoked more than the minNumberOfInvocations times. However, the operation fails if the method is not called minNumberOfInvocations times.
- `atMost(int maxNumberOfInvocations)`: This method is called at the most n times. However, the operation fails if the method is called more than minNumberOfInvocations times.
- `only()`: The only method called on a mock fails if any other method is called on the mock object.
- `timeout(int millis)`: This method is interacted in a specified time range.

```
Example:
verify(portfolio, new Times(0)).sell(otherStock, 10);
```

Verifying zero and no more interactions

The verifyZeroInteractions(Object... mocks) method verifies whether no interactions happened on the given mocks.

The following test code directly calls verifyZeroInteractions and passes the two mock objects. Since no methods are invoked on the mock objects, the test passes:

```
@Test
public void verify_zero_interaction() {
	verifyZeroInteractions(marketWatcher, portfolio);
}
```

The verifyNoMoreInteractions(Object... mocks) method checks whether any of the given mocks has any unverified interaction. We can use this method after verifying a mock method to make sure that nothing else was invoked on the mock.

```
// This test will fail, this is to demonstrate the error
@Test
public void verify_no_more_interaction() {
	Stock noStock = null;
	portfolio.getAvgPrice(noStock);
	portfolio.sell(null, 0);
	verify(portfolio).getAvgPrice(eq(noStock));
	// this will fail as the sell method was invoked
	verifyNoMoreInteractions(portfolio);
}
```

Why do we need wildcard matchers?

Wildcard matchers are used to verify the indirect inputs to the mocked dependencies. The following example describes the context.

In the following code snippet, an object is passed to a method and then a request object is created and passed to service. Now, from a test, if we call the someMethod method and service is a mocked object, then from test, we cannot stub callMethod with a specific request as the request object is local to the someMethod:

```
public void someMethod(Object object) {
        Request request = new Request(); // request is local, hence we can't stub callMethod
        request.setValue(object);
        Response response = service.callMethod(request);
}
```

If we are using argument matchers, all arguments have to be provided by matchers.

We're passing three arguments and all of them are passed using matchers:

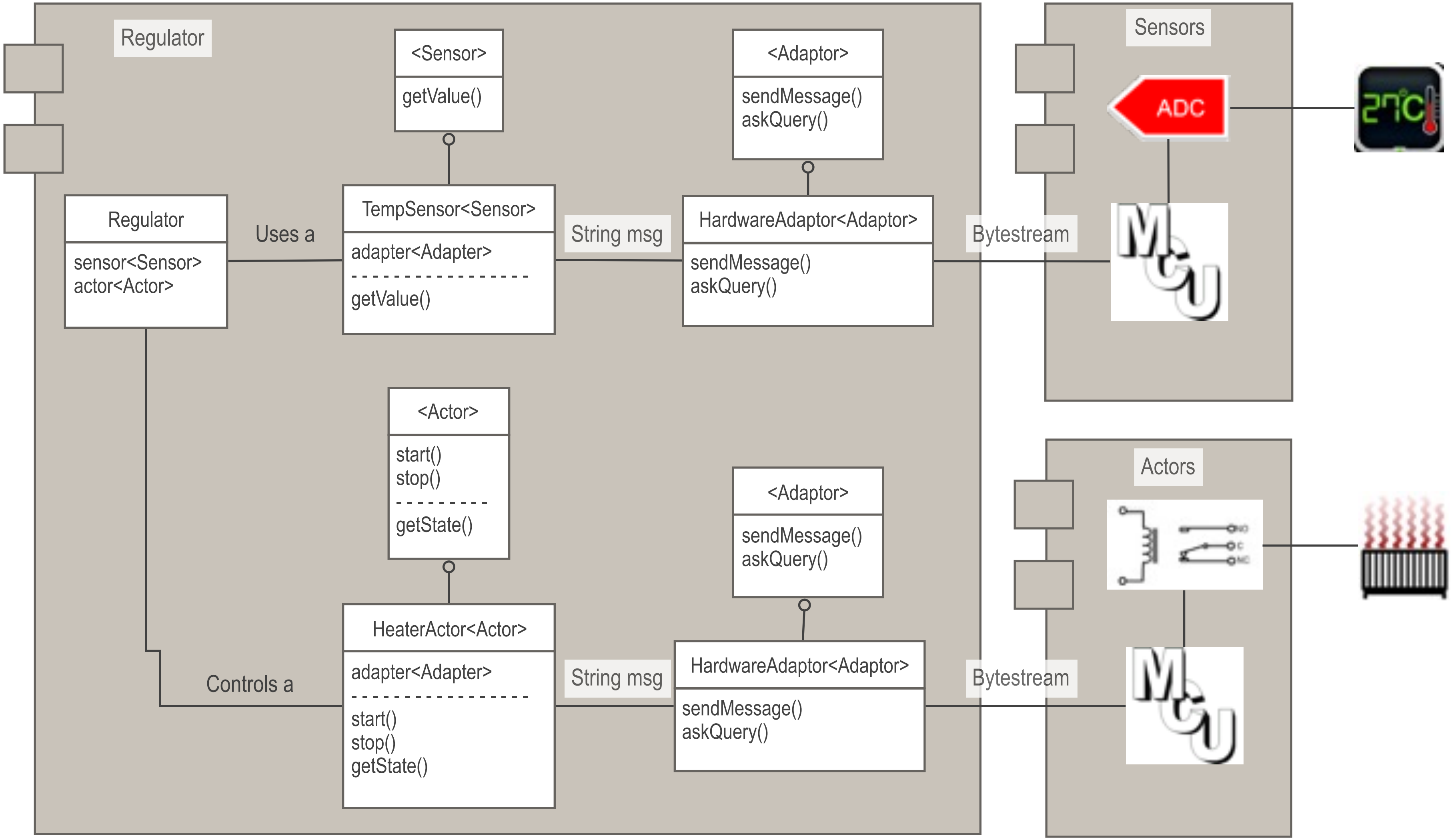
```
@Test
public void testSomeOtherMethod() {

        verify(mock).someOtherMethod(anyInt(), anyString(), eq("third argument"));

        verify(mock).someOtherMethod(1, anyString(), "third argument”); <-- will fail
}
```

The last row in the example will fail because the first and the third arguments are not passed using matcher:

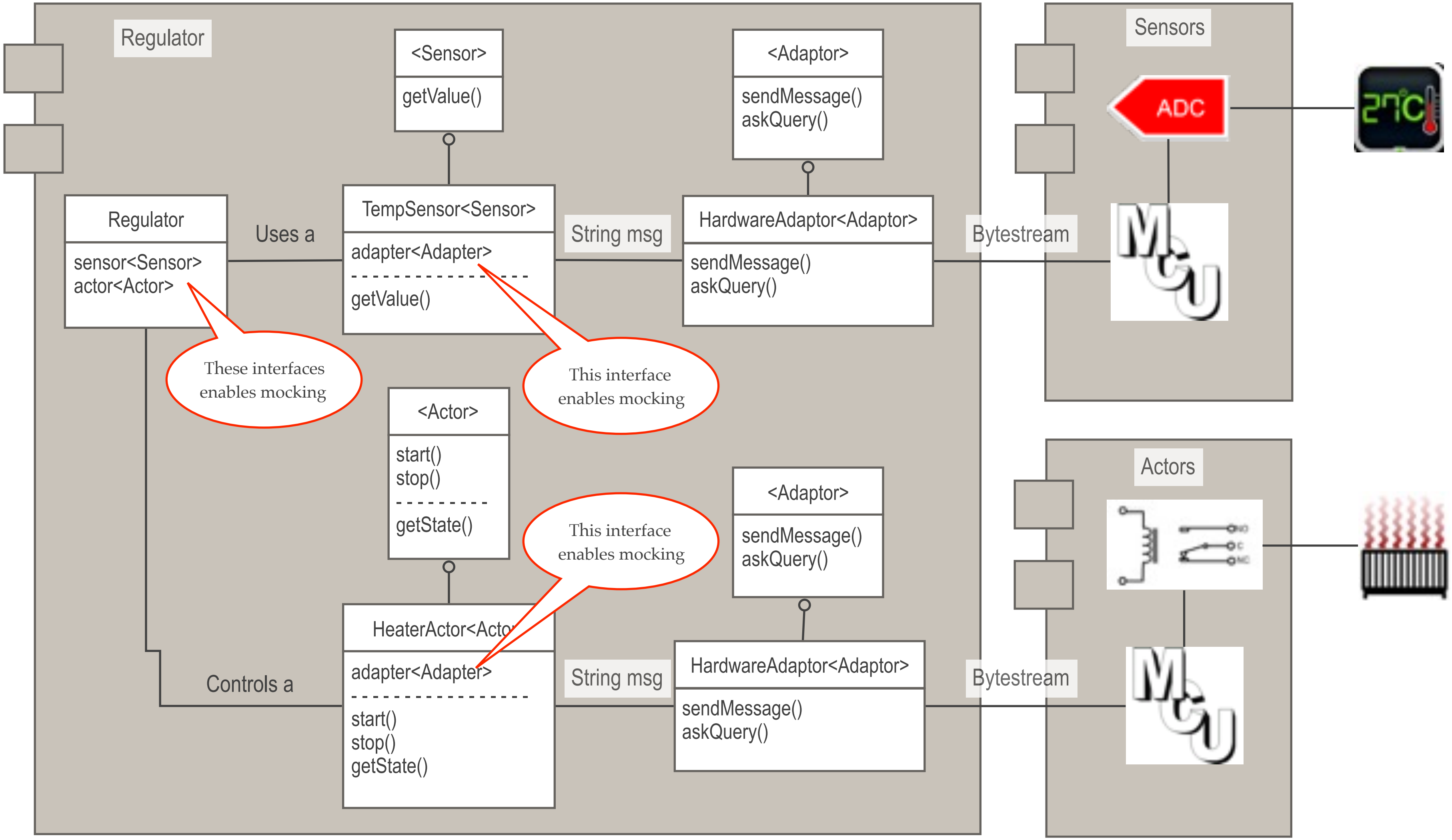
Mocking
a Realworld Example
(with plain Java)

Presentation provided by
Combitech JidokaQ

Real World Example



Design with interface



Regulator

<Sensor>

getValue()

<Adaptor>

sendMessage()
askQuery()

Regulator

sensor<Sensor>
actor<Actor>

Uses a

TempSensor<Sensor>

adapter<Adapter>

getValue()

String msg

HardwareAdaptor<Adaptor>

sendMessage()
askQuery()

Bytestream

These interfaces enables mocking

This interface enables mocking

<Actor>

start()
stop()

getState()

This interface enables mocking

<Adaptor>

sendMessage()
askQuery()

HeaterActor<Actor>

adapter<Adapter>

start()
stop()
getState()

Controls a

String msg

HardwareAdaptor<Adaptor>

sendMessage()
askQuery()

Bytestream

Sensors

ADC

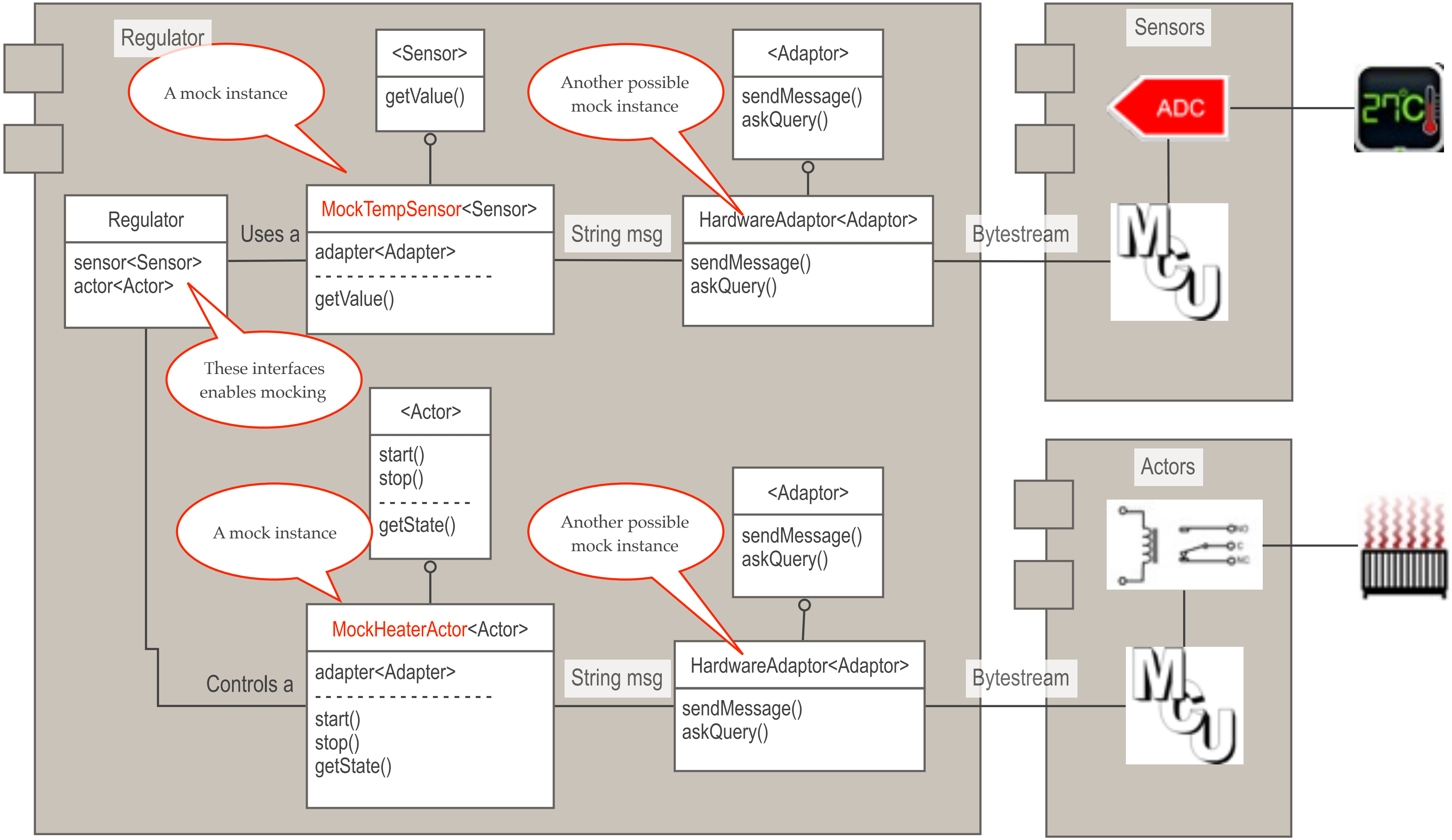
MCU

27°C

Actors

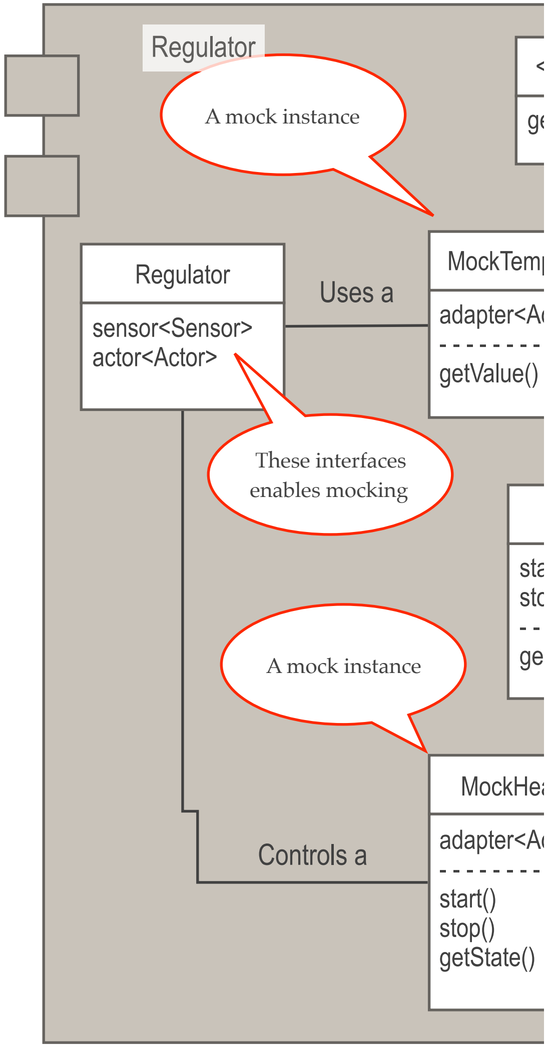
MCU

Replacing real classes with Mock



Regulator class

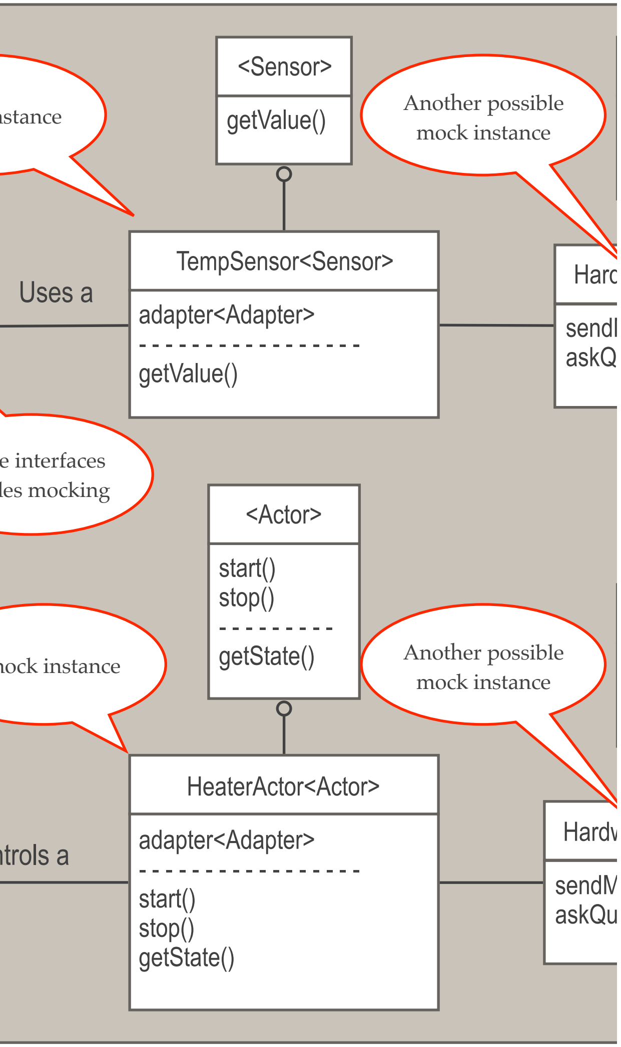
```
public interface Sensor {
    double getValue();
}
public interface Actor {
    void start();
    void stop();
    boolean getState();
}
```



```
public class Regulator {

    private Sensor temperatureSensor;
    private Actor heaterActor;
    private double lowThreshold;

    public Regulator(Sensor temperatureSensor, double lowThreshold,
Actor heaterActor) {
        this.temperatureSensor = temperatureSensor;
        this.lowThreshold = lowThreshold;
        this.heaterActor = heaterActor;
    }

    public void act() {
        double temperature = temperatureSensor.getValue();
        if (temperature < lowThreshold) {
            heaterActor.start();
        } else {
            heaterActor.stop();
        }
    }
}
```

TempSensor class

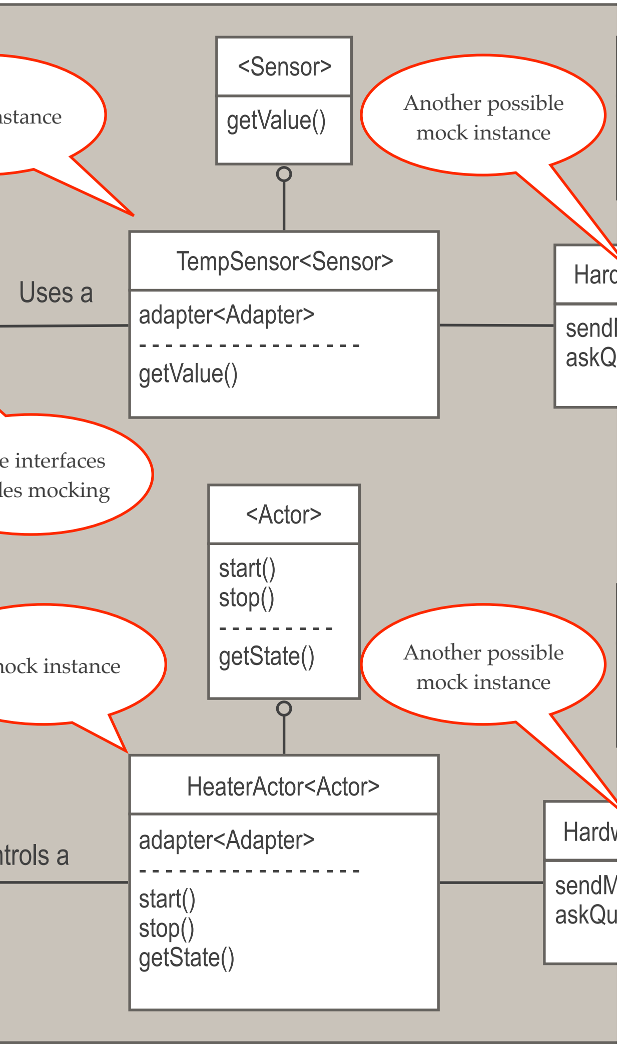
```
public interface Sensor {
    double getValue();
}
```



```
public class TempSensor implements Sensor {

    private Adapter adapter;

    public TempSensor(Adapter adapter) {
        this.adapter = adapter;
    }

    @Override
    public double getValue() {
        String response = adapter.askQuery("TempValue");
        double value = Double.parseDouble(response);
        return value;
    }
}
```

```
public interface Actor {
    void start();
    void stop();
    boolean getState();
}
```

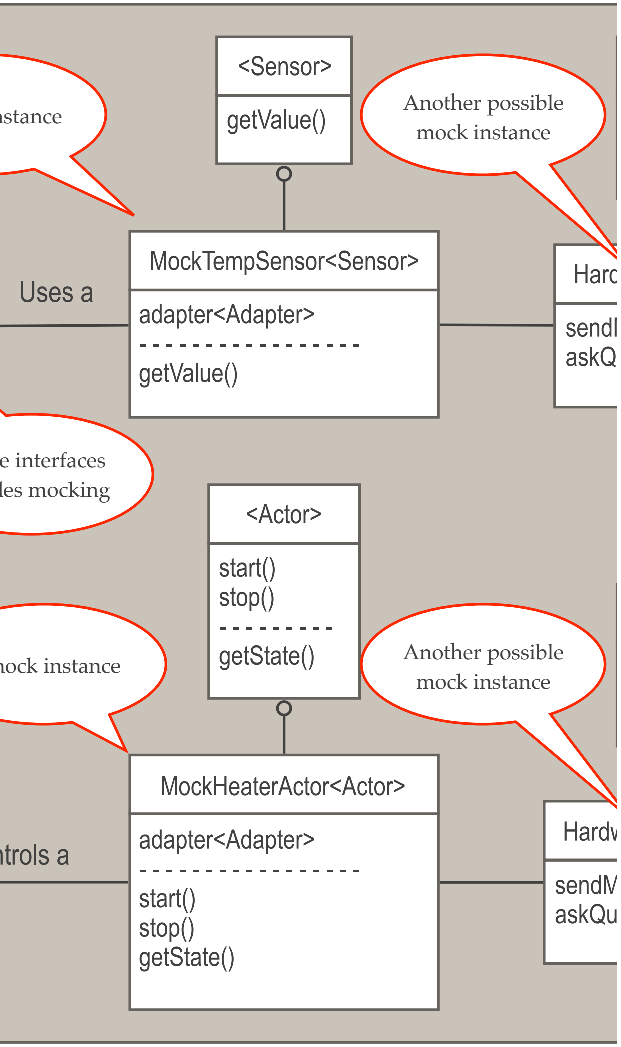
HeaterActor class



```
public class HeaterActor implements Actor {

    private Adapter adapter;

    public HeaterActor(Adapter adaptert) {
        this.adapter = adaptert;
    }
    @Override
    public void start() {
        adapter.sendMessage("start");
    }
    @Override
    public void stop() {
        adapter.sendMessage("stop");
    }
    @Override
    public boolean getState() {
        String state = adapter.askQuery("state");
        return state.equalsIgnoreCase("on");
    }
}
```

MockTempSensor class

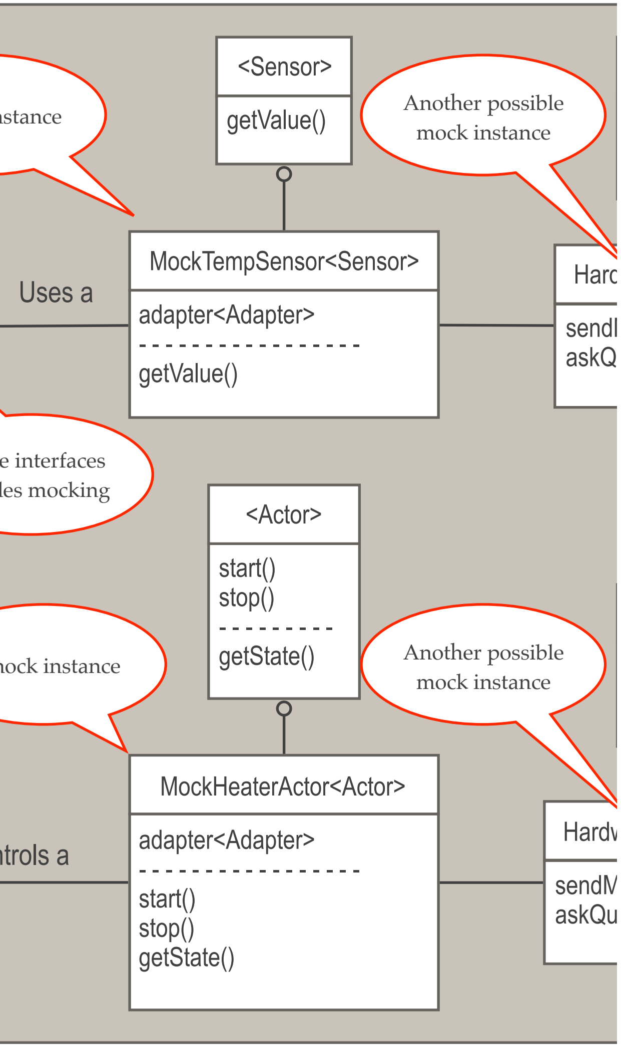
```
public interface Sensor {
    double getValue();
}
```



```
public class MockTempSensor implements Sensor {

    private double actualTemperature;

    public MockTempSensor(double actualTemperature) {
        this.actualTemperature = actualTemperature;
    }

    @Override
    public double getValue() {
        return actualTemperature;
    }
}
```

MockHeaterActor class

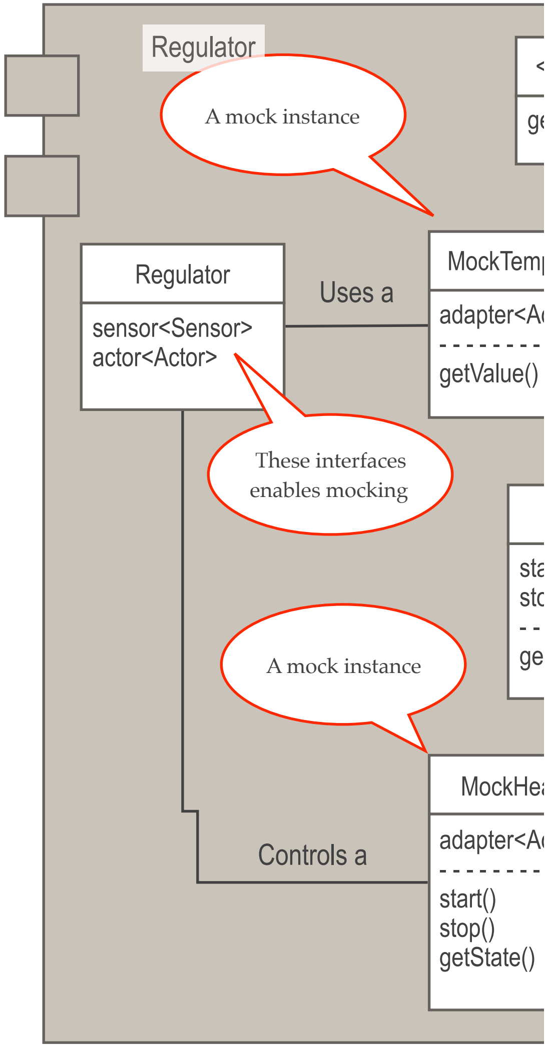
```
public interface Actor {
    void start();
    void stop();
    boolean getState();
}
```



```
public class MockHeaterActor implements Actor {

    private boolean state = false;;

    @Override
    public void start() {
        state  = true;
    }
    @Override
    public void stop() {
        state = false;
    }
    @Override
    public boolean getState() {
        return state;
    }
}
```

RegulatorTest class

```
public interface Sensor {
    double getValue();
}
public interface Actor {
    void start();
    void stop();
    boolean getState();
}
```



```
@Test
public void testTemperatureBelowLowThresholdShouldStartHeater() {

    // Arrange: Set up a regulator with a mocked sensor
     // and a mocked heater
    double actualTemperature = 0.0f;
    Sensor temperatureSensor = new MockTempSensor(actualTemperature);

    Actor heaterActor = new MockHeaterActor();

    Regulator regulator = new Regulator(temperatureSensor,
                  lowThreshold, heaterActor);

    // Act: Run one round in regulator action
    regulator.act();

    // Assert that the heater has started
    boolean heater = heaterActor.getState();
    assertTrue("Expect to have the heater on!", heater);
}
```

- Project MockingHardwareMockito demo
  - Show: `RegulatorTest.java`
  - Show: Mocking on Sensor level and on Adapter level

Mocking
a Realworld Example
(with Mockito)

Presentation provided by
Combitech JidokaQ

- Project MockingHardwareMockito demo
  - Show: `RegulatorMockitoTest.java`
  - Show: `RegulatorHysteresisTest.java`

- Hysteresis Mock demo, to show arrays of test data-result vectors