JidokaQ [ji-do-ka-ku]

TDD
(Test Driven Development)

Goals of this presentation

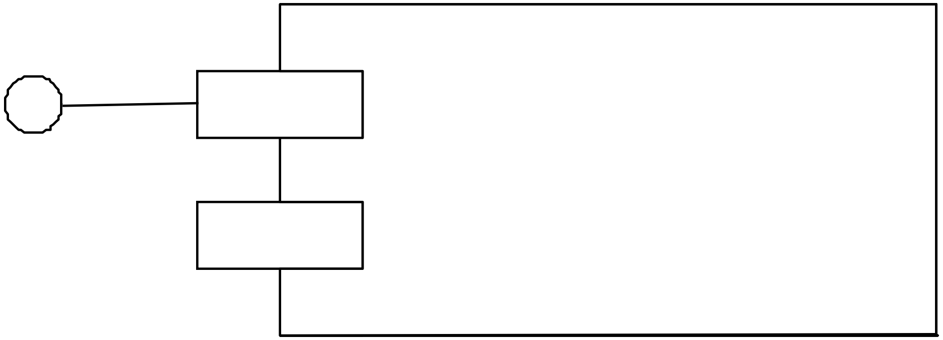
- Automated Testing Theory (Skipped)
  - To introduce and motivate Automated Tests
  - To describe how Automated Testing fits within a Software Development Process
  - To provide a classification of Automated Testing Strategies and Tools
- Unit Testing
  - Provide theoretical background for Unit Testing
  - Hands-on experience with Unit Testing tools and frameworks
- Mocking
  - Provide theoretical background for Mocking and some examples
- Test Driven Development (TDD)
  - Provide theoretical background for TDD and some examples
- Integration Testing
  - Provide theoretical background for and overview of Integration Testing

Introduction to Unit Testing with JUnit and Eclipse

Training provided from CLL
Combitech

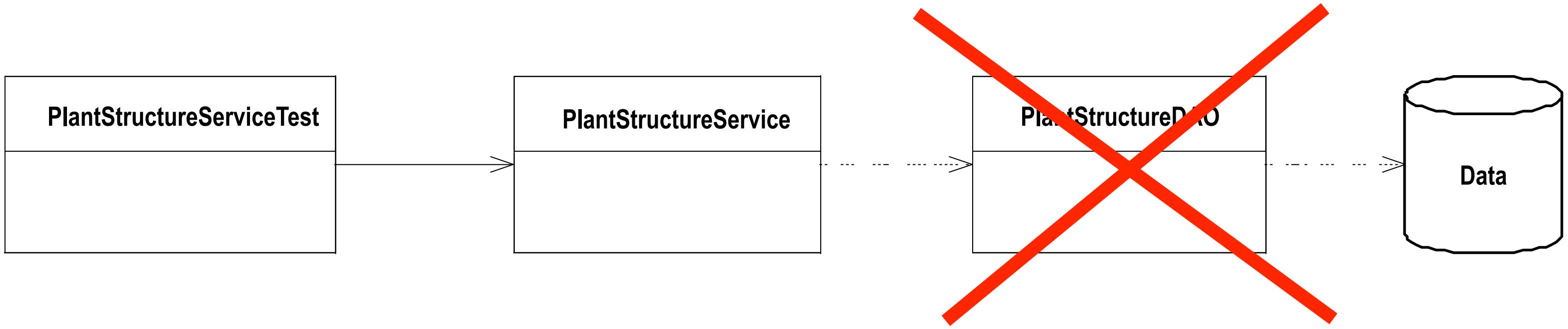
Unit Tests

- Black-box or White-box test of a *logical unit*, which verifies that the logical unit behaves correctly – *honors its contract*.



What exactly is a Unit Test?

- A self-contained software module (in OO languages typically a Class) containing one or more test scenarios which tests a Unit Under Test in isolation.
- Each test scenario is autonomous, and tests a separate aspect of the Unit Under Test.



Smoke Tests

- A set of Unit Tests (which tests a set of logical units) executed as a whole provides a way to perform a *Smoke Test*: Turn it on, and make sure that it doesn't come smoke out of it!
- A relatively cheap way to see that the units "seems to be working and fit together", even though there are no guarantees for its overall function (which requires functional testing)

Developer testing vs Acceptance testing

- Unit Tests are written by developers, for developers.

- Unit Tests do not address formal validation and verification of correctness (even though it has indirect impact on it!) - Unit Tests prove that some code does what we intended it to do

- Unit Tests complements Acceptance Tests (it does not replace it)

Why should I (as a Developer) bother?

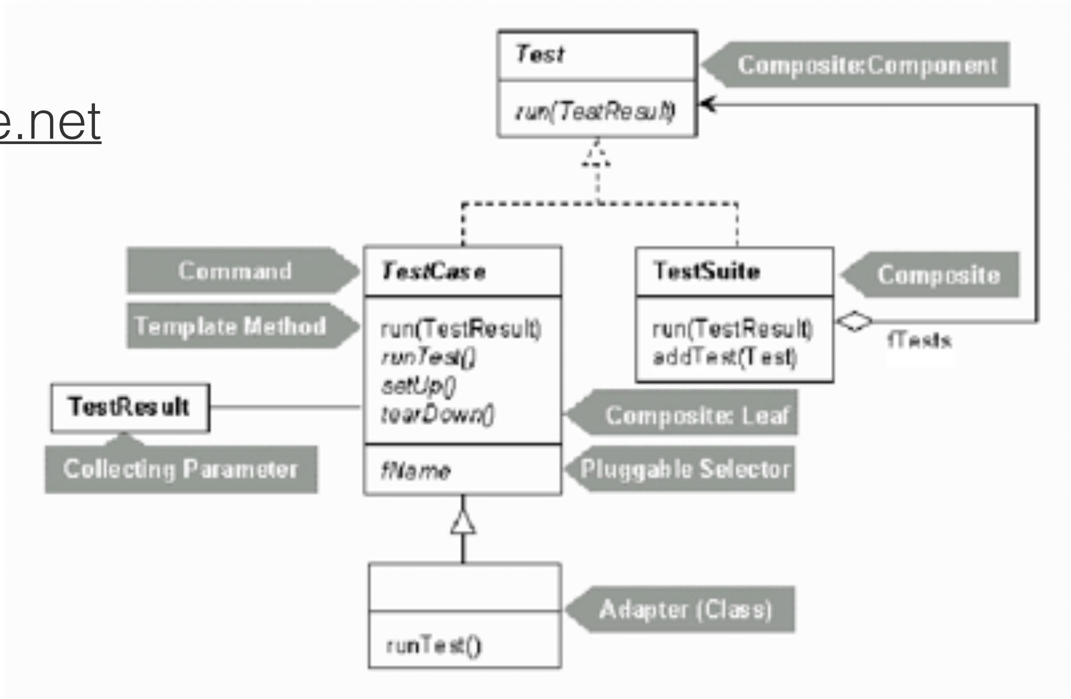
- Well-tested code works better. Customers like it better.
- Tests support refactoring. Since we want to ship useful function early and often, we know that we'll be evolving the design with refactoring.
- Tests give us confidence. We're able to work with less stress, and we're not afraid to experiment as we go.
- Hence Unit Testing will make my life easier
  - It will make my design better
  - It will give me the confidence needed to refactor when necessary
  - It will dramatically reduce the time I spend with the debugger
  - It will make me sleep better when deadlines are closing in

Requirements on Unit Tests

- Easy to write a test class
- Easy to find test classes
- Easy to test different aspects of a contract
- Easy to maintain tests
- Easy to run tests

XUnit: A Framework for Unit Tests

- www.junit.org
- www.csunit.org
- www.vbunit.org
- cppunit.sourceforge.net



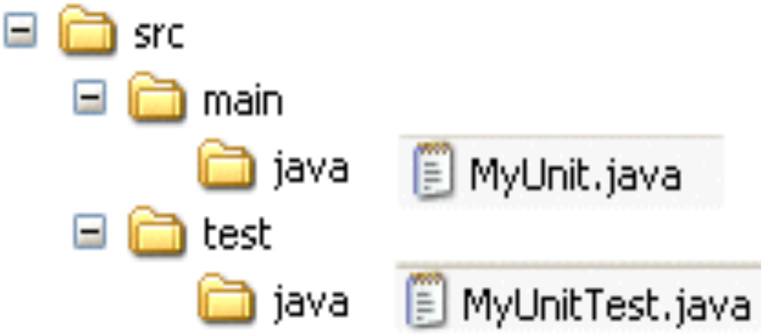
JUnit Test Example

```
public interface Account {
   public void deposit(int amount);
   public void withdraw(int amount) throws AccountException;
   public int getBalance();
      …
}

public class AccountImplTest {
    @Test
    public void testWithdraw() throws AccountException {
        AccountImpl account = new AccountImpl(“1234-9999”, 2000);
        account.withdraw(300);
        Assert.assertEquals(1700, account.getBalance());
     }

    @Test
   public void testWithdrawTooMuch() throws AccountException { … }
     …
}
```

Naming Conventions and Directory Structure

- Unit Tests should be named after the Unit that is tested, with "Test" appended.
  A class usually represents a noun, it is a model of a concept. An instance of one of your tests would be a 'MyUnit test'. In contrast, a method would model some kind of action, like 'test [the] calculate [method]'.

- the MyUnit test --> MyUnitTest
- test the calculate method --> testCalculate()
- JUnit tests should be placed within the same Java package as the Unit under Test, but in a different directory structure.



Test cases and test methods

```
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class AccountImplTest {
    @Test
    public void testWithdraw() throws AccountException {
        AccountImpl account = new AccountImpl(“1234-9999”, 2000);
        account.withdraw(300);
        Assert.assertEquals(1700, account.getBalance());
    }

    @Test
   public void testWithdrawTooMuch() throws AccountException { … }
    …
}
```

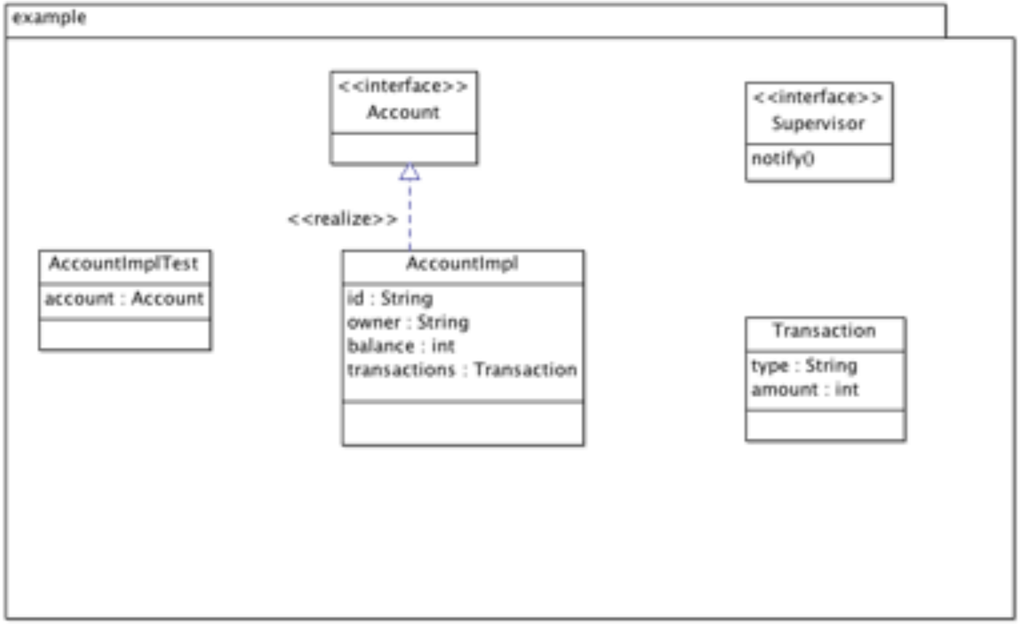
All methods annotated with @Test are considered test scenarios

Assert: Support for verifying conditions

```
static void assertEquals(int expected, int actual);
          // Asserts that two ints are equal.

static void assertEquals(double expected, double actual, double delta);
          // Asserts that two doubles are equal concerning a delta.

static void assertEquals(java.lang.Object expected, java.lang.Object actual);
          // Asserts that two objects are equal.

static void assertFalse(java.lang.String message, boolean condition);
          // Asserts that a condition is false.

static void assertTrue (java.lang.String message, boolean condition);
          // Asserts that a condition is true.

static void assertNull(java.lang.String message, java.lang.Object object);
          // Asserts that an object is null.

static void assertNotNull(java.lang.String message, java.lang.Object object);
          // Asserts that an object isn't null.

// Etc…
```

Exercise 1 description

Sometimes you need to write Unit tests to already existing software when you want to implement a change request e.t.c. In this example we have the source code but no tests, your task is to write them.



Exercise 1

- Create an Unit test case which tests the initial balance of an Account (i e. tests the constructor and GetBalance() method of Account).

```
@Test
public void testInitialBalance() { … }
```

- Add tests for the Deposit() method of Account.

```
@Test
public void testDeposit() { … }
```

Typical unit test scenario – The Three A's

1. **A**rrange - Instantiate Unit under Test and set up test data

2. **A**ct - Execute one or more methods on the Unit Under Test

3. **A**ssert - Verify the results

```
public interface Account {
   public void deposit(int amount);
   public void withdraw(int amount) throws AccountException;
   public int getBalance();
    …
}
public class AccountImplTest {
     @Test
     public void testWithdraw() throws AccountException {
          AccountImpl account = new AccountImpl(“1234-9999”, 2000);  // ARRANGE
          account.withdraw(300);                                     // ACT
          Assert.assertEquals(1700, account.getBalance());           // ASSERT
      }
        … … …
}
```

General Rules of Thumb

- Create a single test class for each non-trivial application class you have.
- Give a readable, meaningful name to each test method. A good name candidates are to name the test method using the same name as the method that it is testing, with some additional info appended to the name. For instance if testing a method called "***Withdraw***" in an Account class, create a few test methods to test different ways of withdrawal:

```
@Test
public void testWithdrawTooMuch() throws AccountException {…}
@Test
public void withdrawBigAmount() throws AccountException {…}
@Test
public void withdrawNegativeAmount() throws AccountException {…}
```

- The scope of how much checking to do in a single test case (test method) is a judgment call. It is usually better to test only one scenario (and hence one potential error condition) in each test method. Remember : tests should be “to the point”.

Setup and teardown

- Methods annotated with **@Before** are executed *before every test method*.
- Methods annotated with **@After** are executed *after every test method*.

```
public class AccountImplTest {

        private AccountImpl account;

        @Before
        public void setUp() {
                account = new AccountImpl(“1234-9999", 2000);
        }
        @Test
        public void testInitialBalance() {
                int actualBalance = account.getBalance();
                Assert.assertEquals(2000, actualBalance);
        }
        @Test
        public void testWithdraw() throws AccountException {
                account.withdraw(300);
                int actualBalance = account.getBalance();
                Assert.assertEquals(1700, actualBalance);
        }
    …
}
```

Working with Exceptions

- Unexpected exceptions thrown during execution of a test will be caught by the JUnit framework and reported as Errors (i.e. test will fail)
- A Test method must declare that it throws any checked exceptions that the Unit under Test may throw. If there are several checked exceptions that may occur, it is perfectly valid for a test method to declare throwing java.lang.Exception.
- Expected exceptions (exceptions that the test is expecting the Unit under Test should throw in a certain situation) are expressed using the **`@Test(expected=ExpectedException.class)`** attribute

```
@Test(expected=NastyException.class)
public void doSomethingNastyTest() {
    SomeUnit target = new SomeUnit();
    target.doSomethingNasty();
}
```

Working with Exceptions (Contd.)

- Or using the following idiom:

```
SomeUnit target = new SomeUnit();
try {
   target.doSomethingNasty();
   Assert.fail("NastyException expected");
} catch (NastyException expected) {
   // Expected
}
```

Ignore a Test

- To temporary ignore a test, use the Ignore attribute:

```
@Test
@Ignore("Not right now, but most definitely later")
public void testThatDoesNotWorkYet(){
    SomeUnit target = new SomeUnit();
    target.doSomethingThatDoesNotWork();
    Assert.assertTrue(target.isValid());
}
```

Exercise 2

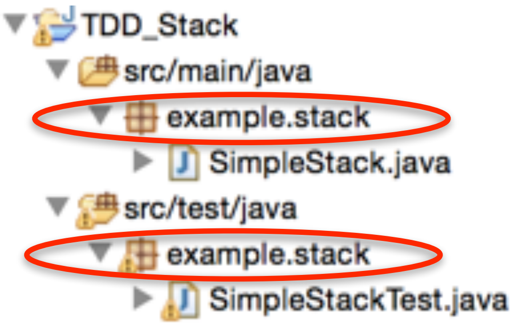
- Refactor your test data from the last example into a @Before annotated setUp() method
- Add tests for the withdraw() method.

Testing private or protected methods/members

JUnit will only test those methods in my class that are public or protected, but…

In principle you got four options

- Don't test private methods. (Good or Bad?)
- Give the methods package-private access. (Good or Bad?)
- Use a inner class or anonymous class. (Does it work?)
- Use reflection. (Is this good?)



```
TDD_Stack
  src/main/java
    example.stack
      SimpleStack.java
  src/test/java
    example.stack
      SimpleStackTest.java
```

http://stackoverflow.com/questions/34571/whats-the-proper-way-to-test-a-class-with-private-methods-using-junit

Testing private or protected methods/members

The best way to test a private method is via another public method. If this cannot be done, then one of the following conditions is true:

1. The private method is dead code.
2. There is a design smell near the class that you are testing.
3. The method that you are trying to test should not be private.

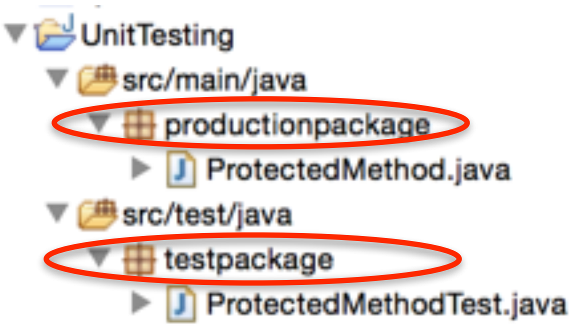
When I have private methods in a class that is sufficiently complicated that I feel the need to test the private methods directly, that could be a code smell: my class is too complicated.

But, it might also be SDK or Framework code or Security or encryption/decryption code. That type of code also need tests, but no publicity…

Testing protected methods (Java)

- Protected methods are visible by default when using the same parallel package structure for tests, but if in different packages, it will not work!

```java
package productionpackage;
public class ProtectedMethod {
     protected String myProtectedMethod (String s) {
          return "MyClass: " + s;    }
}

package testpackage;
public class ProtectedMethodTest {

     @Test
     public void testProtectedMethod() {
          String expected = "MyClass: Hello";
          ProtectedMethod unitUnderTest = new ProtectedMethod();
          String actual = unitUnderTest.myProtectedMethod("Hello");
          boolean equal = actual.equalsIgnoreCase(expected);
          Assert.assertTrue("Strings not equal", equal);
     }
}
```



```
▼ UnitTesting
  ▼ src/main/java
    ▼ productionpackage
      ▶ ProtectedMethod.java
  ▼ src/test/java
    ▼ testpackage
      ▶ ProtectedMethodTest.java
```

Will not work!

Testing protected methods (Java)

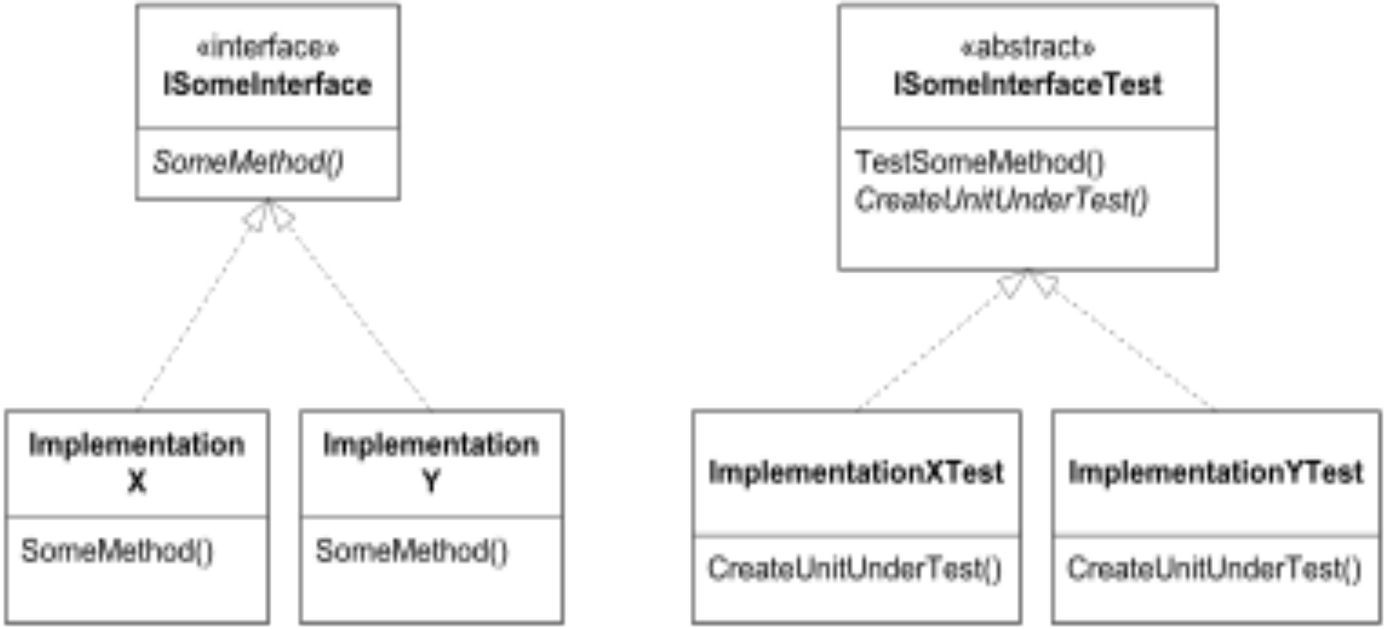
The Subclass and Override idiom is used to write unit tests for protected methods:

```
package productionpackage;
public class ProtectedMethodClass {
        protected String protectedMethod (String s) {
                return "Protected: " + s;    }
}

package testpackage;
public class ProtectedMethodClassTest {

        // Create an inner class to expose the protected method
        class ExposeProtectedMethodClass extends ProtectedMethodClass {
                public String exposeProtectedMethod(String s) {
                        return super.protectedMethod(s);
                }
        }
        @Test
        public void testProtectedMethod() {
                String expected = "Protected: Hello";
                ExposeProtectedMethodClass unitUnderTest = new ExposeProtectedMethodClass();
                String actual = unitUnderTest.exposeProtectedMethod("Hello");
                boolean equal = actual.equalsIgnoreCase(expected);
                Assert.assertTrue("Strings not equal", equal);
        }
}
```

We can live with this since the exposure is done in test package, that will be stripped out in the production code!

Testing Interfaces or Abstract Classes (Java only)

- Sometimes, you want to write tests for an Interface or Abstract Class, and have those tests executed against all implementations.
- Specify the tests in an Abstract Test class, with one concrete Test class for each concrete implementation



Testing Interfaces - Java example

```
package testpackage;
import org.junit.*;

public abstract class AbstractSomeInterfaceTest {
    private SomeInterface unitUnderTest;
    @Before
    public void setUp() {
        unitUnderTest = implementSomeInterfaceTest();
    }
    @Test
    public void testSomeMethodReturnsTrue () {
        Assert.assertTrue("someMethod() should return true", unitUnderTest.someMethod());
    }
    protected abstract SomeInterface implementSomeInterfaceTest();
}
```

```
package testpackage;
public class ImplementationX implements SomeInterface {
        @Override
        public boolean someMethod() {
                return false;
        }
}
```

```
public class ImplementationXTest extends AbstractSomeInterfaceTest {
      @Override
      protected SomeInterface implementSomeInterfaceTest() {
         return new ImplementationX();
      }
}
```

Instances like this one, will run automatically according to test scheme in the abstract class.

What should be tested?

- Everything that could possibly break!
- Corollary: Don’t test stuff that is too simple to break!
- Typical problematic areas:
  - Boundary conditions
    - **C**onformance
    - **O**rdering
    - **R**ange
    - **R**eference
    - **E**xistence
    - **C**ardinality
    - **T**ime
  - Error conditions

Exercise 4

Given the following interface for a fax sender service:

```
/* Send the named file as a fax to the given phone number.
* Phone numbers should be of the form 0nn-nnnnnn where n is
* digit in the range [0-9]
*/
public boolean SendFax(String phone, String filename) {
    . . .
}
```

What tests for boundary conditions can you think of?

Breaking dependencies and Mocking

Training provided from CLL
Combitech

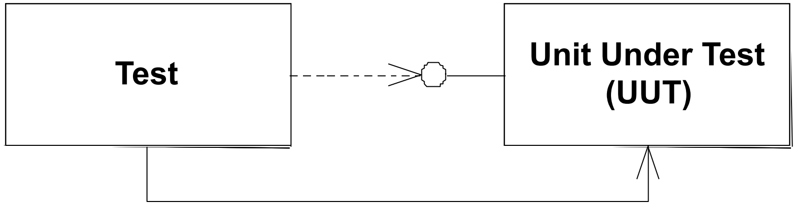
Design properties and Design goals

For Units:

- Modularity
- High cohesion
- Low coupling

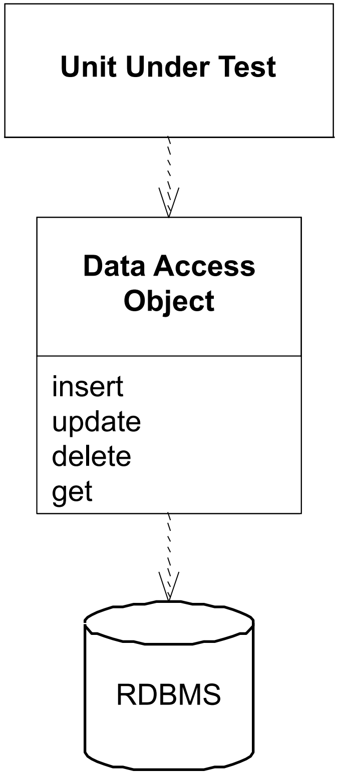
For Tests:

- Modularity
- Locality

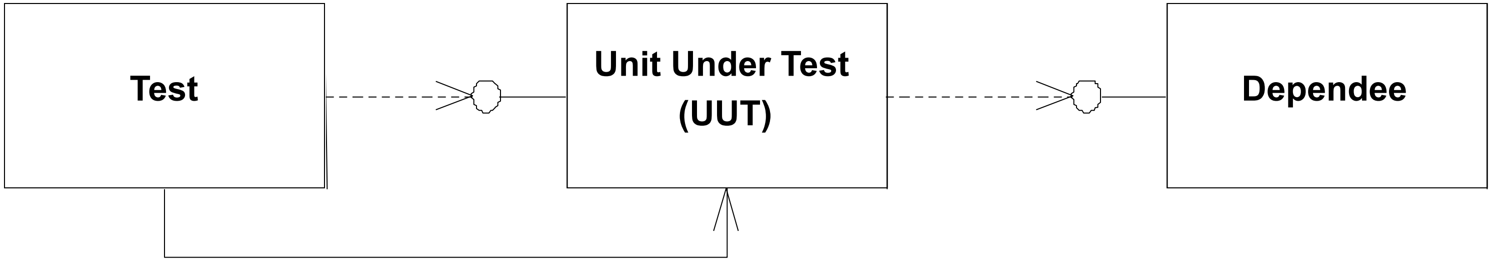


Side effects

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



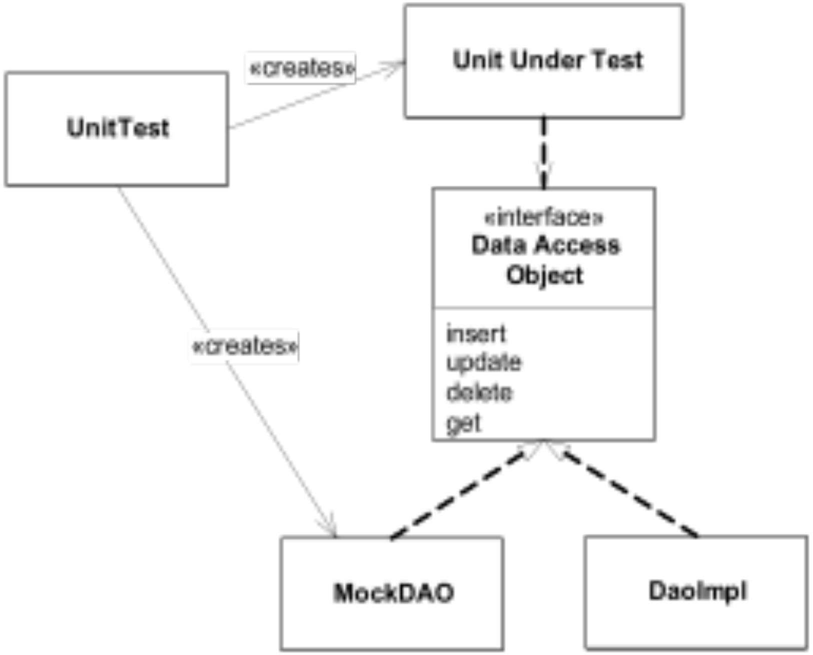
Strategies for testing Units that depend on other units



- 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

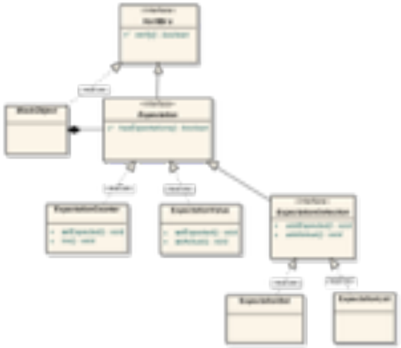
Synthetic context – Mocking

- 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



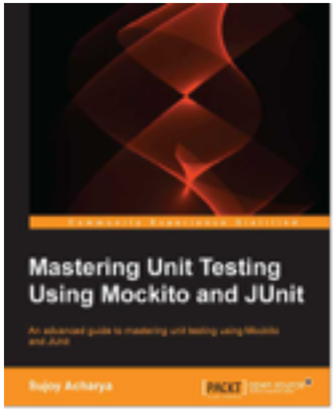
Frameworks and tools for mocking

- code.google.com/p/mockito/ (Active 2015)
  - No expect-run-verify also means that Mockito mocks are often ready without expensive setup upfront
- www.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
- www.easymock.org (Active 2015)
  - Class library which generates Mock Objects dynamically using the Java Proxy class



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 ham crest matchers



Mastering Unit Testing Using Mockito and JUnit

Sujoy Acharya

PACKT

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

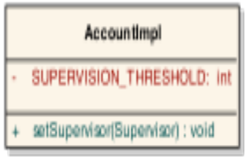
Mockito - example usage

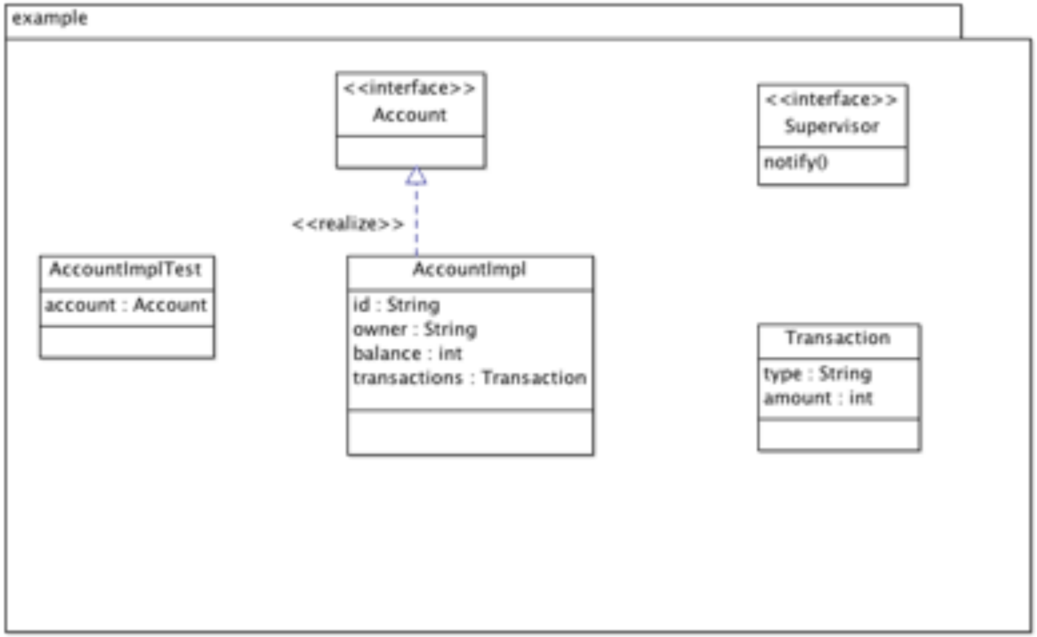
```java
@Test
public void testNotificationVetoShouldBeHonoured() {
    int amount = AccountImpl.SUPERVISION_TRESHOLD;

    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



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

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();
     }
     ...
 }
```

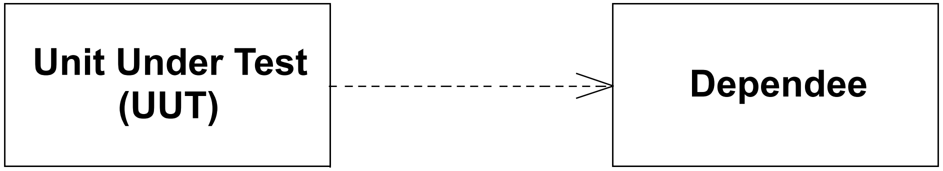
Law of Demeter (Contd.)

- becomes

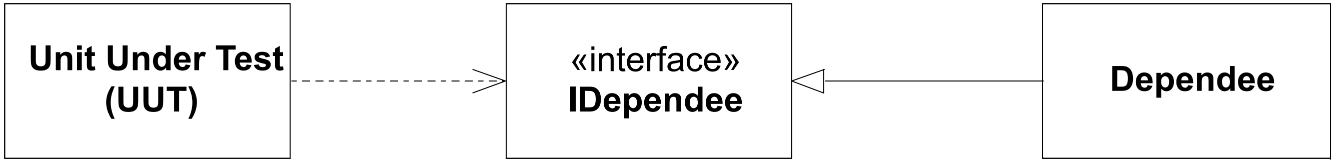
```
public class SomeUnit
    {
           private IDependee dependee;
           public SomeUnit()
           {
           }
           public SetDependee(IDependee dependee)
           {
               this.dependee = dependee;
           }
           ...
    }
```

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!*

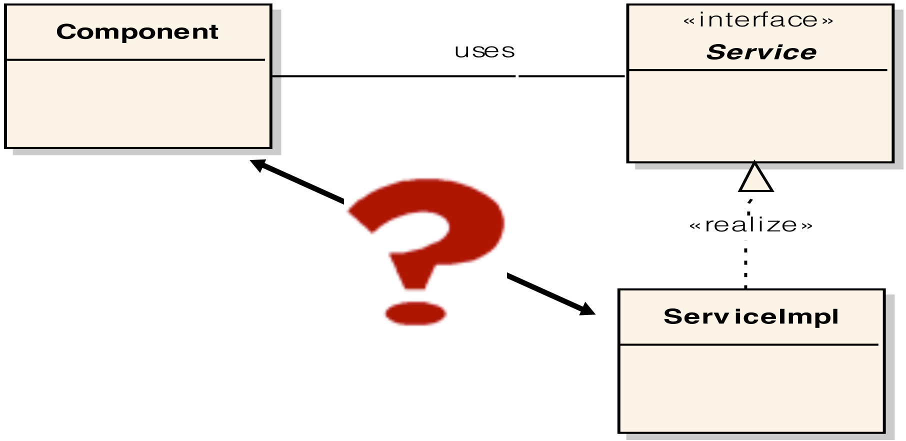


becomes



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



Introduction to TDD (Test-Driven Development)

Training provided from CLL
Combitech

Test-Driven Development

Unit Tests may be written very early. In fact, they may even be written before any production code exists:

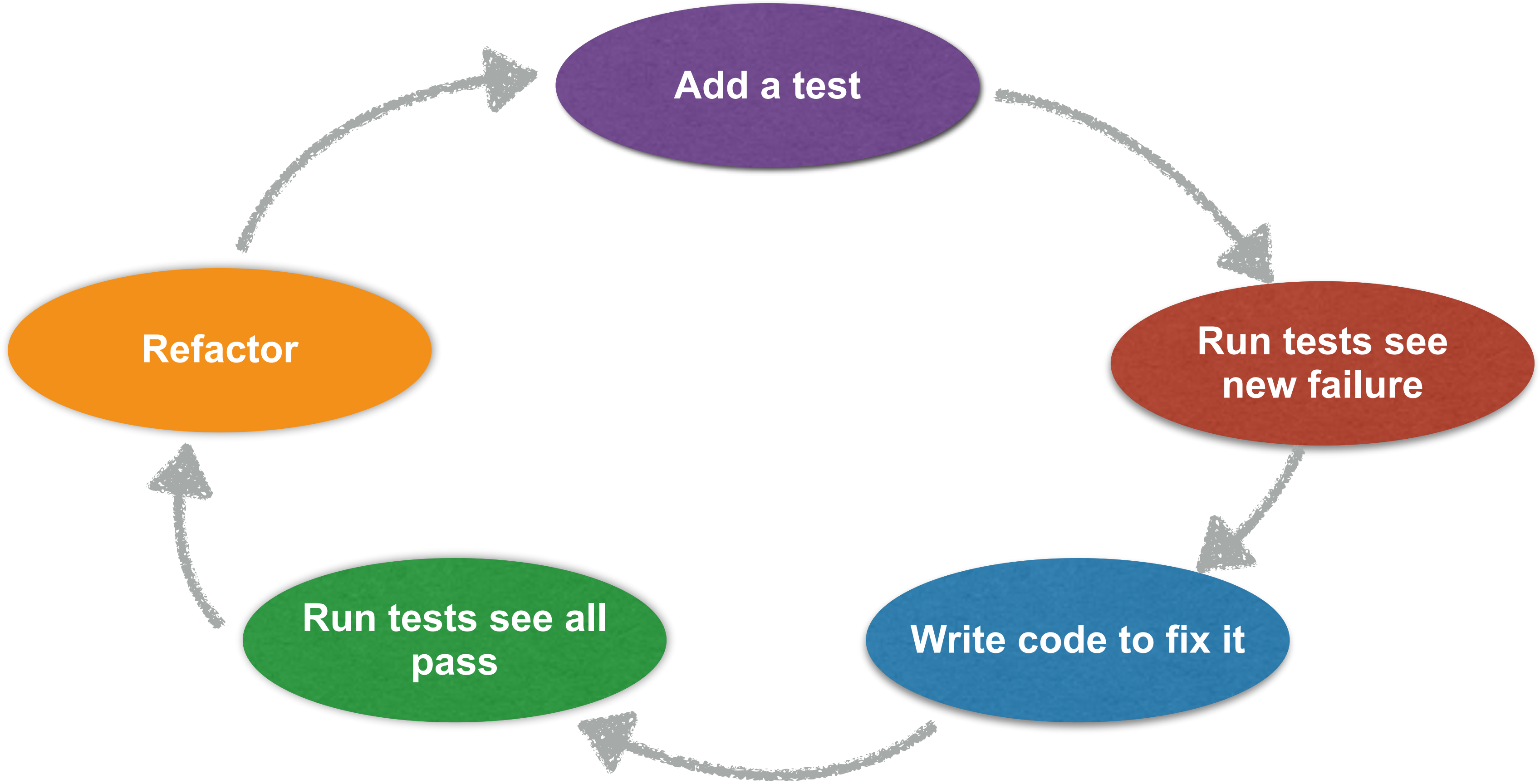
- Write a test that specifies a tiny bit of functionality
- Ensure the test fails (you haven't built the functionality yet!)
- Write the code necessary to make the test pass
- Refactoring the code to remove redundancy

There is a certain rhythm to it: Design a little – test a little – code a little – design a little – test a little – code a little – ...

Test-Driven Development process

1. Think about what you want to do.
2. Think about how to test it.
3. Write a small test. Think about the desired API.
4. Write just enough code to fail the test.
5. Run and watch the test fail (and you'll get the "**Red Bar**").
6. Write just enough code to pass the test (and pass all your previous tests).
7. Run and watch all of the tests pass (and you'll get the "**Green Bar**").
8. If you have any duplicate logic, or inexpressive code, **refactor** to remove duplication and increase expressiveness.
9. Run the tests again (you should still have the "Green Bar").
10. Repeat the steps above until you can't find any more tests that drive writing new code.

Test-Driven Development process (TDD process)



Add a test

Run tests see new failure

Write code to fix it

Run tests see all pass

Refactor

Simple Design

- “Simplicity is more complicated than you think. But it’s well worth it.”

  Ron Jeffries

- Satisfy Requirements
  - No Less
  - No More

You can use your developer intuition to find best choice

Simple Design Criteria

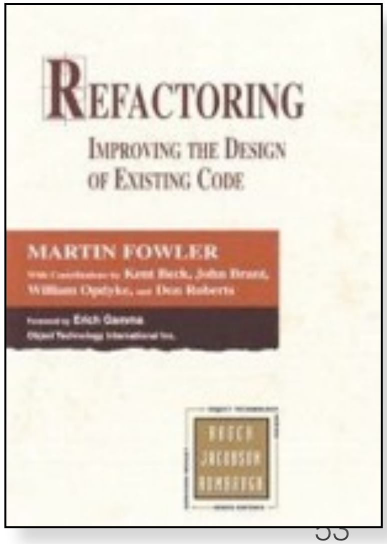
- In Priority Order
  - The code is appropriate for the intended audience
  - The code passes all the tests
  - The code communicates everything it needs to
  - The code has the smallest number of classes
  - The code has the smallest number of methods

Should we then have all code in one class and only have the one method the “main”-method?

Of course not, but why?

Refactoring

- Definition: Improve the code without changing its functionality
- Code needs to be refined as additional requirements (tests) are added
- For more information see *Refactoring: Improve the Design of Existing Code* – Martin Fowler



REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma

Object Technology International Inc.

Working Breadth First - Using a Test List

- Work Task Based
  - 4-8 hour duration (maximum)
- Brainstorm a list of developer tests
- Do not get hung up on completeness… you can always add more later
- Describes completion requirements

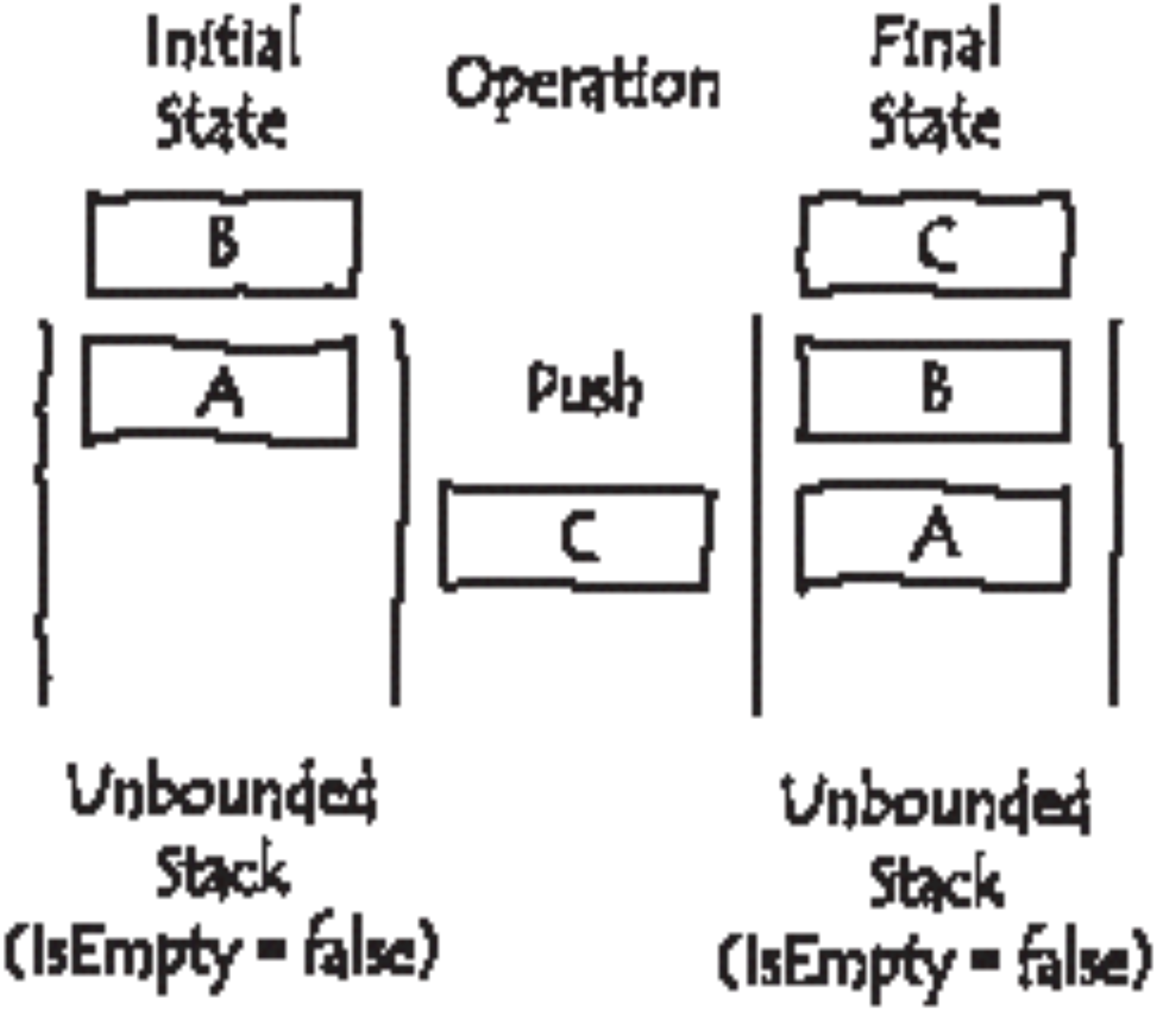
Exercise 5

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.”

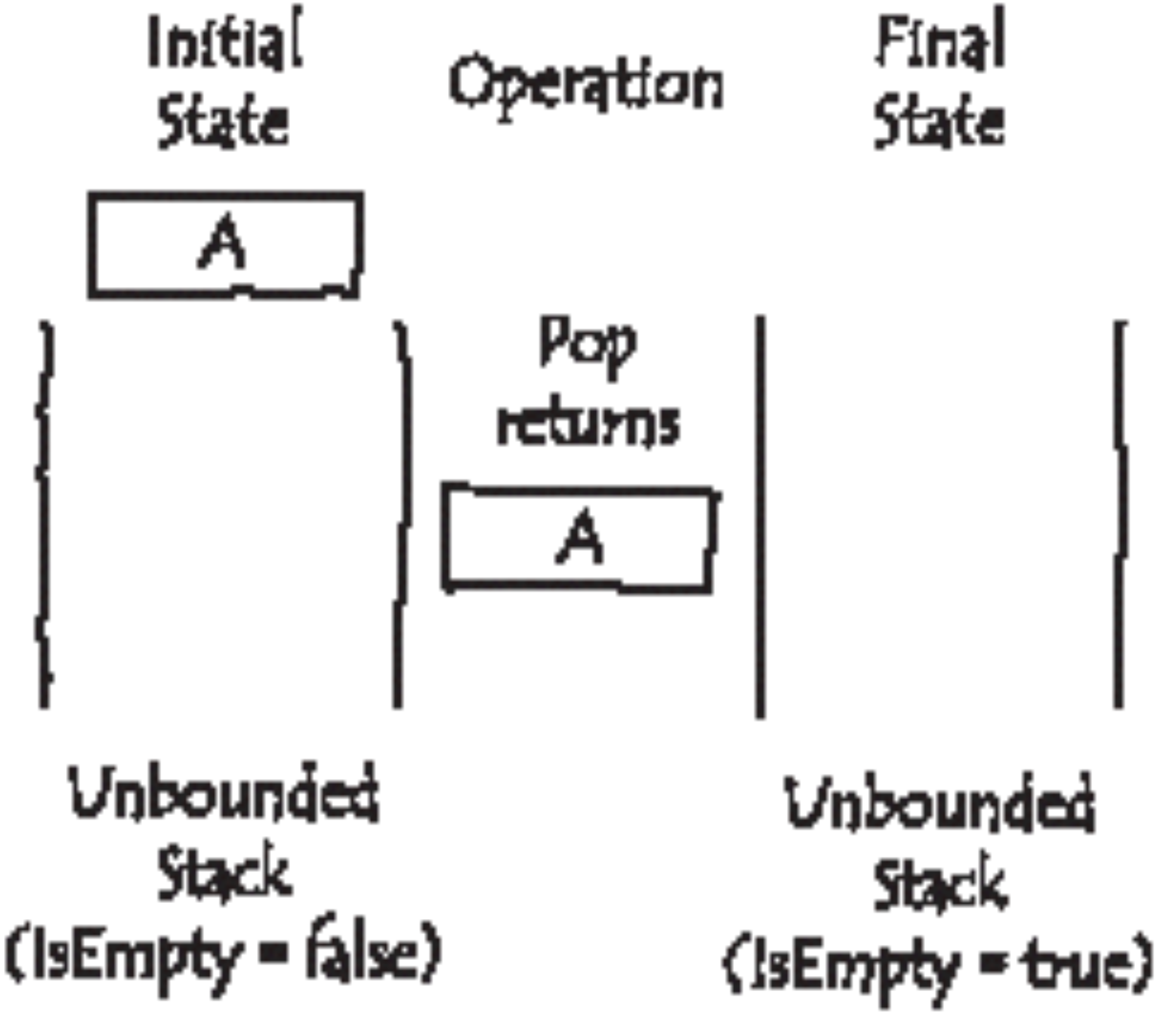
Remember: Every single line of production code written must be motivated by a failing test!

Push operation

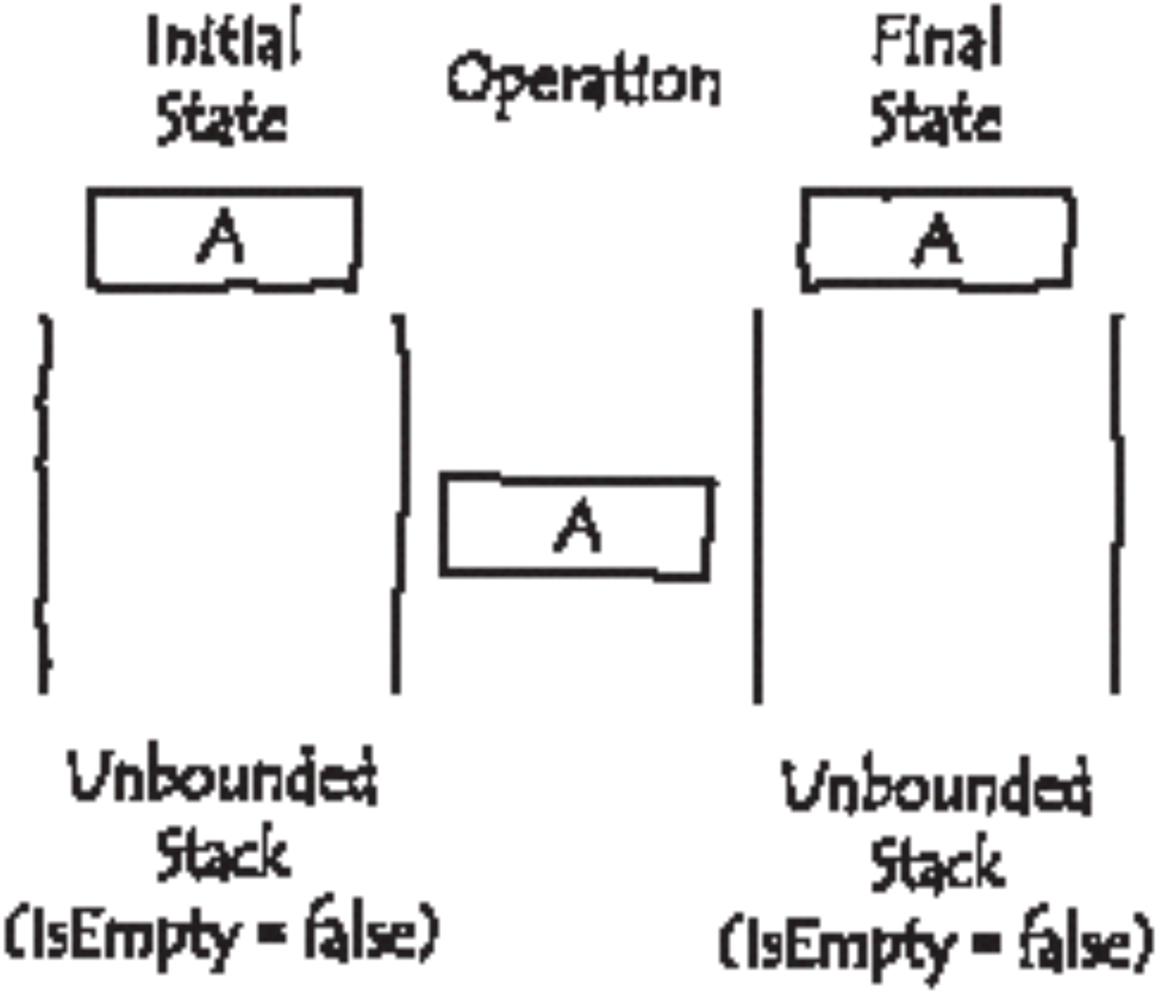


Initial State
Operation
Final State
B
A
Push
C
C
B
A
Unbounded Stack (IsEmpty = false)
Unbounded Stack (IsEmpty = false)

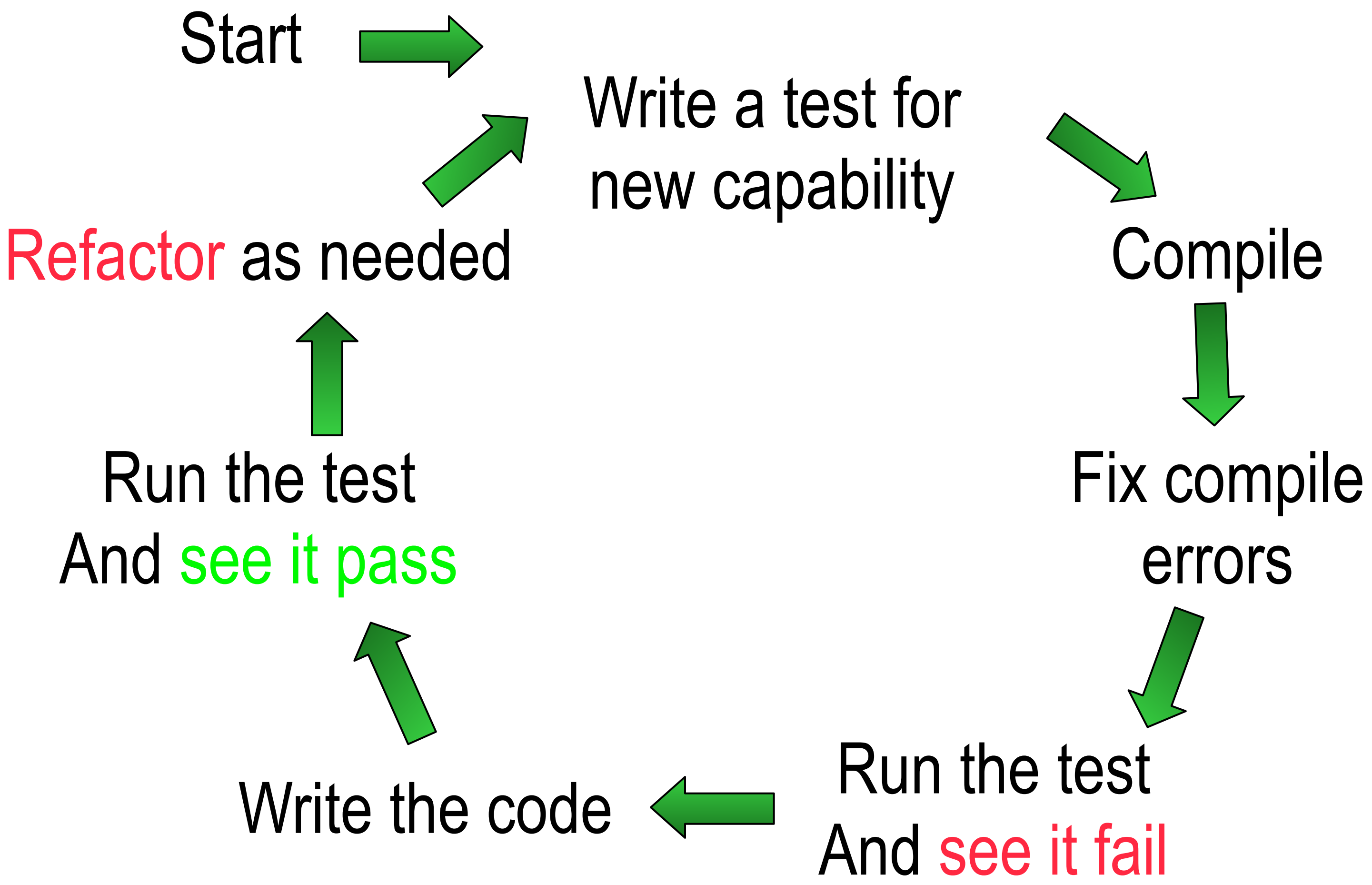
Pop operation


Initial State
Operation
Final State
A
Pop returns
A
Unbounded Stack (IsEmpty = false)
Unbounded Stack (IsEmpty = true)

Top operation


Initial State
Operation
Final State
A
A
A
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

Solution 5 - Step 1a

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

```java
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;              // See it work!
   }
   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();
      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 {
   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)**

```java
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);
}
```

Solution 5 - Step 6b (The production code)

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

```
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() – 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**

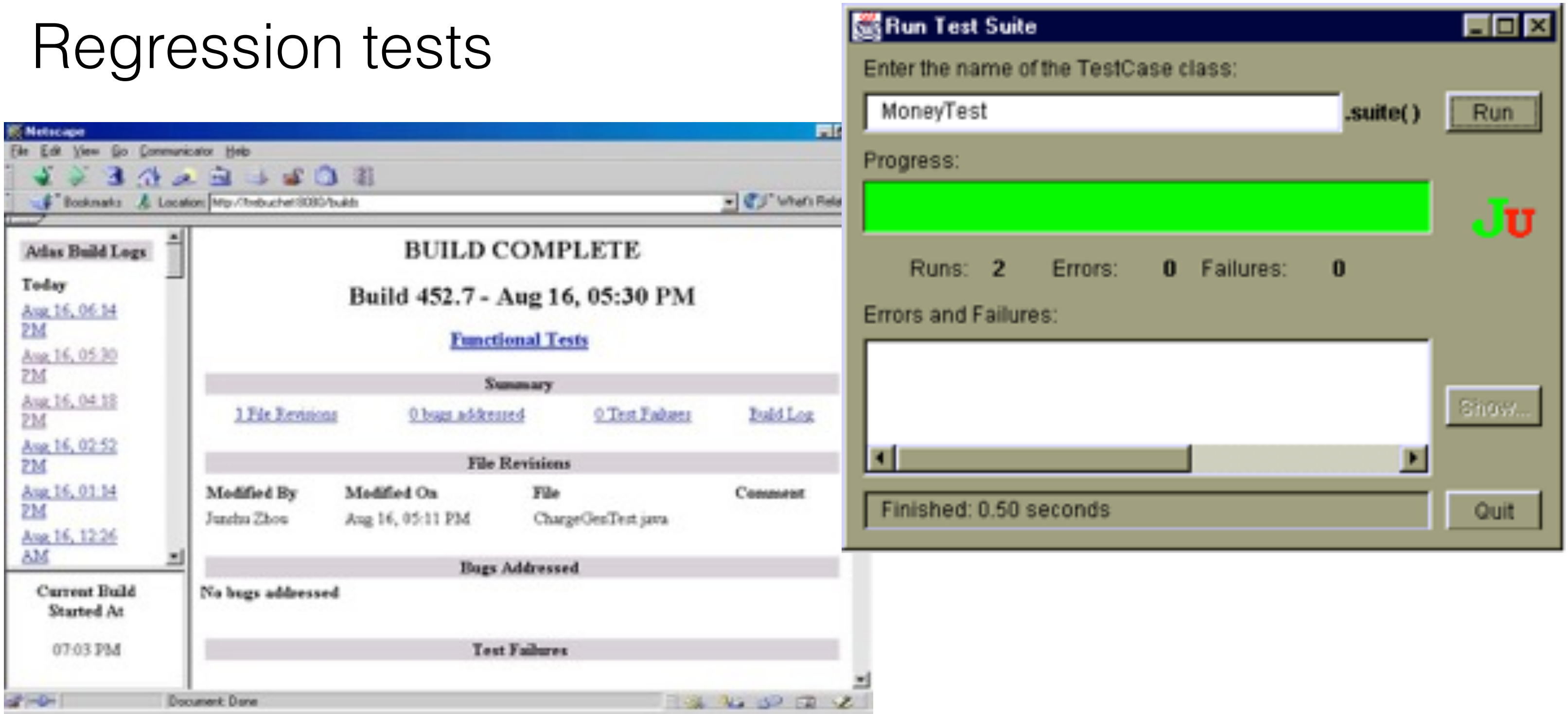
```
@Test
public void testEmptyStackTop() {
   try {
      @SuppressWarnings("unused")
      int top = stack.top();
      Assert.fail("IllegalStateException expected");
   } catch (java.lang.IllegalStateException e) {
      // Expected
   }
}
```

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

```
@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());
}
```

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 write means we spend less time debugging. We get our programs done faster.
- Testing as we write means that we don't have those long testing cycles at the end of our projects. We like working without that death march thing.
- Our tests are the first users of our code. We experience what it is like to use our code very quickly. The 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 week points of TDD ?

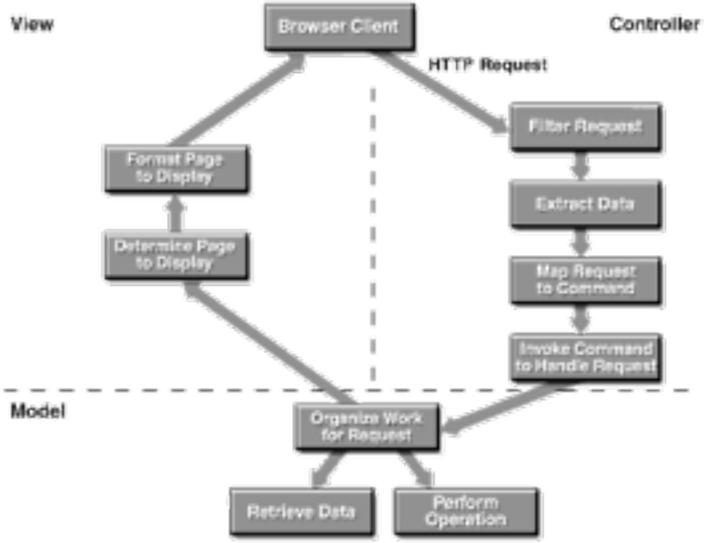
- When test code, very intensively, use production API then it can impact the ability to refactor.
- Buggy tests – tests that is failing because of bugs in the test themselves.
- It will simply not be worth it, when cost for maintenance of the tests will be higher than benefits.

Hey there!

We are developers and should strive to mitigate these week points, shouldn’t we?

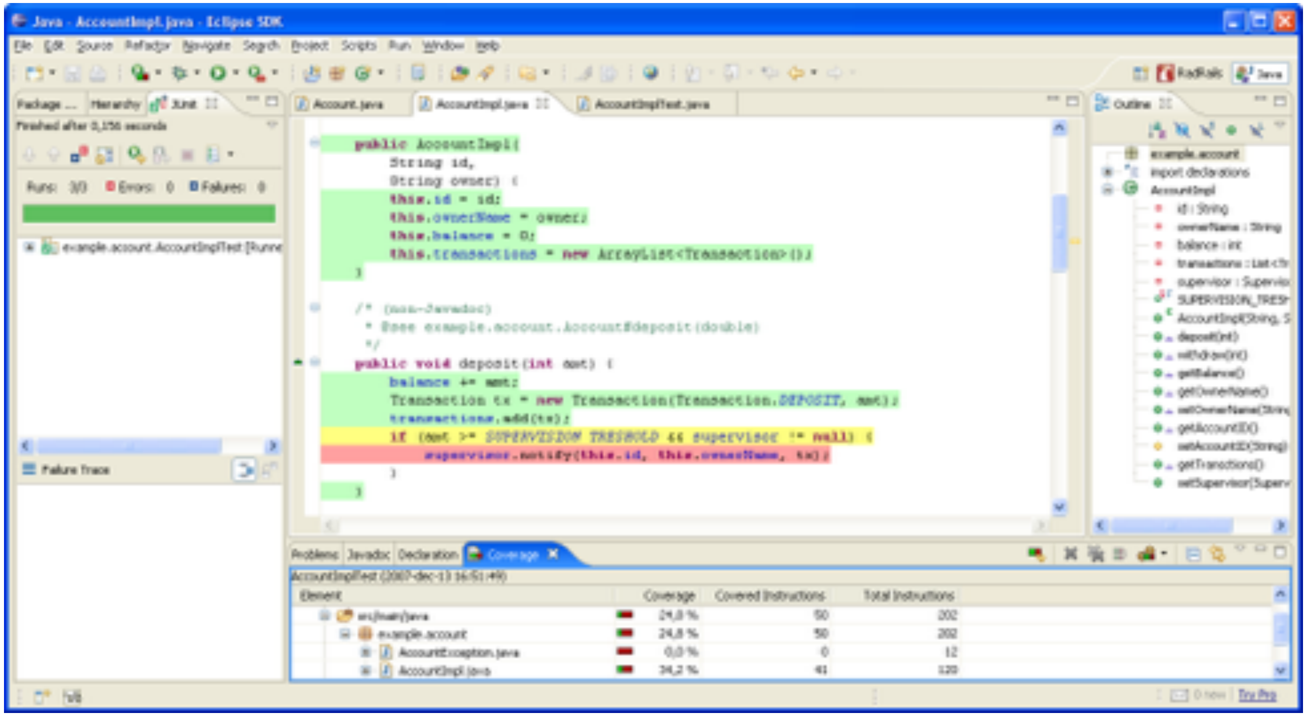
Designing for Testability: Model-View-Control

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



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