Introduction to Unit Testing with JUnit and Eclipse Part 2

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?)

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.

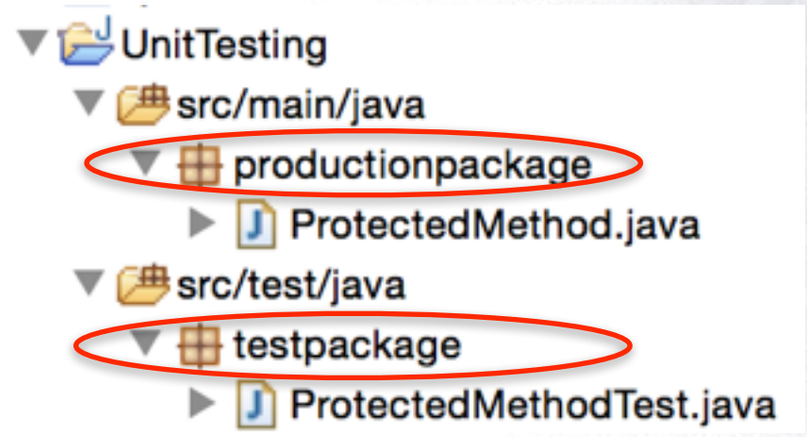
When I have private methods in a class that is sufficiently complicated that I feel the need to test the private methods directly, that could be a code smell: my class is too complicated.

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!

```
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);
    }
}
```

Will not work!

Testing protected methods (Java)

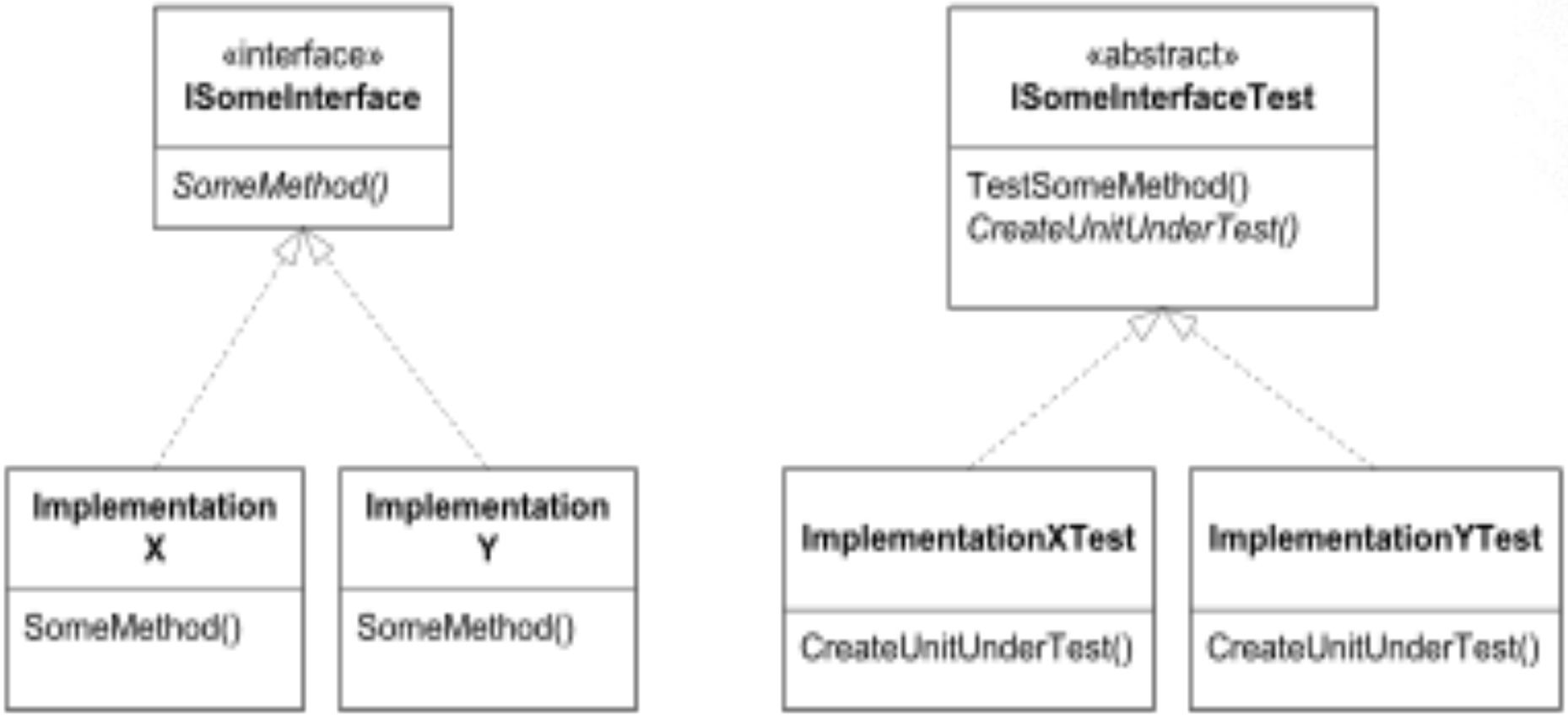
The Subclass and Override idiom is used to write unit tests for protected methods:

```
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

```java
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();
}

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

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

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

What should be tested?

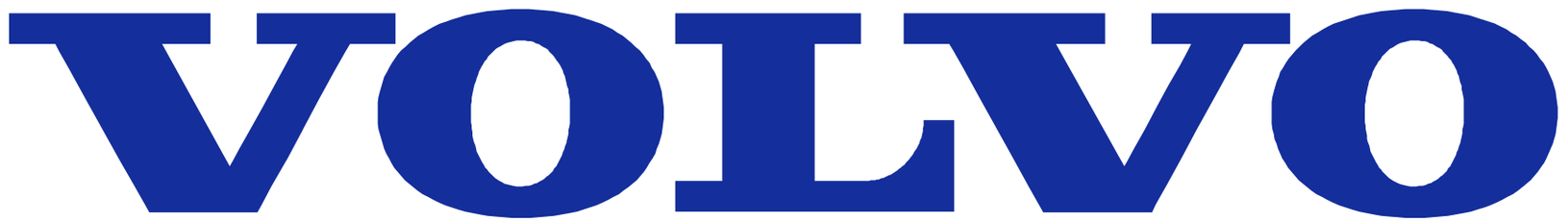
- Everything that could possibly break!
- Corollary: Don’t test stuff that is too simple to break!
- Typical problematic areas:
  - Boundary conditions
    - **C**onformance
    - **O**rdering
    - **R**ange
    - **R**eference
    - **E**xistence
    - **C**ardinality
    - **T**ime

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?



TDD

Breaking Dependencies

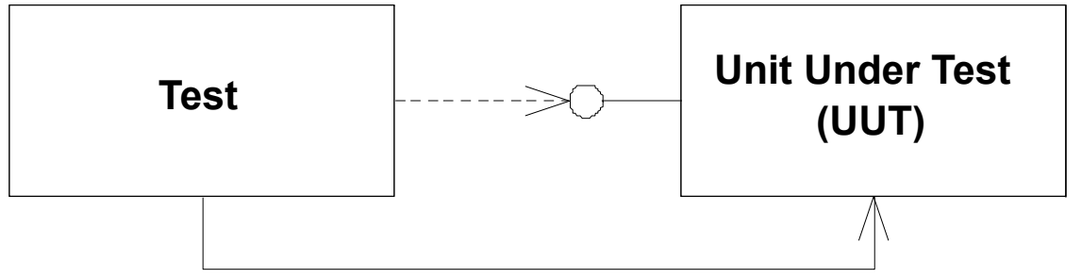
Design properties and Design goals

For Units:

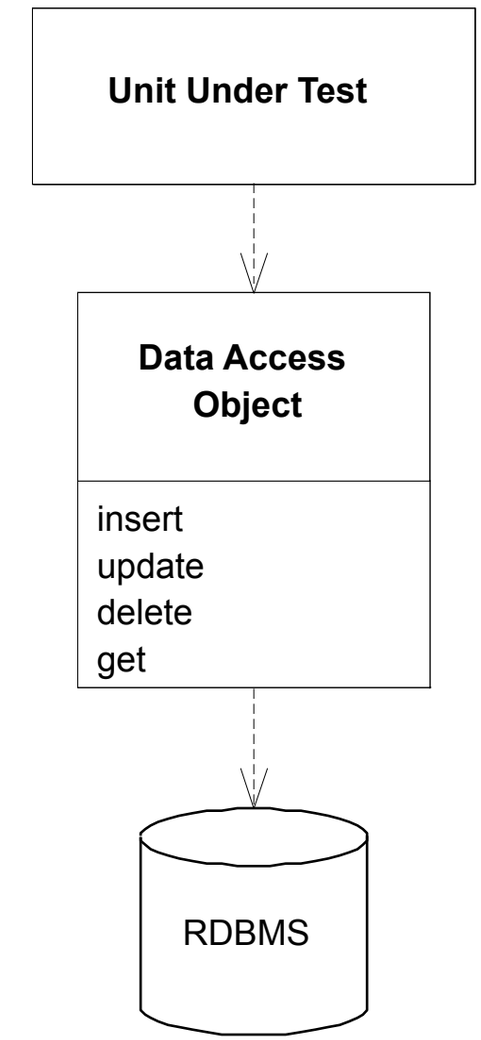
- Modularity
- High cohesion
- Low coupling

For Tests:

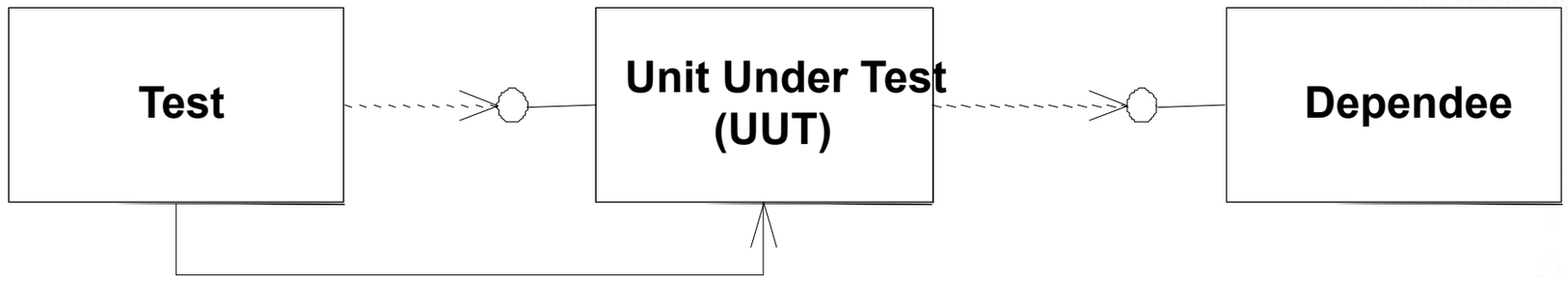
- Modularity
- Locality



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



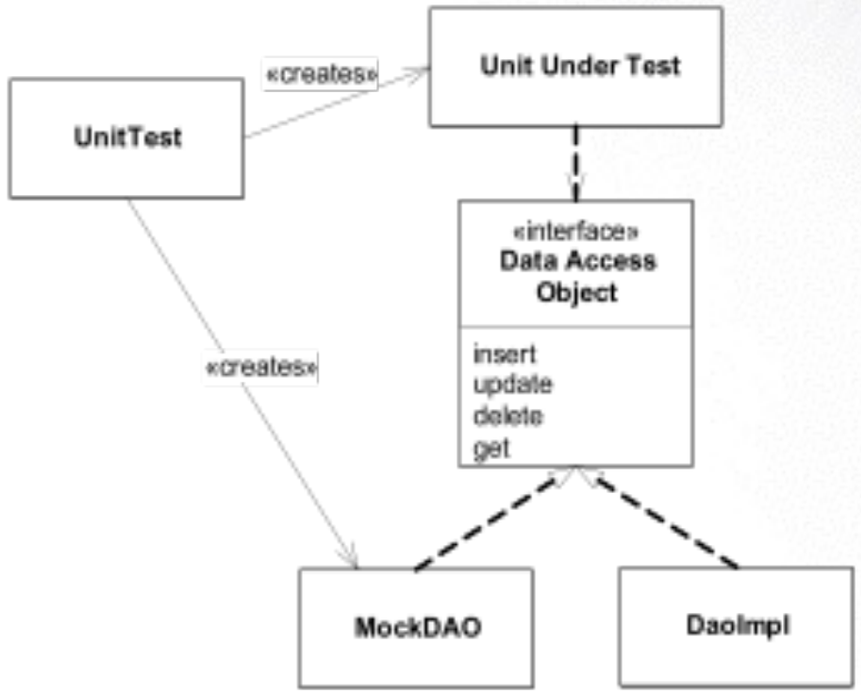
Strategies for testing Units that depend on other units



- 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

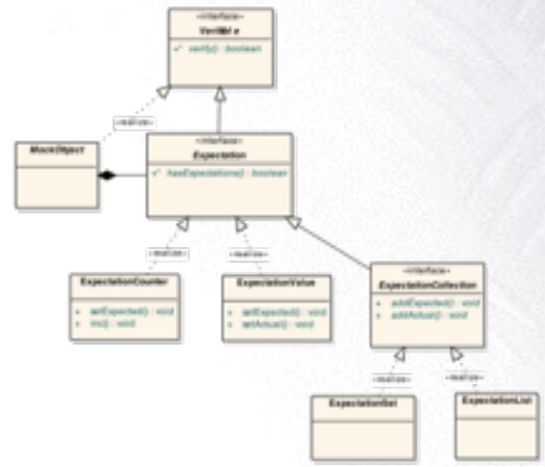
Synthetic context – MockObjects

- 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



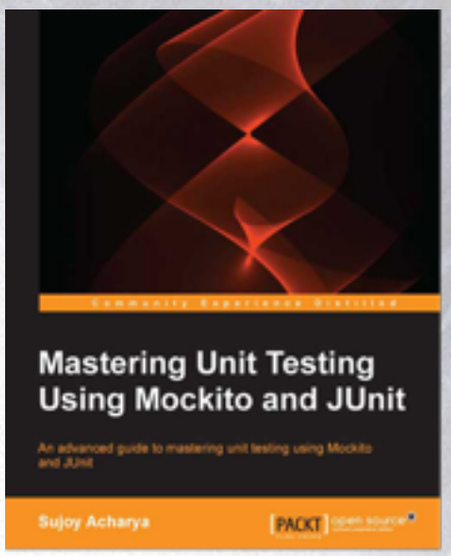
Frameworks and tools for creating MockObjects

- 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



MockMaker

EASYMOCK



- 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

Example usage

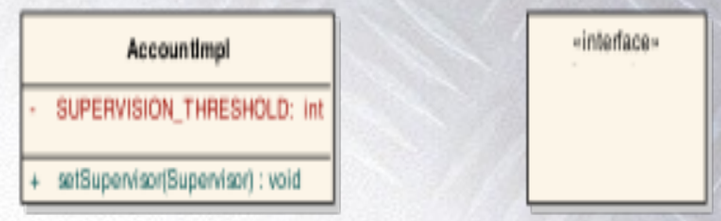
```
@Test
public void testNotificationVetoShouldBeHonoured() {
    int amount = AccountImpl.SUPERVISION_TRESHOLD;

    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

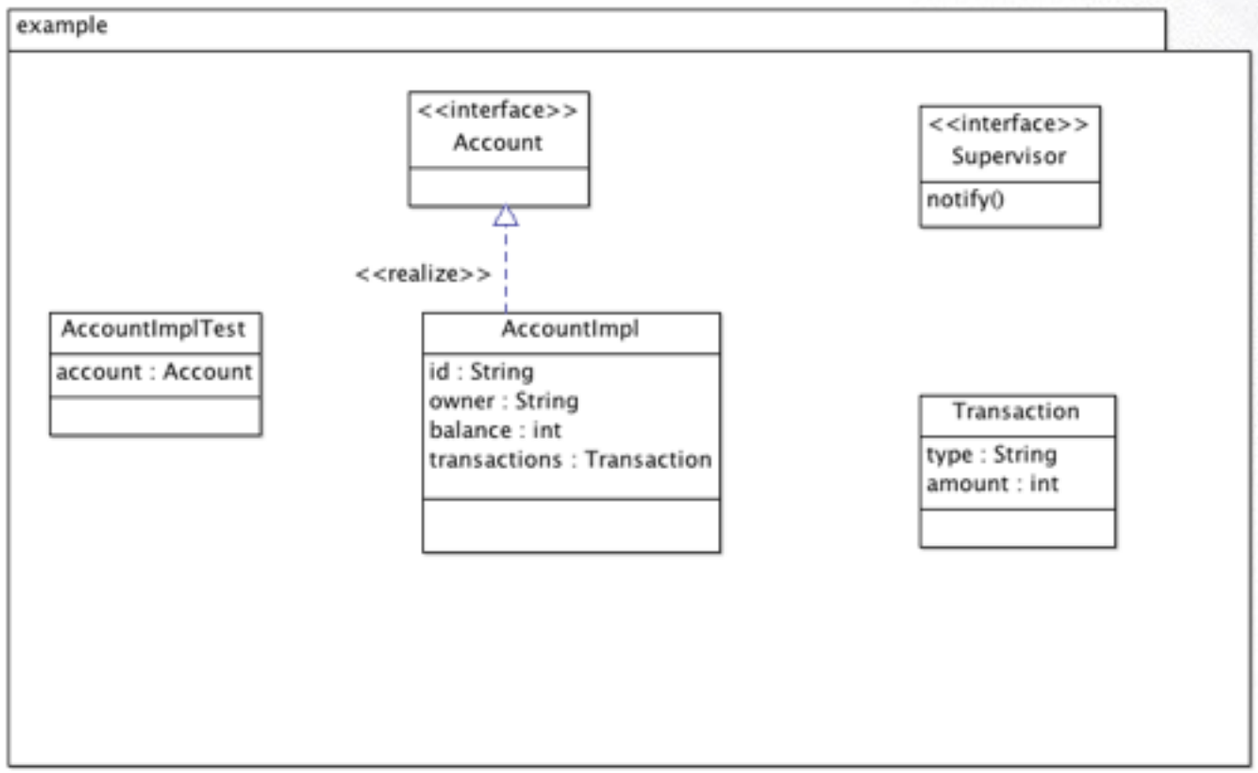
Typical usage scenario for Mock Objects in a TestCase

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

Exercise 7



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



When to use Mock Objects (and when not to)

- Mock Objects are great for
  - Breaking dependencies between well-architected layers or tiers
  - Testing corner cases and exceptional behavior
- Mock Objects are 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;
        }
        ...
}
```

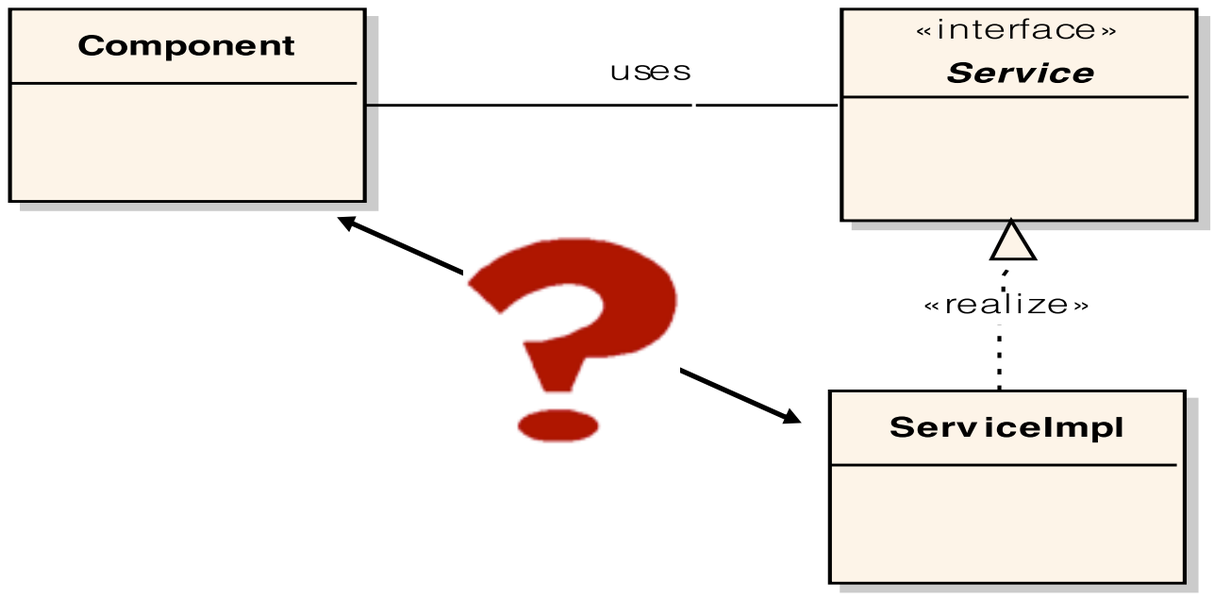
Designing for Testability : LoD - Don’t Talk To Strangers

- 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)

Test-Driven Development

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

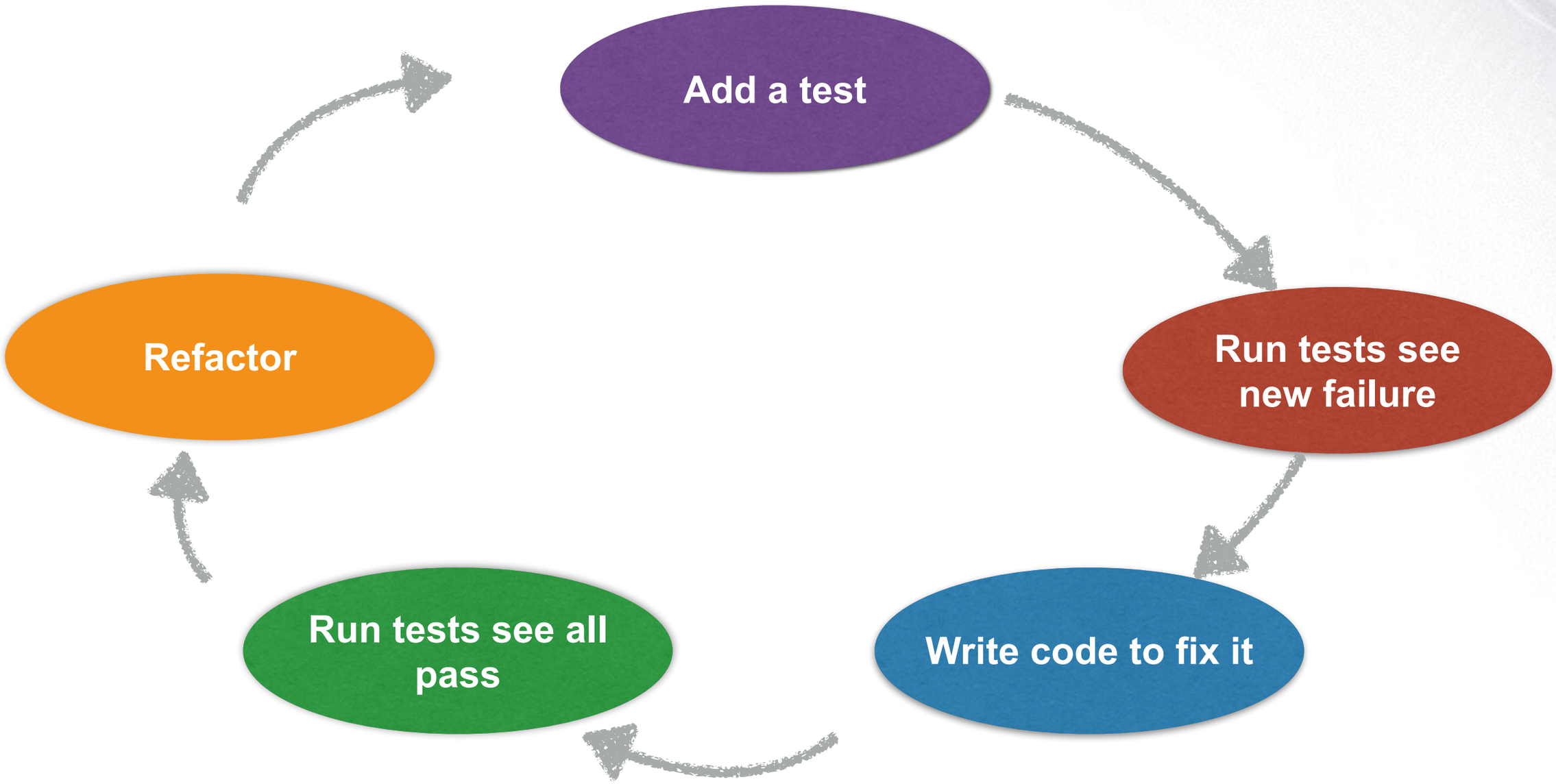
- Write a test that specifies a tiny bit of functionality
- Ensure the test fails (you haven't built the functionality yet!)
- Write the code necessary to make the test pass
- Refactoring the code to remove redundancy

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)



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

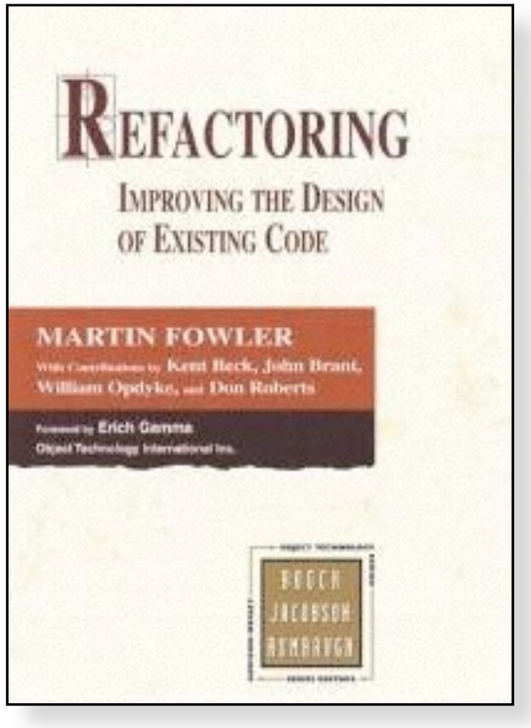
- In Priority Order
  - The code is appropriate for the intended audience
  - The code passes all the tests
  - The code communicates everything it needs to
  - The code has the smallest number of classes
  - The code has the smallest number of methods

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

Of course not, but why?

Refactoring

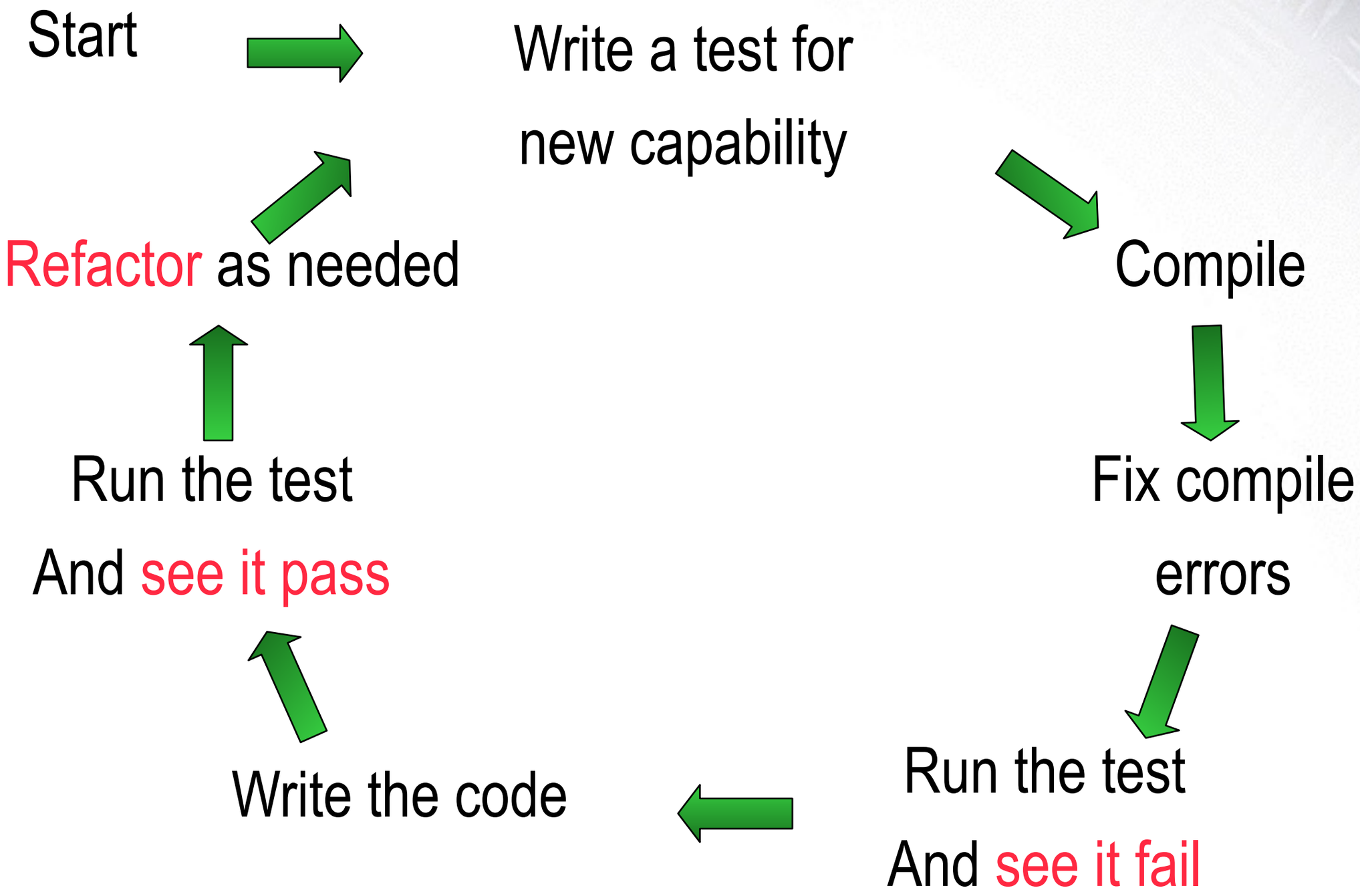
- Definition: Improve the code without changing its functionality
- Code needs to be refined as additional requirements (tests) are added
- For more information see
  Refactoring: Improve the Design of Existing Code – Martin Fowler



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

Red/Green/Refactor



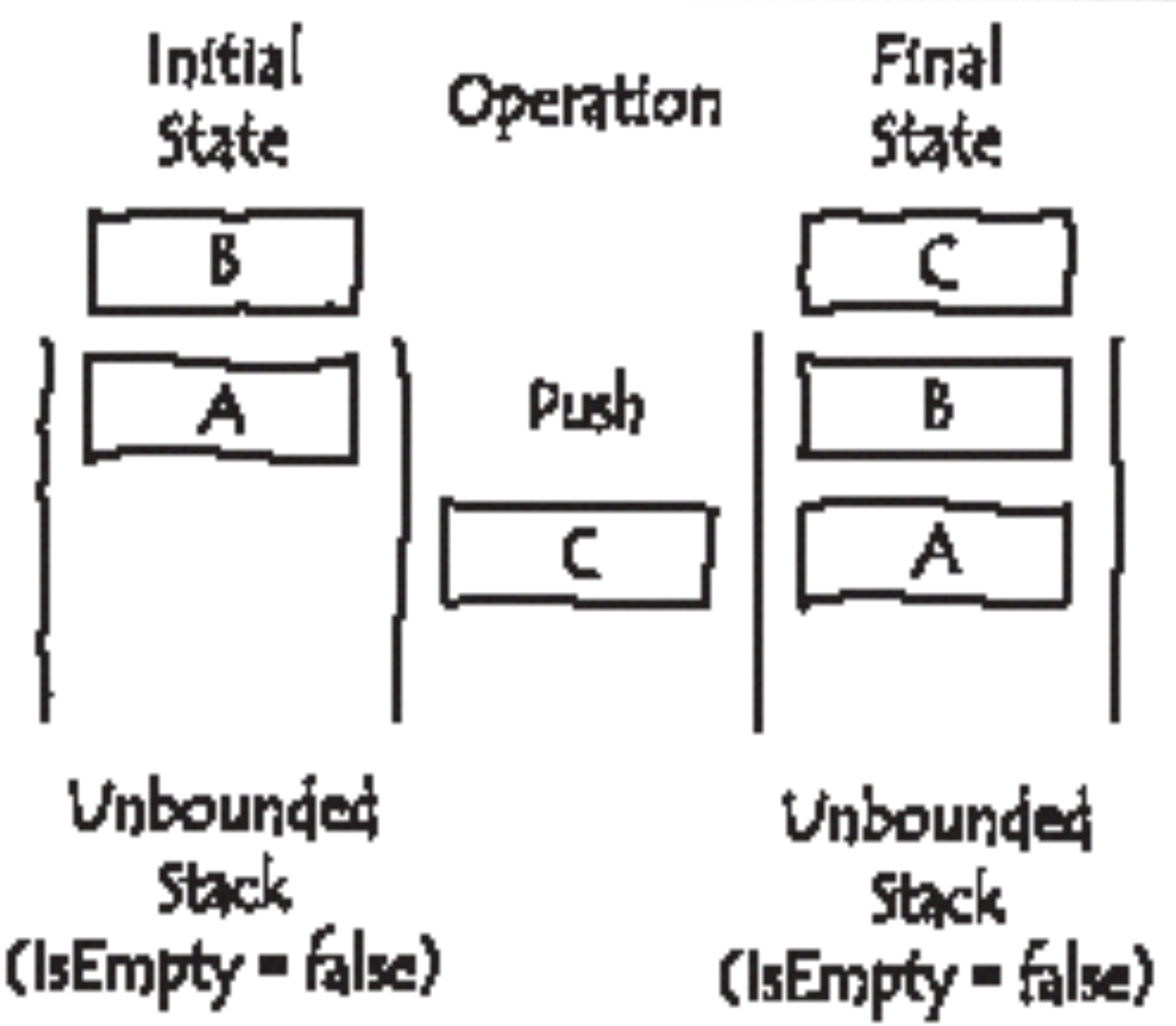
Exercise 5

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!

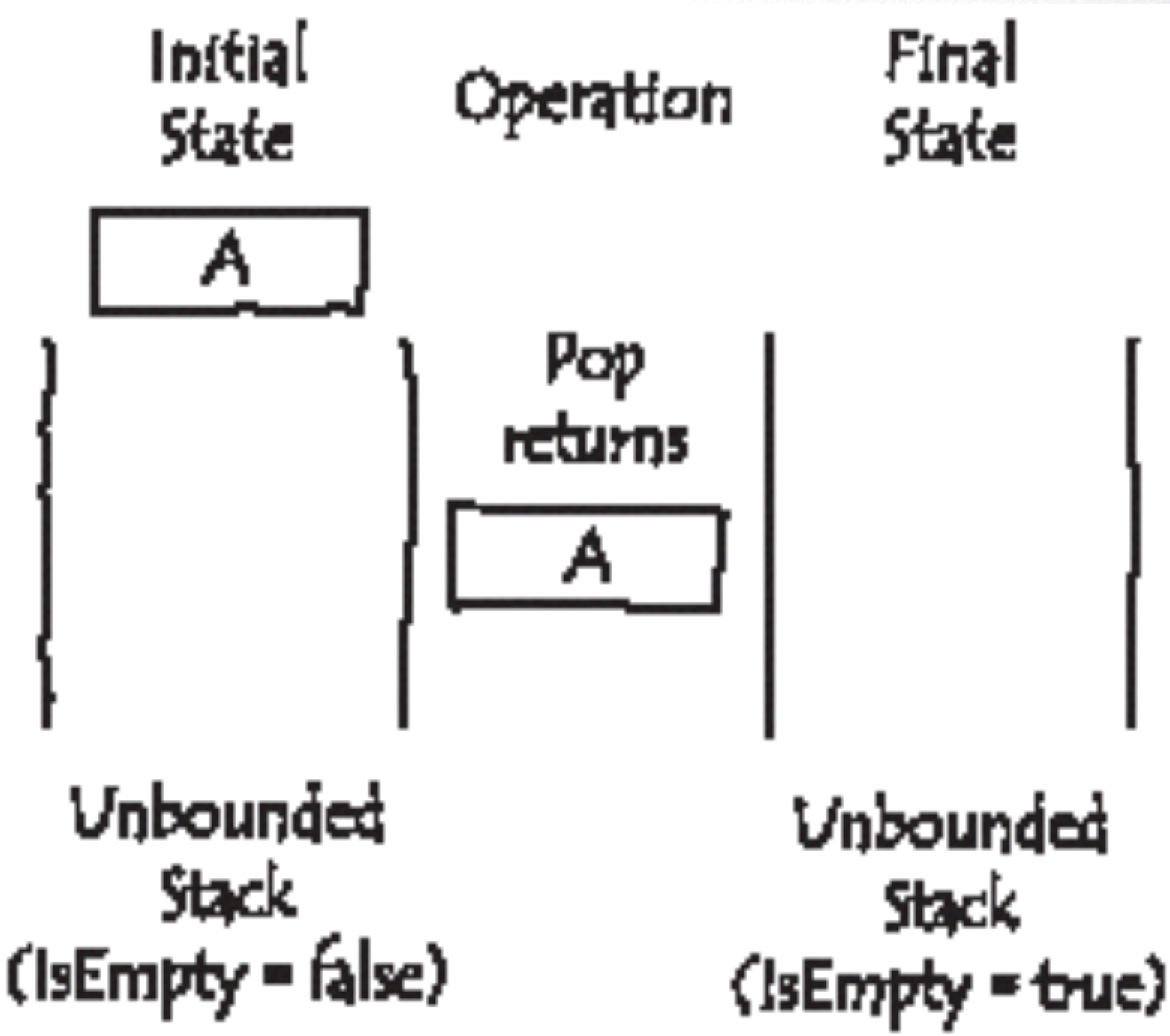
- Specification:
  - "A stack is a data structure in which you can access only the item at the top. With a computer, Stack like a stack of dishes—you add items to the top and remove them from the top."

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

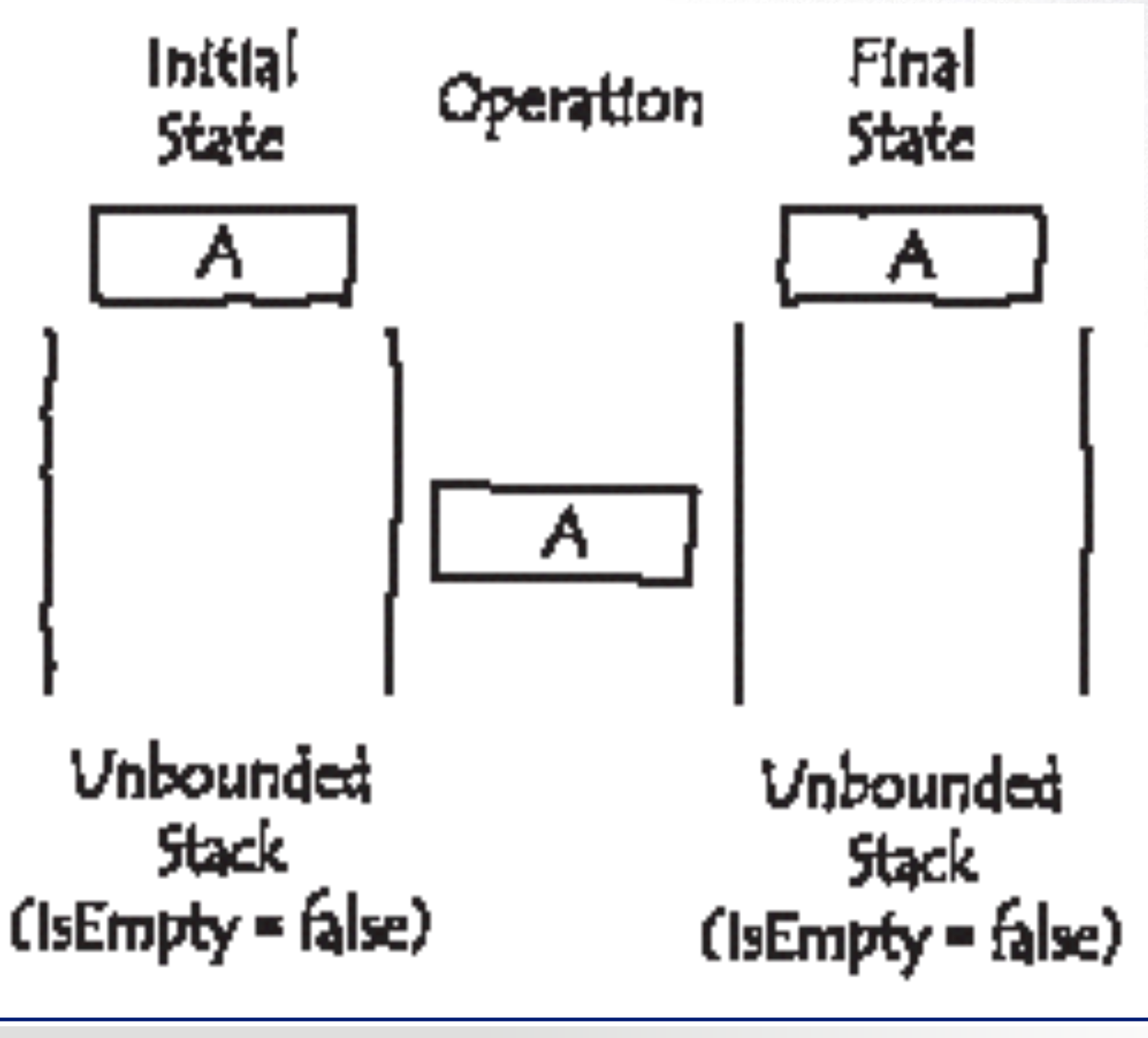
Push operation



Pop operation



Top operation



Recap: The TDD process Red/Green/Refactor