Scaffolding with JMock

Course of Software Engineering II
A.A. 2011/2012

Valerio Maggio, PhD Student
Prof. Marco Faella

Outline

- Brief Recap
  - -Unit Testing
  - -JUnit (case study)
- Test Scaffolding
  - -Stubs
  - -Mocks
- JMock
  - -Working Example

Example Scenario

- (… not properly related to Computer Science :)

- Please, imagine that you have to test a building
  - Test if it has been constructed properly
  - Test if it is able to resist to earthquake
  - ….

- Q: What types of “testing” would you do?
- Q: What should be the “starting point”?
  - Make an educated guess

Unit Testing

- Testing of the smallest pieces of a program
  - Individual functions or methods
- Keyword: Unit
  - (def) Something is a unit if it there's no meaningful way to divide it up further
- Buzz Word:
  - Testing in isolation

Unit Testing (cont.)

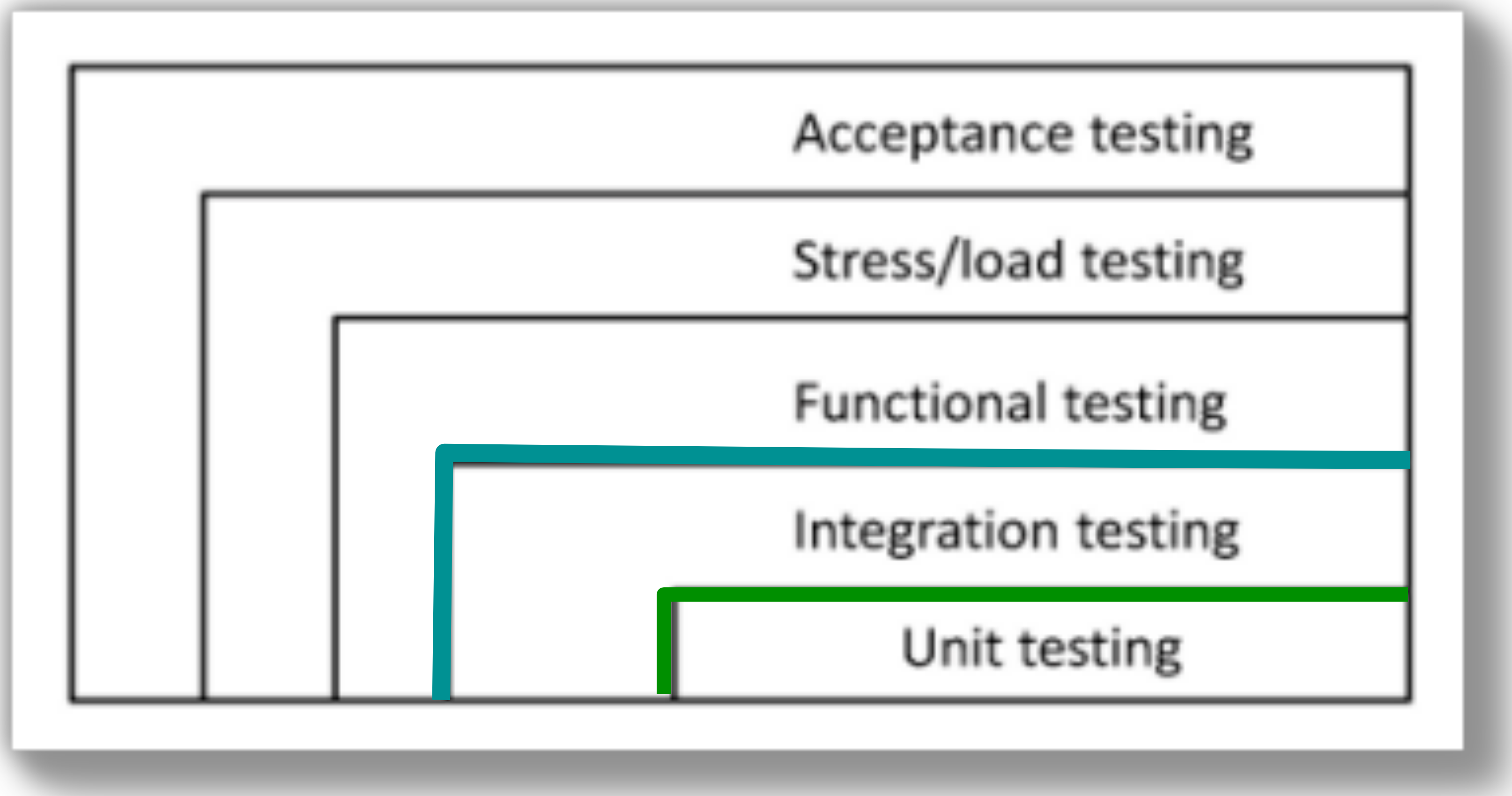
- Unit test are used to test a single unit in isolation
  - Verifying that it works as expected
  - No matter the rest of the program would do
- Possible advantages ?
  - (Possibly) No inheritance of bugs of mistakes from made elsewhere
  - Narrow down on the actual problem

Unit Testing (cont.)

- Is it enough ?
  - Not by itself, but...
  - … it is the foundation upon which everything is based!

- (Back to the example)
  - You can't build a house without solid materials.
  - You can't build a program without units that works as expected.

Testing RoadMap



Acceptance testing

Stress/load testing

Functional testing

Integration testing

Unit testing

Functional Software Testing

- Examine code at the boundary of its public API
  - Testing application Use Cases

- Developers often combine Functional and Integration Testing

- Testing
  - Frameworks (API)
  - GUIs
  - Subsystems (API call enforced)

Integration Software Testing

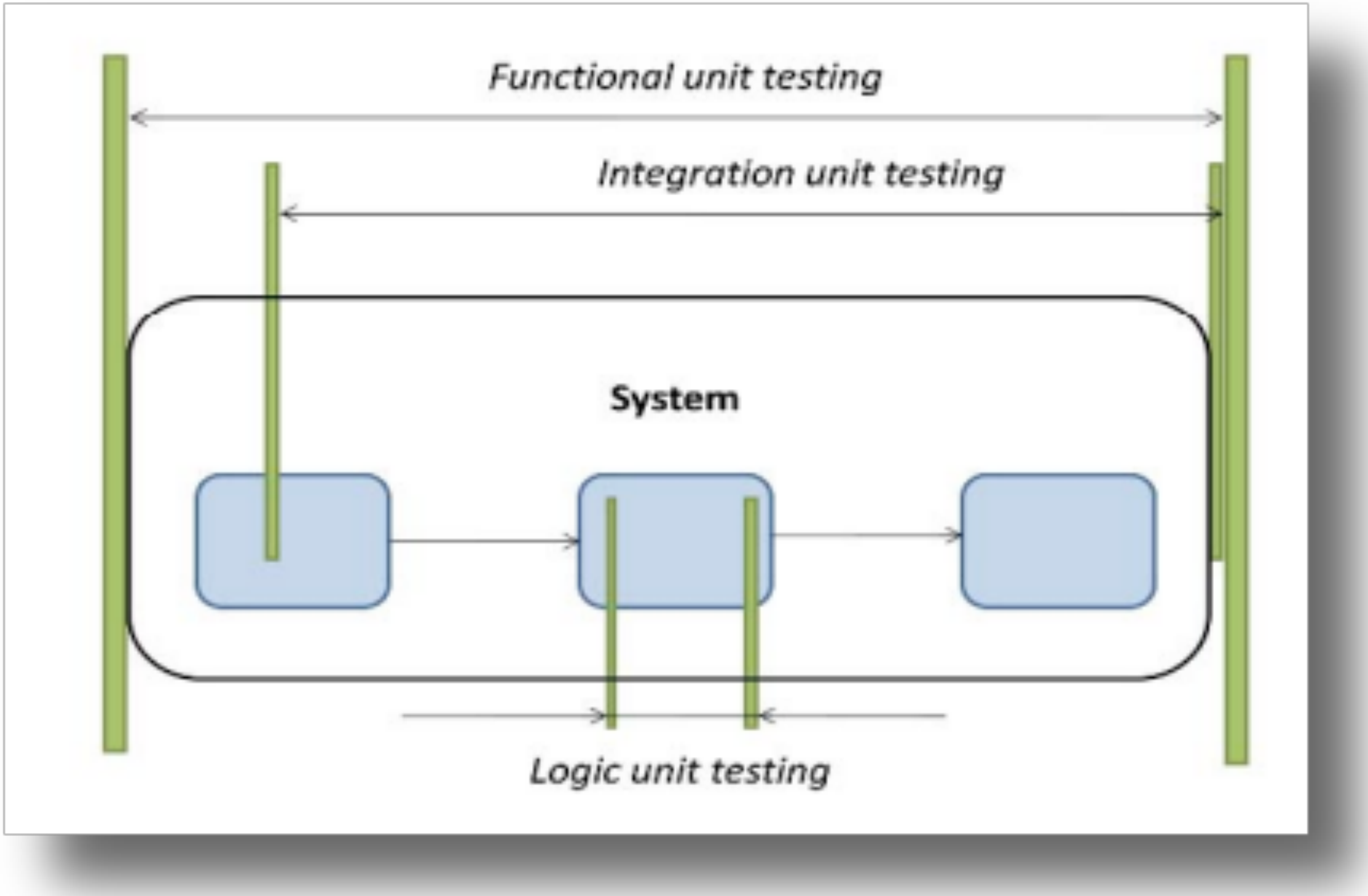
- What happens when different units of works are combined together ?

- Examine the interactions among and writing components:
  - Objects
  - Services
  - Subsystems

Unit Software Testing

- Examine the code of a single module in all of its features
- Starts from the inspection of a simple (small) functionality
- Writing more and more tests means more and more "manifold" test cases
  - Three types of unit testing

Three types of unit tests



Unit Testing main features

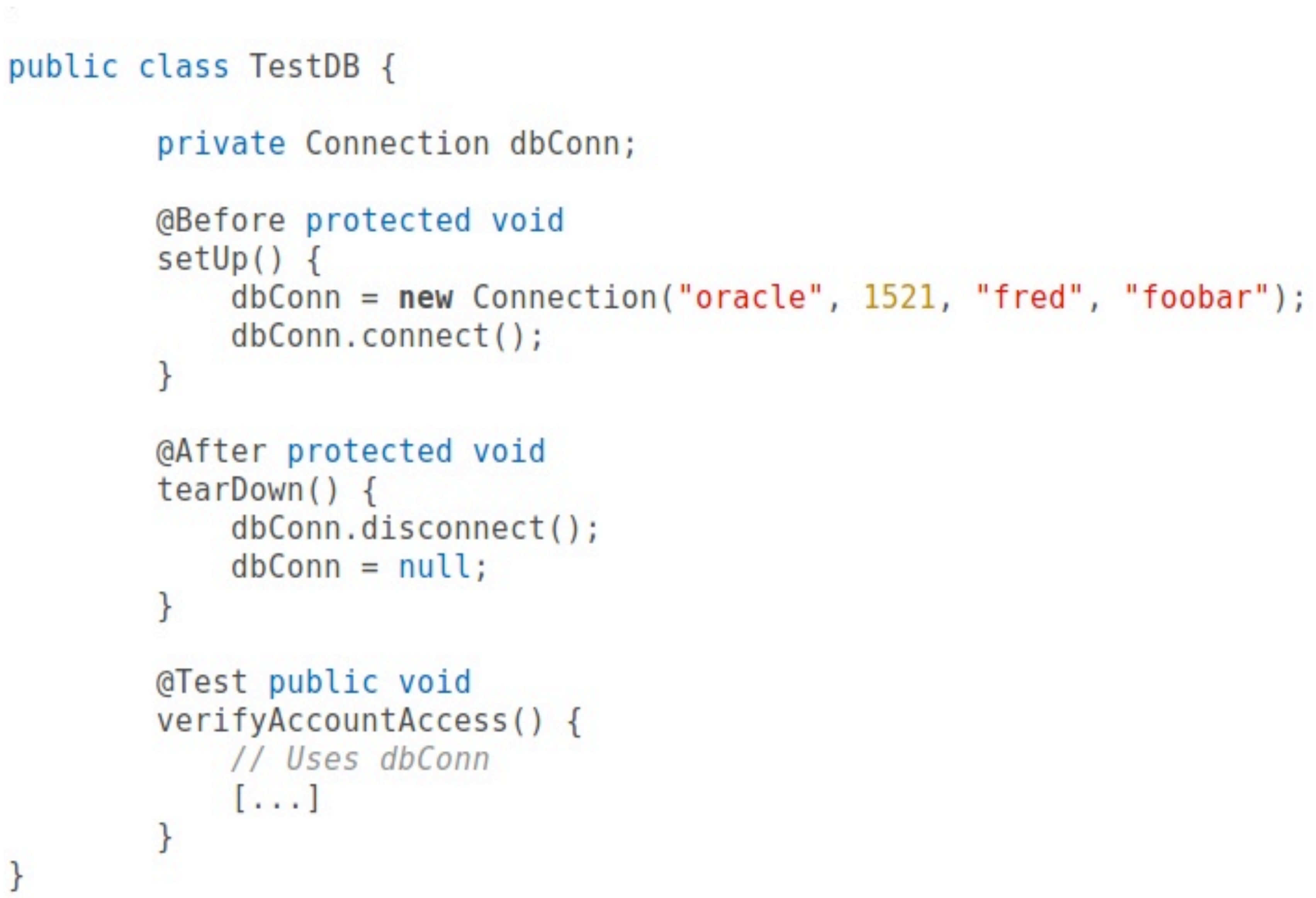
- Greater code coverage percentage
  - Functional Testing coverage about 70%
  - Enable code coverage and other metrics
- Increase team productivity
- Improve implementation
  - Confidence with refactoring
- Document expected behavior

Test Scaffolding

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

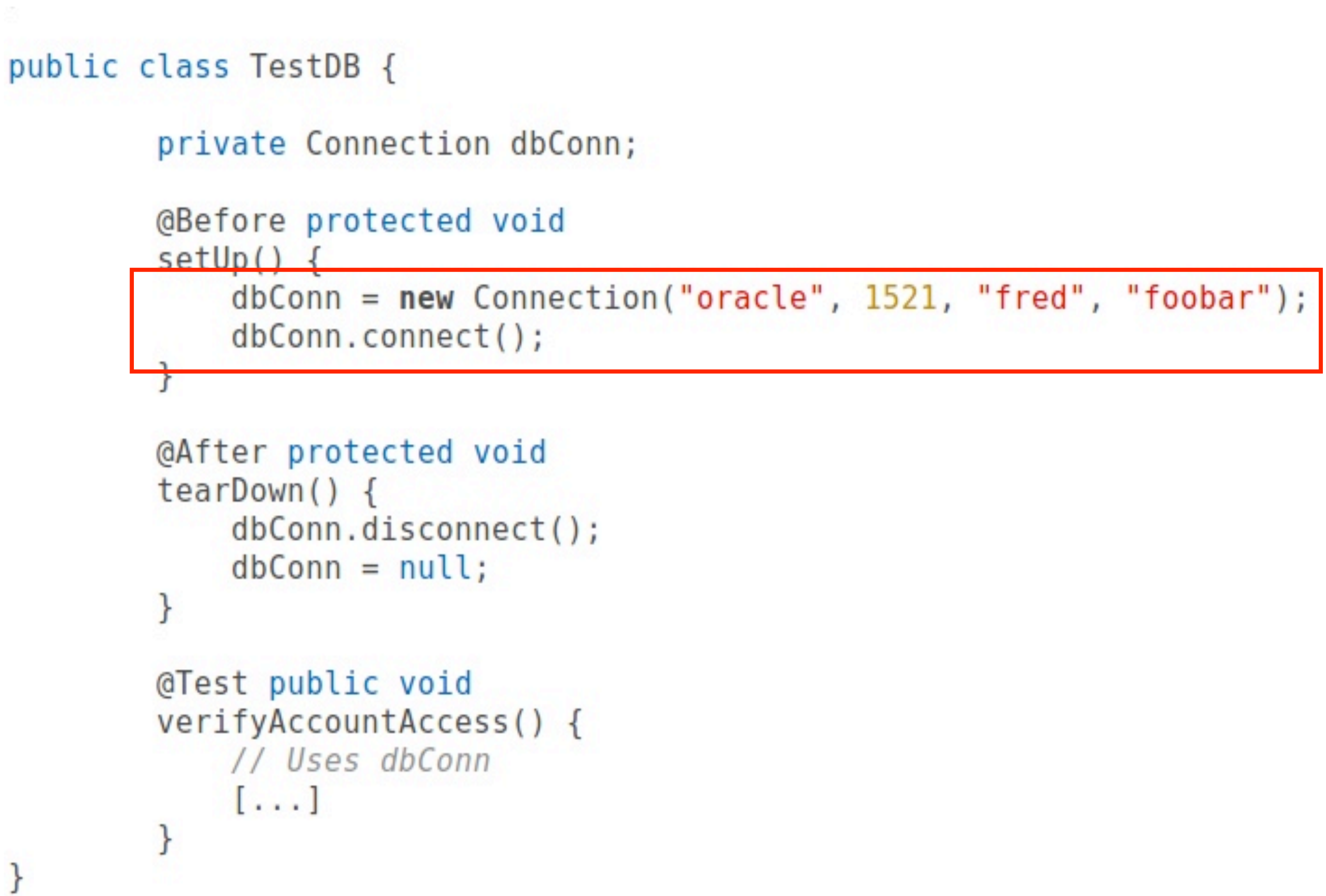
Cit. Rich Cook

Integration Testing Example



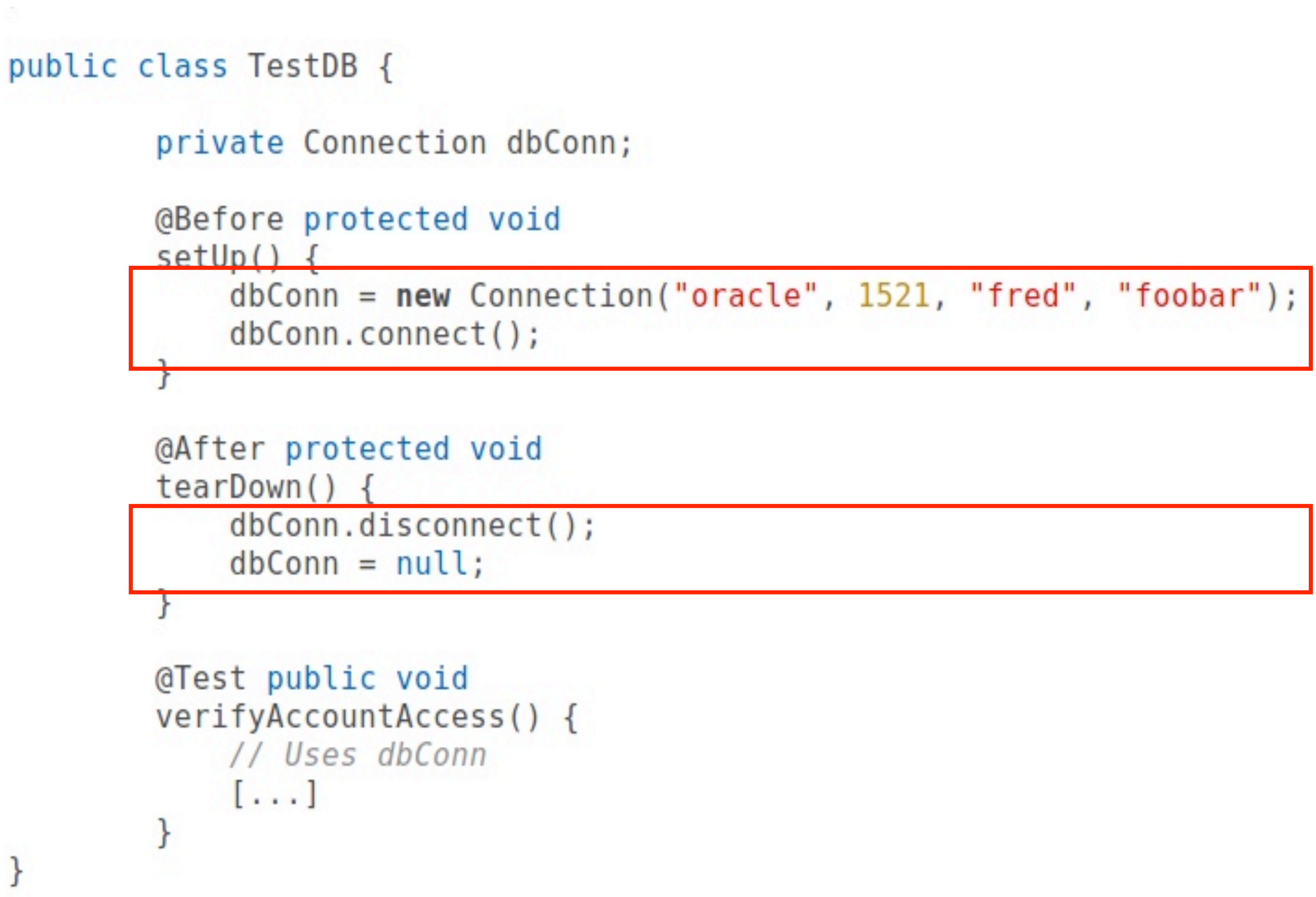
```
public class TestDB {

        private Connection dbConn;

        @Before protected void
        setUp() {
            dbConn = new Connection("oracle", 1521, "fred", "foobar");
            dbConn.connect();
        }

        @After protected void
        tearDown() {
            dbConn.disconnect();
            dbConn = null;
        }

        @Test public void
        verifyAccountAccess() {
            // Uses dbConn
            [...]
        }
}
```

Integration Testing Example



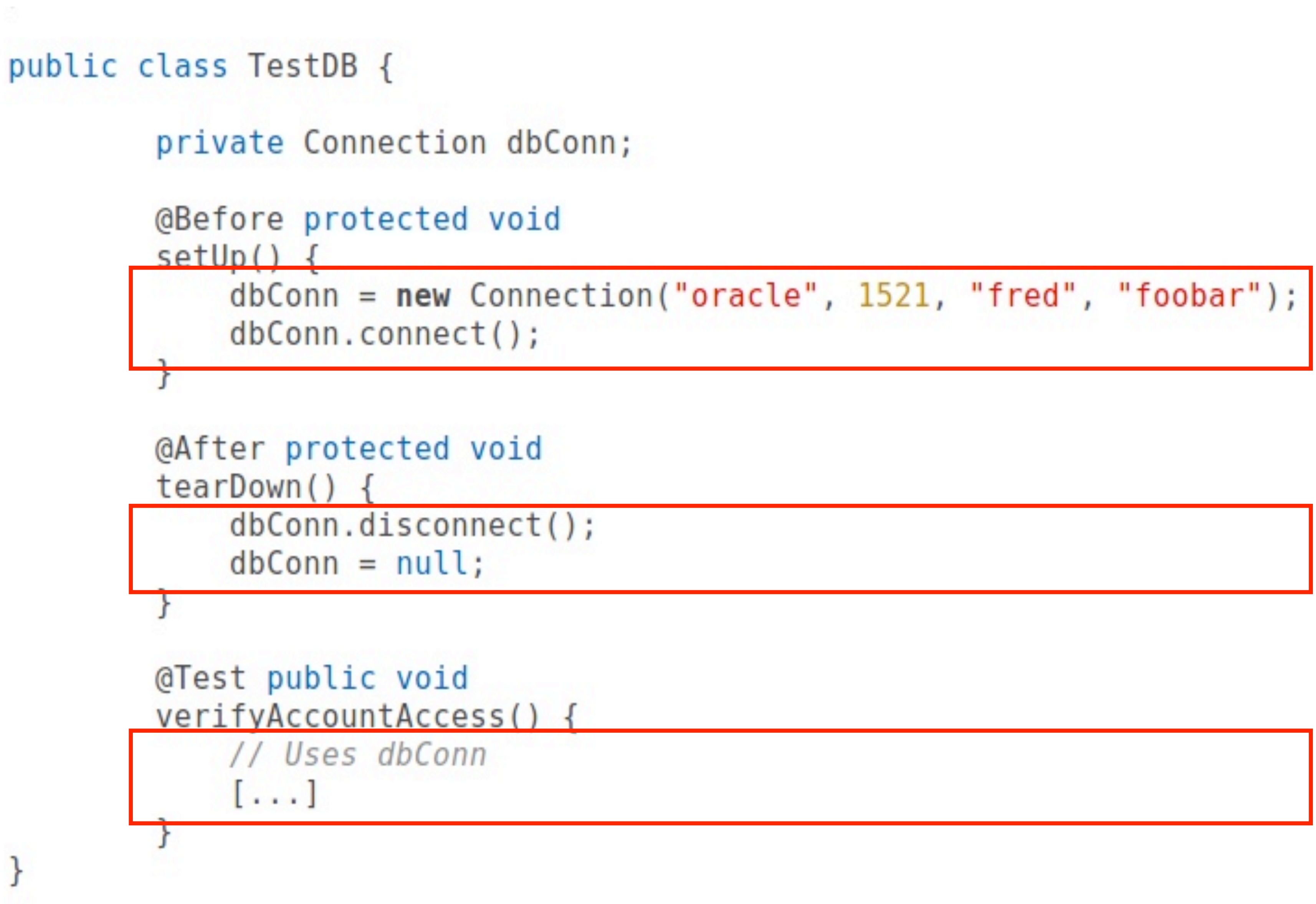
```
public class TestDB {

        private Connection dbConn;

        @Before protected void
        setUp() {
            dbConn = new Connection("oracle", 1521, "fred", "foobar");
            dbConn.connect();
        }

        @After protected void
        tearDown() {
            dbConn.disconnect();
            dbConn = null;
        }

        @Test public void
        verifyAccountAccess() {
            // Uses dbConn
            [...]
        }
}
```

Integration Testing Example



```
public class TestDB {

        private Connection dbConn;

        @Before protected void
        setUp() {
            dbConn = new Connection("oracle", 1521, "fred", "foobar");
            dbConn.connect();
        }

        @After protected void
        tearDown() {
            dbConn.disconnect();
            dbConn = null;
        }

        @Test public void
        verifyAccountAccess() {
            // Uses dbConn
            [...]
        }
}
```

Integration Testing Example



```
public class TestDB {

        private Connection dbConn;

        @Before protected void
        setUp() {
            dbConn = new Connection("oracle", 1521, "fred", "foobar");
            dbConn.connect();
        }

        @After protected void
        tearDown() {
            dbConn.disconnect();
            dbConn = null;
        }

        @Test public void
        verifyAccountAccess() {
            // Uses dbConn
            [...]
        }
}
```

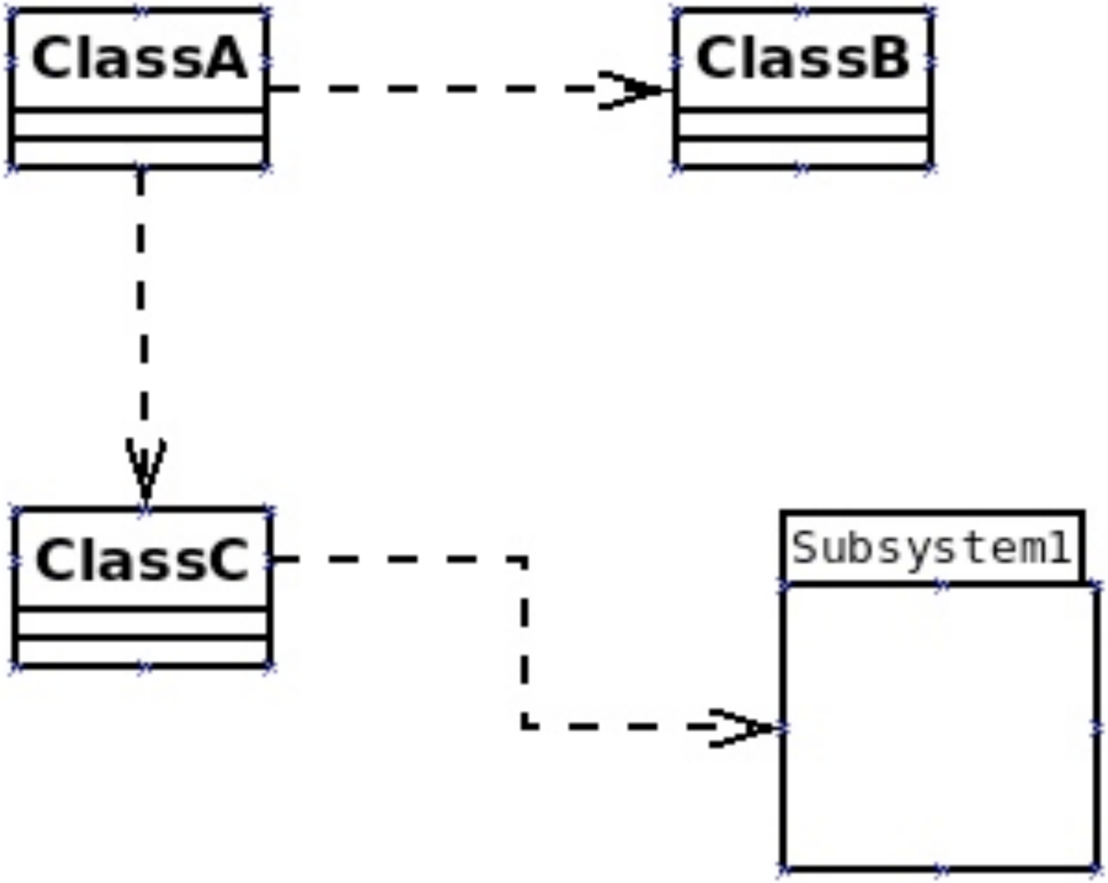
Integration testing problem

- Integrate multiple components implies to decide in which order classes and subsystems should be integrated and tested

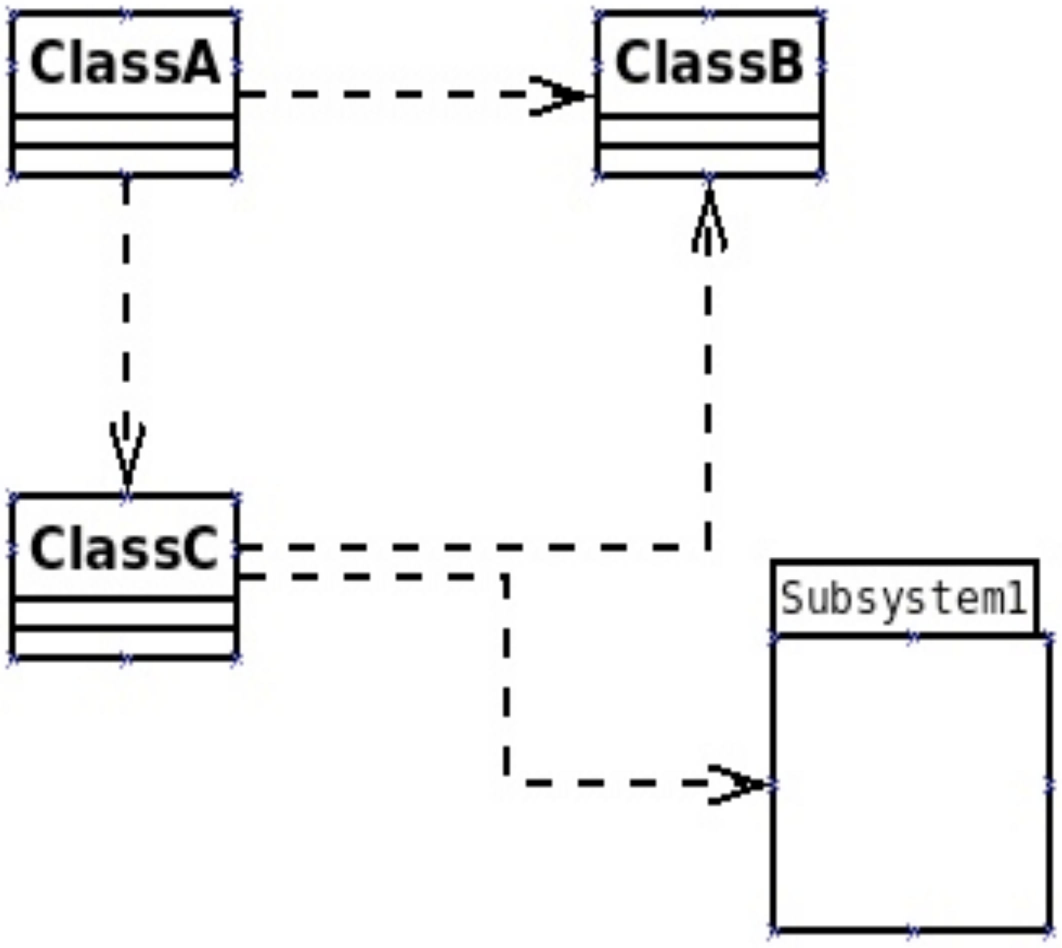
- CITO Problem
  - Class Integration Testing Order Problem

- Solution:
  - Topological sort of dependency graph

Integration testing example



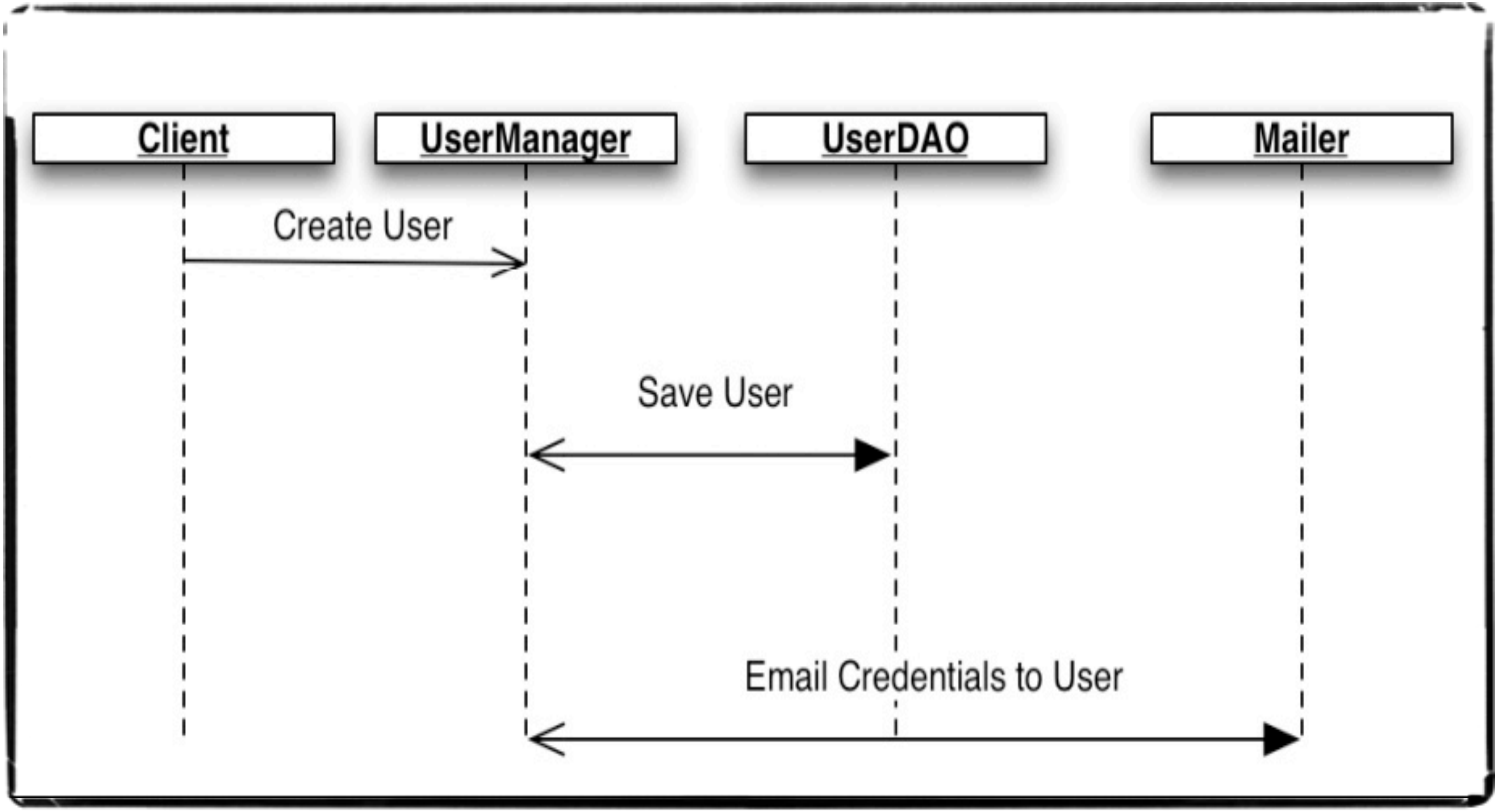
Integration testing example



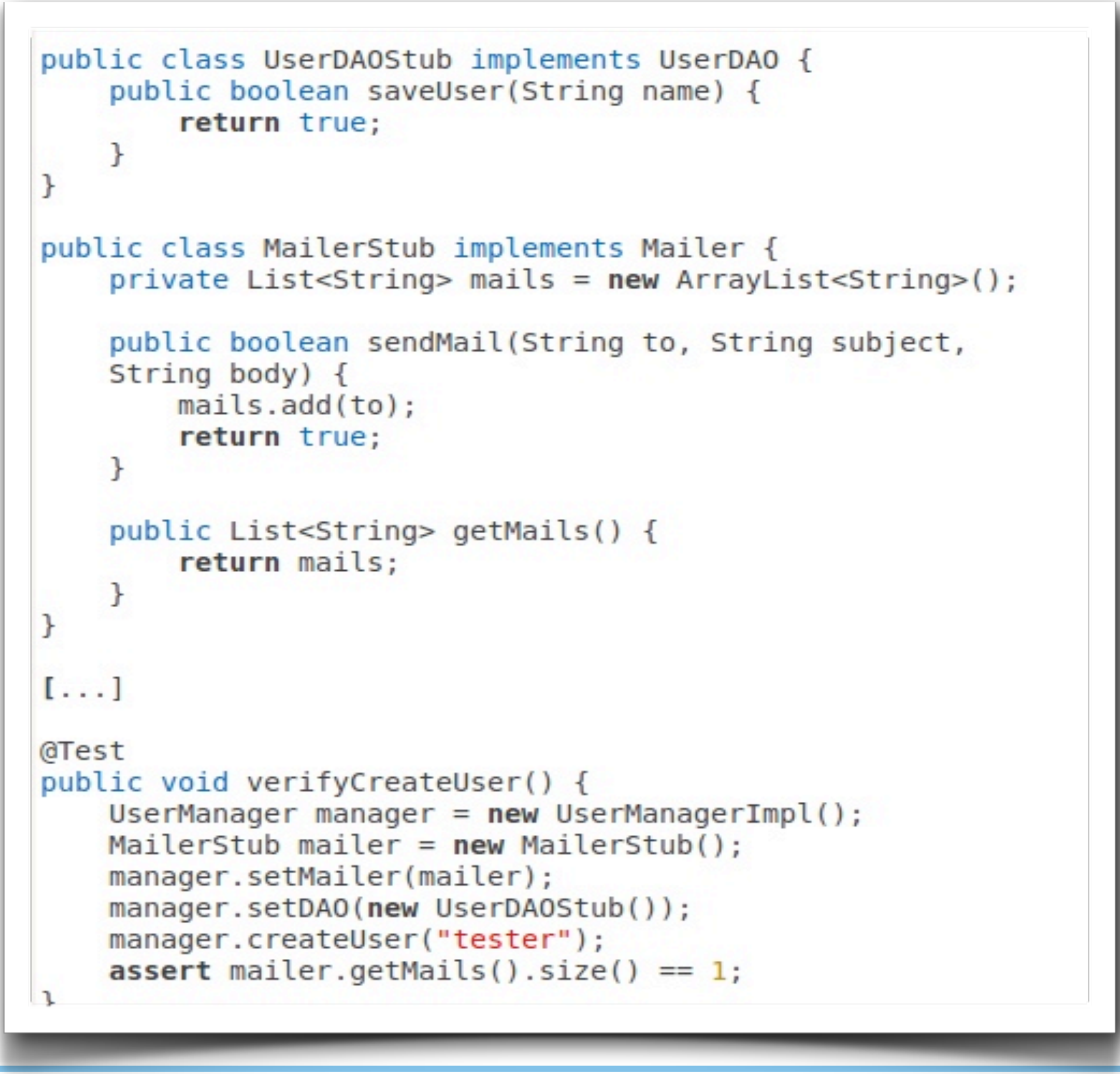
Testing in isolation

- Testing in isolation offers strong benefits
  - Test code that have not been written
  - Test only a single method (behavior) without side effects from other objects
- Solutions ?
  - Stubs
  - Mocks
  - …

Testing in Isolation: example

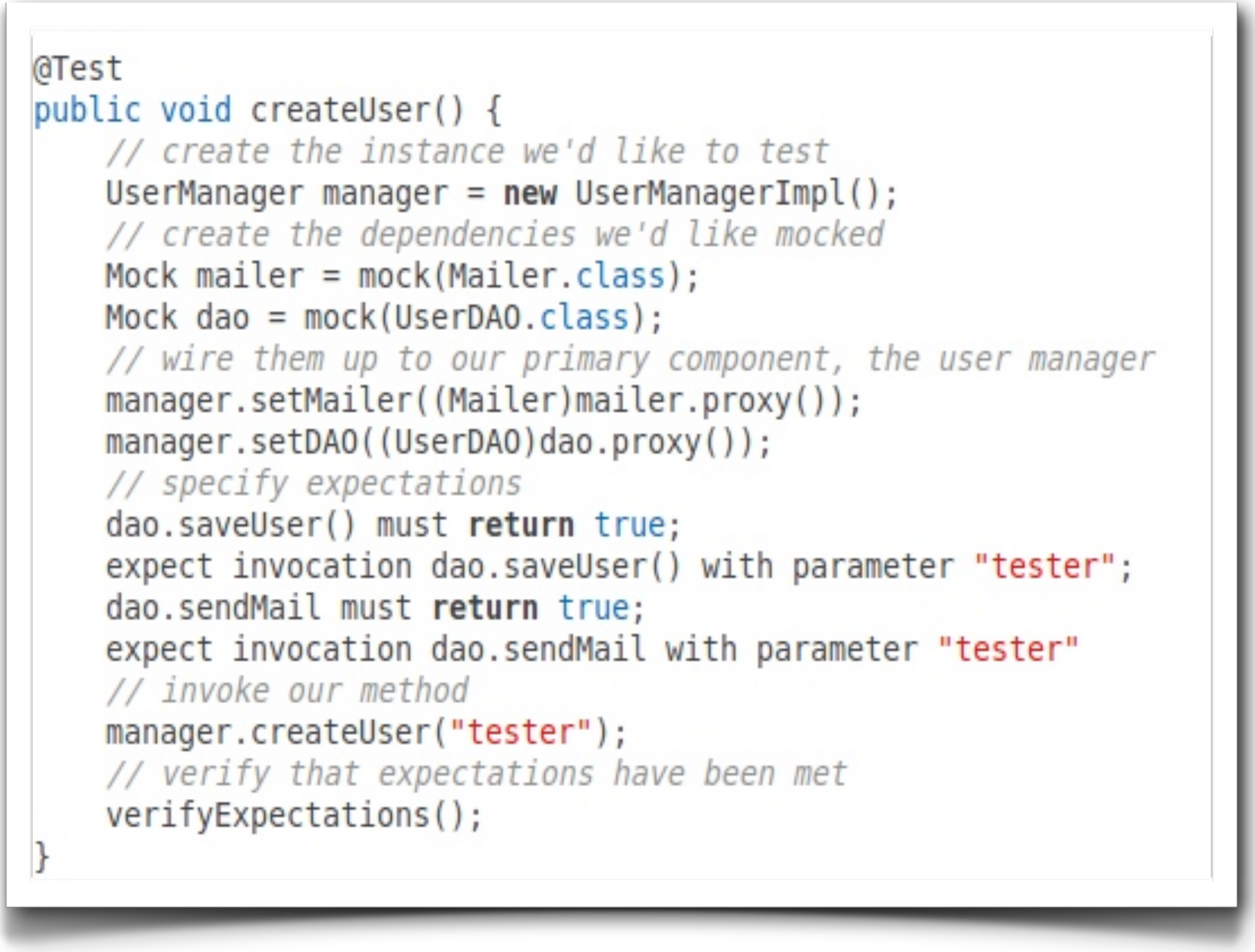


Solution with stubs



```
public class UserDAOStub implements UserDAO {
    public boolean saveUser(String name) {
        return true;
    }
}

public class MailerStub implements Mailer {
    private List<String> mails = new ArrayList<String>();

    public boolean sendMail(String to, String subject,
    String body) {
        mails.add(to);
        return true;
    }

    public List<String> getMails() {
        return mails;
    }
}

[...]

@Test
public void verifyCreateUser() {
    UserManager manager = new UserManagerImpl();
    MailerStub mailer = new MailerStub();
    manager.setMailer(mailer);
    manager.setDAO(new UserDAOStub());
    manager.createUser("tester");
    assert mailer.getMails().size() == 1;
}
```

Solution with (Pseudo) Mocks



```
@Test
public void createUser() {
    // create the instance we'd like to test
    UserManager manager = new UserManagerImpl();
    // create the dependencies we'd like mocked
    Mock mailer = mock(Mailer.class);
    Mock dao = mock(UserDAO.class);
    // wire them up to our primary component, the user manager
    manager.setMailer((Mailer)mailer.proxy());
    manager.setDAO((UserDAO)dao.proxy());
    // specify expectations
    dao.saveUser() must return true;
    expect invocation dao.saveUser() with parameter "tester";
    dao.sendMail must return true;
    expect invocation dao.sendMail with parameter "tester"
    // invoke our method
    manager.createUser("tester");
    // verify that expectations have been met
    verifyExpectations();
}
```

Key Ideas

- Wrap all the details of Code
  - (sort of) Simulation
- Mocks do not provide our own implementation of the components we'd like to swap in
- Main Difference:
  - Mocks test behavior and interactions between components
  - Stubs replace heavyweight process that are not relevant to a particular test with simple implementations

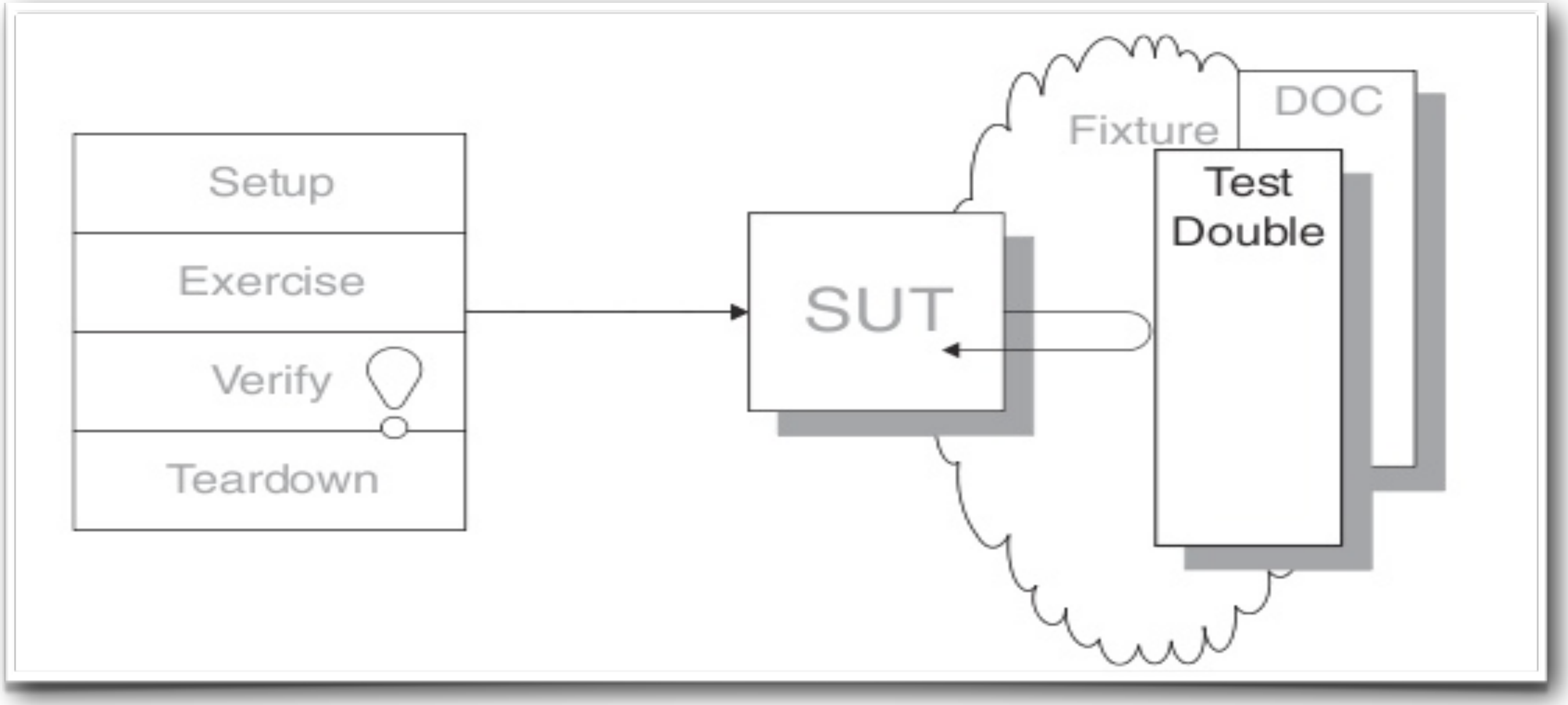
Mock Objects Observations

- Powerful way to implement Behavior Verification
  - while avoiding Test Code Duplication between similar tests.
- It works by delegating the job of verifying the indirect outputs of the SUT
- Important Note: Design for Mockability
  - Dependency Injection Pattern

Naming Confusion

- Unfortunately, while two components are quite distinct, they're used interchangeably.
  - Example: `spring-mock package`
- If we were to be stricter in terms of naming, stub objects defined previously are test doubles
- Test Doubles, Stubs, Mocks, Fake Objects… how we can work it out ?

Test Double Pattern
(a.k.a. Imposter)

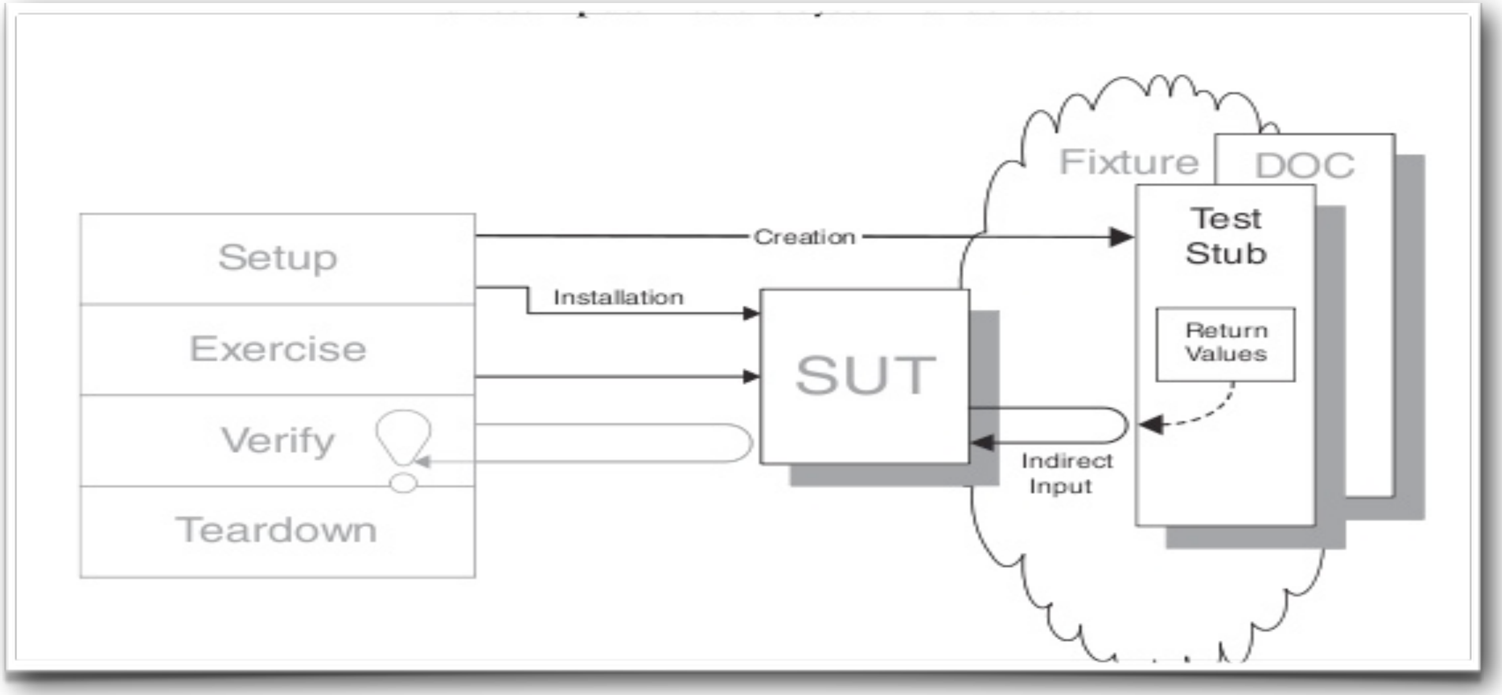


Q: How can we verify logic independently when code it depends on is unusable?

Q1: How we can avoid slow tests ?

A: We replace a component on which the SUT depends with a "test-specific equivalent."

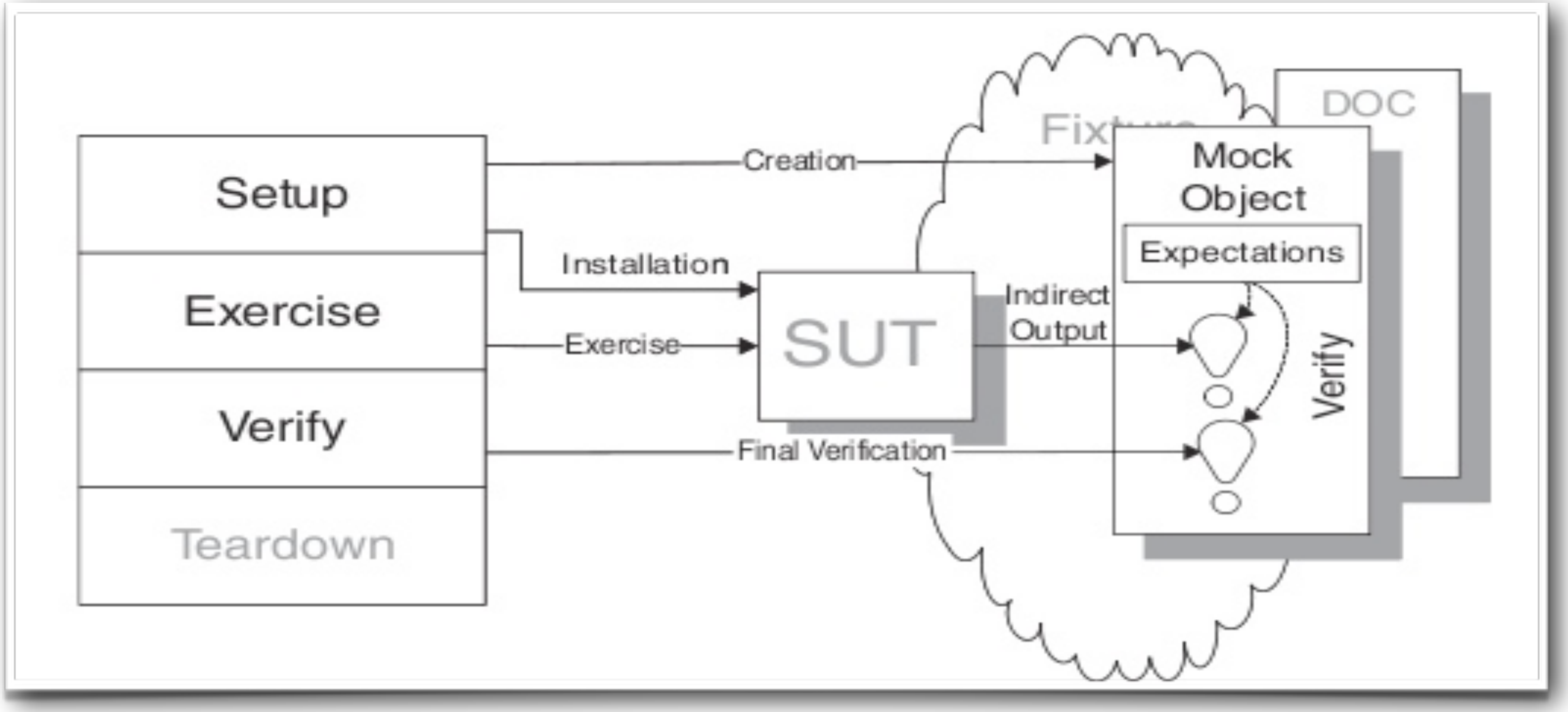
Test Stub Pattern



Q: How can we verify logic independently when it depends on indirect inputs from other software components ?

A: We replace a real objects with a test-specific object that feeds the desired inputs into the SUT

Mocks Objects

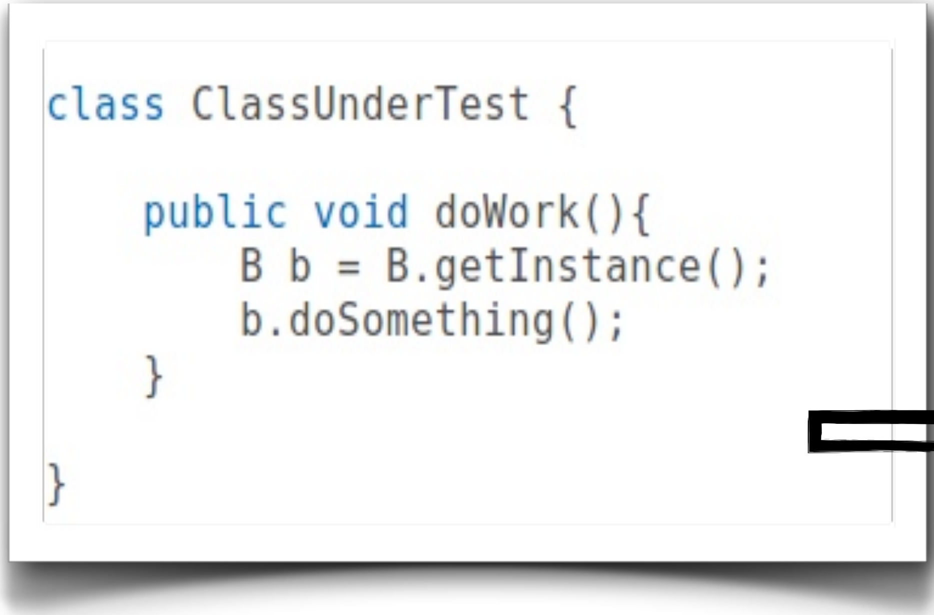


Q: How can we implement Behavior Verification for indirect outputs of the SUT ?

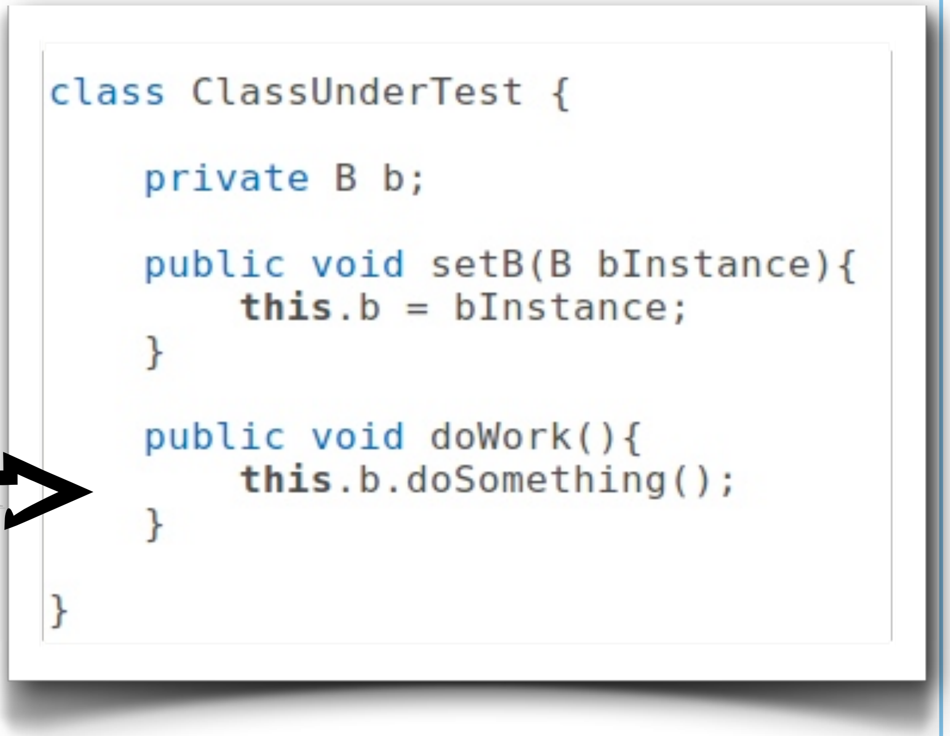
A: We replace an object on which the SUT depends on with a test-specific object that verifies it is being used correctly by the SUT.

Design for Mockability

- Dependency Injection



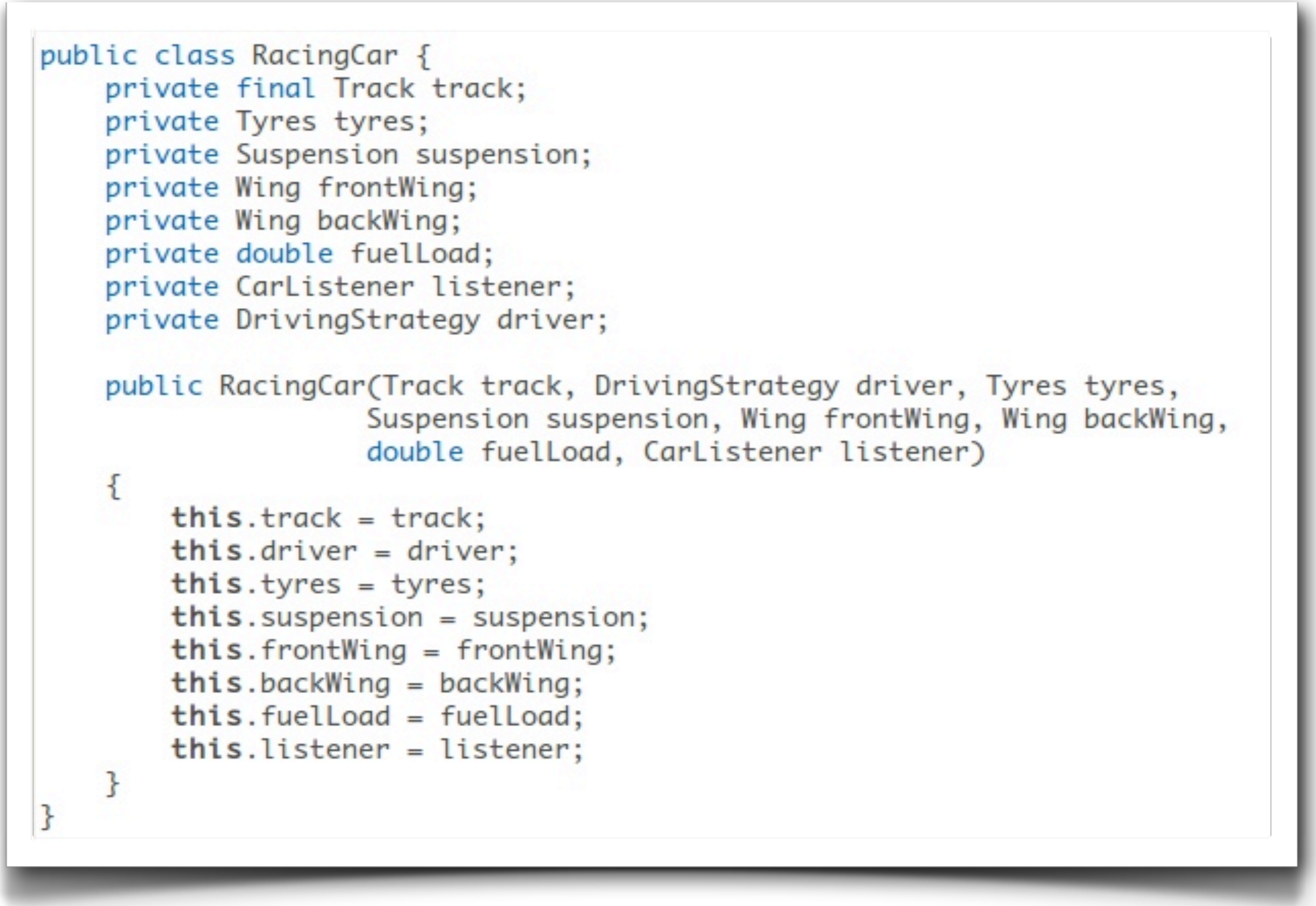
```
class ClassUnderTest {

    public void doWork(){
        B b = B.getInstance();
        b.doSomething();
    }

}
```



```
class ClassUnderTest {

    private B b;

    public void setB(B bInstance){
        this.b = bInstance;
    }

    public void doWork(){
        this.b.doSomething();
    }

}
```

Dependency injection issues?

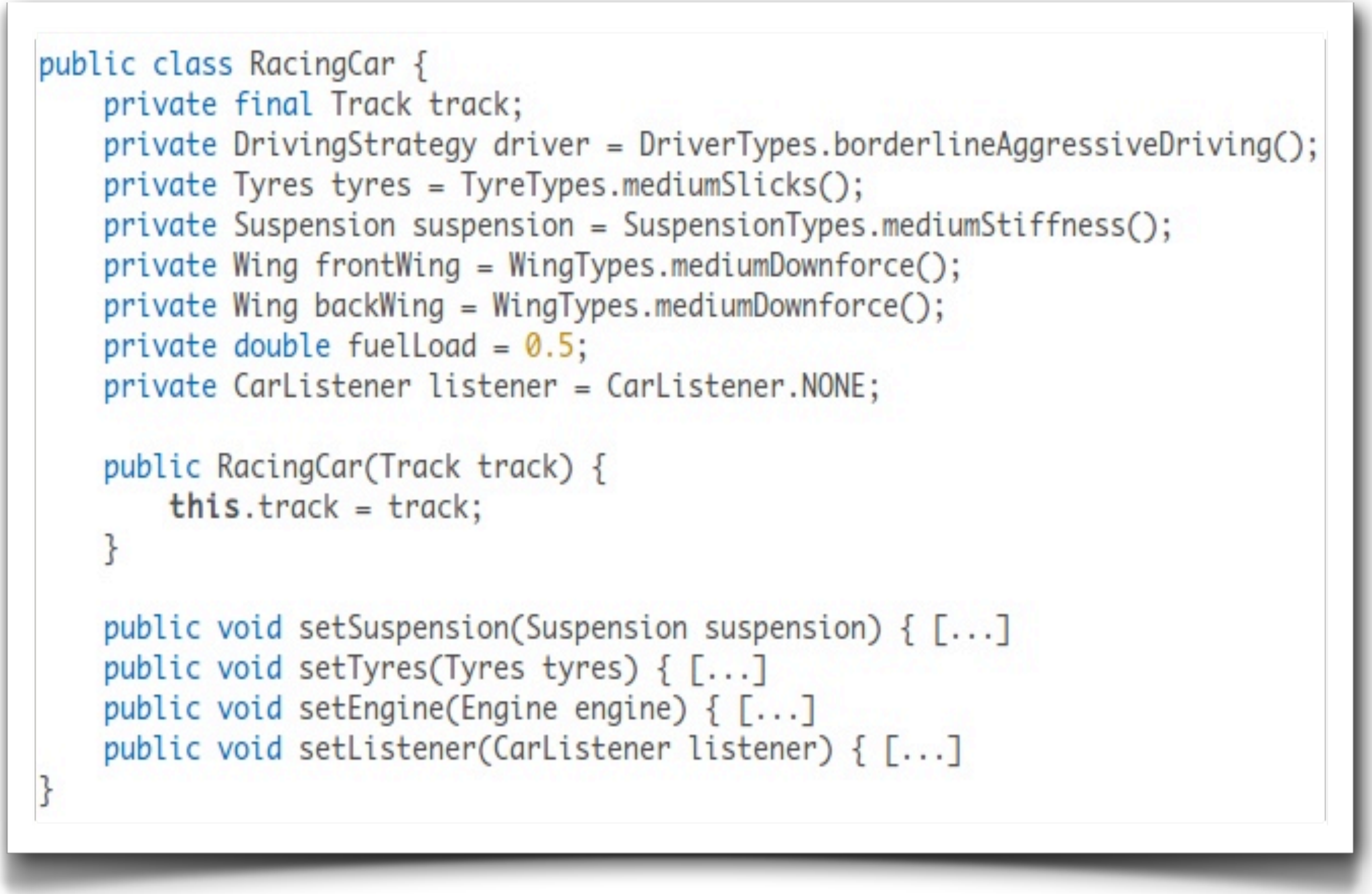
Too Many Dependencies……Ideas??



```java
public class RacingCar {
    private final Track track;
    private Tyres tyres;
    private Suspension suspension;
    private Wing frontWing;
    private Wing backWing;
    private double fuelLoad;
    private CarListener listener;
    private DrivingStrategy driver;

    public RacingCar(Track track, DrivingStrategy driver, Tyres tyres,
                     Suspension suspension, Wing frontWing, Wing backWing,
                     double fuelLoad, CarListener listener)
    {
        this.track = track;
        this.driver = driver;
        this.tyres = tyres;
        this.suspension = suspension;
        this.frontWing = frontWing;
        this.backWing = backWing;
        this.fuelLoad = fuelLoad;
        this.listener = listener;
    }
}
```

Dependency injection issues?

Dependency injection for mockability

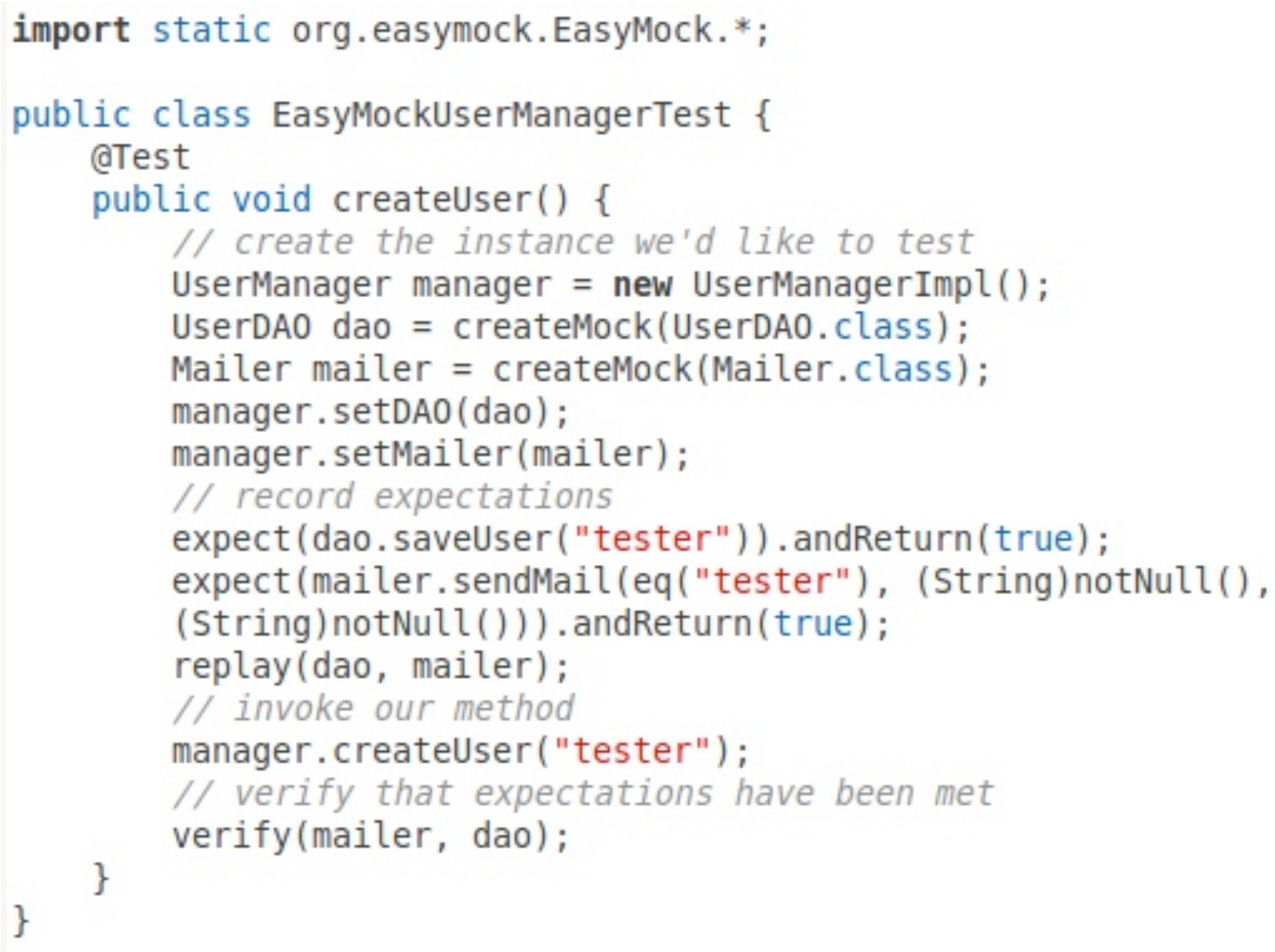


```
public class RacingCar {
    private final Track track;
    private DrivingStrategy driver = DriverTypes.borderlineAggressiveDriving();
    private Tyres tyres = TyreTypes.mediumSlicks();
    private Suspension suspension = SuspensionTypes.mediumStiffness();
    private Wing frontWing = WingTypes.mediumDownforce();
    private Wing backWing = WingTypes.mediumDownforce();
    private double fuelLoad = 0.5;
    private CarListener listener = CarListener.NONE;

    public RacingCar(Track track) {
        this.track = track;
    }

    public void setSuspension(Suspension suspension) { [...]
    public void setTyres(Tyres tyres) { [...]
    public void setEngine(Engine engine) { [...]
    public void setListener(CarListener listener) { [...]
}
```

Mock Libraries

- Two main design philosophy:
  - DSL Libraries
  - Record/Replay Models Libraries
- Record Replay Frameworks
  - First train mocks and then verify expectations
- DSL Frameworks
  - Domain Specific Languages
  - Specifications embedded in “Java” Code

Mocking with EasyMock

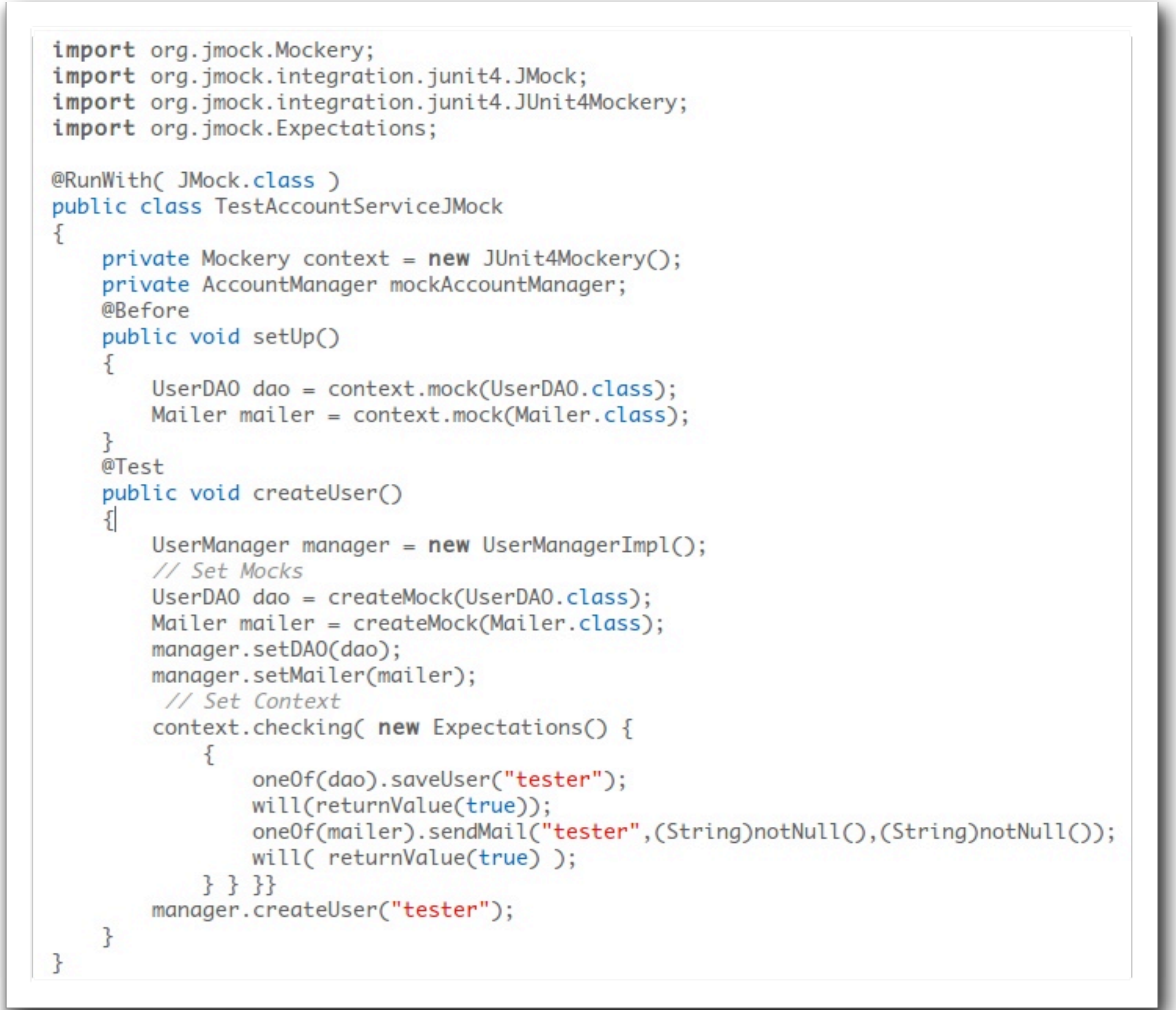


```java
import static org.easymock.EasyMock.*;

public class EasyMockUserManagerTest {
    @Test
    public void createUser() {
        // create the instance we'd like to test
        UserManager manager = new UserManagerImpl();
        UserDAO dao = createMock(UserDAO.class);
        Mailer mailer = createMock(Mailer.class);
        manager.setDAO(dao);
        manager.setMailer(mailer);
        // record expectations
        expect(dao.saveUser("tester")).andReturn(true);
        expect(mailer.sendMail(eq("tester"), (String)notNull(),
        (String)notNull())).andReturn(true);
        replay(dao, mailer);
        // invoke our method
        manager.createUser("tester");
        // verify that expectations have been met
        verify(mailer, dao);
    }
}
```

EasyMock Test

- Create Mock objects
  - Java Reflections API
- Record Expectation
  - expect methods
- Invoke Primary Test
  - replay method
- Verify Expectation
  - verify method

JMock Example



```
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;
import org.jmock.Expectations;

@RunWith( JMock.class )
public class TestAccountServiceJMock
{
    private Mockery context = new JUnit4Mockery();
    private AccountManager mockAccountManager;
    @Before
    public void setUp()
    {
        UserDAO dao = context.mock(UserDAO.class);
        Mailer mailer = context.mock(Mailer.class);
    }
    @Test
    public void createUser()
    {
        UserManager manager = new UserManagerImpl();
        // Set Mocks
        UserDAO dao = createMock(UserDAO.class);
        Mailer mailer = createMock(Mailer.class);
        manager.setDAO(dao);
        manager.setMailer(mailer);
         // Set Context
        context.checking( new Expectations() {
            {
                oneOf(dao).saveUser("tester");
                will(returnValue(true));
                oneOf(mailer).sendMail("tester",(String)notNull(),(String)notNull());
                will( returnValue(true) );
            } } }}
        manager.createUser("tester");
    }
}
```

JMock features (intro)

- JMock previous versions required subclassing
  - Not so smart in testing
  - Now directly integrated with Junit4
  - JMock tests requires more typing
- JMock API is extensible

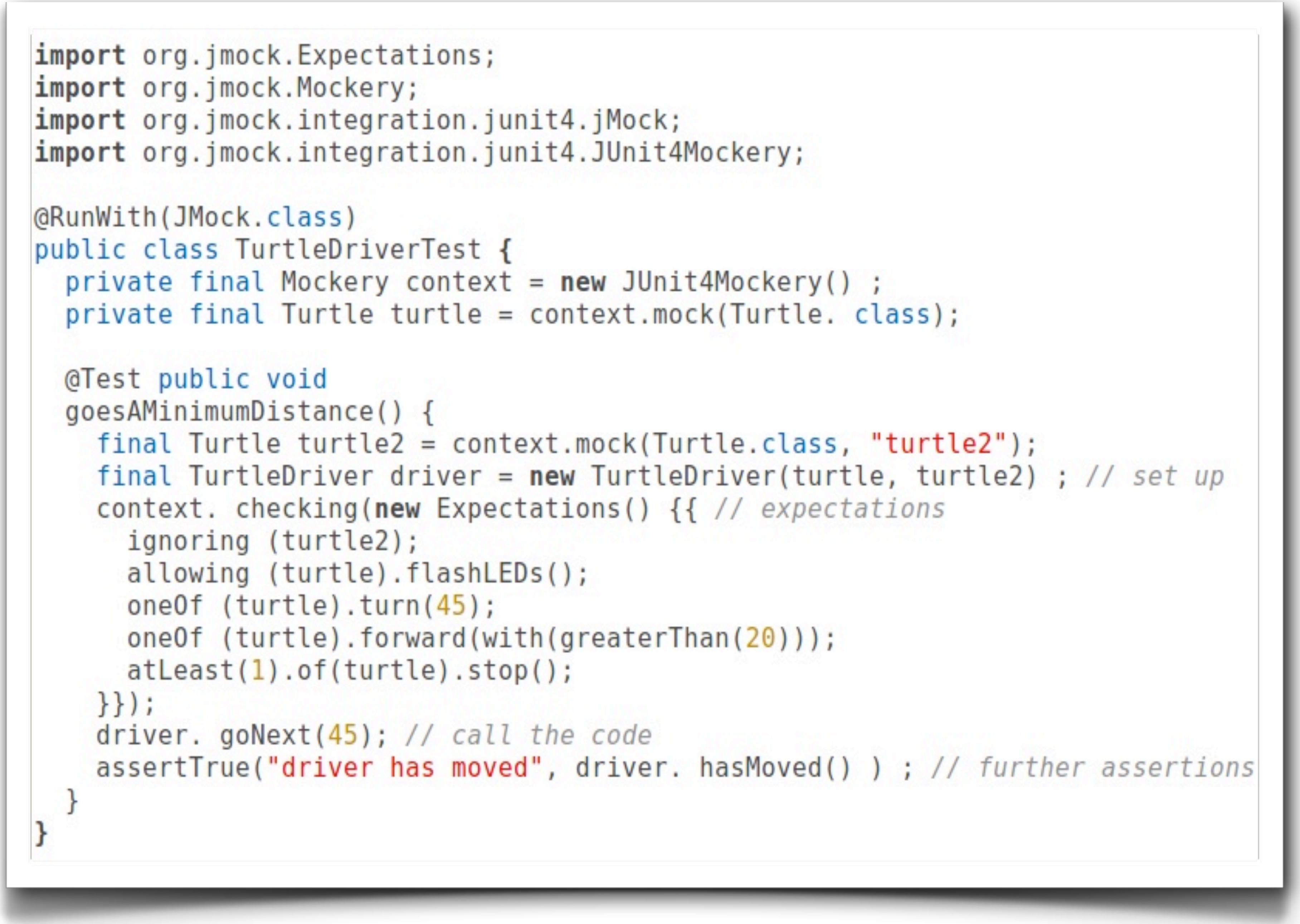
JMock features

- JMock syntax relies heavily on chained method calls
  - Sometimes difficult to decipher and to debug

- Common Patterns:

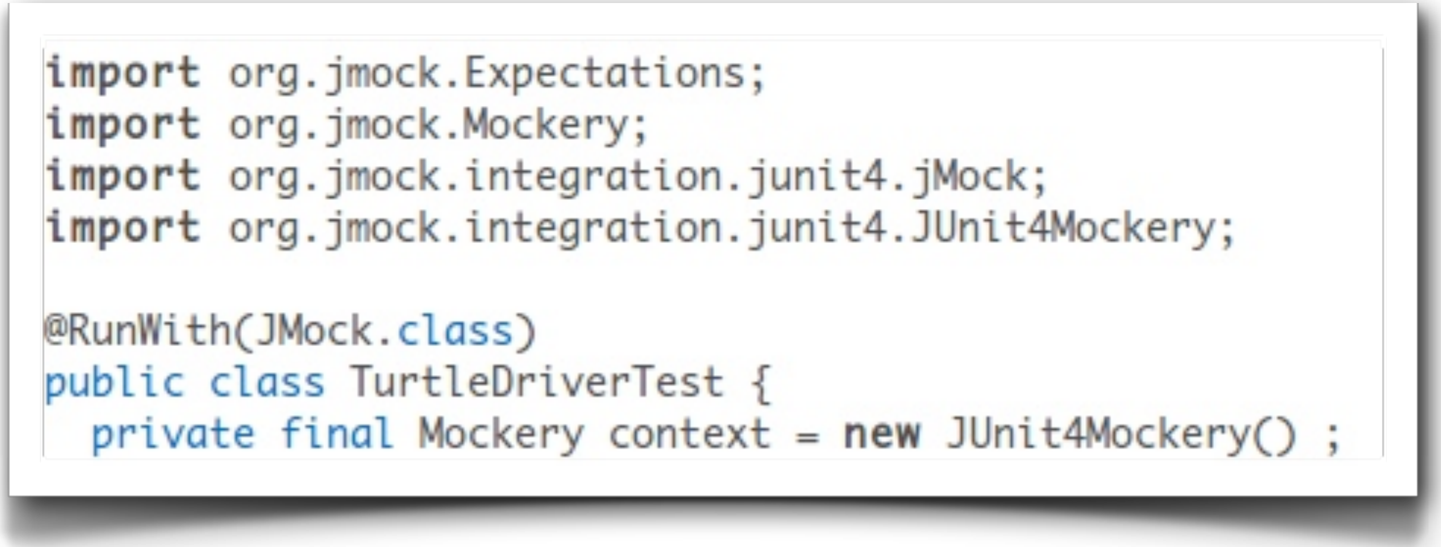
```
invocation-count(mockobject).method(arguments);
inSequence(sequence-name);
when(state-machine.is(state-name));
will(action);
then(state-machine.is(new-state name));
```

JMock Example



```java
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.jMock;
import org.jmock.integration.junit4.JUnit4Mockery;

@RunWith(JMock.class)
public class TurtleDriverTest {
  private final Mockery context = new JUnit4Mockery() ;
  private final Turtle turtle = context.mock(Turtle. class);

  @Test public void
  goesAMinimumDistance() {
    final Turtle turtle2 = context.mock(Turtle.class, "turtle2");
    final TurtleDriver driver = new TurtleDriver(turtle, turtle2) ; // set up
    context. checking(new Expectations() {{ // expectations
      ignoring (turtle2);
      allowing (turtle).flashLEDs();
      oneOf (turtle).turn(45);
      oneOf (turtle).forward(with(greaterThan(20)));
      atLeast(1).of(turtle).stop();
    }});
    driver. goNext(45); // call the code
    assertTrue("driver has moved", driver. hasMoved() ) ; // further assertions
  }
}
```

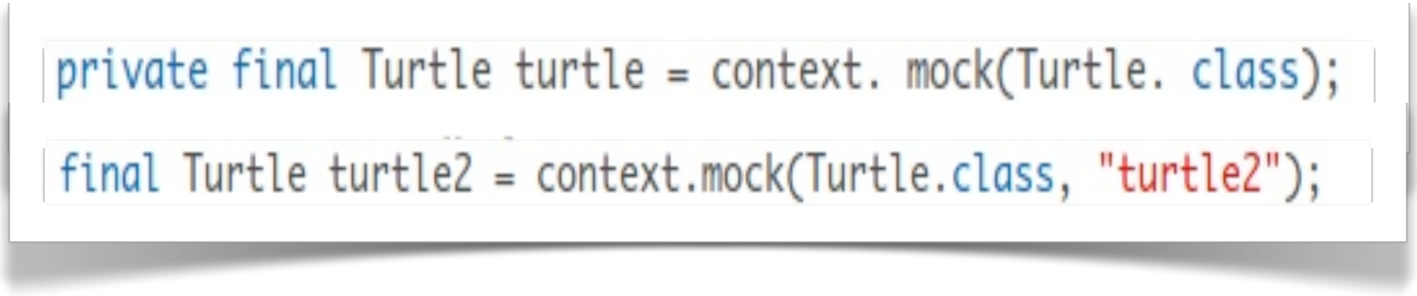
1. Test Fixture



```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.jMock;
import org.jmock.integration.junit4.JUnit4Mockery;

@RunWith(JMock.class)
public class TurtleDriverTest {
  private final Mockery context = new JUnit4Mockery() ;
```

- Mockery represents the *context*
  - Neighboring objects it will communicate with
  - By convention the mockery is stored in an istance variable named `context`
- `@RunWith(JMock.class)` annotation
- `JUnit4Mockery` reports expectation failures as JUnit4 test failures

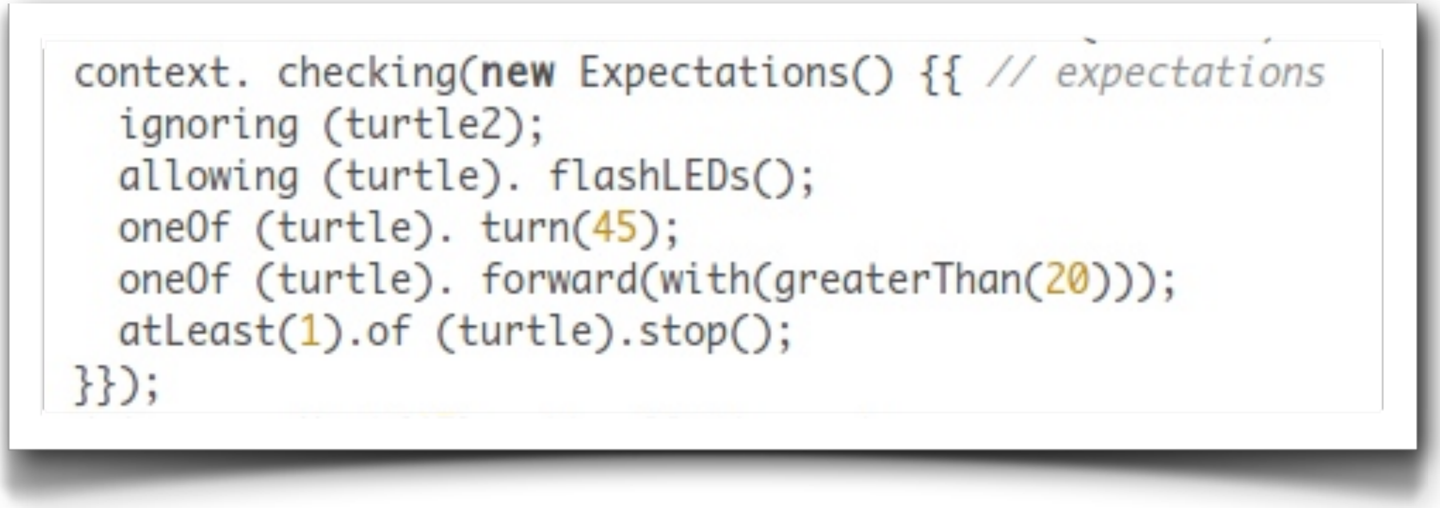
2. Create Mock Objects



```
private final Turtle turtle = context. mock(Turtle. class);

final Turtle turtle2 = context.mock(Turtle.class, "turtle2");
```

- The tests has two mock turtles
  - The first is a field in the test class
  - The second is local to the test
- References (fields and Vars) have to be `final`
  - Accessible from Anonymous `Expectations`
- The second mock has a specified name
  - JMock enforces usage of names except for the first (default)
  - This makes failures reporting more clear

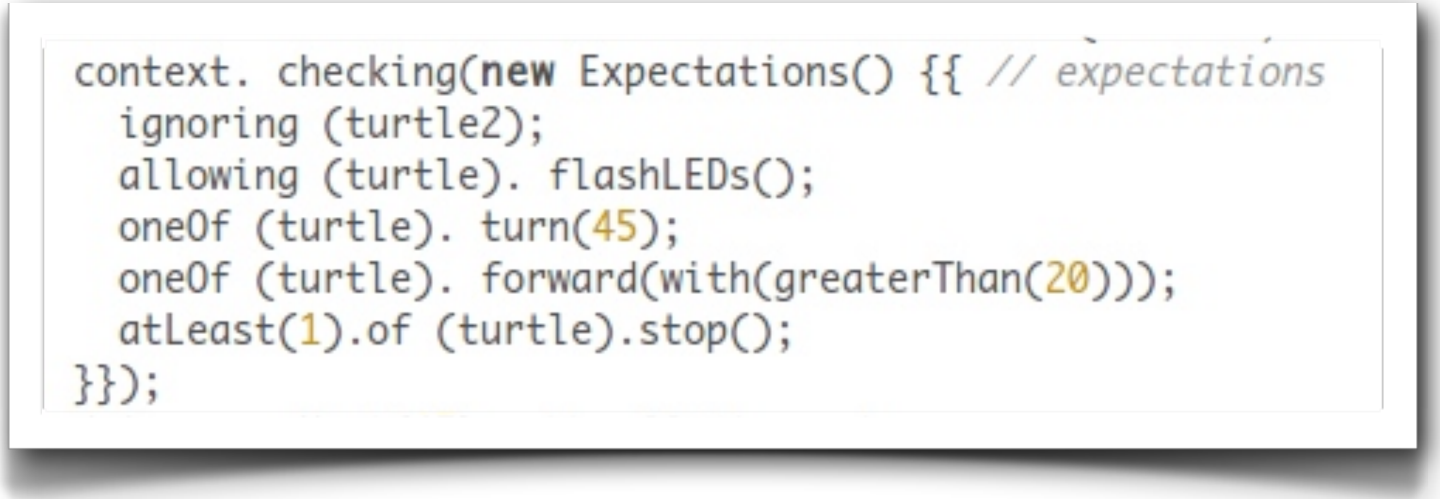
3. Tests with Expectations



```
context. checking(new Expectations() {{ // expectations
  ignoring (turtle2);
  allowing (turtle). flashLEDs();
  oneOf (turtle). turn(45);
  oneOf (turtle). forward(with(greaterThan(20)));
  atLeast(1).of (turtle).stop();
}});
```

- A test sets up it expectations in one or more *expectation blocks*
  - An expectation block can contain any number of expectations
  - Expectation blocks can be interleaved with calls to the code under test.

3. Tests with Expectations



```
context. checking(new Expectations() {{ // expectations
  ignoring (turtle2);
  allowing (turtle). flashLEDs();
  oneOf (turtle). turn(45);
  oneOf (turtle). forward(with(greaterThan(20)));
  atLeast(1).of (turtle).stop();
}});
```

- Expectations have the following structure:

```
invocation-count
  (mockobject).method(arguments);
inSequence(sequence-name);
when(state-machine.is(state-name));
will(action);
then(state-machine.is(new-state name));
```

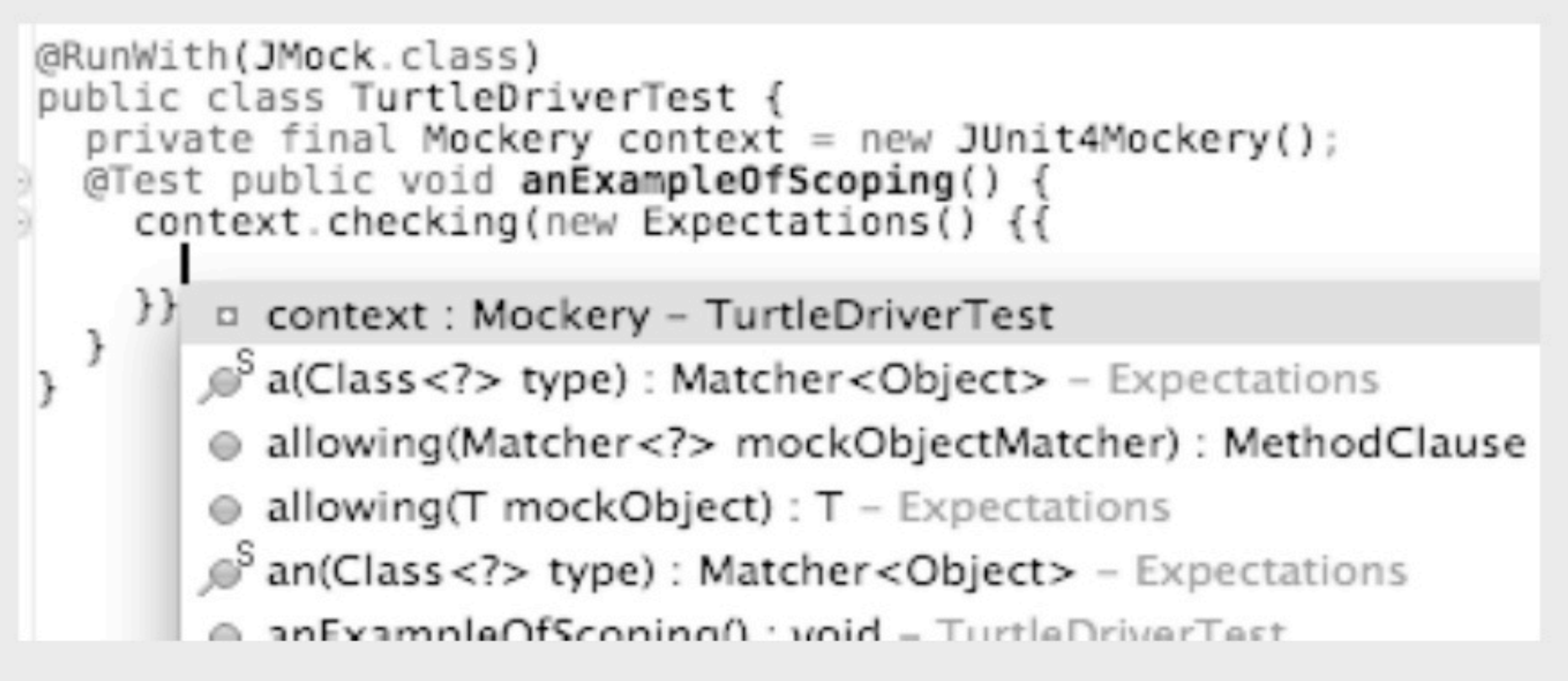
What are those double braces?

```
context.checking(new Expectations(){{
      oneOf(turtle).turn(45);
 }});
```

- Anonymous subclass of `Expectations`
- Baroque structure to provide a scope for building up expectations
  - Collection of expectation components
  - Is an example of **Builder Pattern**
  - Improves code completion

What are those double braces?

```
context.checking(new Expectations(){{
        oneOf(turtle).turn(45);
 }});
```



```
@RunWith(JMock.class)
public class TurtleDriverTest {
  private final Mockery context = new JUnit4Mockery();
  @Test public void anExampleOfScoping() {
    context.checking(new Expectations() {{
      
    }}
  }
}
```

□ context : Mockery – TurtleDriverTest
a(Class<?> type) : Matcher<Object> – Expectations
allowing(Matcher<?> mockObjectMatcher) : MethodClause
allowing(T mockObject) : T – Expectations
an(Class<?> type) : Matcher<Object> – Expectations
anExampleOfScoping() : void – TurtleDriverTest

Allowances and Expectations

```
context.checking(new Expectations(){{
    ignoring (turtle2);
    allowing (turtle).flashLEDs();
    oneOf(turtle).turn(45);
}});
```

- *Expectations* describe the interactions that are **essential** to the protocol we're testing
- *Allowances* **support** the interaction we're testing
  - `ignoring()` clause says that we don't care about messages sent to `turtle2`
  - `allowing()` clause matches any call to `flashLEDs` of `turtle`

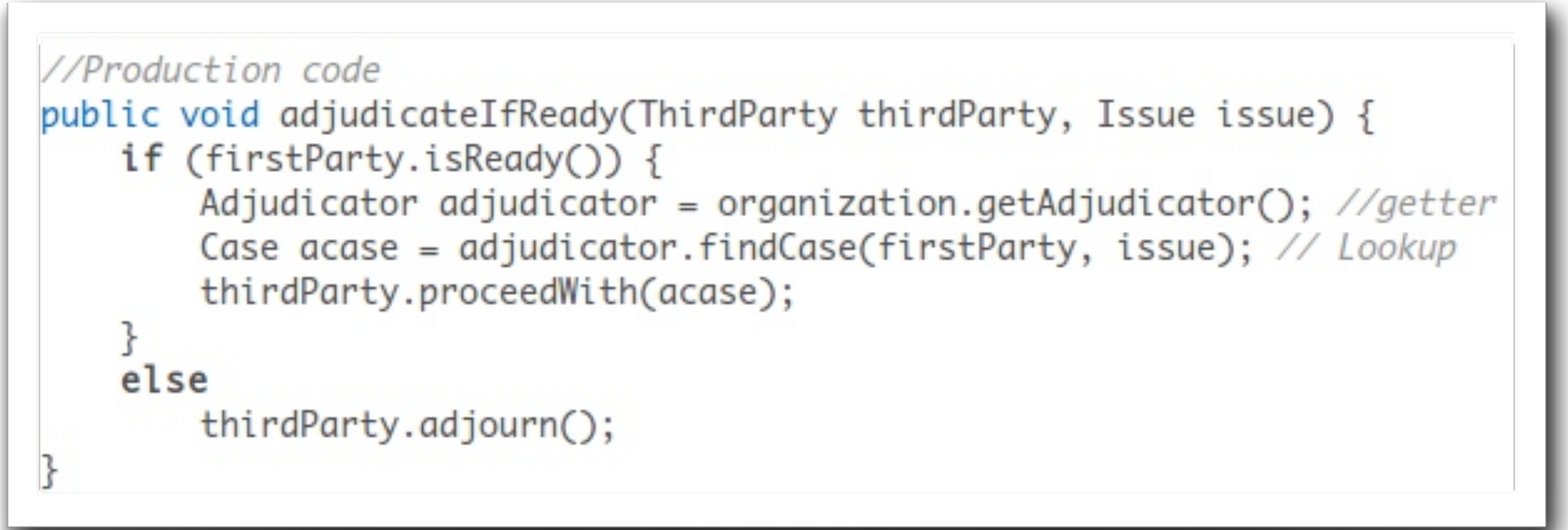
Allowances and Expectations

```
context.checking(new Expectations(){{
    ignoring (turtle2);
    allowing (turtle).flashLEDs();
    oneOf(turtle).turn(45);
}});
```

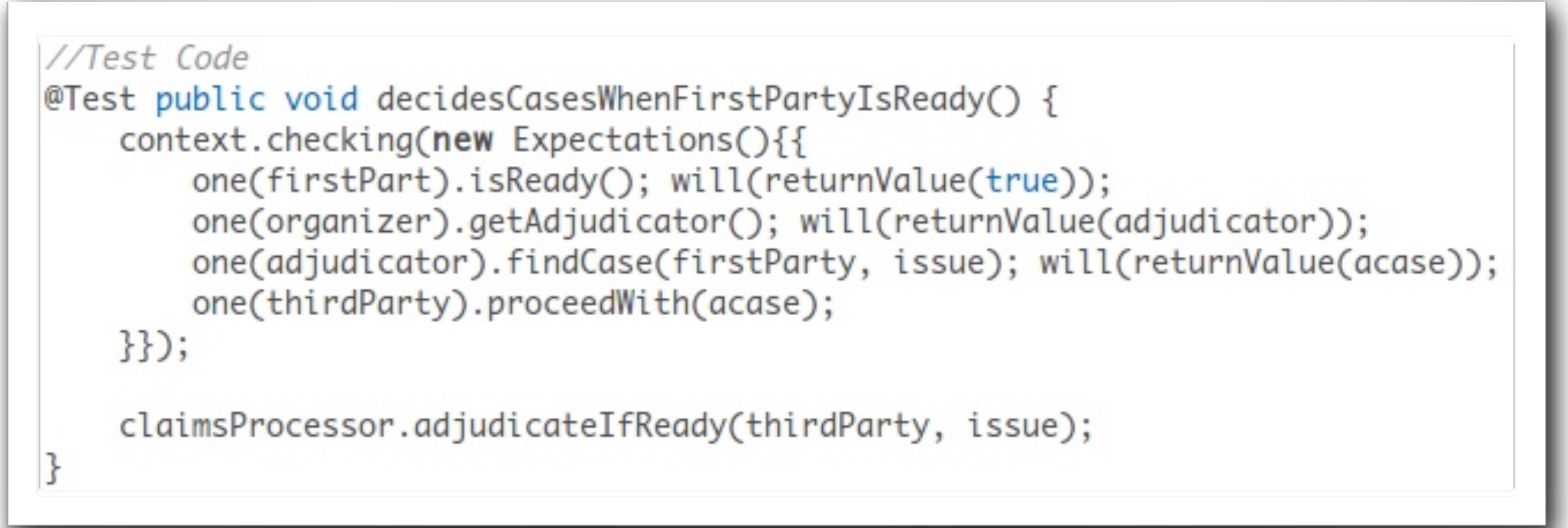
- Distinction between allowances and expectations is not rigid
- **Rule of Thumb**:
  - *Allow queries; Expect Commands*
- **Why**?
  - Commands could have side effects;
  - Queries don't change the world.

Expectations or … ?

Too Many Expectations……Ideas??



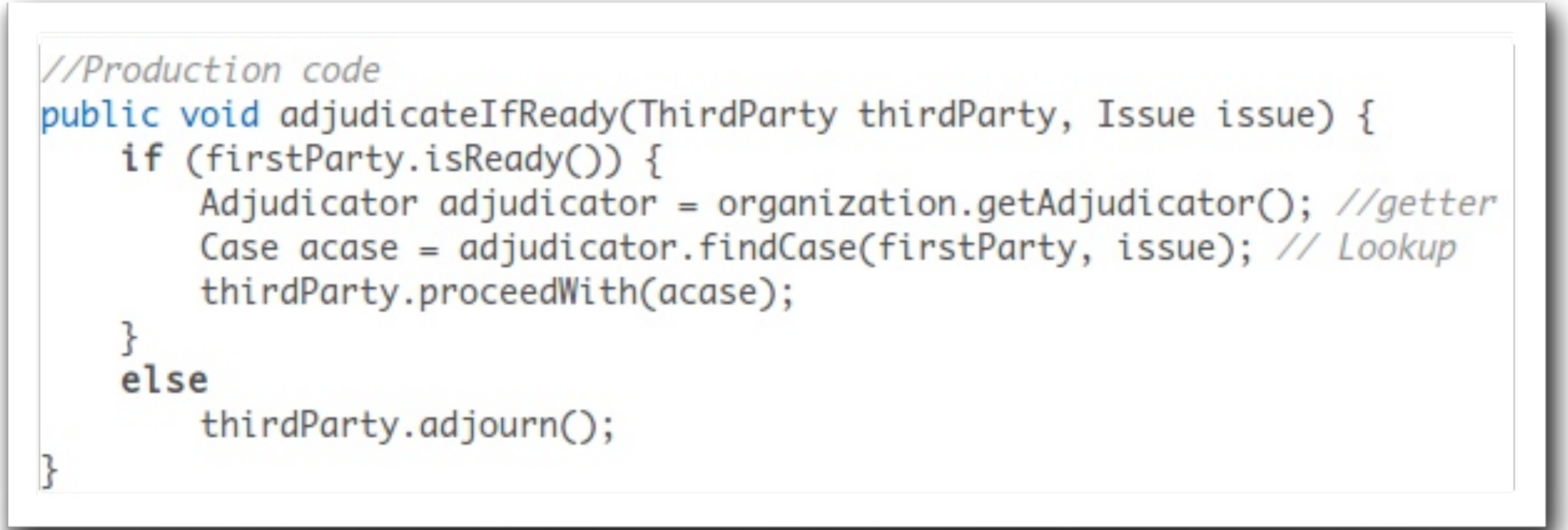
```java
//Production code
public void adjudicateIfReady(ThirdParty thirdParty, Issue issue) {
    if (firstParty.isReady()) {
        Adjudicator adjudicator = organization.getAdjudicator(); //getter
        Case acase = adjudicator.findCase(firstParty, issue); // Lookup
        thirdParty.proceedWith(acase);
    }
    else
        thirdParty.adjourn();
}
```



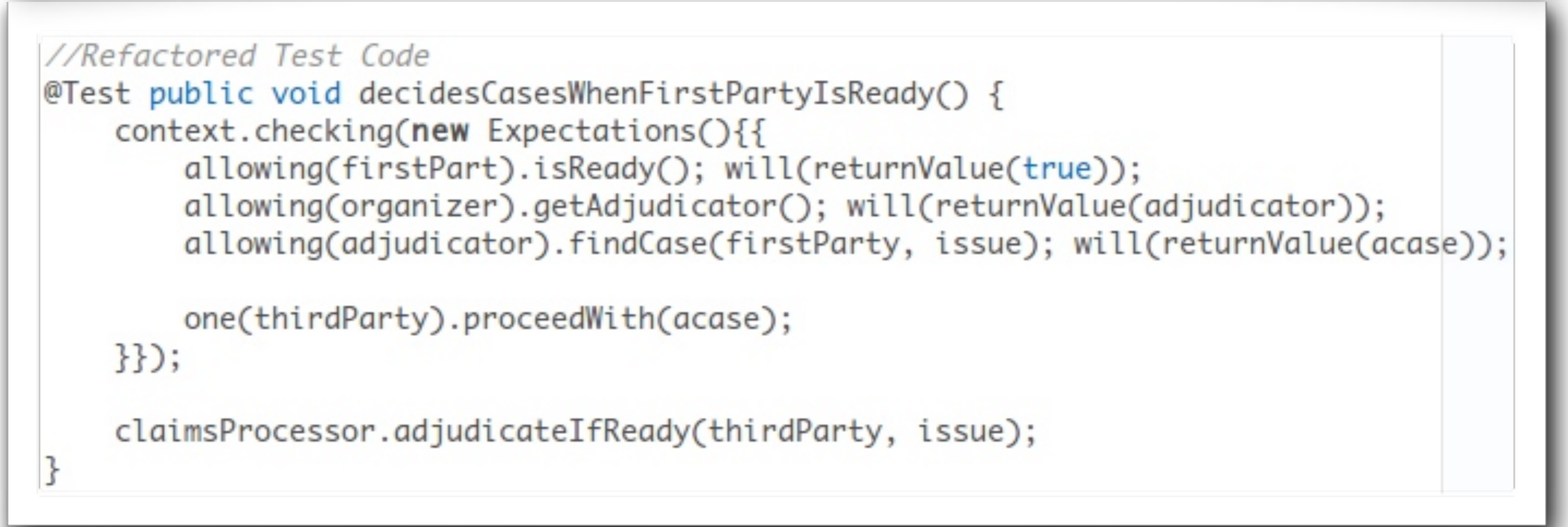
```java
//Test Code
@Test public void decidesCasesWhenFirstPartyIsReady() {
    context.checking(new Expectations(){{
        one(firstPart).isReady(); will(returnValue(true));
        one(organizer).getAdjudicator(); will(returnValue(adjudicator));
        one(adjudicator).findCase(firstParty, issue); will(returnValue(acase));
        one(thirdParty).proceedWith(acase);
    }});

    claimsProcessor.adjudicateIfReady(thirdParty, issue);
}
```

Expectations or … ?

Too Many Expectations……Ideas??



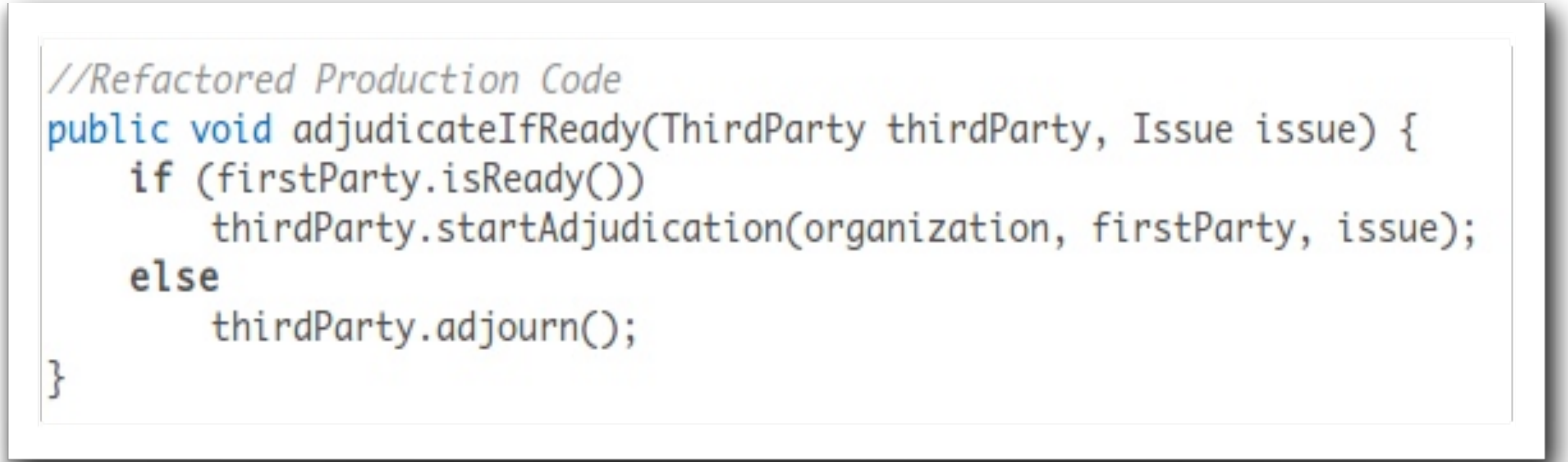
```
//Production code
public void adjudicateIfReady(ThirdParty thirdParty, Issue issue) {
    if (firstParty.isReady()) {
        Adjudicator adjudicator = organization.getAdjudicator(); //getter
        Case acase = adjudicator.findCase(firstParty, issue); // Lookup
        thirdParty.proceedWith(acase);
    }
    else
        thirdParty.adjourn();
}
```



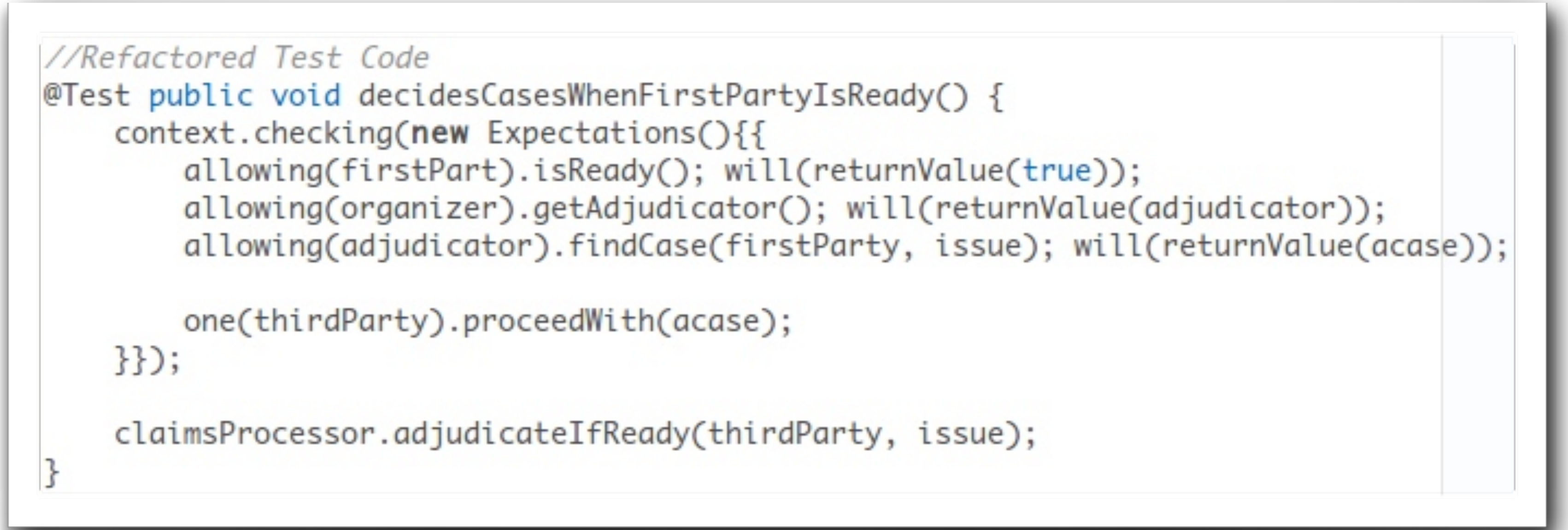
```
//Refactored Test Code
@Test public void decidesCasesWhenFirstPartyIsReady() {
    context.checking(new Expectations(){{
        allowing(firstPart).isReady(); will(returnValue(true));
        allowing(organizer).getAdjudicator(); will(returnValue(adjudicator));
        allowing(adjudicator).findCase(firstParty, issue); will(returnValue(acase));

        one(thirdParty).proceedWith(acase);
    }});

    claimsProcessor.adjudicateIfReady(thirdParty, issue);
}
```

Expectations or … ?

Too Many Expectations……Ideas??



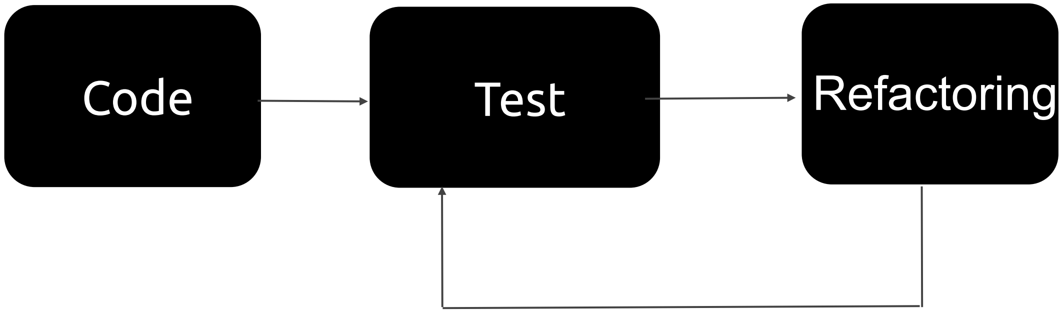
```
//Refactored Production Code
public void adjudicateIfReady(ThirdParty thirdParty, Issue issue) {
    if (firstParty.isReady())
        thirdParty.startAdjudication(organization, firstParty, issue);
    else
        thirdParty.adjourn();
}
```



```
//Refactored Test Code
@Test public void decidesCasesWhenFirstPartyIsReady() {
    context.checking(new Expectations(){{
        allowing(firstPart).isReady(); will(returnValue(true));
        allowing(organizer).getAdjudicator(); will(returnValue(adjudicator));
        allowing(adjudicator).findCase(firstParty, issue); will(returnValue(acase));

        one(thirdParty).proceedWith(acase);
    }});

    claimsProcessor.adjudicateIfReady(thirdParty, issue);
}
```

Development process

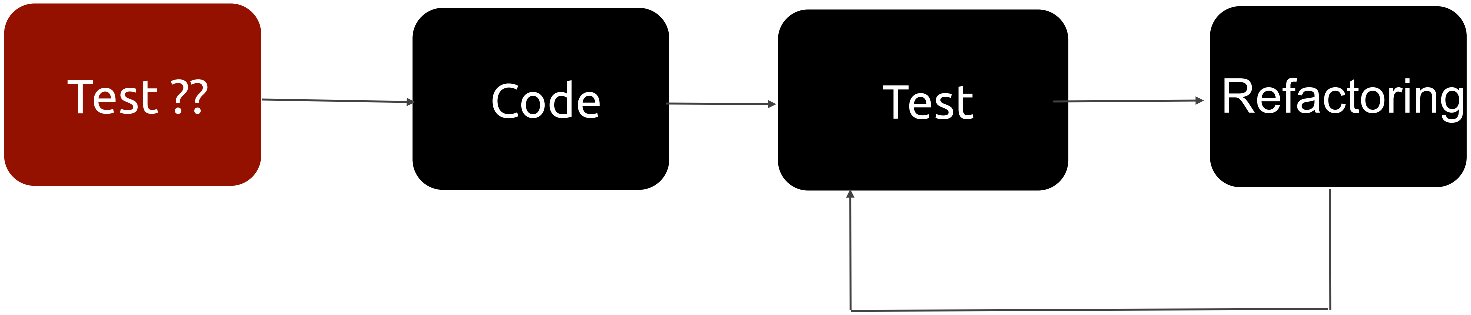
- Let's think about the development process of this example:



- **Q: Does make sense to write tests before writing production code?**
- **A: Two Keywords**
  - ○ **TDD:** Test Driven Development
  - ○ **Test-first Programming**

Development process

- Let's think about the development process of this example:



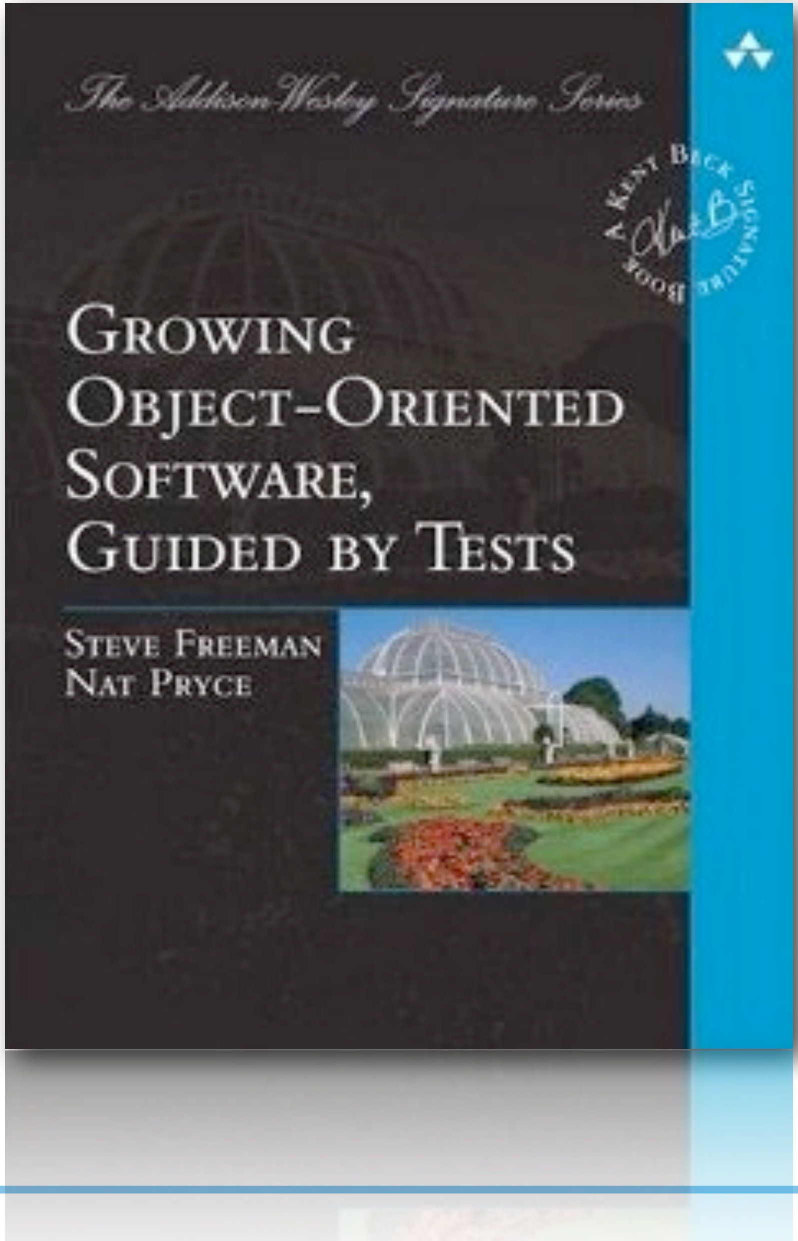
- **Q: Does make sense to write tests before writing production code?**
- **A: Two Keywords**
  - **TDD:** Test Driven Development
  - **Test-first Programming**

References

Growing Object-Oriented Software, Guided By Tests

Freeman and Pryce, Addison Wesley 2010

JMock Project WebSite (http://jmock.org)


The Addison-Wesley Signature Series
A Kent Beck Signature Book
Growing Object-Oriented Software, Guided by Tests
Steve Freeman
Nat Pryce