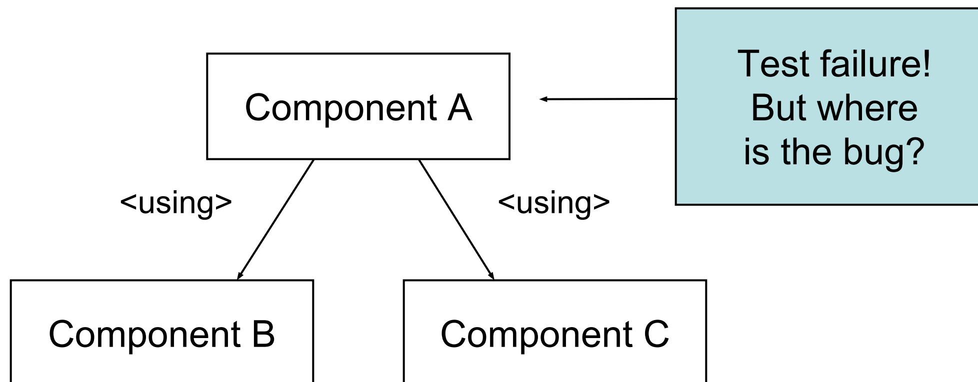
Unit Testing using JUnit

INF5750/9750 - Lecture 4 (Part II)

Problem area

- Code components must be tested!
  - Confirms that your code works
- Components must be tested in isolation
  - A functional test can tell you that a bug exists in the implementation
  - A unit test tells you where the bug is located



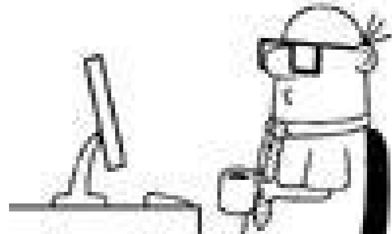
Example: The Calculator

```
public interface Calculator
{
    int add( int number1, int number2 );


    int multiply( int number1, int number2 );
}
```

```
public class DefaultCalculator
    implements Calculator
{
    public int add( int number1, int number2 )
    {
        return number1 + number2;
    }


    public int multiply( int number1, int number2 )
    {
        return number1 * number2;
    }
}
```

Approaches to unit testing

- Write a small command-line program, enter values, and verify output
  - Involves your ability to type numbers
  - Requires skills in mental calculation
  - Doesn’t verify your code when its released



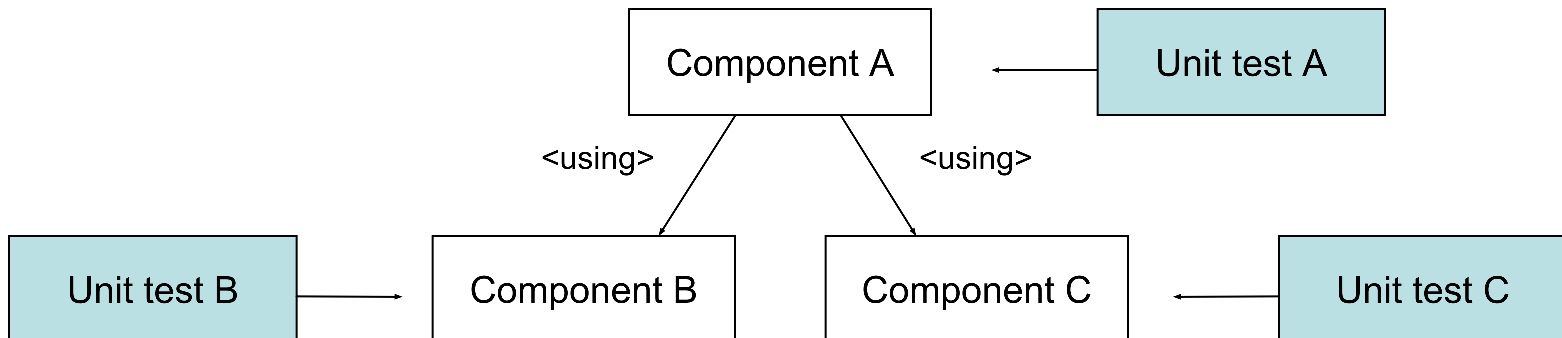
Approaches to unit testing

- Write a simple test program
  - Objective and preserves testing efforts
  - Requires you to monitor the screen for error messages
  - Inflexible when more tests are needed

```
public class TestCalculator
{
    public static void main( String[] args )
    {
        Calculator calculator = new DefaultCalculator();


        int result = calculator.add( 8, 7 );


        if ( result != 15 )
        {
            System.out.println( ”Wrong result: ” + result );
        }
    }
}
```

The preferred solution

- Use a unit testing framework like *JUnit*
  - A *unit* is the smallest testable component in an application
  - A unit is in most cases a *method*
  - A unit does not depend on other components which are *not unit tested themselves*
  - Focus on whether a method is following its *API contract*



JUnit

- *De facto* standard for developing unit tests in Java
  - One of the most important Java libraries ever developed
  - Made unit testing easy and popular among developers
  - Driven by annotations
  - Spring provides integration with JUnit

Using JUnit annotations

- No need to follow naming conventions
  - Tests identified by ***@Test*** annotation
  - Fixture methods identified by ***@Before*** and ***@After*** annotations

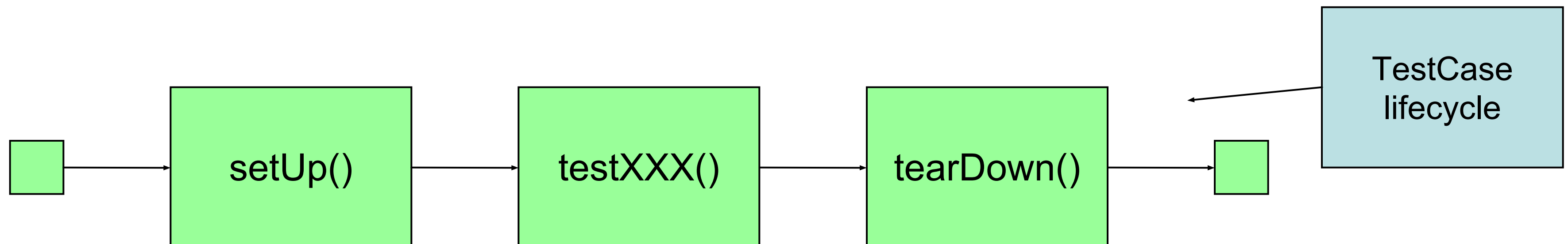
- Class-scoped fixture
  - Identified by **@BeforeClass** and **@AfterClass** annotations
  - Useful for setting up expensive resources, but be careful…

- Ignored tests
  - Identified by **@Ignore** annotation
  - Useful for slow tests and tests failing for reasons beyond you

- Timed tests
  - Identified by providing a parameter ***@Test(timeout=500)***
  - Useful for benchmarking, network, deadlock testing

Test fixtures

- Tests may require common resources to be set up
  - Complex data structures
  - Database connections
- A *fixture* is a set of common needed resources
  - A fixture can be created by overriding the *setUp* and *tearDown* methods from TestCase
  - *setUp* is invoked before each test, *tearDown* after



JUnit Calculator test

Static import of Assert

Fixture

Fixture

Use assertEquals to verify output

```
import static junit.framework.Assert.*;

public class CalculatorTest
{
    Calculator calculator;


    @Before
    public void before()
    {
        calculator = new DefaultCalculator();
    }


    @Test
    public void addTest()
    {
        int sum = calculator.add( 8, 7 );
        assertEquals( sum, 15 );
    }


    @Test
    public void multiplyTest()
    {
    }
}
```

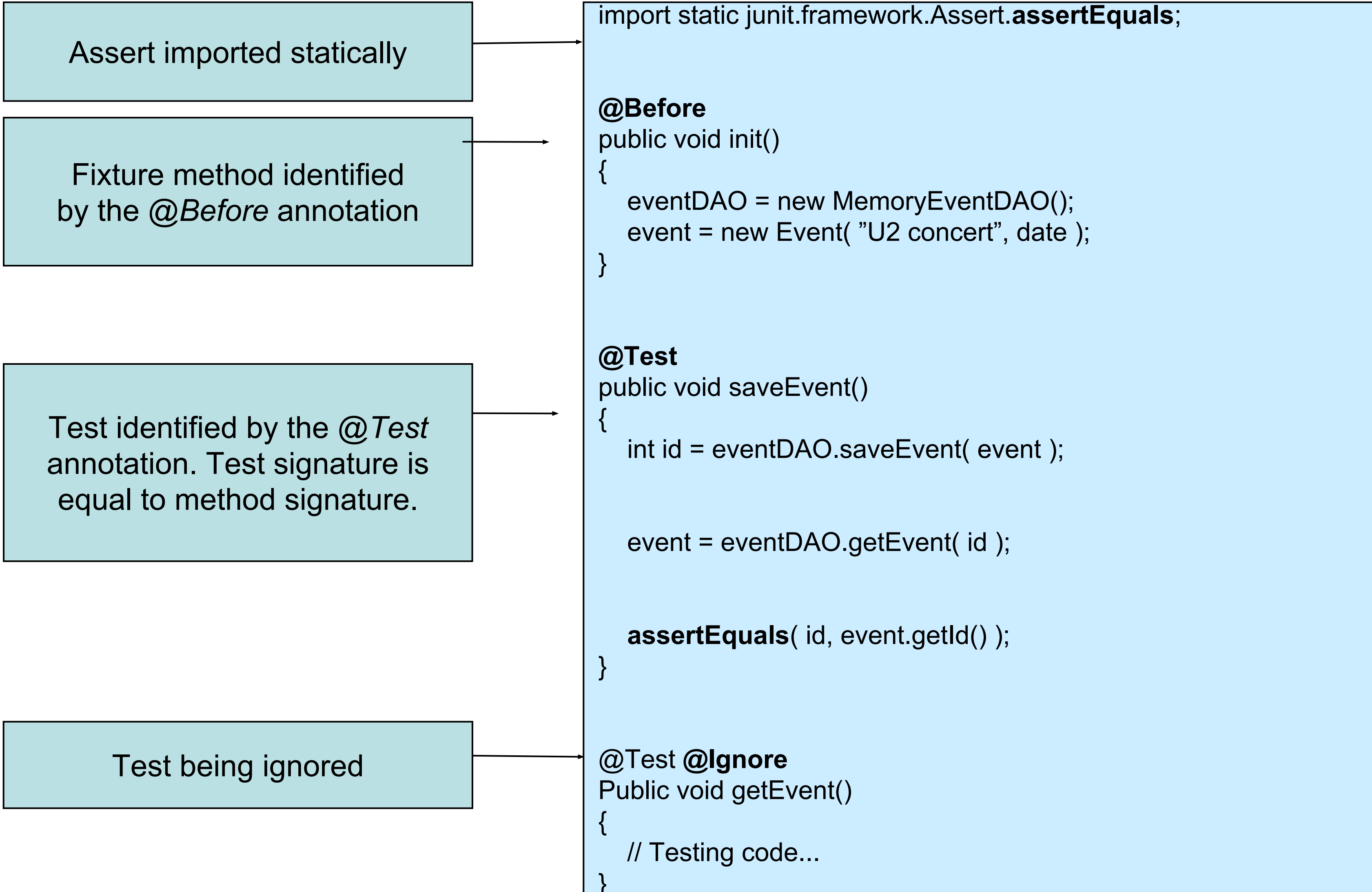
Example: The EventDAO

```
public class Event()
{
    private int id;
    private String title;
    private Date date;


    // constructors
    // get and set methods
}
```

EventDAO interface

```
public interface EventDAO
{
    int saveEvent( Event event );


    Event getEvent( int id );


    void deleteEvent( Event event );
}
```

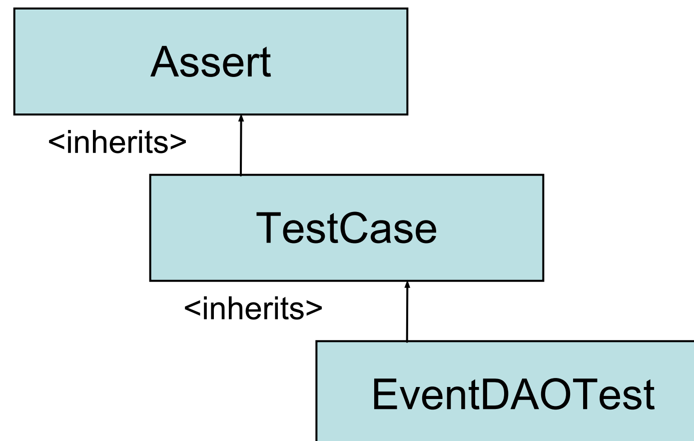
EventDAOTest



The Assert class

- Contains methods for testing whether:
  - Conditions are true or false
  - Objects are equal or not
  - Objects are null or not

- If the test fails, an AssertionFailedError is thrown
- All methods have overloads for various parameter types
- Methods available because *TestCase* inherits *Assert*



Assert methods

Method	***Description***
assertTrue(boolean)	Asserts that a condition is true.
assertFalse(boolean)	Asserts that a condition is false.
assertEquals(Object, Object)	Asserts that two objects are equal.
assertArrayEquals (Object[], Object[])	Asserts that all elements in the two arrays are equal
assertNotNull(Object)	Asserts that an object is *not* null.
assertNull(Object)	Asserts that an object is null.
assertSame(Object, Object)	Asserts that two references refer to the same object.
assertNotSame(Object, Object)	Asserts that two references do *not* refer to the same object.
fail(String)	Asserts that a test fails, and prints the given message.

Assert in EventDAOTest

Asserts that the saved object is equal to the retrieved object

Saves and retrieves an Event with the generated identifier

An object is expected

Asserts that null is returned when no object exists

```
@Test
public void testSaveEvent()
{
    int id = eventDAO.saveEvent( event );
    event = eventDAO.getEvent( id );
    assertEquals( id, event.getId() );
    assertEquals( ”U2 concert”, event.getTitle() );
}


@Test
public void testGetEvent()
{
    int id = eventDAO.saveEvent( event );
    event = eventDAO.getEvent( id );
    assertNotNull( event );
    event = eventDAO.getEvent( -1 );
    assertNull( event );
}
```

Core Hamcrest Matchers - assertThat

Examples	***The matchers***
assertThat("good", allOf(equalTo("good"), startsWith("good")));	**allOf**(...) - same as && (AND) **equalTo**(...) - object equivalence, **startsWith**(...) - string matching
assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));	**not**(...) - same as != (NOT EQUAL)
assertThat("good", anyOf(equalTo("bad"), equalTo("good")));	**anyOf** (...) - same as || (OR)

assertThat(theBiscuit, equalTo(myBiscuit));
assertThat(theBiscuit, is(equalTo(myBiscuit)));
assertThat(theBiscuit, is(myBiscuit));
	is - syntactic sugar - all 3 statements are same

- Don’t use too much sugar!! - Only use for readability

Testing Exceptions

- Methods may be required to throw exceptions
- Expected exception can be declared as an annotation
  - @Test(expected = UnsupportedOperationException.class)

Annotation declares that an exception of class UnsupportedOperationException is supposed to be thrown

```
@Test( expected = UnsupportedOperationException.class )
public void divideByZero()
{
    calculator.divide( 4, 0 );
}
```

Running JUnit

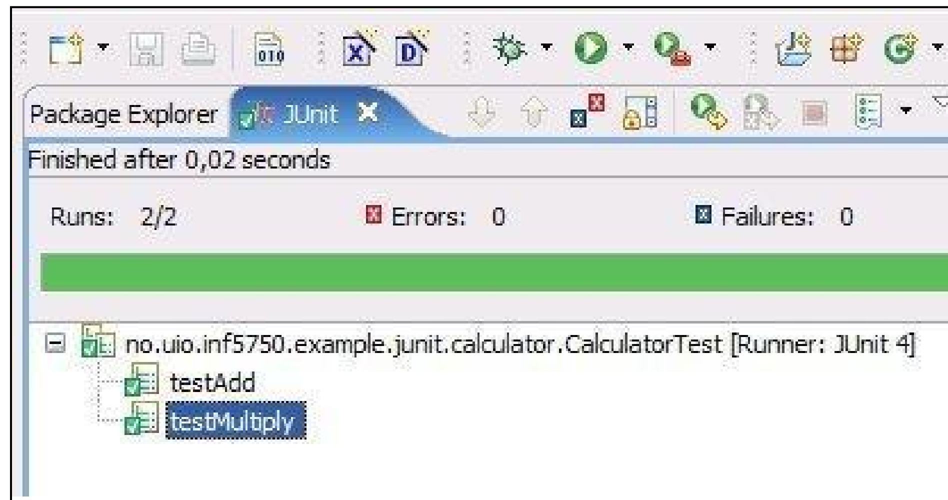
- Textual test runner
  - Used from the command line
  - Easy to run

- Integrated with Eclipse
  - Convenient, integrated testing within your development environment!

- Integrated with Maven
  - Gets included in the build lifecycle!

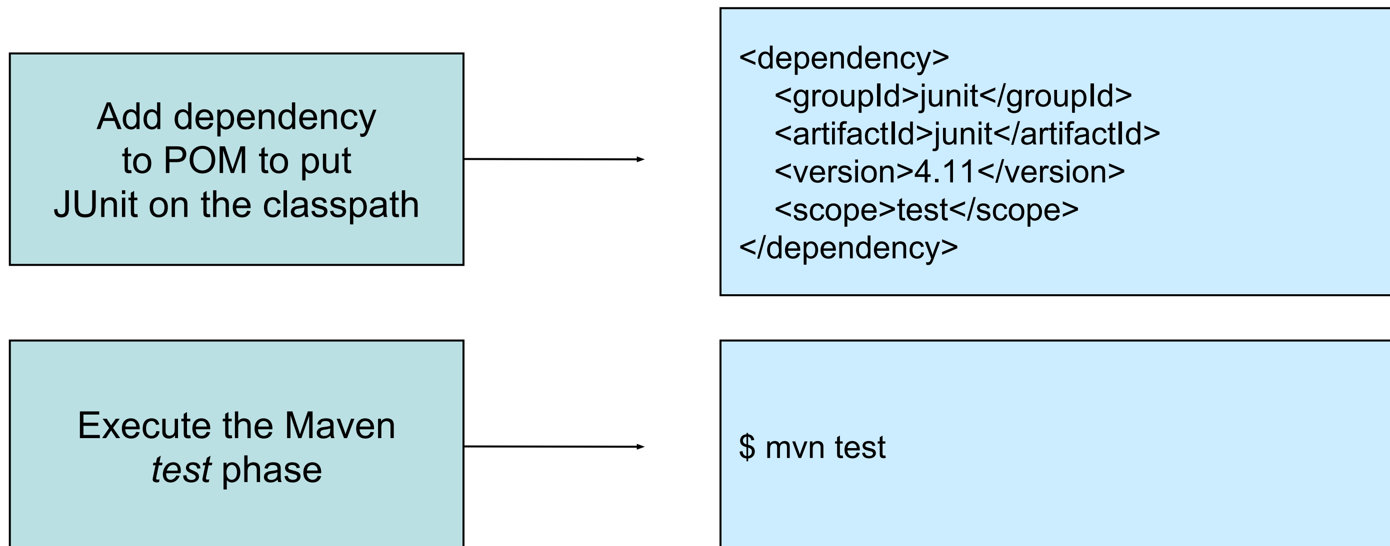
JUnit with Eclipse

- Eclipse features a JUnit view
- Provides an informative GUI displaying test summaries
- Lets you edit the code, compile and test without leaving the Eclipse environment



JUnit with Maven

- Maven provides support for automated unit testing with JUnit
- Unit testing is included in the build lifecycle
  - Verifies that existing components work when other components are added or changed



JUnit with Maven

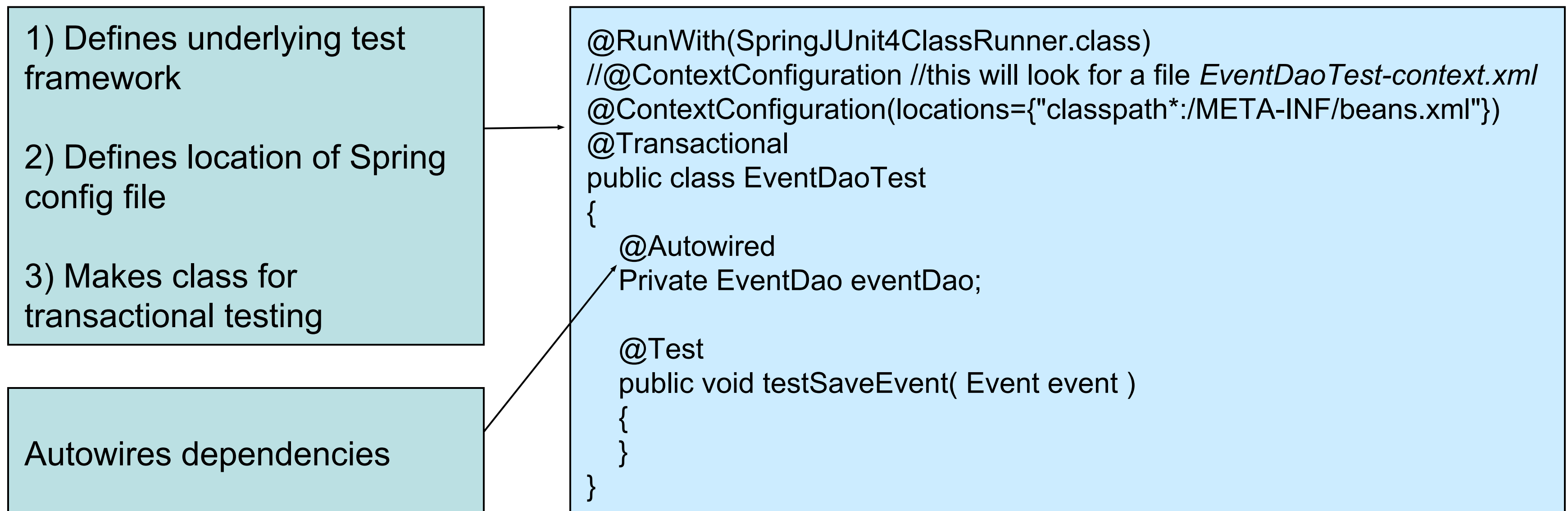
- Maven requires all test-class names to contain *Test*
- Standard directory for test classes is src/test/java
- The *test* phase is mapped to the ***Surefire*** plugin
- Surefire will generate reports based on your test runs
- Reports are located in *target/surefire-reports*

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running no.uio.inf5750.example.junit.calculator.CalculatorTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.09 sec
Running no.uio.inf5750.example.junit.event.EventDAOTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.03 sec

Results :
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Sun Sep 16 13:40:58 CEST 2007
[INFO] Final Memory: 3M/127M
[INFO] ------------------------------------------------------------------------
```

Spring test support

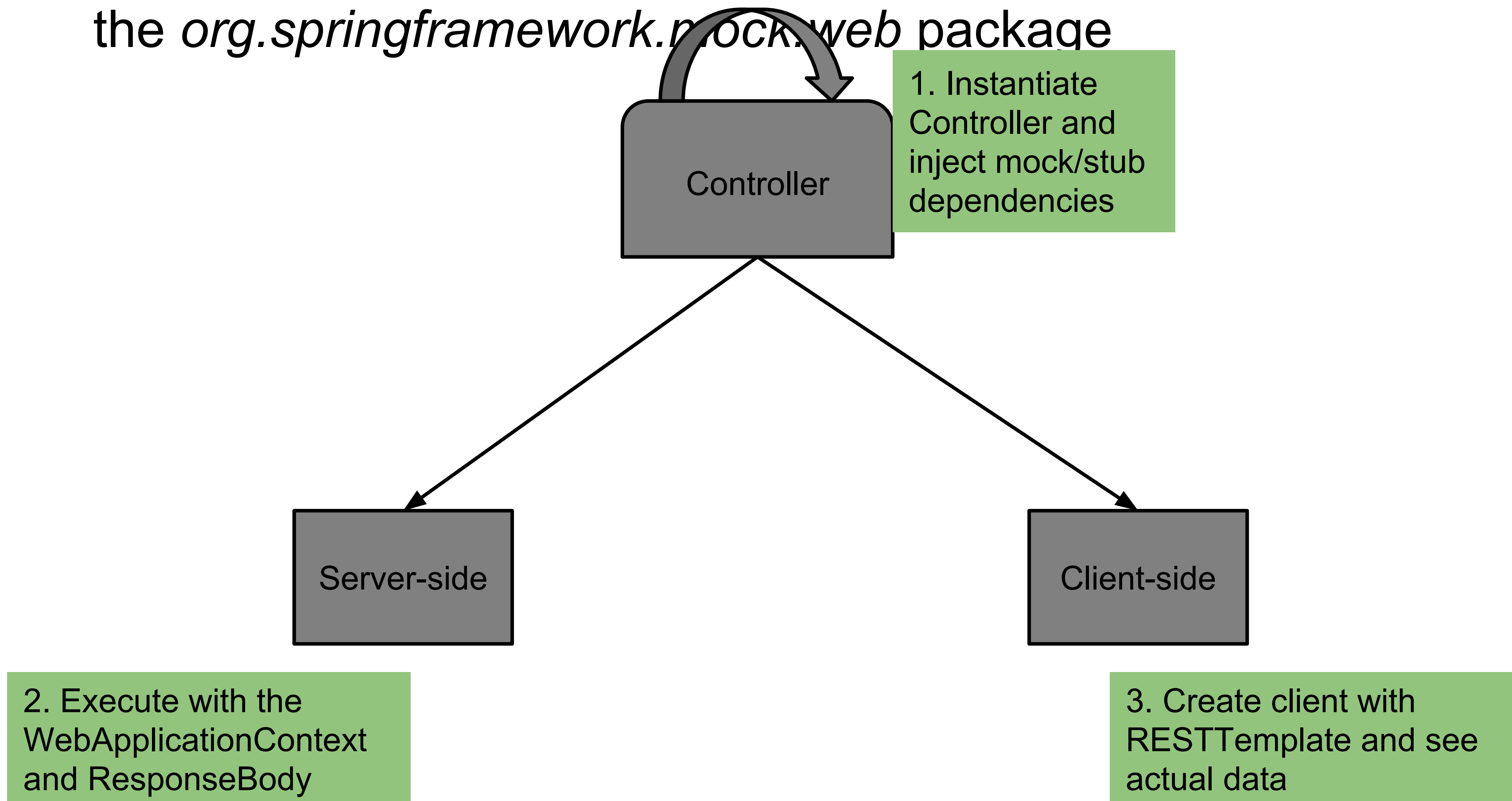
- Spring has excellent test support providing:
  - IoC container caching
  - Dependency injection of test fixture instances / dependencies
  - Transaction management and rollback

- Spring (spring-test) integrates nicely with JUnit



Unit Testing Spring MVC Controllers

- To test your Spring MVC Controllers, use *ModelAndViewAssert* combined with *MockHttpServletRequest*, *MockHttpSession*, and so on from the *org.springframework.mock.web* package



Spring MVC - Server-side

- Due to autowiring, find the correct ***@WebAppConfiguration***
- ***MockMvc*** class provides testing methods for method, headers, request, response etc.
- *standaloneSetup* will create WebApplicationContext with default Spring configuration for a webapp

With full Web Application Spring Configuration

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration("my-servlet-context.xml")
public class MyWebTests {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc =
            MockMvcBuilders.webAppContextSetup(this.wac).build();
    }
    // ...
}
```

With controller-specific setup (more unit-like)

```
public class MyWebTests {

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.standaloneSetup(
                new AccountController()).build();
    }
    // ...
}
```

MockMvc usage

- *perform()* - does requests

```
@Test
public void testReadId() throws Exception {
    this.mockMvc.perform(get("/read/1")
      .accept(MediaType.parseMediaType("text/html;charset=UTF-8")))
      .andExpect(status().isOk())
      .andExpect(content().contentType("text/html"))
      .andExpect(message.value("Message number 1 was hello"));
}
```

- *model()* - access the returned model
- *status()* - for HTTP Status
- *accept()* - accept mime type
- For repeated actions, in standalone setup you can:

```
standaloneSetup(new BaseController()) {
    .alwaysExpect(status().isOk())
    .alwaysExpect(content().contentType("application/json;charset=UTF-8"))
    .build()
}
```

Best practices

- One unit test for each tested method
  - Makes debugging easier
  - Easier to maintain
- Choose descriptive test method names
  - TestCase: Use the testXXX naming convention
  - Annotations: Use the method signature of the tested method
- Automate your test execution
  - If you add or change features, the old ones must still work
  - Also called *regression testing*
- Test more than the ”happy path”
  - Out-of-domain values
  - Boundary conditions

Advantages of unit testing

- Improves debugging
  - Easy to track down bugs
- Facilitates refactoring
  - Verifies that existing features still work while changing the code structure
- Enables teamwork
  - Lets you deliver tested components without waiting for the whole application to finish
- Promotes object oriented design
  - Requires your code to be divided in small, re-usable units
- Serving as developer documentation
  - Unit tests are samples that demonstrates usage of the API

Mocking Tests...

- We deliberately skip mocking in this course
- And integration tests
- Results in faster testing of the whole system

- Not because we think its not useful, but limitation of time

Resources

- Vincent Massol: *JUnit in Action*
  - Two free sample chapters
  - http://www.manning.com/massol
- JUnit home page (www.junit.org)
  - Articles and forum
- Spring documentation chapter 11
  - http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/testing.html