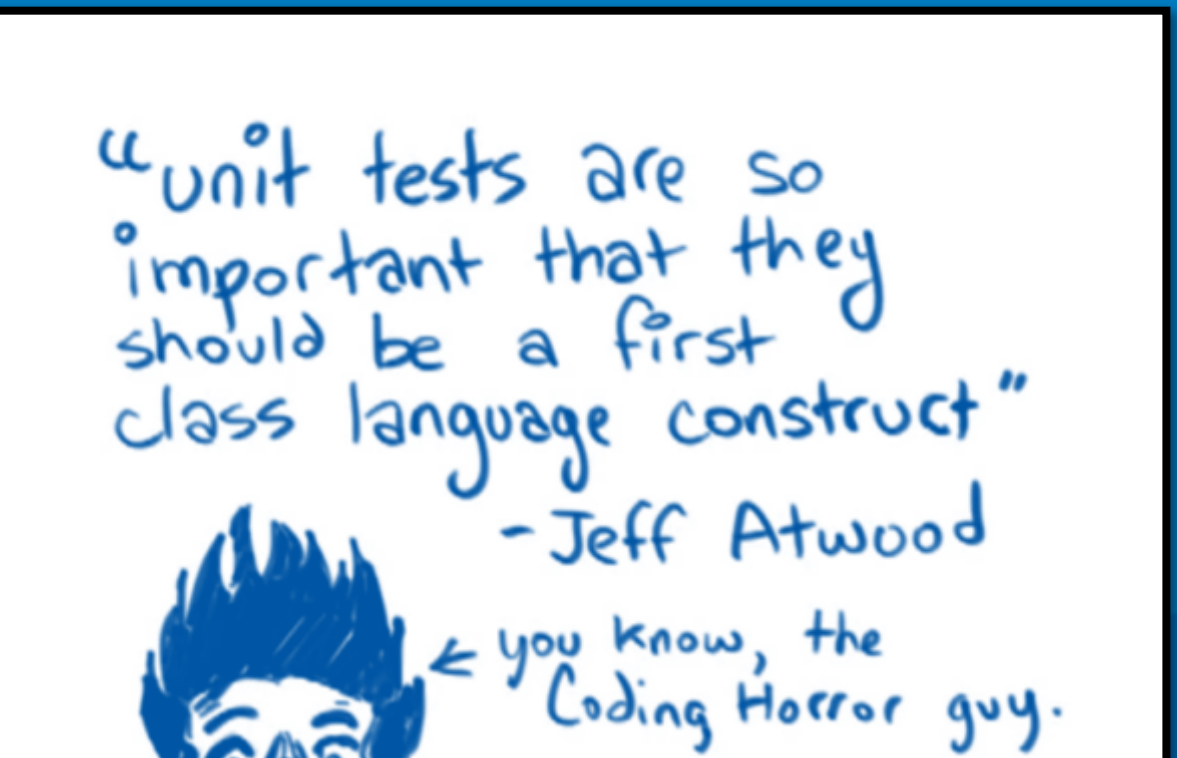
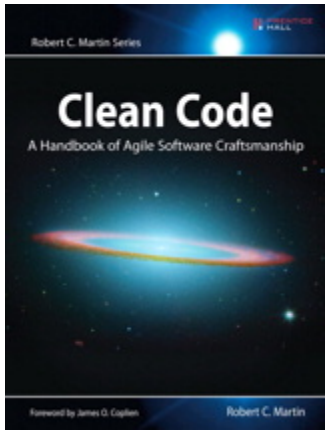
Unit testing



Introduction to testing

- You are programmer



- How to deliver working and clean code?

Keyword: Software Craftsmanship

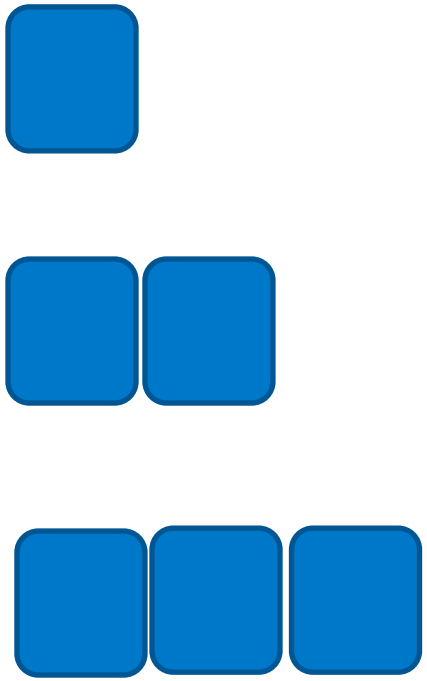
Introduction to testing

- Solution: Test your code
- Ok, but when?
- Test first

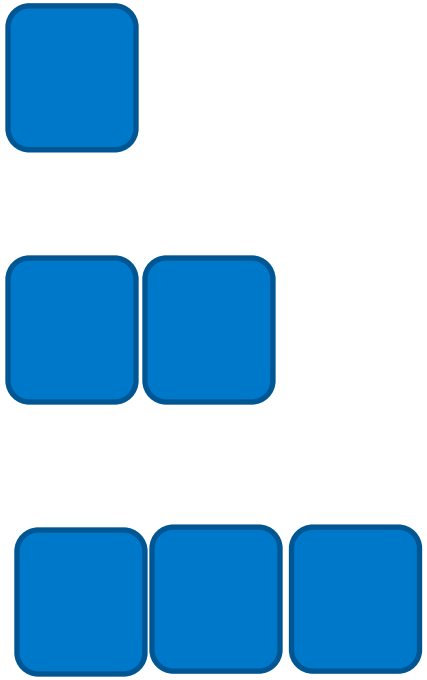
Test Driven Development

Introduction to testing

- Tests



Production code



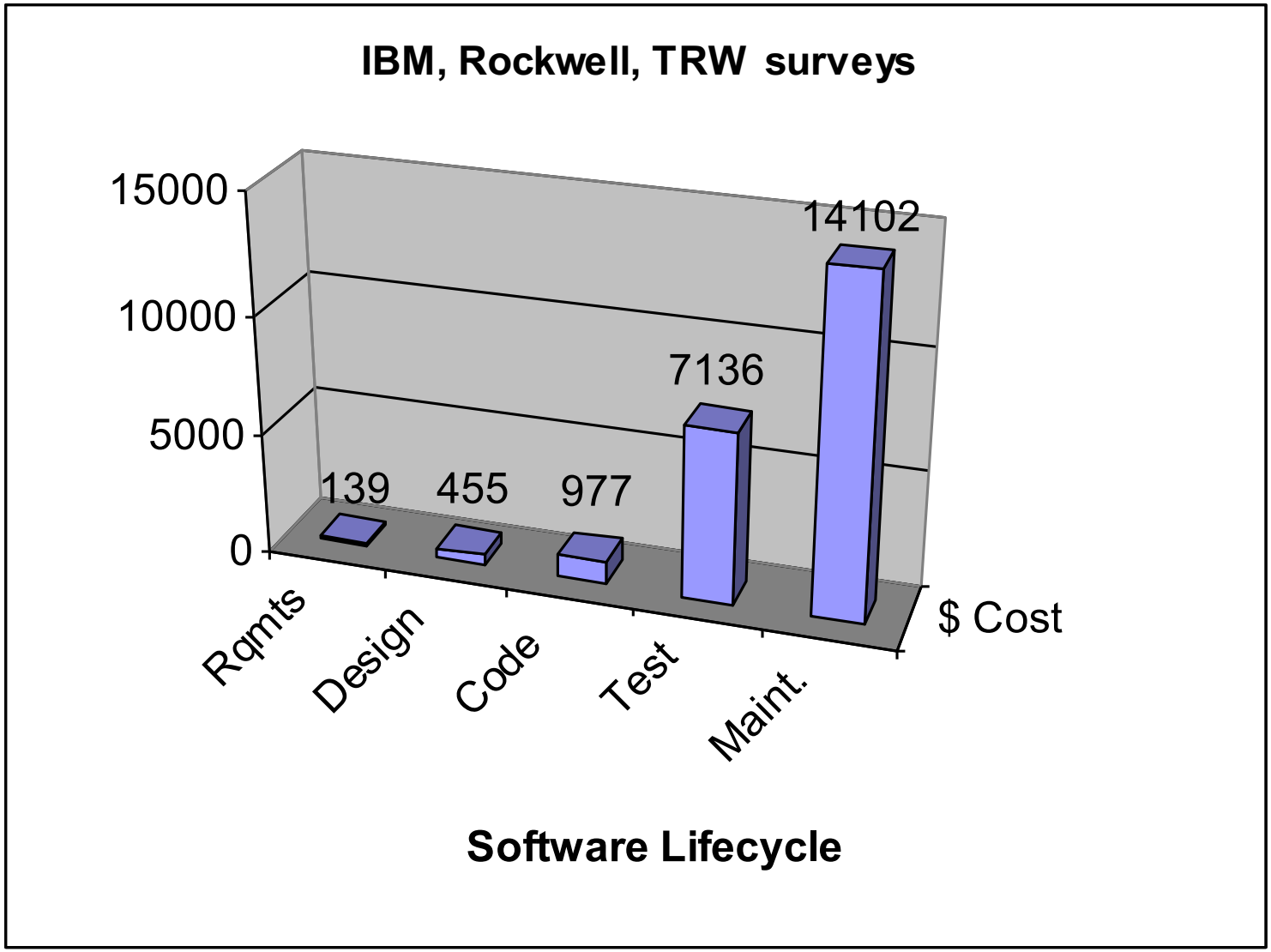
What is Test Driven Development?

- TDD is a method of designing software
- TDD is more than a way of testing
  - TDD is a way of working
- Began to receive publicity as part of Extreme Programming (XP)
- Now seen as the best way to develop quality software
- Unit Tests play a central role during development

Why TDD ?

- TDD helps in developing quality software
  - Writing tests help in creating a clean design
- Makes sure that code behaves according to specification
- Makes sure that new code doesn't break existing code
  - Regression testing
- Test code early
  - Bugs become more expensive to fix after coding has finished

Cost of bugs


IBM, Rockwell, TRW surveys
15000
10000
5000
0
139
455
977
7136
14102
Rqmts
Design
Code
Test
Maint.
$ Cost
Software Lifecycle

TDD and productivity

- TDD improves overall developer's productivity
  - Less defects
  - Better code
- It will add some coding time however
  - This extra time becomes almost nothing however when becoming more experienced with TDD

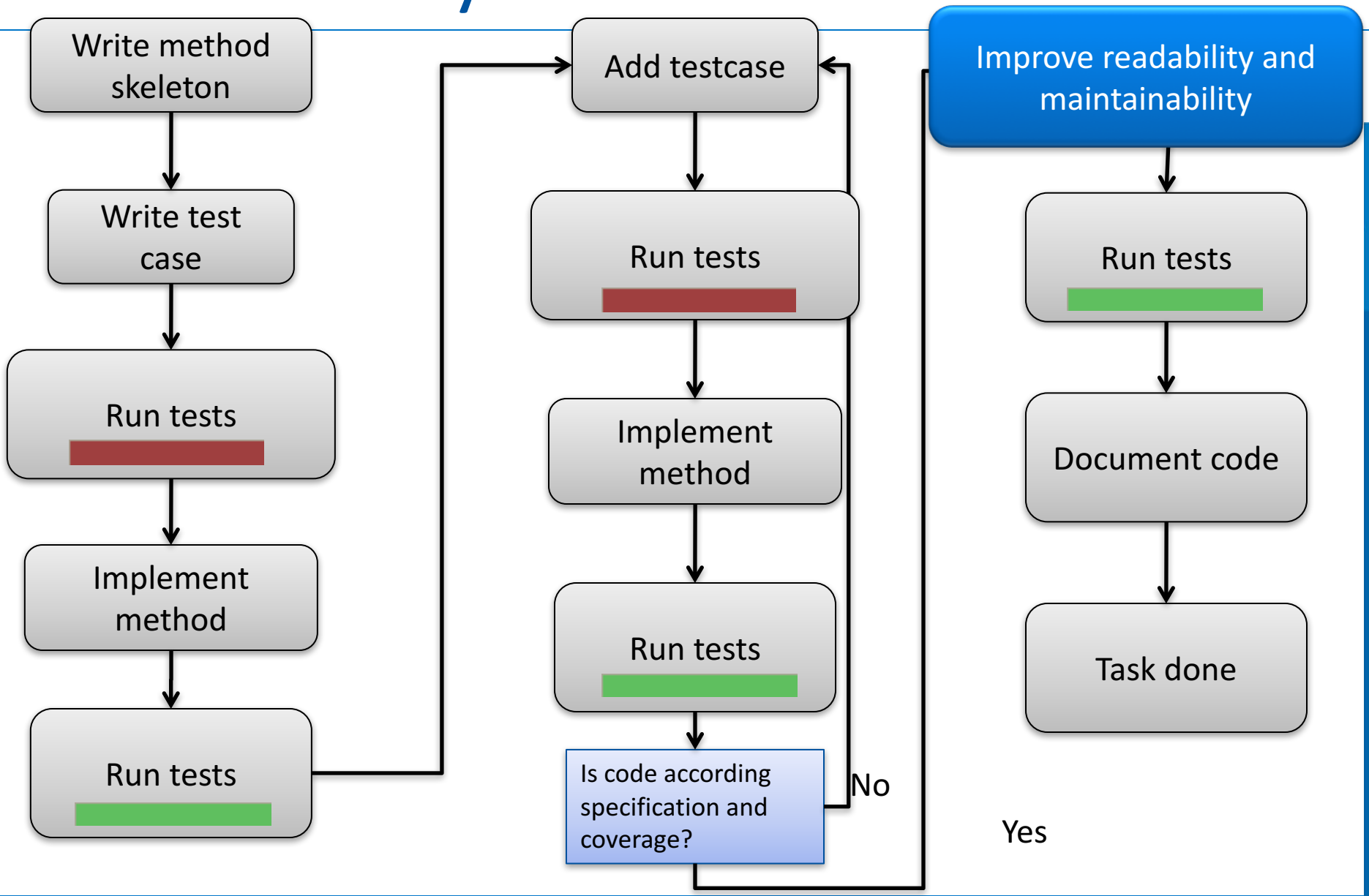
Arguments not to unit test

- Writing tests takes more time
- Other tests will find the bugs eventually
- When I run the tests, they will break
- They take too long to run
- Our architecture is too complex
- We have testers
- We also have to test the Unit Tests
- My code is great, I don't need Unit Tests!

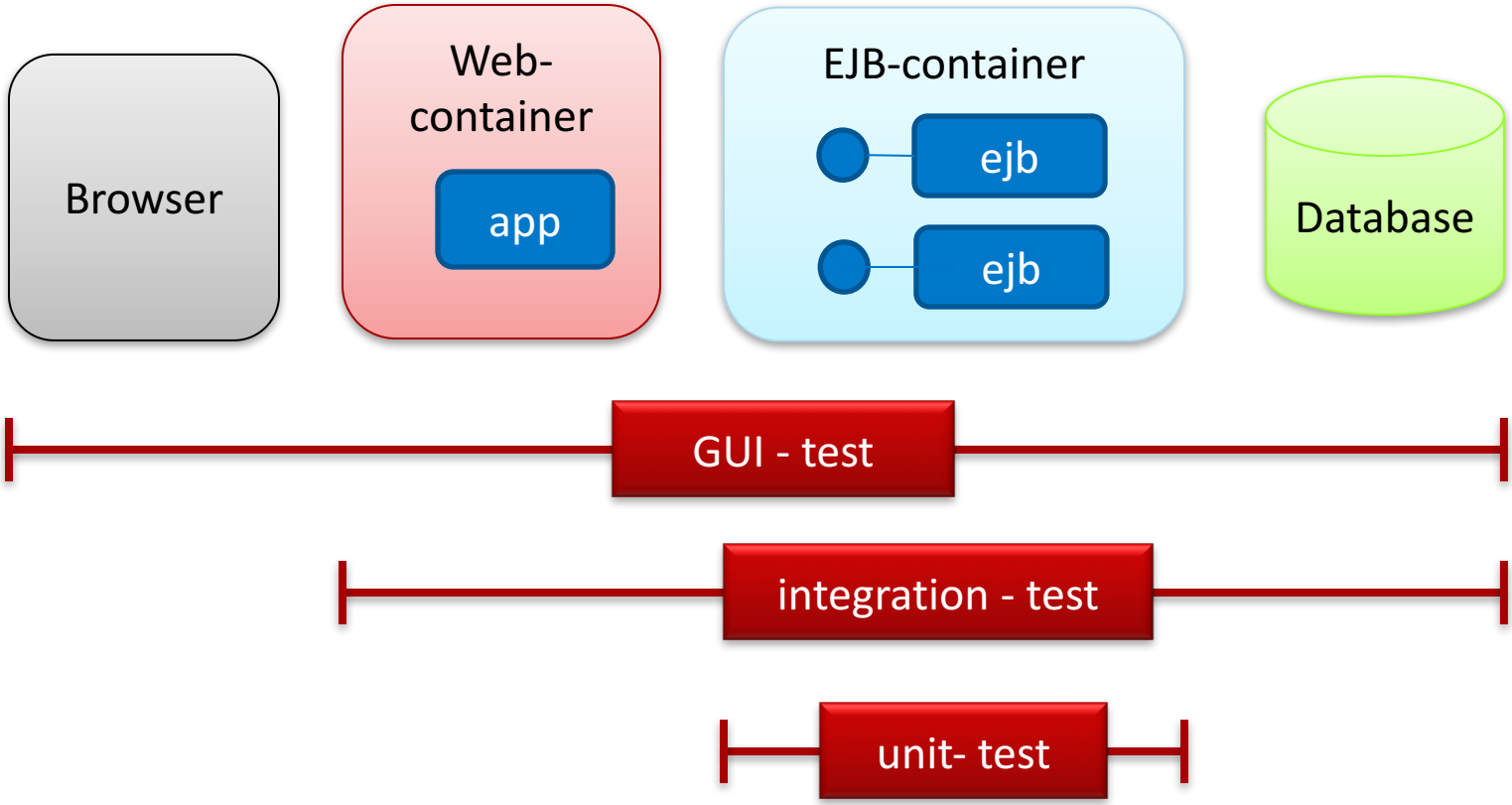
TDD principles

- First write a test and then write code
  - Writing tests afterwards doesn’t work
- Implement code in very small iterations
- A test should always fail first
- Tests are written by developers
- Writing tests improves productivity
  - Testable code enforces a clean design
  - Find bugs early

TDD workcycle



Sort of tests



Unit Testing

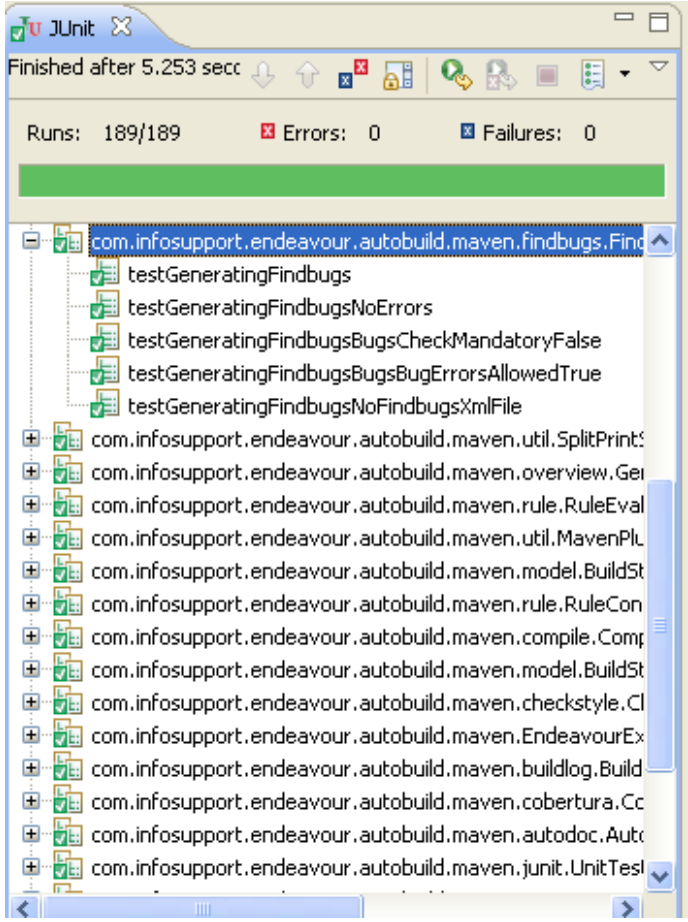
- Be able to validate the correct behavior for each individual unit in the system
- Test a unit in isolation
- What is a Unit?
  - the smallest compilable unit, practically a class
- A unit test runs automatically
- A unit test runs fast
- Unit testing does not make other levels of testing unnecessary!

HOW TO DO THIS?

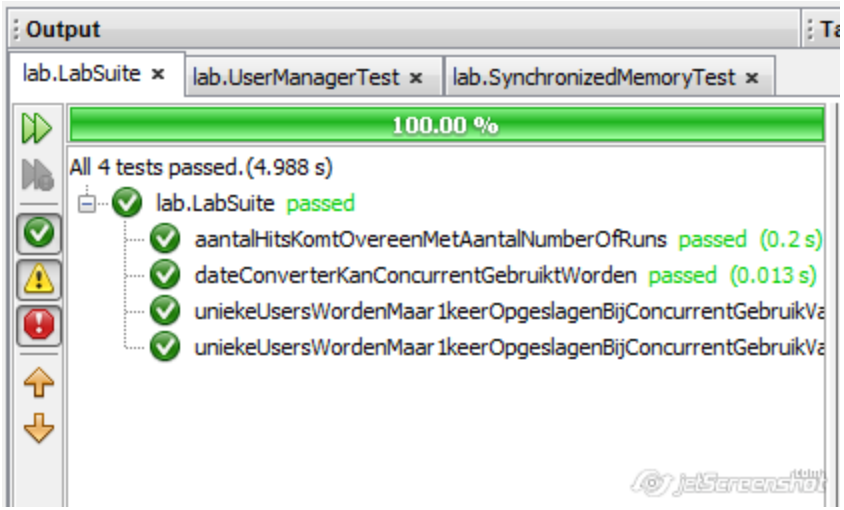
Using JUnit

- JUnit is used to run unit tests
  - JUnit is called a test ‘driver’
- Available in all popular IDEs
- JUnit 4 uses annotations to define tests
- Junit 3 uses a TestCase baseclass and naming conventions
  - Every testmethod name must start with ‘test’

JUnit in popular IDE's



JUnit in Eclipse



JUnit in NetBeans

Choosing what tests to write

- Test a unit's public interface
  - All public methods
    - including static methods
  - Private methods are tested implicitly
  - Protected methods might be useful to test

Choosing what tests to write

- Test invalid input
  - But don’t forget to also test valid input
- Test every method
  - Unless it’s impossible to break

  (e.g. getters / setters)

A simple JUnit test

```
import static org.junit.Assert.assertEquals;

@Test
public void calcAddsTwoPositiveNumbers() {
  Calculator calc = new Calculator();
  int addResult = calc.add(4, 7);

  int expectedResult = 11;
  assertEquals(expectedResult, addResult);
}
```

Annotation is handled by JUnit

In old JUnit versions are assertEquals, assertFalse popular. For now, use matchers

Corresponding implementation

```
/**
 * @param a
 * @param b
 * @return the addition of number a and b
 */
public int plus(int a, int b) {
    return a + b;
}
```

Scenario's

- The previous test tests only one scenario
- What about negative numbers, max numbers?

```
@Test
public void calcRefuseResultAboveMaxNumber () {
  Calculator calc = new Calculator();
  int addResult = calc.add(Integer.MAX_VALUE, 1);

  long expectedResult = Integer.MAX_VALUE + 1L;
  assertEquals(expectedResult, addResult);
}
```

Failure Trace

```
java.lang.AssertionError: expected:<2147483648> but was:<-2147483648>
  at webshop.dao.CalculatorTest.calcRefuseResultAboveMaxInteger(CalculatorTest.java:24)
```

Expecting an Exception

- In this case, we expect an exception

```
@Test(expected=IllegalArgumentException.class)
public void calcRefuseResultAboveMaxNumber () {
  Calculator calc = new Calculator();
  calc.add(Integer.MAX_VALUE, 1);
}
```

Other @nnotations

```
@RunWith(Suite.class)

public class OrderServiceTest {
    private OrderService orderService;
    private Order order;

    @BeforeClass
    public static void setUp() throws Exception { }

    @Before
    public void orderIsSavedInDatabase() {  }

    @Test  (expected=InvalidOrder.class)
    public void orderIsRemovedFromDatabaseOnOrderRemove() { }

    @AfterClass
    public static void tearDownClass() throws Exception { }

    @After
    public void tearDown() {  }

}
```

Writing Junit tests

```
public class OrderServiceTest {

    @Test
    public void orderIsRemovedFromDatabaseOnOrderRemove() {
        StringTemplate template = new StringTemplate("{a}{b}");
        HashMap<String,Object> macros = new
                                HashMap<String,Object>();
        macros.put("a", "A"); macros.put("b", "B");

        String expanded = template.expand(macros);

        assertThat(expanded, equalTo("AB"));
    }

}
```

Setup

Execute

assert

Testing void methods

- Test side effects of the method
  - There are always side effects (or the method doesn't do anything)
- For example, test if internal state has been updated correctly
- If internal state does not change
  - Use Mock Objects to verify outgoing method calls

Testing return values

- Most straight forward way of testing
- Old assert methods
  - assertTrue(actual)
  - assertFalse(actual)
  - assertEquals(expected, actual)
  - assertAssertEquals(expected, actual)
  - assertNotNull(actual)
  - assertNull(actual)

Testing return values

- More flexible is to use matcher methods

```
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;

assertThat(plus, is(expectedResult));
```

- Why use of matchers?
  - Readable
  - Flexible

Matchers

- Matchers not (fully) implemented by JUnit
- JUnit uses the library Hamcrest

- Hamcrest:

"Provides a library of matcher objects (also known as constraints or predicates) allowing 'match' rules to be defined declaratively, to be used in other frameworks. Typical scenarios include testing frameworks, mocking libraries and UI validation rules."

Hamcrest common matchers

- Core
  - anything(…)
  - describedAs(…)
  - is(…)
- Logical
  - allOf(…)
  - anyOf(…)
  - not(…)
- Object
  - equalTo(…)
  - hasToString(…)
  - instanceOf(…)
  - notNullValue(…), nullValue(…)
  - sameInstance(…)
- Beans
  - hasProperty(…)
- Collections
  - array(…)
  - hasEntry(…), hasKey(…), hasValue(…)
  - hasItems(…), hasItem(…)
  - hasItemInArray(…)
- Number
  - closeTo(…)
  - greaterThan(…), greaterThanOrEqualTo(…), lessThan(…), lessThanOrEqualTo(…)
- Text
  - equalToIgnoringCase(…)
  - equalToIgnoringWhiteSpace
  - containsString(…), endsWith(…), startsWith(…)

Hamcrest examples

```
assertThat("choco chips",
      theBiscuit.getChocoChipCount(), equalTo(10));
assertThat("hazelnuts",
      theBiscuit.getHazelnutCount(), equalTo(3));
```

```
assertThat(x, is(3));
assertThat(x, is(not(4)));
assertThat(responseString,
  either(containsString("color"))
  .or(containsString("colour")));
assertThat(myList, hasItem("3"));
```

Lab 1:

JUnit

infoSupport
Solid Innovator

Matcher internals

- To clarify how a matcher works, a custom matcher example will be shown

- In this example we write a custom matcher to test for the property: NotANumber
- Example usage of this matcher is:

```
public void testSquareRootOfMinusOneIsNotANumber() {
   assertThat(Math.sqrt(-1), is(notANumber()));
}
```

Custom matcher

```
import org.hamcrest.*;

public class IsNotANumber extends TypeSafeMatcher<Double> {

  @Override public boolean matchesSafely(Double number) {
    return number.isNaN();
  }

  public void describeTo(Description description) {
    description.appendText("not a number");
  }

  @Factory public static Matcher<Double> notANumber() {
    return new IsNotANumber();
  }
}
```

number represents the result of the first expression in assertThat

Automatically called when Evaluating a matcher

Called when the comparison fails

Create instance of the matcher

Static method to call from test

Using the custom matcher

- Following statement:

```
assertThat(1.0, is(notANumber()));
```

- fails with the message:

```
java.lang.AssertionError:
Expected: is not a number
     got : <1.0>
```

Why use matchers?

- Readability of code
- Reuse of comparison code
- Loosely coupled with equal- and hash method

Loosely coupling with equals/hash

- Very often a businessobject is compared by ID

- But in unit tests you want to compare not only by ID, but by all fields

```
Person expectedPerson = new Person("Jan", "Dorpie");
Person givenPerson = dao.find("Jan", "Dorpie");
assertEquals(expectedPerson, givenPerson);
```

In this case hashcode and equals is used

Create a custom person matcher

```
public class IsPersonMatcher extends TypeSafeMatcher<Person> {
  private Person expectedPerson;
  public IsPersonMatcher(Person expectedPerson) {
    this.expectedPerson = expectedPerson;
  }

  @Override protected boolean matchesSafely(Person current) {
    boolean result = current.getName().equals(expectedPerson.getName())
                &&
              current.getAddress().equals(expectedPerson.getAddress());
    return result;
  }

  @Factory public static IsPersonMatcher person(Person expectedPerson) {
    return new IsPersonMatcher(expectedPerson);
  }

  @Override  public void describeTo(Description description) {
    description.appendText("A person  with name: ")
    .appendValue(expectedPerson.getName())
    .appendText(" and with address: ")
    .appendValue(expectedPerson.getAddress());
  }
```

Using the matcher

- Add a static import
- Use the matcher in an assertThat statement

```
import static nl.infosupport.IsPersonMatcher.person;
```

```
Person expectedPerson= new Person("Jan", "Dorpie");
Person givenPerson = dao.find("Jan", "Dorpie");
assertThat(givenPerson, is(person(expectedPerson)));
```

Questions