Unit Testing

Slide Series 5

Unit Testing

- Have a few running UCs
    - Partial implementations of involved classes
- No we go for quality
- By testing smaller parts (units) we try to increase the quality (i.e. remove bugs)
- If units have high quality, chances for overall quality increases (but not guaranteed)
- If we’re unsure of unit quality … how can we ever hope to get a high quality working application?

Unit

So what’s a unit?

Not well defined! In this course:

- A single non-dependent class (i.e. not depending on any other of <u>our</u> classes)
- … or small aggregate of dependent (model) classes
- … or a subsystem

NOTE: We test implementations (classes), don’t use interfaces in tests

Organizing Test Code

Keep test code separate from application code

- Use (or create) a source folder "test"
- Use exactly same package structure as application, will make it possible to test package private classes

JUnit

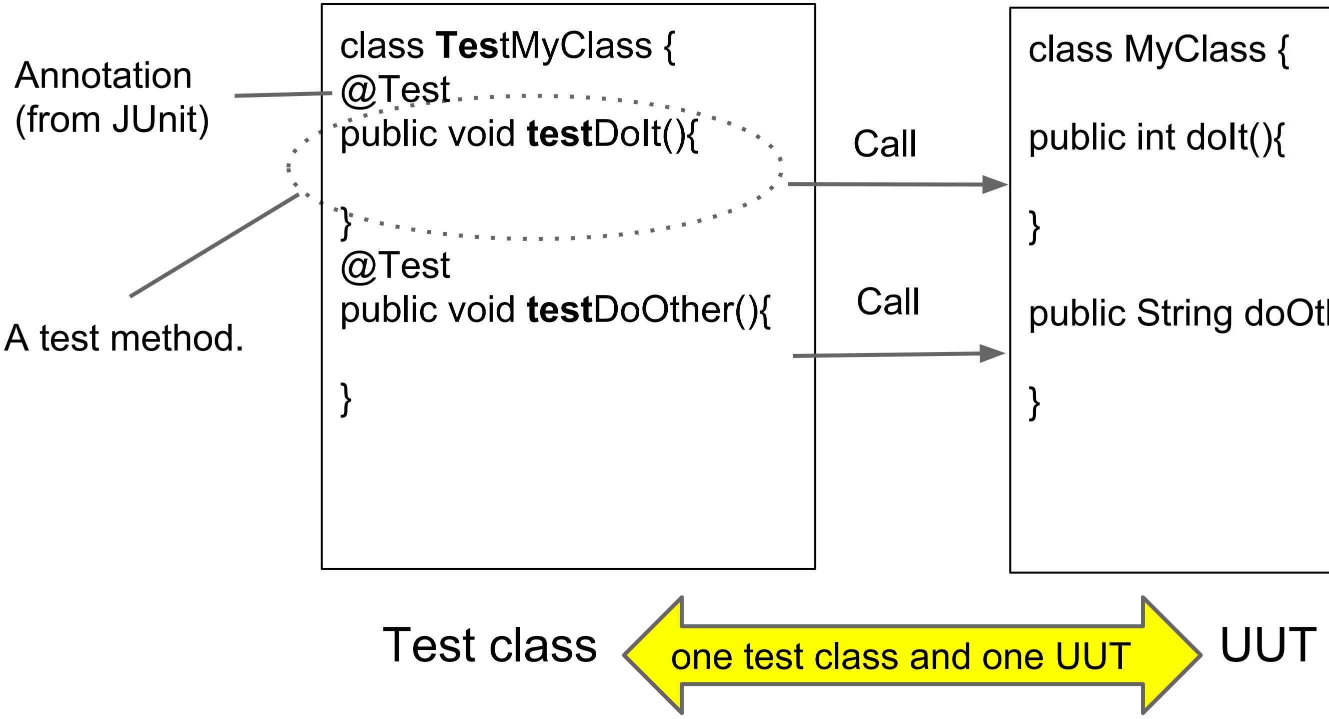
JUnit is a simple framework to write repeatable unit tests

- Bundled with NetBeans
- .. think also Eclipse and IntelliJ (?)
- We use Junit 4.x or above (don’t use Junit 3.x)
- Framework = Semi-implemented application. You must fill in some parts, in some specific way, before executing (in JUnit we fill in with test classes)

Using JUnit

- We have a unit under test, the UUT
- Write a "test"-class (a Java class) for the UUT
- Let JUnit run the test class (tests calling public non-trivial methods on UUT)
- JUnit will report outcome of tests (success or not)
  - Note: No human interaction, no System.out.println()
  - Aka automated testing!

Test a Single Non-Dependent Class



Annotation (from JUnit)

A test method.

```
class TestMyClass {
@Test
public void testDoIt(){

}
@Test
public void testDoOther(){

}
```

Call

Call

```
class MyClass {

public int doIt(){

}

public String doOther(){

}
```

Test class

one test class and one UUT

UUT

Test Methods

- public void *someLongAndDescritiveName*() (no params, no return value, descriptive name)
- Must be annotated with @Test
- Use JUnit “assert”-methods to pass or fail

```
@Test
public void testDoIt(){
        MyClass m = new MyClass();
        int i = m.doIt();
        // Must know correct value
        assertTrue( i == 4 );
}
```

assertTrue method from JUnit (import static)

Test expression true -> test will pass else fail. Junit will report

Test expression

Test Expressions

How do I know which test expressions to use?

- No rules, creativity needed!
- Some common: <, >, ==, equals(), size(), contains(), … also loops ...
- Use inverses (example: add/remove)
- Always check “corner cases” (examples: null, empty list/String, first/last element, size()+1, i < 0, …)

More on Test Methods

- Should not depend on other test methods
- Should not be dependent on execution order
- If calling void methods, possibly must add get()-method to UUT (to inspect state)
- If need for testing private methods, problems … (fix for now, change visibility … not good …)
- If exception expected use: @Test (expected=IllegalStateException.class)

Test-Isolation

To be able to test units there must be units (i. e. separable parts)

- Single non-dependent class, ... no problem
- Subsystems shouldn’t be too problematic
- … but most classes are dependent!!

Isolation

To isolate a unit, i.e. minimize dependencies to make testing possible

- Avoid inline use of static classes, Singletons or any global dependency
- No inline use of **new** (use factories)
- Dependencies passed in via constructor (makes it possible to mock dependencies)
- Advanced: Use a dependency injection framework

Fixtures

Have a unit with a few dependencies (say a few model classes). To create a fixture (i.e. connect the classes in the unit) annotate some method with:

@BeforeClass, method run once before all tests

@Before, method run before each test

Create all objects needed in the methods

(also @AfterClass and @After, probably not very useful to us)

Start Testing of MP

Testing of Board

- Class nearly independent (need trivial class Space)
- Some methods trivial, no tests
- A few should be tested
    - getSpace(), hasPassedGo()

Testing of Player

- Class rather dependant but can start testing specific methods

See Test Packages MP v 0.2

Testing a Service

A service delivers functionality to the application (model)

- A service should be independent, a very well separated part of application (more in Iteration 2 slides)
- Normally there should be return values
- Use return values in test expressions

Test Suite

Possible to collect all testes into a test suite

- Create a class for the suite (i.e. TestSuite)
- Add some class annotations
- See MP v 0.3

Junit and Debugging

Very efficient to debug tests (instead of full application)!

- Highly recommended!

Code Coverage

How do we know we have tested all code (example: if statement executed for both true and false)?

- After running the tests use code coverage
- Need plugin JaCoCo or similar (in project pom.xml), see MP v 0.3
- Should design test to cover as much as possible (of non-trivial code)
- High coverage is a quality factor

Test Driven Development

TDD is a way of working using unit tests

1. For some unit: Write unit tests (tests will fail because there is no code yet*)
2. For the unit: Implement code until all tests work
3. Goto 1 until finished

Possibly a bit advanced for us. But at least try to write test in parallel

*) Add initial false test expression

Other Benefits of Tests

Test will make refactoring comfortable

- Refactor and run tests, everything should pass!

Tests will act as documentation

- Kind of specification
- Much more probable that test are in sync with code (vs RAD, SDD)

Summary

Our primary means to increase the quality of our application is unit testing

We must design the code to be testable (independent units)

JUnit will run automated test suites

Important to have high code coverage