Practical Unit Testing

“Good programmers write code, great programmers write tests.”

Peter Kofler, ‘Code Cop’

JSUG, June 2009

Who am I?

- Ph.D. in Applied Mathematics
- Java developer since 1999
- fanatic about code quality since 2004
- appointed ‘Code Cop’ in 2006
- Senior Software Engineer at s-IT Solutions (Spardat), Erste Group

Agenda

- JUnit Basics
  - Test Methods, Assertions, Fixtures
- Advanced Topics
  - Privates, Mocks, Timings, Singletons, J2EE
- Tuning
- Code Coverage
- JUnit Extensions
  - Tools, Scripting, JUnit and the Build

A Little Survey...

- Who knows xUnit?
- Who knows JUnit 4?
- Who ever wrote a unit test?
- Who writes tests and tries to write them first?
- Who checks the coverage?
- Who ever produced a bug?

We Make Mistakes

- at least I do... ☺
- number of bugs proportional loc
  - 2 bugs per 1.000 loc (7 or even more...)
  - 1 bug per 10.000 loc in critical software
- be paranoid when you write software
  - Assume you have lots of bugs.
  - Try to find these bugs aggressively.

I find your lack of tests disturbing.

Wait - We Have Tests

- almost every project has some “tests”
- almost all of them are useless ☹
  - experiments how to use some library
  - main methods, waiting for user input, ...
  - tests that initialise the whole application and check nothing
  - tests that fail since long, etc.

No You Don’t!

JUnit

- a unit testing framework
- active, dynamic black-box tests
  - some call it white-box tests (tbd)
- works best with a number of small tests
- You should know it (no excuses!)
  - You should use it (no excuses!)
  - I will not explain it here → www.junit.org

“Keep the bar green to keep the code clean”

Test Methods

- unit test tests the methods of a single class
- test case tests the response of a single method to a particular set of inputs
  - multiple test cases for a single method
  - **`public void`** `testMethod()` or `@Test`
  - test methods should be short, simple
  - tests without test methods are pointless
    → **Findbugs** and **PMD**

Assertions

- Don’t do any output from your unit tests!
- check expectations programmatically
  - `assertEquals,assertNull,assertTrue,…`
  - test method without assert is pointless (→PMD)
  - one test method - one assertion (tbd)
    - some work around PMD with `assertTrue(`**`true`**`)`
    - → PMD: `UnnecessaryBooleanAssertion`
- test runner reports failure on each test

Proper Assertions

- add messages to asserts (tbd) (→ PMD)
- `assertTrue(a.equals(b))` no message, better use `assertEquals(a,b)` (→ PMD)
- assert in `Thread.run()` not noticed (→ Findbugs: `IJU_...`)
- assert float in ranges of precision:

```
assertEquals(expected, actual,
                  5*Math.ulp(expected))
```

Assertions (JUnit 4)

- `assertArrayEquals(.)` for atom arrays and `Object`
- but `assertEquals(int,int)` removed
  - not needed any more (auto boxing)
  - problems with mixed params, e.g. `(int,byte)`
  - JUnit 3: promoted to `(int,int)`, succeeds
  - JUnit 4: boxed to `(Integer,Byte)`, fails

Asserting Exceptions

- JUnit 3 `try-catch` code:

```
try {
  // code that should cause an exception
  fail("no exception occurred");
} catch (SomeException success) {
  // check exception type/parameters
```

- JUnit 4: `@Test(expected)` annotation:

```
@Test(expected=SomeException.class)
public void testThrows() {
  // code that should cause an exception
```

Fixtures (JUnit 3)

- sets up data needed to run tests
- JUnit 3: `setUp()`, `tearDown()`
  - don't forget to call `super.setUp()` first
  - don't forget to call `super.tearDown()` last
  - don't forget it (!)
  - Findbugs: `IJU_SETUP_NO_SUPER`, `IJU_...`
- for fixture in JUnit 3.x that runs only once, use the `TestSetup` decorator

JUnit 3 `TestSetup` Decorator

```java
public class TheTest extends TestCase {
 // test methods ...

 public static Test suite() {
  return new TestSetup(new TestSuite(TheTest.class)) {
   protected void setUp() throws Exception {
    super.setUp();
    // set-up code called only once
   }
   protected void tearDown() throws Exception {
    // tear-down code called only once
    super.tearDown();
   }
  };
```

Fixtures

- JUnit 4: `@Before`, `@After`
  - run methods of super classes
  - only once: `@BeforeClass`, `@AfterClass`
- test data in database is problematic
  - test has to insert its own preconditions
  - large data sets → **DbUnit**
- Remember: Test data is more likely to be wrong than the tested code!

Test Code Organisation

- test code loc ~ functional code loc
- same quality as production code
  - always built with test code
  - execute tests as soon/often as possible
- parallel package hierarchy
  - no `*.test` sub-packages! (tbd)
  - folder `test` (simple), `src/test/java` (Maven)
  - package-access!

Test Class Organisation

- create your own base test case(s)
  - named `*TestCase` or `*TC` (not `*Test`)
  - common methods, initialisation code
  - custom asserts, named `assert*` (PMD)

- name test classes `<tested class>Test`

QNH
29.92
CSTR
WPT
VOR.D
NDB
ARPT
SPD
HDG
LAT
060
HDG
V/S
ALT
LVL/CH
V/S
02500
AP1
AP2
A/THR
LOC
EXPD
APPR
ROSE
NAV
ARC
PLAN
ADF
VOR
OFF
FD
LS
CHRONO
SIDE STICK PRIORITY
N835UA
AG-HK
4
0
3
5
ILS APP
TERR ON ND
LDG GEAR
AUTO BRK
LO
MED
MAX
A/SKID &
N/W STRG
EPR
EGT
FOB: 33480 LBS
FLAP
NO SMOKING
PARK BRK
TCAS STBY
APU BLEED
23:27:57
UP
DOWN
DOOR/OXY
OXY 1670 PSI
SLIDE
TAT +20 °C
SAT +20 °C
23 H 27
GW
LBS
SWITCHING
ACARS-MAIN INDEX
<PREFLIGHT
ATS>
<ENROUTE
UA COMM>
<POSTFLIGHT
REPORTS>
<MESSAGE LOG
REQUESTS>
F-PLN DISCONTINUITY

Agenda

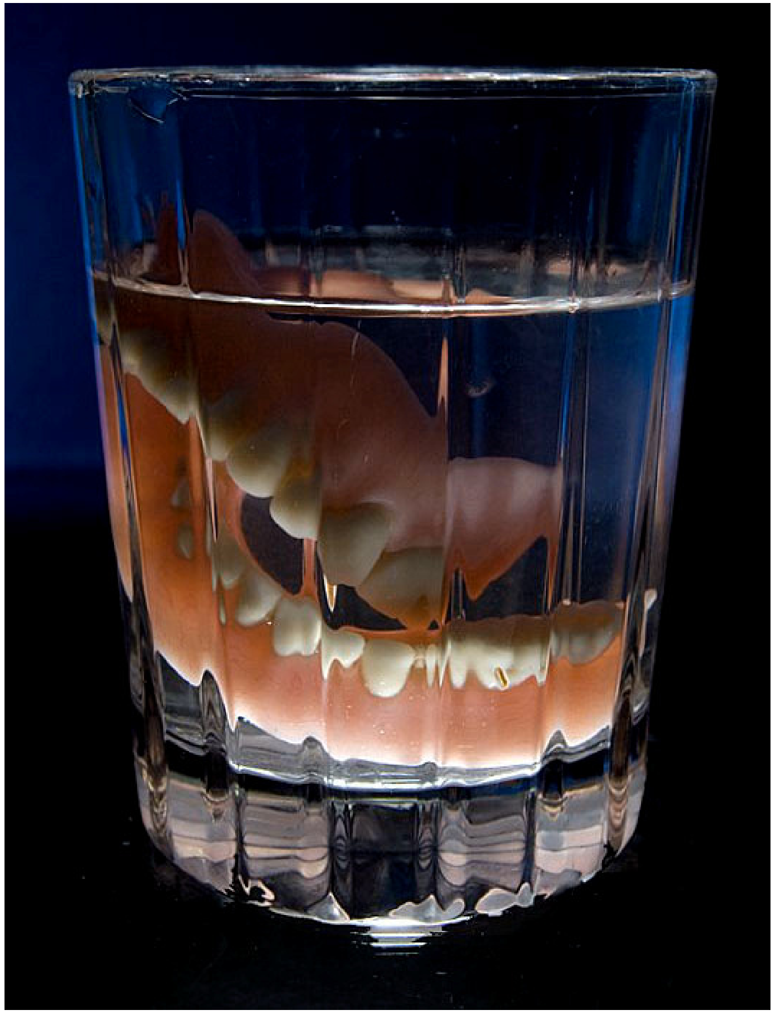
- JUnit Basics ✓
  - Test Methods, Assertions, Fixtures
- **Advanced Topics**
  - Privates, Mocks, Timings, Singletons, J2EE
- Tuning
- Code Coverage
- JUnit Extensions
  - Tools, Scripting, JUnit and the Build

Testing Private Data

- “Wishing for White Box Testing (i.e. check a private field) is not a testing problem, it is a design problem.”
  - If you want to check internals - improve design.

- if you have to:
  - Reflection: `member.setAccessible(true)`

Mocks



When to Use Mocks

- To have a “real” unit test (cut dependencies)
- “It is much simpler to simulate behaviour than it is to recreate that behaviour.”
- use a mock when the real object is
  - non-deterministic (e.g. current time)
  - problematic during execution (e.g. user input)
  - difficult to trigger (e.g. network error)
  - not existing yet (team collaboration)

How to Mock an Object

- by hand
  - implement its interface (Eclipse Ctrl-1)
  - subclass it (beware complex constructors)
- `java.lang.reflect.Proxy`
  - since 1.3
  - only for interfaces
  - nasty for more than 1 method

Dynamic Mock Frameworks

- **EasyMock**, **jMock**, ... (in fact since 1.5)
- mock interfaces (Proxy)
- mock non final classes (cglib)

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

SomeInt mock = createMock(SomeInt.class);
expect(mock.someMethod("param")).andReturn(42);
replay(mock);
// run the test which calls someMethod once
verify(mock);
```

Mocks in Spring

- IoC make it easy, just set the mock
- combination of context/mocks
  - needs mocks inside Spring:

```
<bean id="someMock" class="org.easymock.EasyMock"
      factory-method="createMock">
  <constructor-arg index="0" value="...SomeBean" />
</bean>
```

  - see http://satukubik.com/2007/12/21/spring-tips-initializing-bean-using-easymock-for-unit-test/

Enabling Mocking

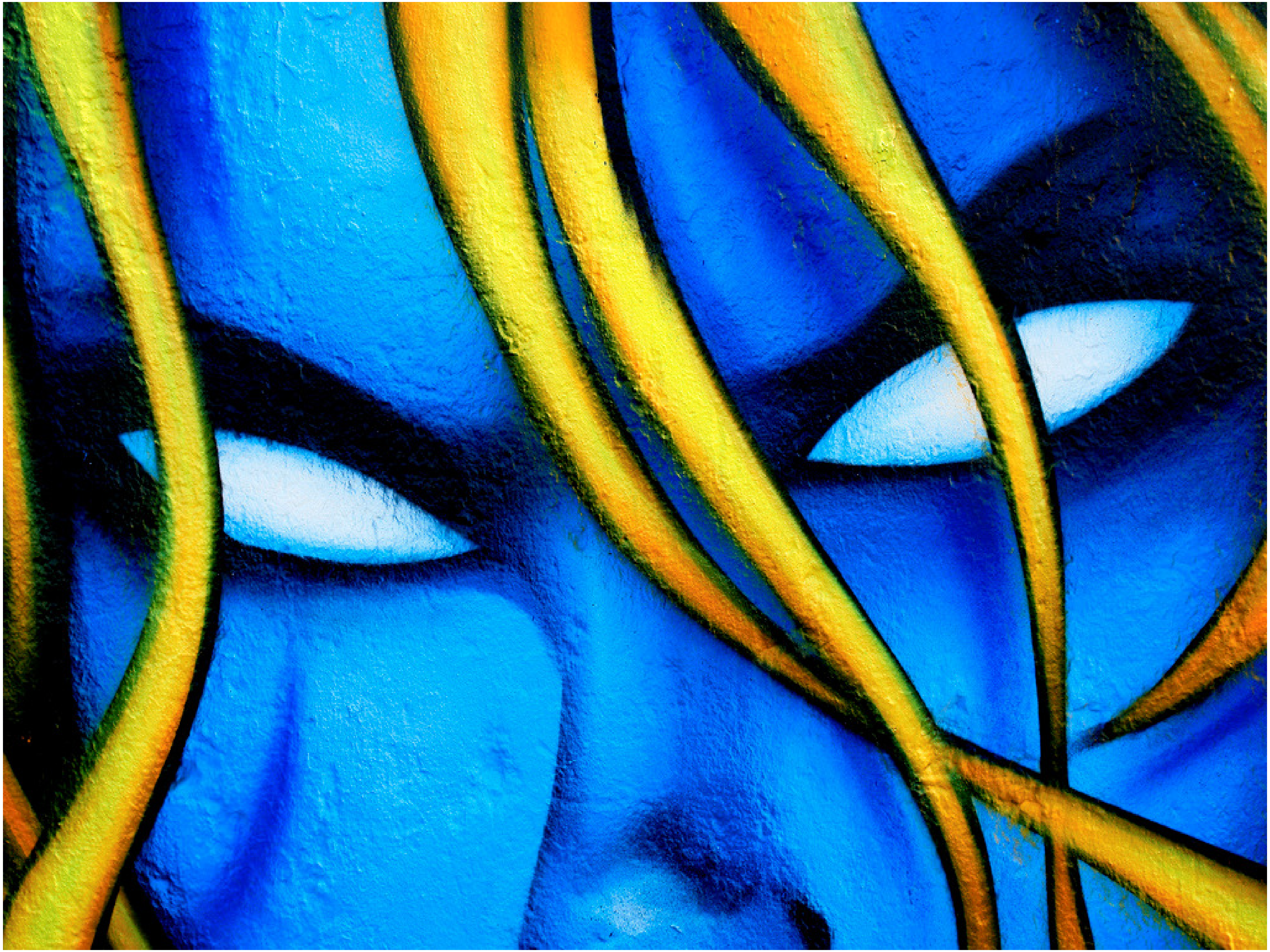
- Program to interfaces, not implementations.
  - interfaces are easier to mock
- Law of Demeter
  - style guideline
  - “Only talk to your immediate friends.”
  - calling methods on objects you get from other collaborators is trouble - your mocks must expose internal state through these methods

Limits of Mocking

- behave accordingly to **your** expectations
  - Do you know the mocked class good enough?
- complex mocks are error prone
  - e.g. state machines
  - refactor using Law of Demeter
- replace the right classes
  - not the tested ones!
  - focus on what goes inside than what comes out

Testing Timings

- timings (e.g. timeouts) are difficult
  - timing/scheduling is not guaranteed
  - short timings almost always fail
  - long timings slow down the execution
- You will never get it right!
  - esp. not for Windows **and** Unix at same time
- → mock the timer



Singletons are evil!

- most overused design pattern
  - typical: **`public static`** `Instance getInstance()`
  - static methods (“mingletons”), e.g. `System.currentTimeMillis()`
  - static fields (“fingletons”)
  - `ThreadLocal`
- most likely you have too many of them
- see http://c2.com/cgi/wiki?SingletonsAreEvil

Testing Singletons

- problems for testing
  - evil
  - unknown dependencies
  - initialisation often expensive (fixture)
  - side effects in same class loader
  - concurrency issues when testing in parallel
  - can’t mock

Testing Singletons “Brute Force”

- straight forward
  - (fake) initialise singleton in fixture (`setUp`)
  - use Ant’s `forkmode=“perTest”`
  - slow^2
- if singletons can be reset
  - cleanup singleton in `shutDown`
  - make sure double initialisation fails
  - still slow, still no mock

Testing Singletons “AOP”

- context-sensitive modification with **AspectJ**
- returning a mock instead of proceeding (around advice)
- per-test-case basis (using various pointcuts)
  - `execution(public void SomeTest.test*())`
  - `cflow(inTest()) && //other conditions`
- see http://www.ibm.com/developerworks/java/library/j-aspectj2/
- mock ✓, but .aj files get nasty

Refactor Singletons

- for new code - avoid singletons
- refactor
  - pass singleton instance from outside to certain methods as argument, mock it
  - create a global registry for all singletons, which is the only singleton then, register mocks there
  - make whole singleton a Spring bean with singleton scope, mock it

Testing J2EE - JNDI

- use mocks like **Simple-JNDI** or **MockEJB**

```
protected void setUp() throws Exception {
  super.setUp();
  MockContextFactory.setAsInitial();
  new InitialContext().bind("name", stuff);
}
protected void tearDown() throws Exception {
  MockContextFactory.revertSetAsInitial();
  super.tearDown();
}
```

Testing J2EE - JMS

- use mock implementation like MockEJB
- use in memory JMS like Apache**ActiveMQ**

```
<bean id="factory" class="..ActiveMQConnectionFactory">
  <property name="brokerURL” value="vm://broker?
                  broker.persistent=false&amp;
                  broker.useJmx=false" />
</bean>
<bean id="queue" class="...command.ActiveMQQueue">
  <constructor-arg value="SomeQueue" />
</bean>
```

Testing J2EE - Servlet

- call them (**HttpClient**, **HttpUnit**)
  - needs deployment and running server ☹
  - integration test
  - beware GUI changes
- run them in container (**Cactus**)
- embedded server (**Jetty** `ServletTester`)
- mock container (**ServletUnit** of HttpUnit)
- mock/implement interfaces yourself

Testing J2EE - EJB

- embedded server (Glassfish) ?
  - all since EJB 3.1
- run them in container (Cactus)
- mock container (MockEJB)
- using an aspect to replace EJB lookups

- EJB 3.x are just POJOs ✓

NORWICH UNION
No.1 Sponsor of
uk:athletics
NORWICH UNION
UK CHALLENGE
BBC SPORT
IAAF GRAND PRIX SERIES
ukathletics.net
BBC SPORT
OMEGA
19
20
FRATE
4
4

Agenda

- JUnit Basics ✓
  - Test Methods, Assertions, Fixtures
- Advanced Topics ✓
  - Privates, Mocks, Timings, Singletons, J2EE
- **Tuning**
- Code Coverage
- JUnit Extensions
  - Tools, Scripting, JUnit and the Build

Tune Test Performance

- profile test suite - it’s run very often!
- Ant/JUnit report contains execution times
- target longest running tests
  - tune as any Java program (CPU, heap)
  - mock expensive/slow objects
  - avoid expensive set-up (e.g. Spring Context)
  - move expensive set-up to `@BeforeClass`

Test Performance - Database

- database access is slow
- mock out database
  - difficult for complex queries
- use embedded memory database
  - e.g. **HyperSQL DataBase** (HSQLDB), **H2**
  - beware of duplicating schema info
  - Hibernate's `import.sql`

DB/Integration Test Performance

- with database more an integration test
  - no problem - we want to test this too
- don't use fixtures
- do not commit
- connection pool
- tune database access (as usual)

الجمعية الشرعية للتقوية
الرياضيات
التاريخ
يزياء

Agenda

- JUnit Basics ✓
  - Test Methods, Assertions, Fixtures
- Advanced Topics ✓
  - Privates, Mocks, Timings, Singletons, J2EE
- Tuning ✓
- **Code Coverage**
- JUnit Extensions
  - Tools, Scripting, JUnit and the Build

Code Coverage

- tracks comprehensiveness of tests
  - % of classes/methods/lines that got executed
  - identifies parts of program lacking tests
- 85-90% is “good enough”
  - can’t reach 100% (catch-blocks etc.)
  - no need to test everything (getters etc.)
  - at least focus on core systems (business critical)

Code Coverage Tools

- **EMMA**
  - instrument classes offline or on the fly
  - detects partial coverage (if/short circuit)
  - Ant, Maven, Eclipse (**EclEmma**)
  - even able to track Eclipse plugins
  - also used in test staging to test the testers
- **Cobertura**
- etc.

“Don’t Be Fooled”

- comprehensiveness ≠ quality!
  - high coverage does not mean anything
  - tools like **AgitarOne** create it
- see http://www.ibm.com/developerworks/java/library/j-cq01316/
- “Test state coverage, not code coverage.” (Pragmatic Tip 65)
  - difficult to measure
- **Crap4J** “metric”

Development Process

- code test & class (or class & test)
- run tests with EclEmma (or on build)
  - all important methods executed?
  - all relevant if-branches executed?
  - most common error cases executed?
  - just browse the report line by line...

How to Get Coverage

- difficult to add tests to an existing program
- wasn’t written with testing in mind
- better to write tests before
- → Test Driven Development (TDD)

Red/Green/Refactor

- Design to Test (Pragmatic Tip 48)

But How To Test This?

Legacy Code

- ... is code without test. (Michael Feathers)
- write test for new features
- create tests for bug reports, then fix bugs
  - Find Bugs Once (Pragmatic Tip 66)
- find insertion points/bring them under test
  - for more see “Working Effectively with Legacy Code”
- refactor for testability (**TestabilityExplorer**)
  - see http://code.google.com/p/testability-explorer/

But Management Won’t Let Me

- Testing is a mindset - **You** have to want it.
- A thoroughly tested program will take twice as long to produce as one that's not tested.
  - – you need time to write tests
  - – argue for it
  - – or just lie →
    - hide time in your estimations
    - say the feature is not finished
    - write tests before, so you can’t finish without tests

HELPING HAND
DROP FORGED

Agenda

- JUnit Basics ✓
  - – Test Methods, Assertions, Fixtures
- Advanced Topics ✓
  - – Privates, Mocks, Timings, Singletons, J2EE
- Tuning ✓
- Code Coverage ✓
- **JUnit Extensions**
  - – Tools, Scripting, JUnit and the Build

JUnit Extensions (e.g.)

- **DbUnit** - database fixtures
- **HtmlUnit/HttpUnit** - GUI-less browser
  - typical for functional/integration tests
- **JUnitPerf** - measure performance
  - no ordinary unit test → different package
- **SWTBot** - UI testing SWT/Eclipse
- **XMLUnit** - XML asserts

New Trend: Scripting Languages

- “Testing is a scripting problem.”
  - dynamically typed, easier to write tests
- “If I can write tests in a rapid manner, I can view their results quicker.” (Andy Glover)
- need tight Java integration
- e.g. using Groovy
  - `GroovyTestCase` **`extends`** `TestCase`
  - see http://www.ibm.com/developerworks/java/library/j-pg11094/

(J)Ruby `Test::Unit`

- typical xUnit implementation
- asserts like
  - `assert_raise, assert_throws`
- advanced frameworks
  - **JtestR** - JRuby integration “so that running tests is totally painless to set up”
  - **RSpec** - Behaviour Driven Development framework for Ruby

ScalaTest

- run JUnit in ScalaTest
  - with wrapper `JUnit3WrapperSuite`
- run ScalaTest in JUnit (`JUnit3Suite`)
- **Specs** - Behaviour Driven Development
- **ScalaCheck** - property-based testing
  - automatic test case generation
  - `specify("startsWith", (a:String, b:String) => (a+b).startsWith(a) )`

JUnit and The Build

- the build must be fast (max. 10 minutes)
  - typically tests take large part of build time
  - monitor and tune test performance
- execute tests from very beginning (or die)
- make it impossible to deploy failed builds
- programmatically assessing and fixing blame is a bad practice

Ant and Maven

- Integration ✓
- Ant < 1.7
  - add junit.jar to Ant boot classpath (lib)
  - each JUnit 4.x test class needs to be wrapped as a JUnit 3.8 suite with `JUnit4TestAdapter`
- Maven
  - Hudson (uses Maven) continues if tests failed
  - build is marked as unstable

Running JUnit in Parallel (Ant)

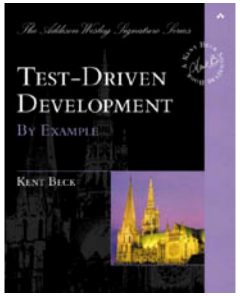
- causes lots of problems ☹
  - separate class loaders - more PermSpace
  - same class loader - singletons
    `(<junit … reloading="false">)`
  - separate VM instances = high start-up cost
    `(<junit ... fork="yes">)`
    `forkmode="perBatch"` only since Ant 1.6.2
  - load balancing of worker threads/VMs?
  - database race conditions, dead locks, ...

Distributed JUnit

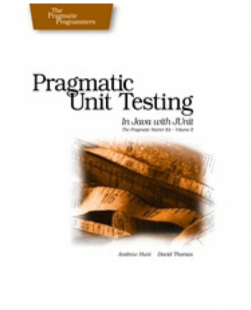
- not all tests are the same...
- small/fast tests should not be distributed
  - distributing takes up to 90% of total time
- performs best with a few long running tests
- **Distributed JUnit** (ComputeFarm & Jini)
- **GridGain**’s `JunitSuiteAdapter`
- commercial build servers/agent technology

THE EDINBURGH ANGUS CLUB
COMPANIONS OF COLUMBUS
DUTY OF MAN
LAST ESSAYS OF ELIA
C. LAMB

Some Good Books...

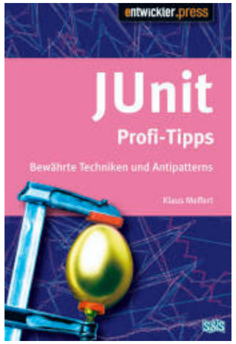


- Kent Beck - Test Driven Development. By Example (2002)

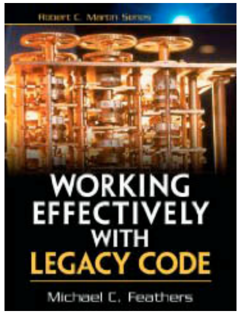


- Andy Hunt, Dave Thomas - Pragmatic Unit Testing in Java with JUnit (2003)

Some Good Books...



- Klaus Meffert - JUnit Profi-Tipps (2006)



- Michael Feathers - Working Effectively with Legacy Code (2007)

Now go and write some tests!

Q&A

- Thank you for listening.
- http://www.code-cop.org/presentations/

Image Sources

- http://rubystammtisch.at/

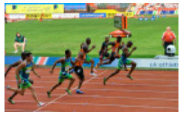
- http://www.flickr.com/photos/43442082@N00/296362882/

- http://www.flickr.com/photos/eszter/144393616/

- http://www.flickr.com/photos/sneddon/2413980712/

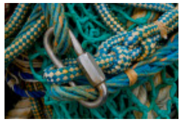
Image Sources

- http://www.flickr.com/photos/paopix/184238679/



- http://www.flickr.com/photos/teotwawki/164966631/

- http://www.flickr.com/photos/paulk/3166328163/



- http://www.flickr.com/photos/otolithe/1831281833/

Image Sources

- http://www.flickr.com/photos/rainforestactionnetwork/420117729/

- http://www.flickr.com/photos/ppdigital/ 2054989998/

- http://www.flickr.com/photos/good_day/ 48642035/

- http://www.flickr.com/photos/shelley13/ 2653029303/