VOLVO

Micael Andersson
Senior Lead
Architect & Developer

Testing

Introduction to developers

Micael Andersson
Senior Lead Architect & Developer

October 2023

Pitch


Okay let me put this code live
Wait a minute, where are your unit tests?
No need for that, I have something much better
Oh really?
Yeah, look
Push to production done
Now listen...
Hey what the...
Guys, the app is broken, what the hell did you do?!
Shit
So there is a bug!
CommitStrip.com

VOLVO

This training will cover these areas:

- Unit testing
- Test Driven Development
- Test Coverage
- Design for testability
- Mocking
- Refactoring

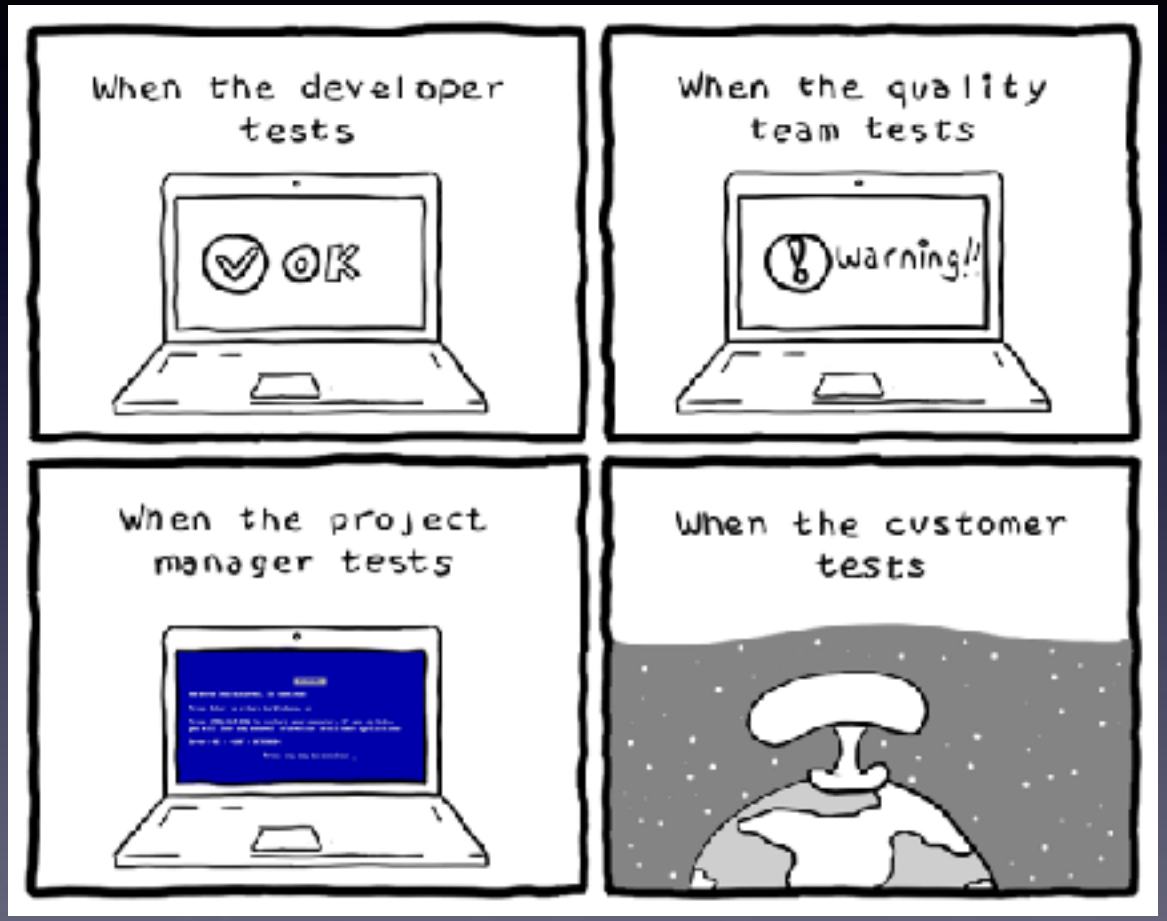
Goals of this presentation

- To **introduce and motivate** the Unit Testing practice.
- To describe how Unit Testing **fits within** a Software Development Process.
- Show **examples** of Unit Testing using a simple yet interesting enough sample project.
- Share **experiences** from Testing area, hopefully **have fun** (and do **pair programming**.)

Questions are most welcome!

Testing

and Automated tests



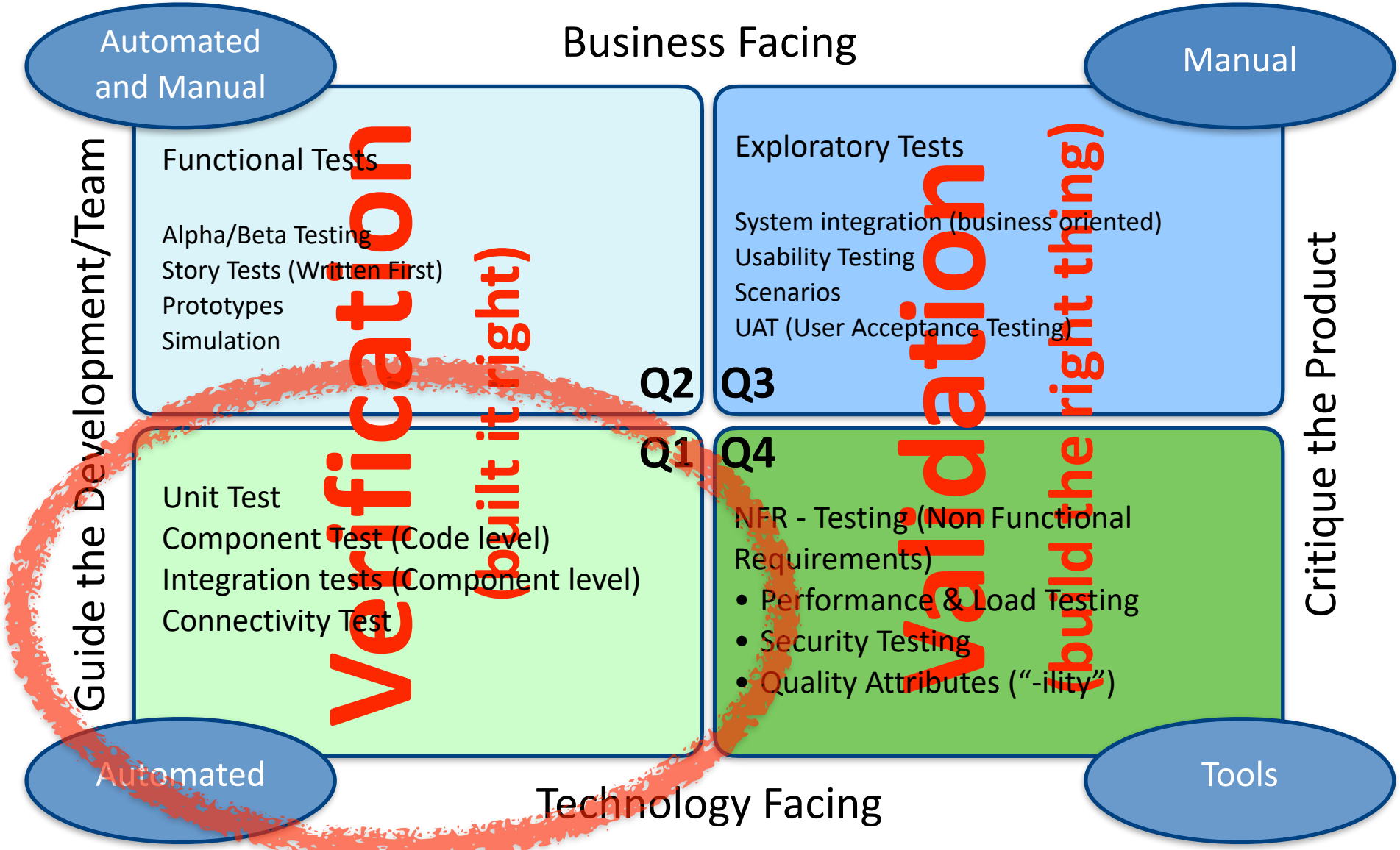
About Tests …

Everybody knows they should, but few actually do.

“Why isn’t this tested before”?

- Because it has been too expensive, difficult, cumbersome to test
- Because we have been too busy
- Because things have changed

Test Automation & Agile Test Quadrant



from Brian Marick
popularised by Lisa Crispin and Janet Gregory

Quality Assurance precedes Quality Assessment

Testing is about **Quality Assurance**, not just Quality Assessment

Quality Assessment only indirectly affect quality

Testing **reveals** information about weaknesses in the System Under Test (SUT)

Testing helps to **focus** project activity.

Goals for Test Automation…

… should be S.M.A.R.T:

Self Contained (and **S**tateless)

Maintainable

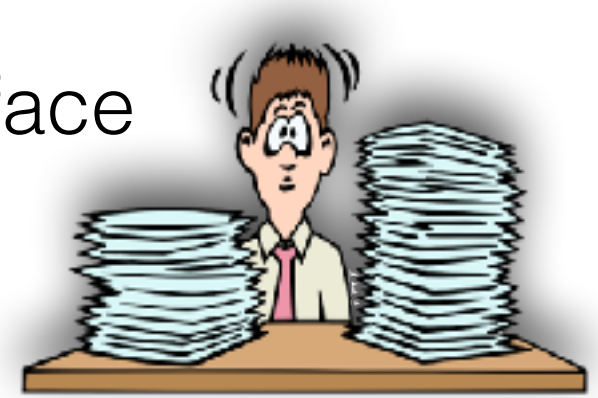
Act as documentation

Repeatable and Robust

To the point – provide good ”defect triangulation”

Manual Tests are …

- Tedious, when done repetitive…
- Error-prone, because we are only human…
- Difficult to test other units than the User Interface



yet …

a (manual) Test Process must be present in order to automate it!

Critical Success Factors for Automated Tests

Repeatability and Consistency

- Once the test is complete, it should **pass repeatedly**, whether it executes by itself or within a test suite.
- When a completed test fails, we need too quickly and **accurately pinpoint** the cause: did the test uncover a bug in the system, or is the test itself faulty?

Readability

- The tests are the definitive **reference for the system requirements**.

Maintainability

- Iterative, test-first development yields much more test code than production code (typically 5-10 times more)
- Thus we have to be as concerned (or more) with the maintenance costs of test code as compared to system code.

Testability, of the production system

Testability consists of two fundamental characteristics:

- **Visibility** – the tester can see (and understand) what happens within the production system (i.e. can observe important aspects of the internal state of the system)
- **Control** – the tester can force interesting things to happen within the production system (i.e. can control its behaviour)

Testability doesn't just happen. **It must be designed and built into a production system**.

Writing tests before designing and building the system (a.k.a. Test-First or Test-Driven Development) is a great way of achieving good testability.

Classifying Automated Tests

Granularity

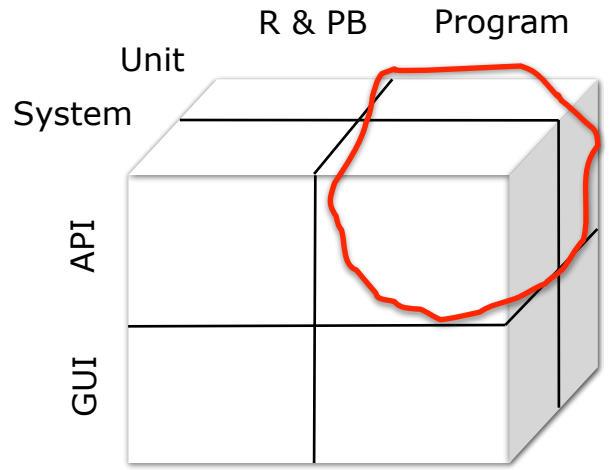
- –Entire system
- –Individual units

Point of Contact

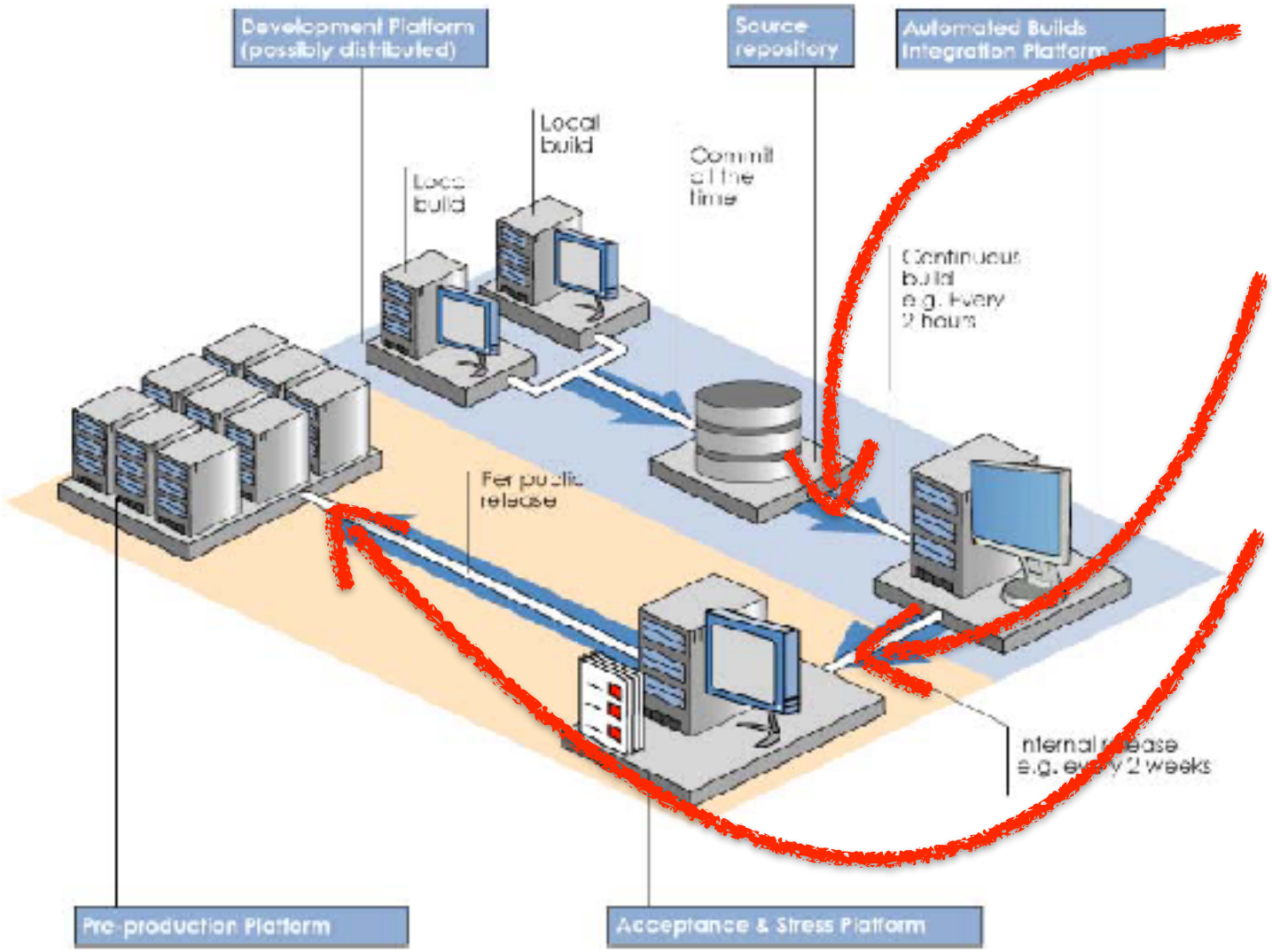
- –Through existing UI (User Interface)
- –Testability API

Test Case Production

- –Record & Play Back
- –Hand Written (programmatic)



Test and build automation (CI or CD)



Continuous Integration basically just means that the developer's working copies are synchronised with a shared mainline several times a day.

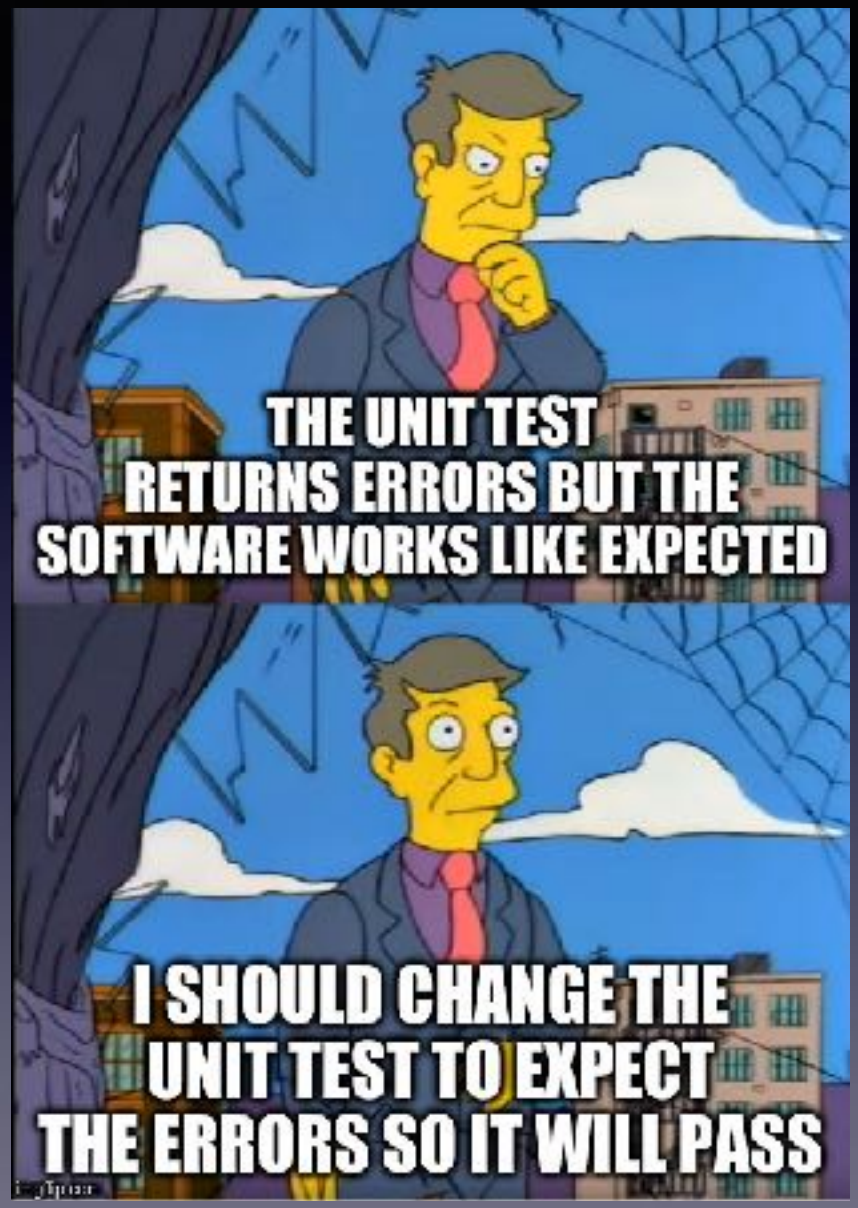
Continuous Delivery is described as the logical evolution of continuous integration: Always be able to put a product into production!

Continuous Deployment is described as the logical next step after continuous delivery: Automatically deploy the product into production whenever it passes QA!

N.B: Sometimes the term "Continuous Deployment" is also used if you are able to continuously deploy to the test system.

Testing

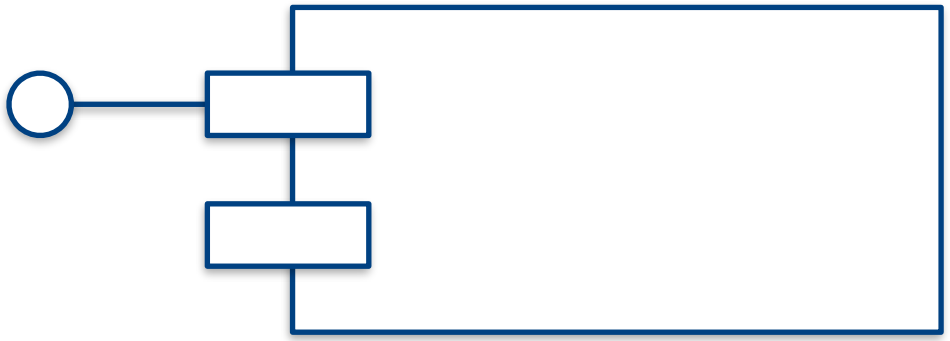
Introduction to Unit Testing



Unit Tests

Black-box test of a ***logical unit***, which verifies that the logical unit behaves (functionality wise) correctly – ***honors its contract***.

White-box test of a ***logical unit***, verifies that the internal structure or workings behaves correctly

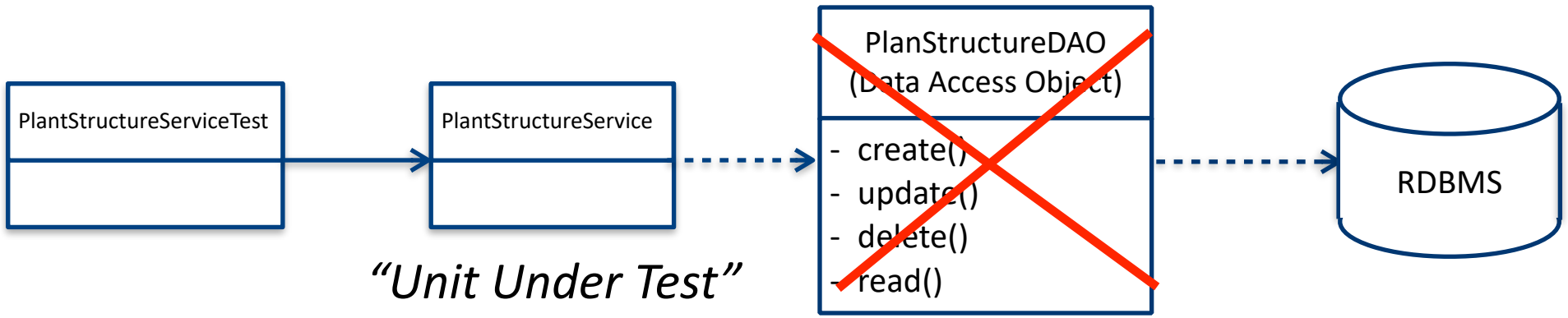


"Unit Under Test"

What exactly is a Unit Test?

A self-contained software module (typically a Class) containing one or more test scenarios which tests a “Unit Under Test” in isolation.

Each test scenario is autonomous, and tests a separate aspect of the “Unit Under Test”.



Smoke Tests

A set of Unit Tests (which tests a set of logical units) executed as a whole provides a way to perform a ***Smoke Test***:
Turn it on, and make sure that it doesn't come smoke out of it!

A relatively cheap way to see that the units "**seems to be working and fit together**", even though there are no guarantees for its overall function (which requires functional testing)

Developer testing vs. Acceptance testing

Unit Tests are written **by developers, for developers**.

Unit Tests **do NOT address formal validation and verification of correctness** (even though it has indirect impact on it!) - Unit Tests prove that some code does what we intended it to do

Unit Tests **complements** Acceptance Tests (it does not replace it)

Why should I (as a developer) bother?

Well-tested code works better. **Customers like** it better.

Tests **support refactoring**. Since we want to ship useful function early and often, we know that we'll be evolving the design with refactoring.

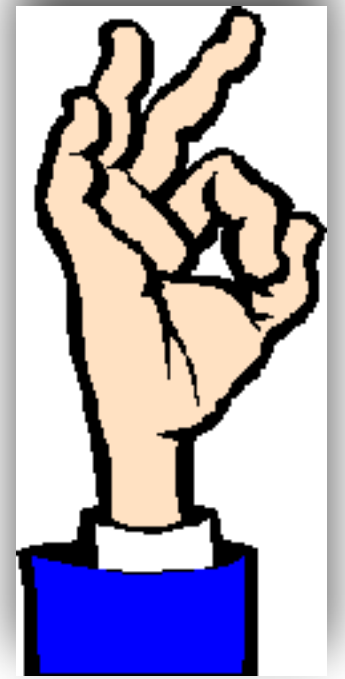
Tests **give us confidence**. We're able to work with less stress, and we're not afraid to experiment as we go.

Hence Unit Testing will **make my life easier**

- It will make my **design** better
- It will give me the **confidence needed to refactor** when necessary
- It will dramatically **reduce the time** I spend with the debugger
- It will make me **sleep better** when deadlines are closing in

Requirements on Unit Tests

- Easy to **write** a test class
- Easy to **find** test classes
- Easy to test different **aspects** of a contract
- Easy to **maintain** tests
- Easy to **run** tests

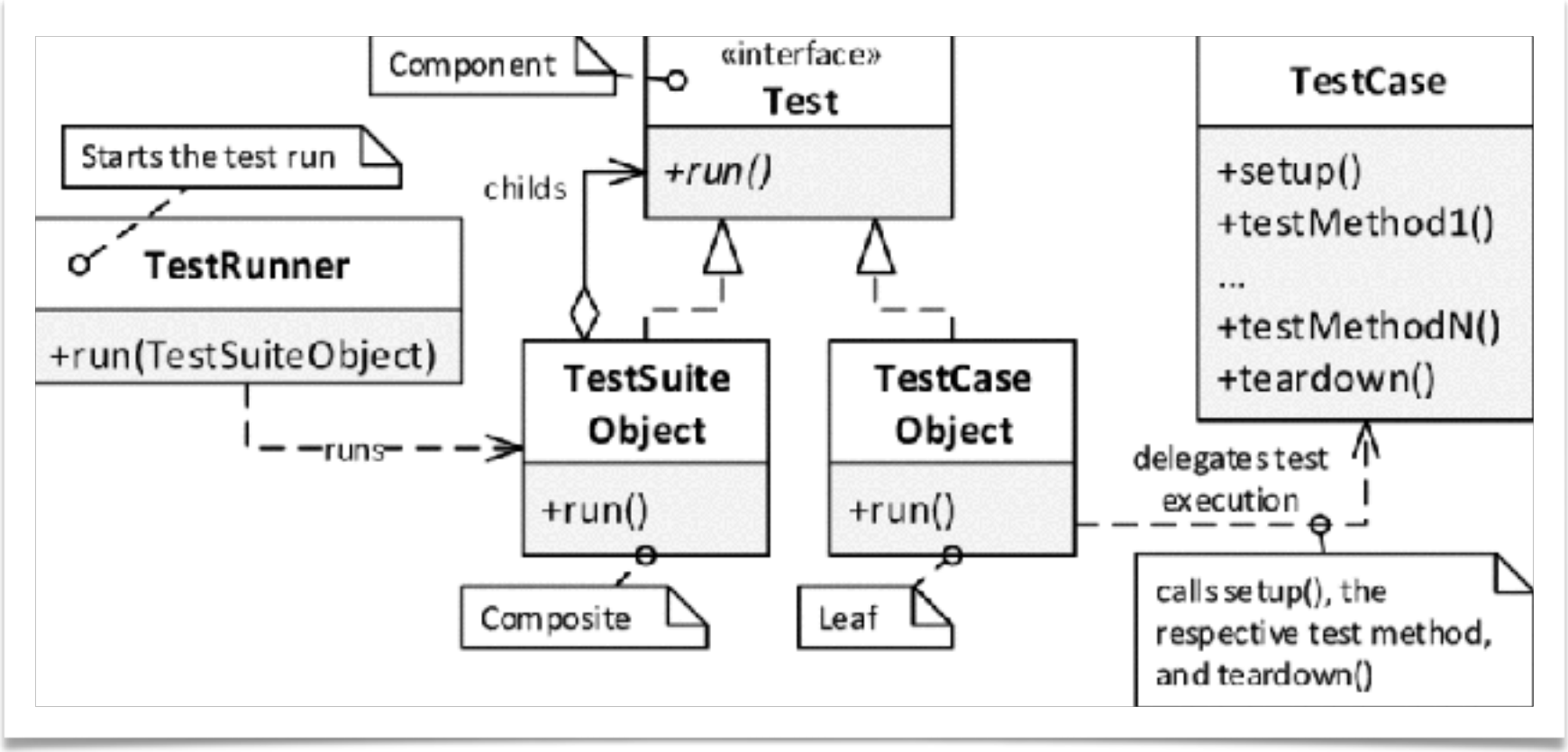


xUnit: Frameworks for Unit tests

www.junit.org

www.csunit.org

www.nunit.org



Catch2 framework for C++

https://sourceforge.net/projects/cppunit/

Testing

Unit testing with JUnit

JUnit Test Example

```
public interface IAccount {
   public void deposit(int amount);
   public void withdraw(int amount) throws AccountException;
   public int getBalance();
      …
}

public class AccountImplTest {
    @Test
    public void testWithdraw() throws AccountException {
        AccountImpl account = new AccountImpl("1234-9999", 2000);
        account.withdraw(300);
        Assert.assertEquals(1700, account.getBalance());
    }
    @Test
    public void testWithdrawTooMuch() throws AccountException {
    …
    }
}
```

All methods annotated with @Test are considered test scenarios

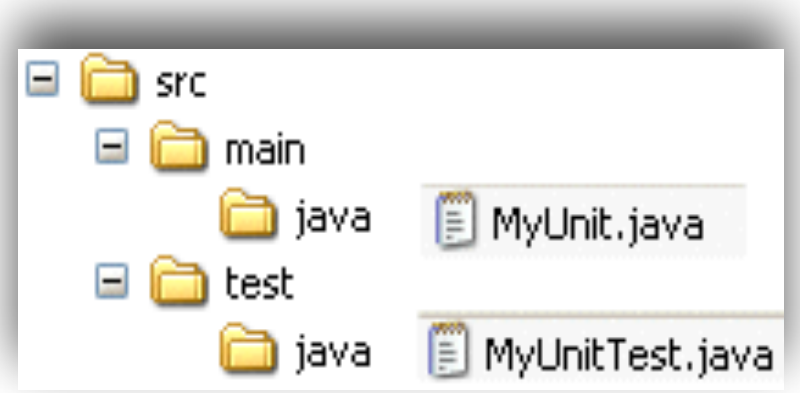
Naming Conventions and Directory Structure

Unit Tests should be named after the Unit that is tested, with "Test" appended.

A class usually represents a noun, it is a model of a concept. An instance containing tests (a suite) of one would be a 'MyUnit tests'.

In contrast, a method would model some kind of action, like 'test [the] calculate [method]'.

- the tests for MyUnit is in --> MyUnitTests
- test of the calculate method --> testCalculate();
- JUnit tests should be placed within the same package as the Unit under Test, but in a different parallell directory structure.



Assert: Support for verifying conditions (JUnit 4)

```
static void assertEquals(“A message!”, int expected, int actual);
static void assertEquals(“msg”, double expected, double actual, double delta);
static void assertEquals(“A message!”, Object expected, Object actual);

static void assertFalse(String message, boolean condition);
static void assertTrue(String message, boolean condition);

static void assertNull(String message, Object object);
static void assertNotNull(String message, Object object);

static void fail(“An explaining message!”);
```

Assert: Support for verifying conditions (JUnit 5)

```
static void assertEquals(int expected, int actual, “A message!”);
static void assertEquals(double expected,double actual, double delta, “Msg”);
static void assertEquals(Object expected, Object actual, “Informative msg”);

static void assertFalse(boolean condition, String message);
static void assertTrue(boolean condition, String message);

static void assertNull(jObject object, String message);
static void assertNotNull(Object object, String message);

static void fail(“An explaining message!”);
```

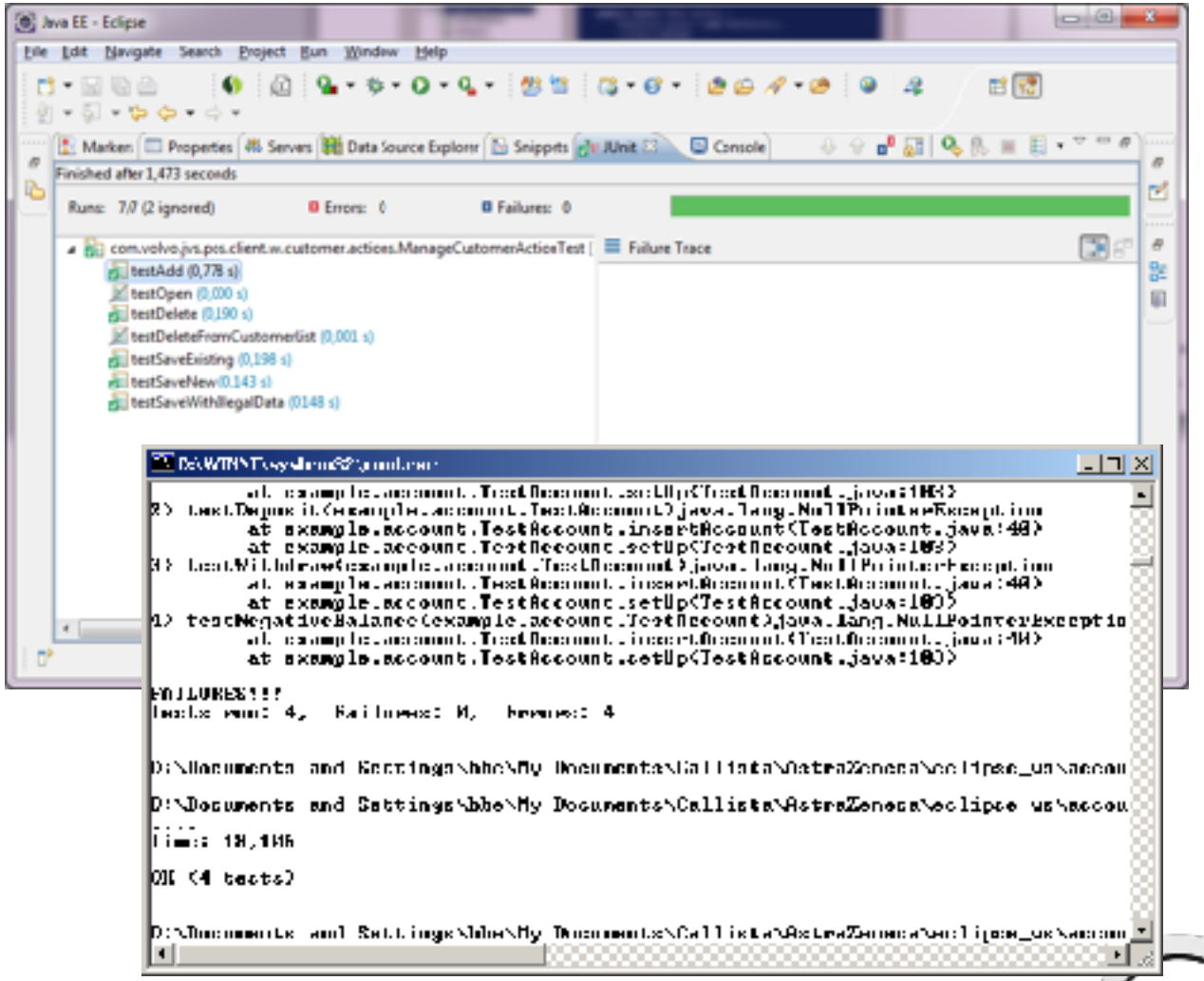
Executing/Running JUnit Tests

From within the IDE (like Eclipse)

From command line (like gradle)

From automated builds:

- Build servers
- Jenkins
- CI systems e.t.c.



Typical unit test scenario – The Three A's

1. **A**rrange - Instantiate Unit under Test and set up test data
2. **A**ct - Execute one or more methods on the Unit Under Test
3. **A**ssert - Verify the results

```
public interface IAccount {
    public void deposit(int amount);
    public void withdraw(int amount) throws AccountException;
    public int getBalance();
}

public class AccountImplTests {
     @Test
     public void testWithdraw() throws AccountException {
          AccountImpl account = new AccountImpl("1234-9999", 2000);    // ARRANGE
          account.withdraw(300);                                       // ACT
          Assert.assertEquals(1700, account.getBalance());            // ASSERT
     }
     … … …
}
```

General Rules of Thumb

Create a single test class for each non-trivial application class you have.

Give a readable, **meaningful name** to each test method. A good naming convention is to name test methods using the same name as the method that it is being tested, with some additional info appended to the name. For instance if testing a method called "***withdraw(...)***" in the Account class, create a few test methods to test different aspects of withdrawal:

```
@Test
public void testWithdrawTooMuch() throws AccountException {…}
@Test
public void testWithdrawBigAmount() throws AccountException {…}
@Test
public void testWithdrawNegativeAmount() throws AccountException {…}
```

The scope of how much checking to do in a single test case (test method) is a judgment call. It is usually better to test only one scenario (and hence one potential error condition) in each test method. Remember : **tests should be "to the point"**.

setUp() and tearDown()

```
public class AccountImplTest {

    private AccountImpl account;

    @Before
    public void setUp() {
        account = new AccountImpl("1234-9999", 2000);
    }
    @Test
    public void testInitialBalance() {
        int actualBalance = account.getBalance();
        Assert.assertEquals(2000, actualBalance);
    }
    @Test
    public void testWithdraw() throws AccountException {
        account.withdraw(300);
            int actualBalance = account.getBalance();
        Assert.assertEquals(1700, actualBalance);
    }
  …
}
```

- Methods annotated with `@Before` are executed *before every test method*. (`@BeforeEach` in JUnit 5)
- Methods annotated with `@After` are executed *after every test method*. (`@AfterEach` in JUnit5)

Working with Exceptions (JUnit 4)

Unchecked (i.e null, div by zero) exceptions thrown during execution of a test will be caught by the JUnit framework and reported as Errors (i.e. test will fail)

A Test method must declare that it throws any checked exceptions that the Unit under Test may throw. If there are several checked exceptions that may occur, it is perfectly valid for a test method to declare throwing java.lang.Exception.

Expected exceptions (exceptions that the test is expecting the "Unit under Test" to throw in a certain situation) are expressed using the **`@Test(expected=ExpectedException.class)`** annotation attribute

```
@Test(expected=NastyException.class)
public void doSomethingNastyTest() throws NastyException {
   NastyUnit unitUnderTest = new NastyUnit();
   unitUnderTest.doSomethingNasty();
}
```

Working with Exceptions (Contd. & JUnit 5)

Or using the following idiom:

```
@Test
public void doSomethingNastyTryCatchTest() {
   NastyUnit unitUnderTest = new NastyUnit();
   try {
      unitUnderTest.doSomethingNasty();
      Assert.fail("Expected a NastyException!");
   } catch (NastyException expected) {
      // We expect to end up here!
   }
}
```

@Ignore (@Delete in JUnit 5)

To temporary ignore a test, use the `@Ignore` annotation:

```
@Ignore("Problem right now, trying to fix the bug”)
@Test // @Test annotation must still be there
public void testThatDoesNotWorkYet(){
   SomeUnit target = new SomeUnit();
   target.doSomethingThatDoesNotWork();
   Assert.assertTrue(target.isValid());
}
```

Testing

at different access levels

Testing private or protected methods/members

Java (and others) have different access levels, where can we test:

Easy {

Hard {

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	X
no modifier	Y	Y	X	X
private	Y	X	X	X

Testing

How to find what to test.

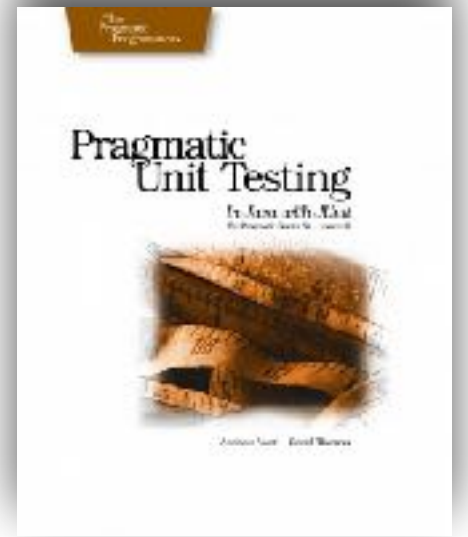
Exhaustive Testing ???

Exhaustive Testing vs. Pragmatic Testing

Exhaustive testing, means cover every possible aspect of the function or non function.

That will take “forever and a day” and might not even be possible. Especially not in real life.

Pragmatic testing is our approach…



by Andrew Hunt & David Thomas 2003

What should be tested?

It **can be hard** to look at code and try to come up with all the ways it might fail.

With enough experience, you start to get a feel for those things that are **“likely to break”** and can effectively concentrate testing in those areas first.

But **without a lot of experience**, it can be hard and frustrating trying to discover possible failure modes.

End-users are quite adept at finding our bugs, but that’s both **embarrassing and damaging** to our careers!

What to test?

Everything that could possibly break!

Corollary:
Don’t test stuff that is too simple!
(example: setters/getters and DTOs e.t.c.)

Typical problematic areas:

- **Error** conditions
- **Boundary** conditions (should be CORRECT)

Boundary conditions being CORRECT

Conformance

- Does the value conform to an expected format? (like, 1 000 SEK)

Ordering

- Is the set of values ordered or unordered as appropriate?

Range

- Is the value within reasonable minimum and maximum values? (like, 0-100%)

Reference

- Does the code reference anything external that isn't under direct control of the code itself?

Existence

- Does the value exist (e.g., is non-null, non- zero, present in a set, file exists, etc.)?

Cardinality

- Are there exactly enough values? (Like, exactly two parents)

Time (absolute and relative)

- Is everything happening in the right order? At the right time? In time?

What to test? - The Right-BICEP!

We need some guidelines, some reminders of areas that might be important to test.
Six specific areas to test that will strengthen your testing skills, using your Right-BICEP:

Right — Are the results right?

B — Are all the **boundary conditions** CORRECT?

I — Can you check **inverse** relationships (x^2 <-> sqrt()) ?

C — Can you **cross-check** results using other means?

E — Can you force **error conditions** to happen?

P — Are **performance** characteristics within bounds?

Thank you for listening!

We just started our way to better testing…