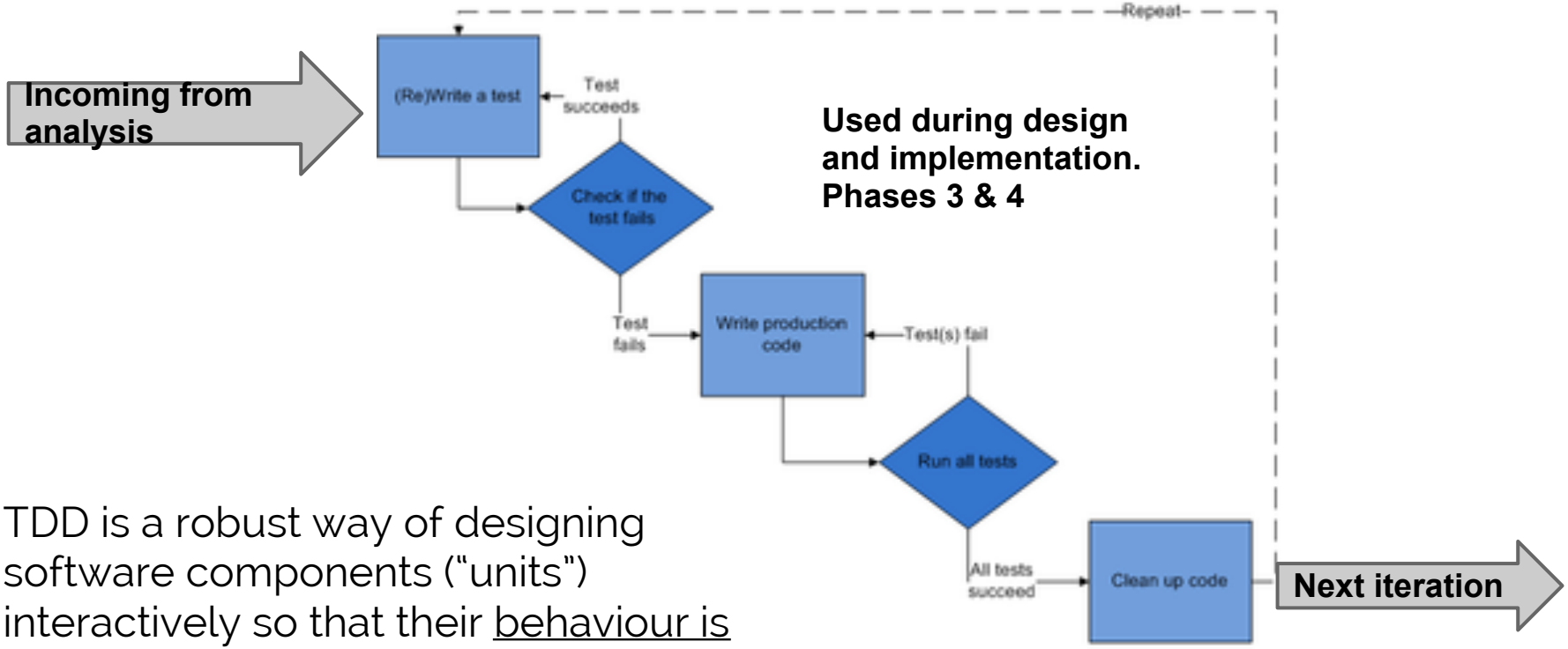
Test Driven Development

Slide Series 4

Test Driven Development

Is a development approach

- So it's not a testing approach



TDD is a robust way of designing software components ("units") interactively so that their behaviour is specified through **unit tests.**

TDD in Course

A bit relaxed

- We'll get a few UC's up and running before starting

...when UC's running and design started to stabilize we start writing tests

- Ok to write test in parallel to code (i.e. not before)

Testability

To make it possible to test the application we must design it for testability

- Minimize dependencies
- Avoid inline use of static classes, Singletons or any global dependency
- No inline use of **new**, use factories
- All dependencies passed in via constructor (later possible to mock)
- Use interfaces
- More later ...

Organizing Test Code

Keep test code separate from application code

- Use (create) a source folder "test" in Eclipse
- Use same package structure as application, more to come...

The Units

Single class (basic functionality)

Small aggregate of classes (to fulfill a use case)

Service (package with unified interface), more to come...

No "visual" units (i.e. GUI tests)

Using JUnit

JUnit is test framework for Java (many similar for other platforms)

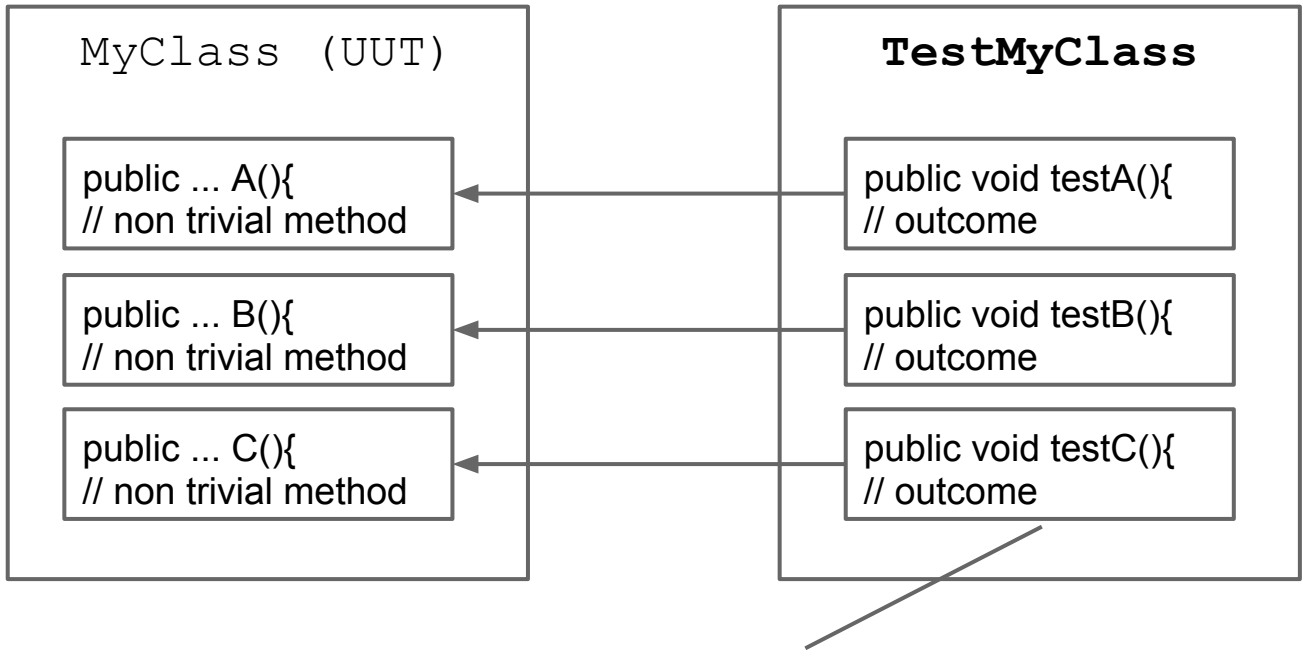
- Installed plugin in Eclipse

Procedure

- We have a unit under test, UUT
- Write a "test"-class (in Java) for the UUT
- Let JUnit run the test class (exercising the UUT)
- JUnit will report outcome of tests

UUT is a Single Class

Test class has one test-method for each public non-trivial method of UUT



Outcome is a logical true/false i.e. test passed or not

Implementing Test Methods

A test method must have signature and annotation as (i.e. no return value no params);

```
@Test
public void anyNameHere(){
  // Code to run test
}
```

Possible if exception should be thrown

```
@Test(expected=IllegalStateException.class)
```

Implementing Test Methods, cont

Last in test method a "logical test"-expression (assert*-expressions, part of JUnit, built in)

- assertTrue(o.getName().equals("Olof"))
- assertFalse(o.getValue <= 199))
- No System.out println() in test methods (only acceptable during development of test, use debugger more to come...)

Test shouldn't need any interaction, we run, tests pass... (if fail have to fix)

- Just a mouse click to run all tests

Test Method Names

Should be looooooong and descriptive

```
@Test
public void playerLandOnPropertyOwnedByOther(){
    ... // Similar to UC
}
```

Private and Void

If UUT method private, normally not in test.

- If in need, change to public and back (bad.., just for now)

If UUT method void, need to inspect state

- Possible extra "test"-method getNNN(), not part of any public API

If UUT (class) private

- Possible to test if we have same package-structure in test source folder as in application (src-folder)

UUT is an Aggregate

If more objects needed have to create a fixture

- = A setup to create and connect involved objects
- JUnit has special setup-methods (and tear down)

```
// In test class
@BeforeClass  // or @AfterClass
public static void anyNameHere(){
    ... // Run once before all test
}

@Before // or @After
public void anyNameHere(){
... // Run before **each** test
}
```

Designing Tests

Try to find UUT-method possible to run in isolation

- If not possible let test-method use as few UUT-methods as possible
- Use UUT-methods already tested in later test-methods

Make each test orthogonal

- I.e. independent to all the others, tests should not rely on each other
- Should not use global state (test must run with well defined start state)

"Very" Graphical Application

Games and alike

- Should be possible to use TDD but maybe not to the same extent
- Model should be tested, the graphics is just a view of model!

What's happening in the model?

Debugging using Tests

Very efficient

- Much easier than to run complete application
- Put breakpoint in test class ...
- .. continue a usual

Code Coverage

*"Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested." //*Wikipedia

Possible to see how much of code is run during tests, high percentage good (90% or more)

Many tools, ECLEmma a plugin for Eclipse

TDD: Pros and Cons

Pros

- Kind of specification
- Confidence in doing changes (refactoring), just re-run the tests!
- Documentation

Cons

- All tests passed does not imply application pass (i.e. no bugs)
- No specific technique for this in course, have to find...

Summary

We use TDD as a way to increase the quality of the units of the application

Will make increase confidence when refactor

Will be part of documentation