

1 Team

Complementary Skills; Mutual Accountability;
Common Commitment; Shared Goal; Collective Work
Products; More than the sum of its parts

1.1 Meetings

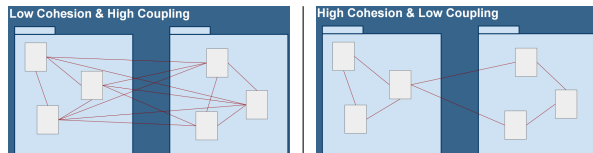
Hogging – talking too much; Flogging – beating an
issue to death; Frogging – jumping from topic to
topic; Bogging – getting stuck on an issue; Ignoring
the Elephant in the Corner

2 Software Architecture

2.1 Quality

Cohesion – Degree to which elements of a module fit
together

Coupling – Degree of interdependence between
modules



3 Continuous Integration

Merge Frequently; Don't push broken code; Don't
push untested code; Don't push when the build is
broken; If the build is broken, fix it

4 Test-Driven Development

A software development methodology based
on: Short development iterations, Satisfying
pre-prepared test cases. An independent offshoot
of Agile methodologies. Based on using automated
unit testing to drive software development.

```
import org.junit.Test;
import static org.junit.Assert.*;

public class FlooperTest {

    @Test
    public void testWorking() {
        // do some set up

        int result = flooper.flooper();
        assertEquals("Wasn't 3", 3, result);
    }
}

import org.junit.Test;
import static org.junit.Assert.*;

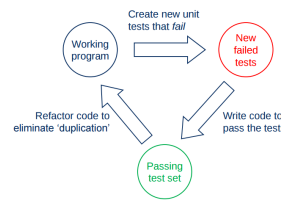
public class FlooperTest {

    @Before
    public void beforeEachTest() {
        // some set up that's run before each test
        // in this class
    }

    @BeforeClass
    public static void beforeAnyTests() {
        // some set up that's run once before
        // any of the tests in this class
    }

    // etc
}
```

4.1 Test-Driven Dev: Red, Green, Refactor



Applying Test-Driven
Development relies
on the existence of an

automated unit testing
environment. You are
obliged to maintain a
suite of test cases. Code
must not be released
until it has associated
tests. The tests are written
before the code.

4.2 Refactoring

Code that needs refactoring has: Duplication, Unclear
intent, Tight coupling, Pure data classes, Over-sized
or under-sized classes, Complex or long methods,
Switch statements instead of polymorphism.

4.3 Designing Unit Tests

Test one thing only (Use few assertions per test).
Work in isolation (Without relying on other
tests). Test boundary conditions early. Avoid
testing against “real” resources, i.e. GUIs or
databases, to support testing determinism (Use
mock objects and services - with fixed data)

How Many Tests?

4.4 Mocking

Dummies - test objects which are never used but
exist only to satisfy syntactic requirements

Stubs - test objects whose methods return fixed
values, and support the specific test cases only

Fakes - test objects whose methods work but have
only limited functionality

Mocks - test object which know how they're
meant to be used, e.g. the sequence in which their
methods should be called (allowing behavioural
verification instead of just state verification)

5 Pair Programming

Constant review from two people ensures fewer

defects. Works well for mentoring: inexperienced
staff, new team members, learning new techniques
or tools.

Driver - person at the keyboard

Navigator - focusing on design

Both need to be actively engaged - keep a running
commentary

Switch roles frequently - every few minutes

5.1 Ping-Pong Programming

Driver writes a failing unit test. Driver
& Navigator switch roles. New driver
implements code to pass test - then write
a new failing unit test. Switch roles again

6 Class Model

6.1 Class Icon

Employee (Class Name)
-employeeNumber:String (Attribute)
-nextEmployeeNumber:String (Static Attribute)
-qualification:Qualification[]
+addQualification(qual:Qualification) (Operation)
+getDepartment():Department
+changeDepartment(dept:Department)

6.2 Association

```
classDiagram
    Employee "1" -- "*" EmployeeDAO
    class EmployeeDAO {
        +getAll():List<Employee>
        +getEmployee(employeeNum:String):Employee
        +update(employee:Employee):int
    }
```

Navigability; arrow in direction of usage,
no arrow is bi-directional. Multiplicity;
min..max (or n), * is unlimited.

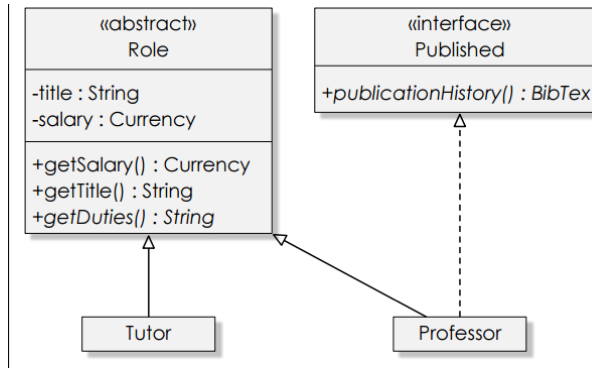
6.3 Aggregation

Strong relationship “has a”. Hollow diamond

6.4 Composition

Strong relation “is part of”. When
composite is destroyed so is the part
coincident life-span. Filled diamond

6.5 Inheritance/Subtyping



Generalisation/Specialisation: solid line, hollow arrow head. Implements: dashed line, hollow arrow head. Italics == abstract.

6.6 Packages

Dashed line = dependency. Solid line = Nesting

7 System

Jenkins – Builds the changes to the repo; SonarQube – checks for codesmells and code errors; Gradle – For testing and running on local machine