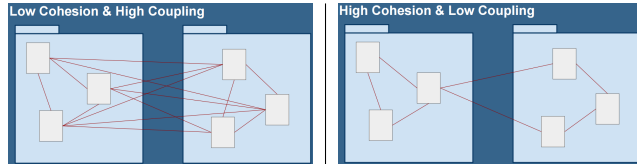


# 1 Software Architecture

## 1.1 Quality

Cohesion – Degree to which elements of a module fit together  
Coupling – Degree of interdependence between modules



## 2 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

## 3 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.

### How Many Tests?

Test or both black-box and glass-box. As the programmer add glass-box tests for: Conditionals, Loops, Operations, Polymorphism.

```
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 org.junit.Before;
import org.junit.BeforeClass;
import static org.junit.Assert.*;

public class FlooperTest {

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

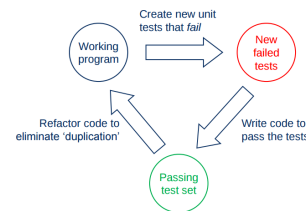
    @BeforeClass
    public static void beforeAnyTests() {
        // some set up that's run once before
        // any of the tests in this class
    }

    // etc
}
```

### 3.1 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.

### 3.2 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.

### 3.3 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)

## 4 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

### 4.1 Ping-Pong Programming

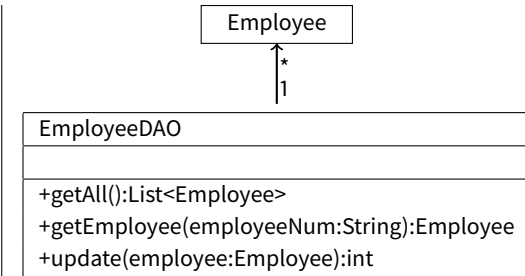
Driver writes a failing unit test. Driver & Navigator switch roles. New driver implements code to pass test - then write a new failing unit test. Switch roles again

## 5 Class Model

### 5.1 Class Icon

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

### 5.2 Association



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

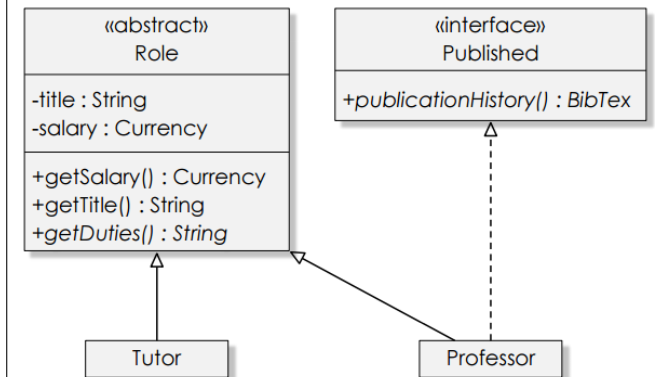
### 5.3 Aggregation

Strong relationship "has a". Hollow diamond

### 5.4 Composition

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

### 5.5 Inheritance/Subtyping



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

### 5.6 Packages

Dashed line = dependency. Solid line = Nesting

## 6 Design Patterns

Apply at various levels of abstraction. Are not reusable classes. Are not complex, domain-specific designs. Are limited in scope. Capture design intent, but not the full detail.

### 6.1 Common Language

Provide a common language for describing solutions; Each window is a composite - with decorators providing titles and scroll bars, To save memory - each image is a flyweight.

### 6.2 Pattern Form (GOF)

Pattern Name - short descriptive moniker and any aliases.  
 Intent - short statement summarizing what problem it solves.  
 Motivation - scenario that describes a problem this pattern solves.  
 Applicability - how to identify when to use this pattern.  
 Structure - description of class relationships and the object interactions.  
 Participants - classes making up the pattern and their responsibilities.  
 Collaborations - how the classes collaborate to perform their responsibilities.  
 Consequences - benefits and trade-offs of using the pattern.  
 Implementation - hints for implementing the pattern (consider language specific issues).  
 Known Uses - examples of the use of this pattern in real systems (the Rule of Three).  
 Related Patterns.

### 6.3 Common Patterns

#### Singleton

**Intent:** Ensure a class only has one instance, and provide a global point of access to it (`getInstance()` function)

#### Observer

**Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Consequences:** Loose coupling between Subject and Observer, Broadcast communication.

#### Composite

**Intent:** Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**Examples:** Menu Bars, Graphics.

#### Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests.

#### Visitor

Represent an operation to be performed on the elements of an object structure.

#### State

Allow an object to alter its behavior when its internal state changes.

#### Lazy Loading

Avoid creating a large/complex/expensive object until you actually need it. Create a placeholder object that can be substituted out when the real object is actually needed.

**Proxy** - loads real object when data is accessed then forwards messages to real object (delegates).

**Ghost** - real object is created but without any data. Loads data when a method is invoked.

### Radial Menus

**Intent:** Present user commands or choices radiating out from a central point.

**Similar Patterns:** Icons instead of Text.

**Advantages:** Shorter average distance to each item, greater distance between each item, faster interaction for expert users, works well for selecting graphical options.

**Disadvantages:** Doesn't scale/nest as well as vertical menus (best for 3-12 items or so), not as intuitive to read for the novice user, less familiar, text has to remain horizontal.

### 6.4 Anti-Patterns

Some things look like patterns, but make things worse, not better. Knowing these is almost as valuable as knowing the "good" patterns.

## 7 Code Smells

Code smells refer to any symptom in the source code that could possibly indicate a deeper problem. Code smells are not the problem. They do not produce compile errors and are not even bugs. Simply, they are evidence that there might be a bug or other issue nearby.

### 7.1 Code Smells v Anti-Patterns

Code smells are not the problem, however the benefit of understanding code smells is to help you discover and correct the anti-patterns and bugs that are the real problems.

## 8 User-Centered Design

Understand -> Define -> Ideate (Sketch) -> Prototype -> Test -> Deliver!

## 9 Logging

### 9.1 java.util.logging

Levels: SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST (in order, also OFF and ALL). Level of the logger is a cutoff for recording messages

### 9.2 Log4J

Levels: FATAL, ERROR, WARN, INFO, DEBUG, TRACE (also OFF, ALL)

### 9.3 slf4j

Brings all the loggers together, enabling switching during runtime or on compilation. Avoids every library having its own logging facility. Uses all the log levels of log4j but FATAL.

## 10 System

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

## 11 JDBC(Java DataBase Connectivity)

Standard library for relational databases. Standardises: Connecting, Queries and Updates, Results. Does not standardize SQL syntax (Sends strings).

### 11.1 Drivers

JDBC Driver Manager - Communicates with vendor-specific drivers that perform the real communication with the database.

### 11.2 Plan for changes

Limit data access to single area of code and don't distribute JDBC calls throughout the code. Don't return JDBC-specific objects from the data-access layer (Return ordinary Java Objects).

## 12 Apache Derby

Written in Java. Good for small/medium applications (less than gig size, few queries/second). **Embedded mode:** Database runs in same VM as Java app. Does not accept network connections. Perfect for simple self-contained applications (easy setup).

**Standalone mode:** Runs in separate VM. Accepts network connections.

## 13 Prepared Statements

Repeatedly executing query or update where format stays consistent, but values change.

### 13.1 Advantages

Move convenient than string concatenation. Much less susceptible to SQL injection attacks.

## 14 Advanced Features

### 14.1 Transactions

By default, after each SQL statement is executed the changes are automatically committed to the database. Change with `connection.setAutoCommit(false)`;

### 14.2 More Features

Stored procedures, Changing buffer size, Connection pooling, Hibernate/JPA and other ORM tools.

## 15 Flyway

Store versions of database in metadata. Migrate from one version to the next using SQL or Java. Select database version in Gradle.

## 16 DBUnit

Used for DataBase testing. Assumes there is a test database. Runs tests over the database.

## 17 JDBC v JDBI v JPA v JDO

**JDBC:** Oriented towards database usage. **JDBI:** Oriented towards object usage. **JPA:** Focus on modelling. **JDO:** Focus on object modelling.