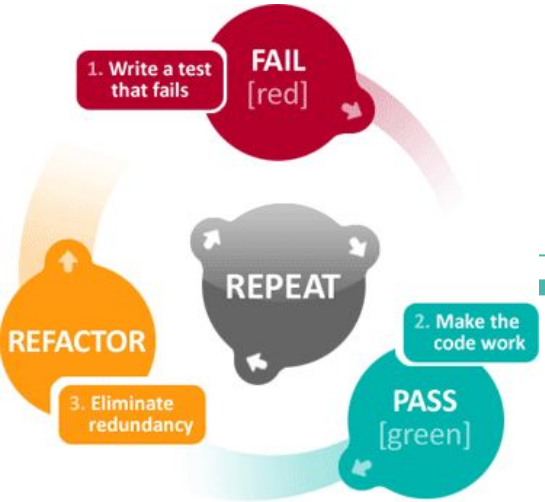
Test Driven Development



A practical course with interactive exercises

Agenda

1. An introduction to Agile Quality Processes and TDD
2. What is TDD and how it is supposed to work?
3. Growing a design with TDD and SOLID - theory
4. The “Shapes” TDD and SOLID Example
5. Code Coverage and Eclemma
6. Unit Testing vs Integration Testing: using Mockery
7. Mockito and Hamcrest, a simple 2-tier example
8. Maven and Unit Test Automation
9. Integration Testing with Spring Framework: an Introduction
10. Integration Testing with Soap UI
11. Quiz

Labs

The Shapes Demo: https://github.com/thimotyb/shapes

The Mockito Demo: https://github.com/thimotyb/mockito-tdd-demo

The Maven Demo: https://github.com/thimotyb/tdd-maven

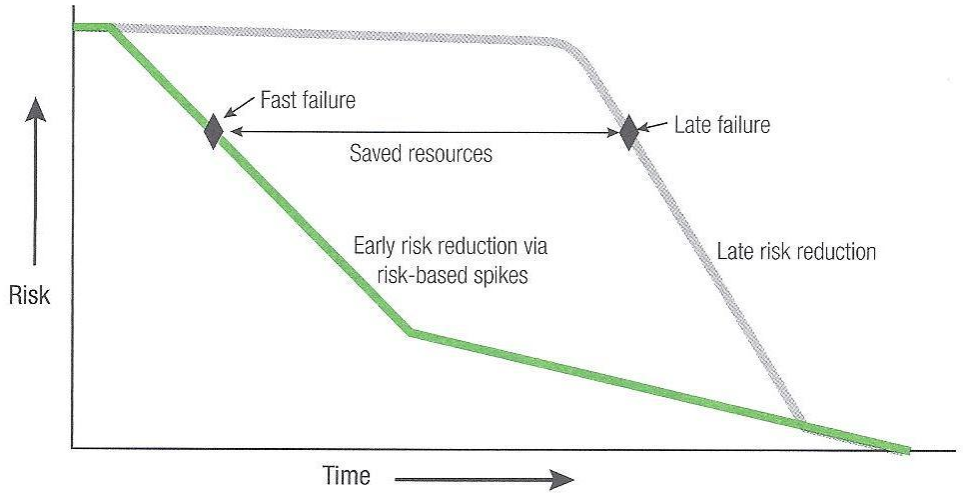
The Spring Demo:

Problem Detection and Resolution

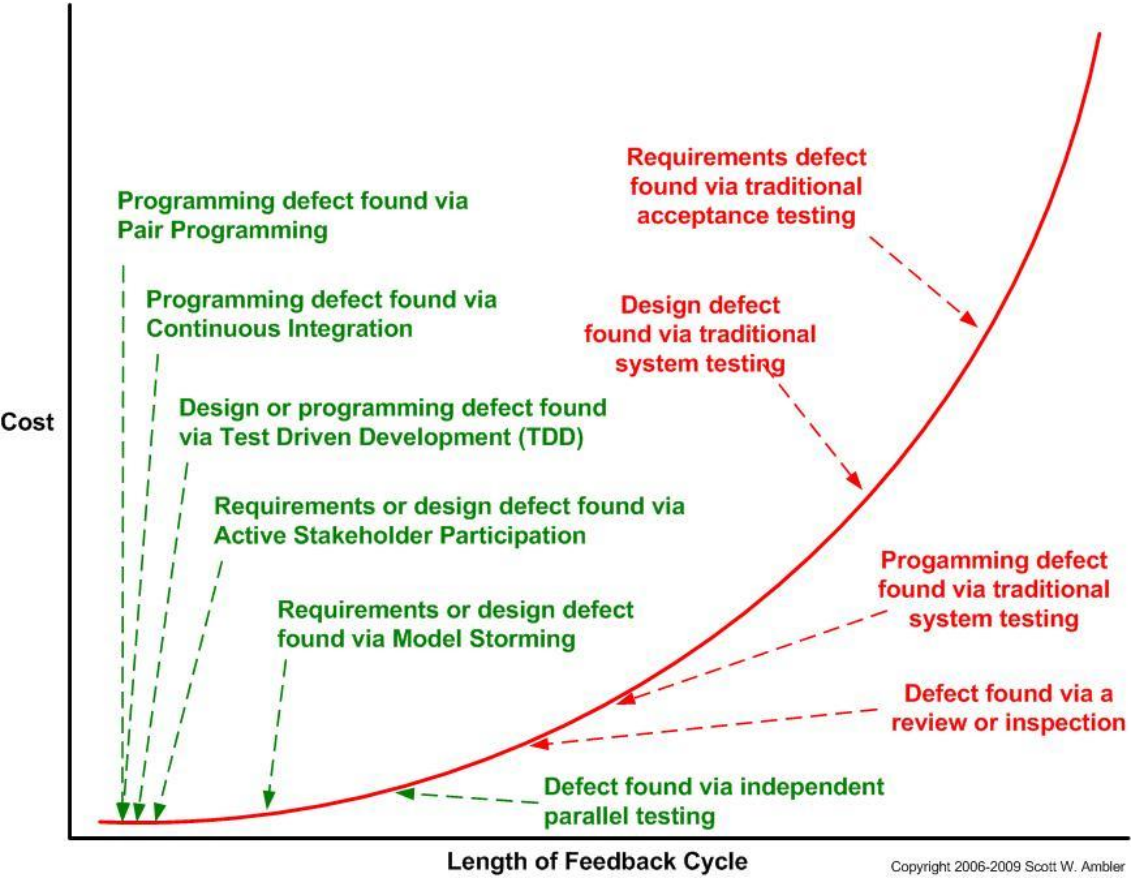
Fast Failure is a production mode in which we choose to use «Risk Spikes» to investigate early if an approach fails or is going to work.

The cost of failure is greatly reduced if it happens as early as possible.

The cost of fixing defects is less the sooner the defect is detected in the production cycle.



Cost of Change Curve

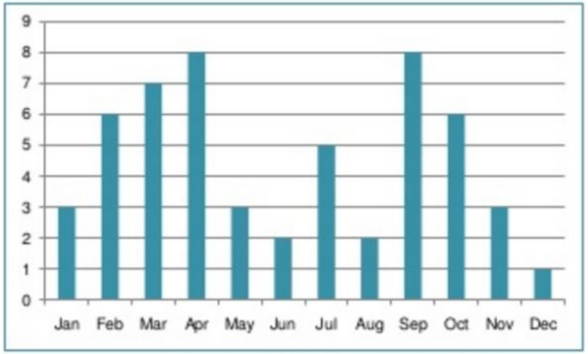


Detection: Escaped Defects

Defects that are not discovered during testing and validation, and make it to release. These are the most costly to fix.

Keeping track of Escaped Defect per release can help assessing the effectiveness of QA procedures we have in place

The graph is typically done Per Release, or Per Month (meaning the number of new escaped defects reported in each month across all releases)



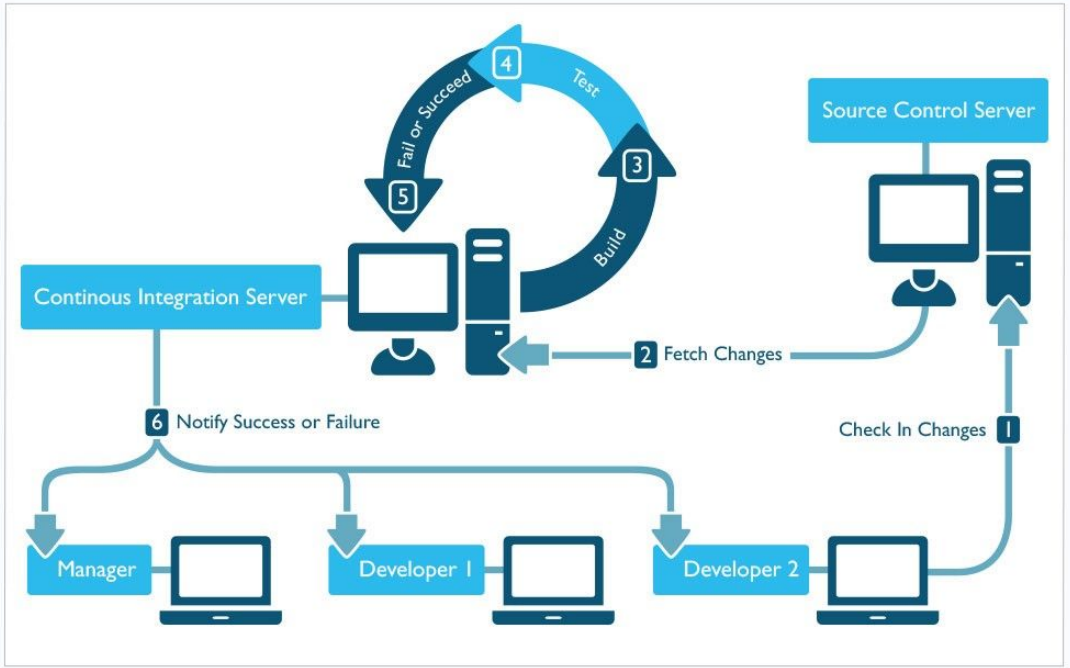
Escaped Defects over Time

Resolution: Continuous Integration

CI systems support continuous testing and integration, hence allowing to identify early integration problems.

- Source control (GIT)
  - Build Tools
  - Test Tools (Arquillian, Junit)
  - Scheduler or Trigger
  - Notifications

Daily Snapshot and Smoke Test – Early detection of problems

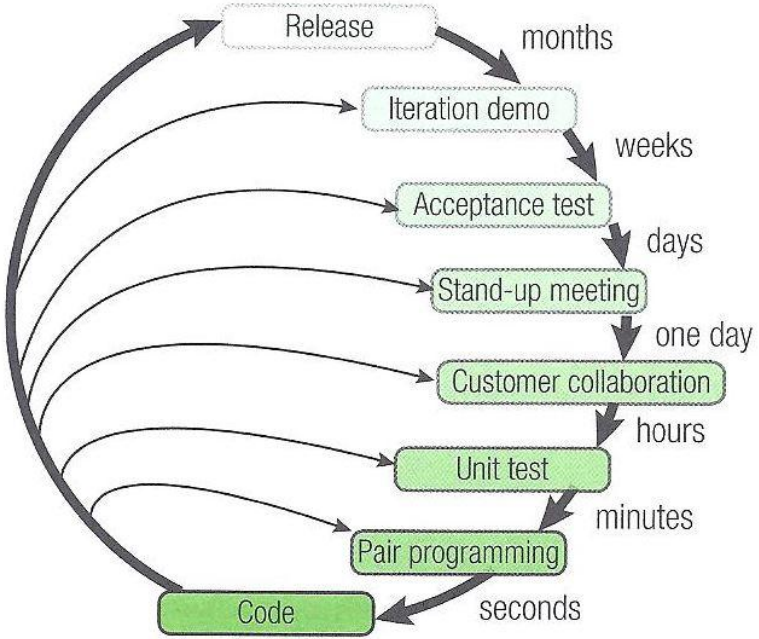


Resolution: Frequent Verification and Validation

Verification and Validation on Agile Projects can be practiced at all levels, from code writing (within seconds) to release (within months).

The idea is that validation and verification provide built-in quality during the creation on the software, and not after the software is made ready for a downstream QA department.

Verification and Validation can be performed also at design time, to check correctness of requirements (Harmony model)



Resolution: Test-Driven Development

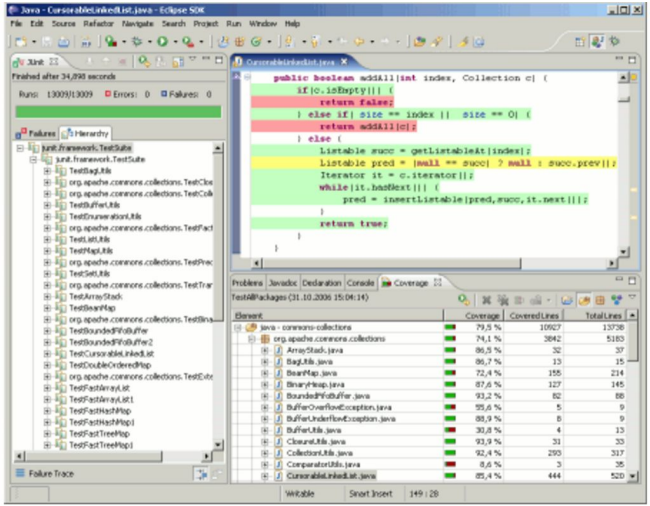
TDD is a crucial practice in Agile programming

Write the test first, then the code!!!

This is the only way to build a regression safety net to pair with Continuous Integration practices

Tipically Test Coverage Metrics (JaCoCo, EclEmma) can be paired as a Test Quality level to be required (e.g. 80% coverage on Controllers and DAOs level)

Fundamental for continuous refactory activities and incorporating changes in Agile

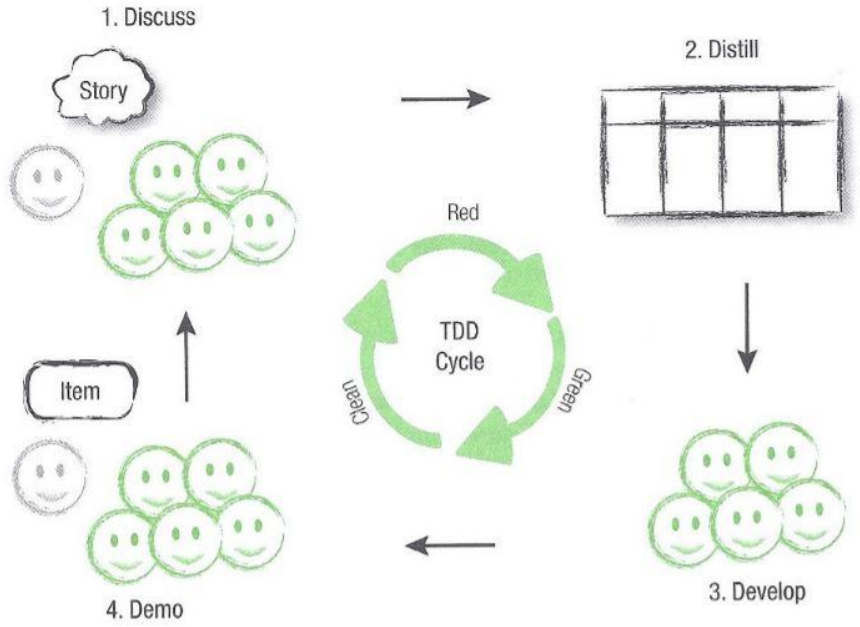


Resolution: Acceptance Test-Driven Development (ATDD)

ATDD moves the focus from testing the code to testing the implementation of the business requirement.

Test scenarios are created before implementation to reflect how the feature is expected to behave.

«FIT» or «FitNesse» are ATDD tools that can be used for ATDD



ATDD model developed by Elisabeth Hendrickson, Grigori Melnick, Brian Marick, and Jim Shore, based on a model by Jim Shore. Copyright © 2012 Quality Tree Software, Inc. Licensed Creative Commons Attribution.

A Brief Introduction to Unit Testing

The essence of TDD

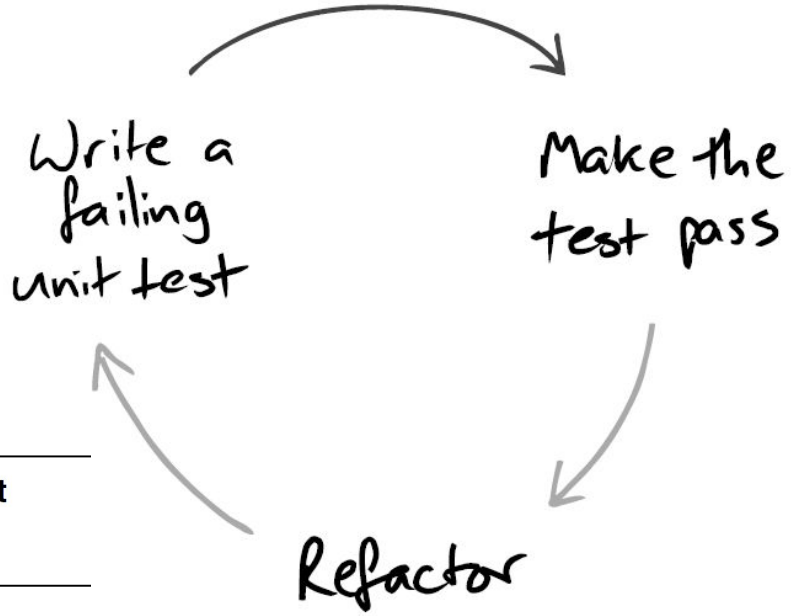
In Test-Driven Development (TDD) we write our tests before we write the code. Instead of just using testing to verify our work after it's done, TDD turns testing into a design activity.

The effort of writing a test first also gives us rapid feedback about the quality of our design ideas— making code accessible for testing often drives it towards being cleaner and more modular.

If we write tests all the way through the development process, we can build up a safety net of automated regression tests that give us the confidence to make changes.

The TDD Basic Cycle

The cycle at the heart of TDD is: write a test; write some code to get it working; refactor the code to be as simple an implementation of the tested features as possible. Repeat.



The Golden Rule of Test-Driven Development

Never write new functionality without a failing test.

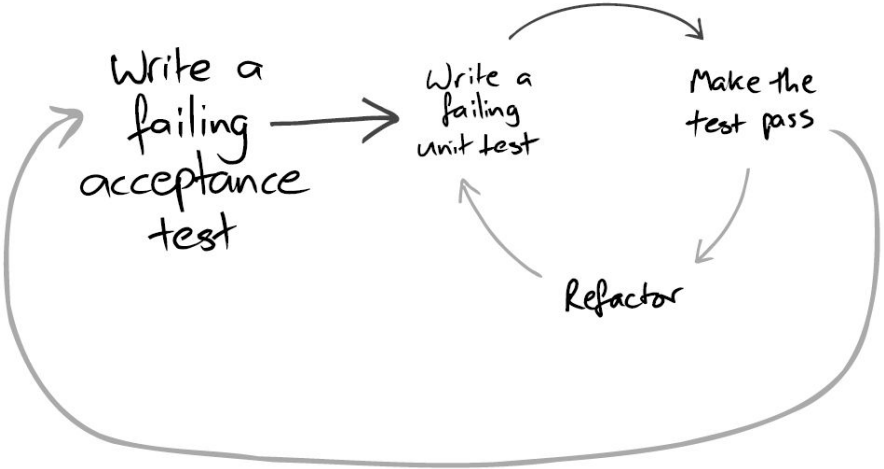
Refactoring

Refactoring is a disciplined technique where the programmer applies a series of transformations (or “refactorings”) that do not change the code’s behavior.

Each refactoring is small enough to be easy to understand and “safe”; for example, a programmer might pull a block of code into a helper method to make the original method shorter and easier to understand.

The programmer makes sure that the system is still working after each refactoring step, minimizing the risk of getting stranded by a change; in test-driven code, we can do that by running the tests.

Acceptance Tests



When working on a feature, we use its acceptance test to guide us as to whether we actually need the code we're about to write—we only write code that's directly relevant. Underneath the acceptance test, we follow the unit level test/implement/refactor cycle to develop the feature.

The outer test loop is a measure of demonstrable progress, and the growing suite of tests protects us against regression failures when we change the system.

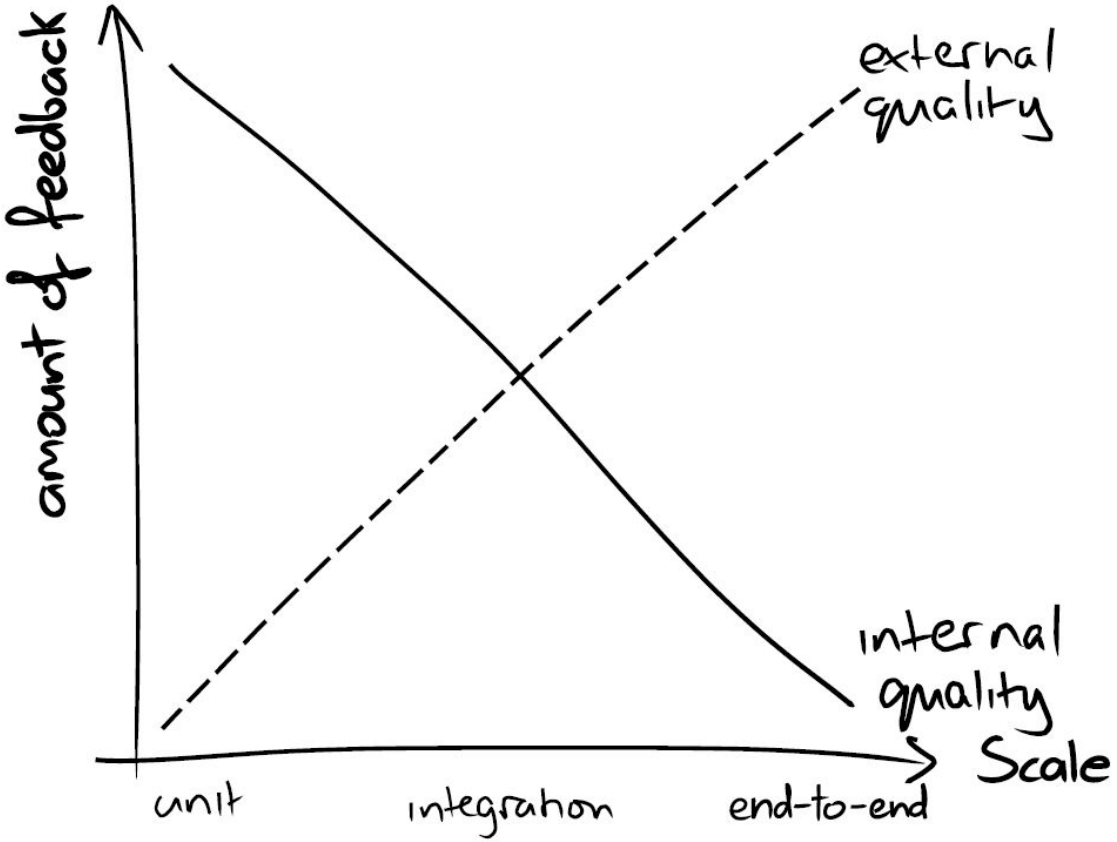
The inner loop supports the developers. The unit tests help us maintain the quality of the code and should pass soon after they've been written. Failing unit tests should never be committed to the source repository.

Levels of Testing

Acceptance: Does the whole system work?

Integration: Does our code work against code we can't change?

Unit: Do our objects do the right thing, are they convenient to work with?



TDD Tools

JUnit 4: Framework for Unit Testing

Mockito: Framework for Mocking and Spying Objects

Hamcrest: Advanced Matcher

Maven: Automation Tool for Continuous Integration

Let’s do it

- Let’s perform some TDD
  - The Shapes Demo https://github.com/thimotyb/shapes

- Let’s perform some TDD with Mockery
  - The Mockito Demo https://github.com/thimotyb/mockito-tdd-demo

SOLID principles

The Agile design

OOD - Object Oriented Design

Goals:

- **Scalability**: potential of the system to accomodate workload growth
- **Extendability**: potential to accomodate new components addition
- **Concurrency**: potential to let many people work on the system without having to know deeply other people work

Main conceptual components:

- **Object:** ADT (Abstract Data Type) with attributes and methods
- **Class:** representation of an object
- **Instance:** concrete implementation of a class

Design smells - the odors of rotting software

- **Rigidity**: the design is hard to change (changes can force to make changes to other parts of the system)
- **Fragility**: the design is easy to break (changes cause breaks in the parts that have no conceptual relationships with the part that changed)
- **Immobility**: the design is hard to reuse (it is hard to disentangle the system in reusable subcomponents)
- **Viscosity**: the design-preserving methods are harder to employ than the hacks
- **Needless complexity:** overdesign (some parts of the infrastructure adds no benefits)
- **Needless repetition:** some repeating structures can be unified in an abstraction
- **Opacity:** the system is hard to read and understand

OOD - Object Oriented Design

Main principles:

- **Inheritance:**
  - Put in common commonalities in order to avoid repetition (**DRY**)
  - Parent + child (inherits parent's attributes and methods and extends it) → **direction**
  - **Overriding**: to concretely implement abstract method of an interface
- **Polymorphism (πολύς + μορφή)**
  - use a variable of a superclass type to hold a reference to an object whose class is the superclass or any of its subclasses
- **Dynamic binding**
  - The JVM will decide at runtime which method implementation to invoke based on the class of the object (it automatically selects the objects in the most specific way)
  - Based on overriding

OOD - Cohesion vs Coupling

- **Cohesion**: *measure of whether responsibilities of a component form a meaningful unit*

All that is contained in a module references to a set of common features: eg. a module completely dedicated to I / O. This allows you to focus on the development of a specific function in just one of the components of the system architecture

- **Loose coupling:** *a change in a component has not to force a change in another*

The key to dominate the complexity of system: the modules must be sufficiently independent of each other, so that they can be independently manipulated, without having a profound knowledge of the others. Well decoupled modules have few short, simple, regular (and therefore always controllable) connections

E.g. as a driver, you would make certain assumptions about how the engine of your car works. The day you buy a new car with an engine that works differently, your previous assumptions would be incorrect.

OOD - Information Hiding

For the purposes of loose coupling, it is important that the content of each module refers only to the module itself, and if it changes, the change does not affect other modules. The modules must be **black boxes** that communicate with each other just in the ways they are allowed to, through the definition of their **interface**

→ **Information hiding:** the module has to be divided into two parts:

- the *interface* is fully visible on the outside and represents the 'border' of the module and the 'rules' to communicate with it
- the *implementation* constitutes the true 'internal mechanism' of the module and must be kept hidden

SOLID principles: the ingredients to enlightment

- SRP: Single Responsibility Principle
- OCP: Open Closed Principle
- LSP: Liskov Substitution Principle
- ISP: Interface Segregation Principle
- DIP: Dependency Inversion Principle

The goal is to be able to deal with changes:

- **Flexibility**
- **Reusability**
- **Maintainability**

Remember...

Overconformance to principles can introduce needless complexity!

Ockham's razor: « Pluralitas non est ponenda sine necessitate. »

Design is a process, not an event!

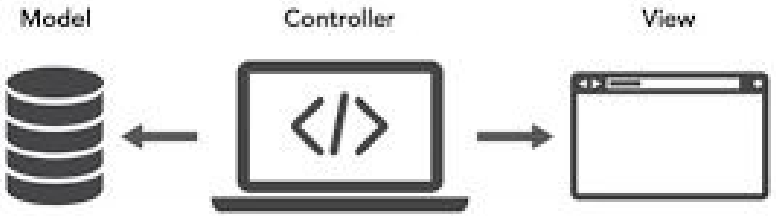
SRP - Single Responsibility Principle

A class should have only one reason to change

Each responsibility is an axis of change → if a class has more responsibilities, they can be coupled and changes to one of them can affect also the other

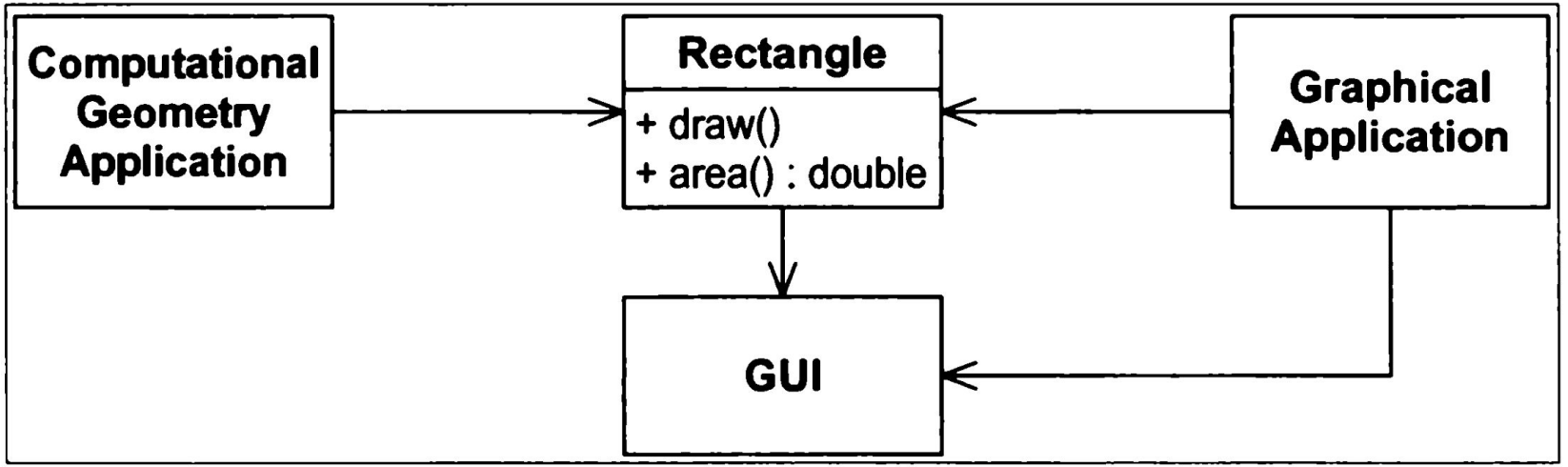
→ **High cohesion, loose coupling**

EX: business rules (controller) and persistence controls (model) should never be mixed! Use the **MVC pattern** instead!



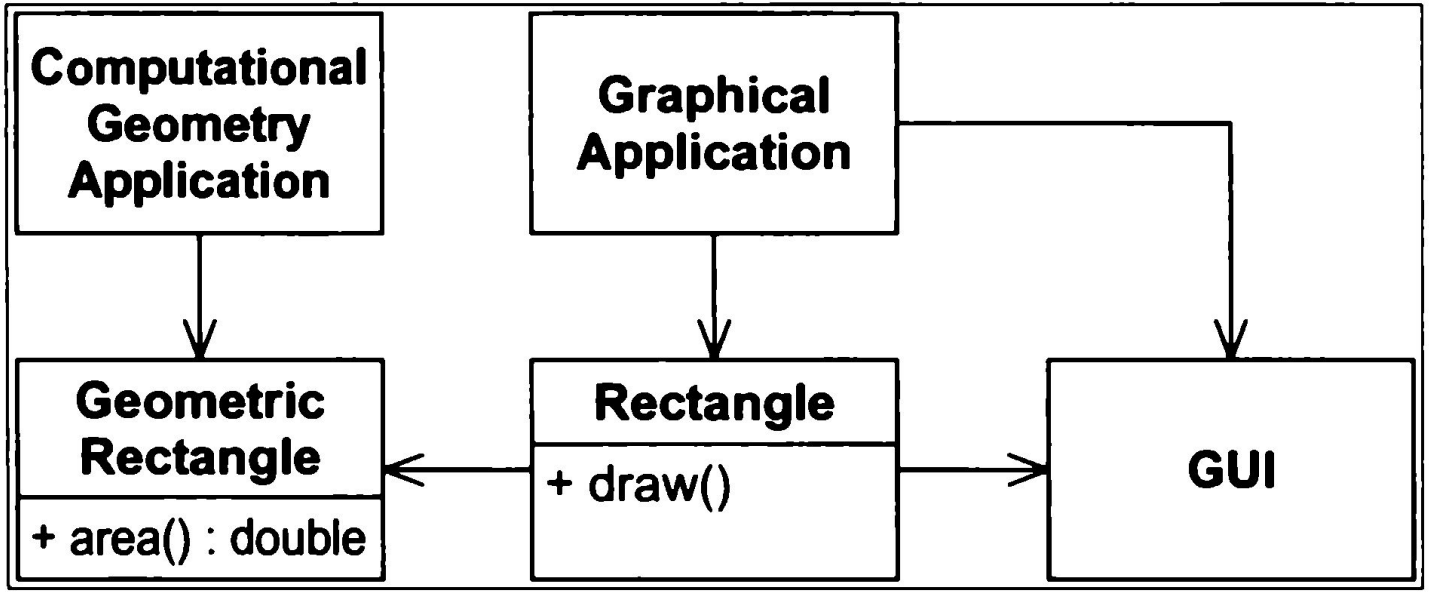
SRP - Single Responsibility Principle

EX: more than one responsibility



SRP - Single Responsibility Principle

EX: separated responsibilities: math model vs graph model of the rectangle



OCP - Open Closed Principle

A class should be open to extension and closed to modification

- Changes should create code extension and not code modification
- Versus Rigidity (if/else, switch)

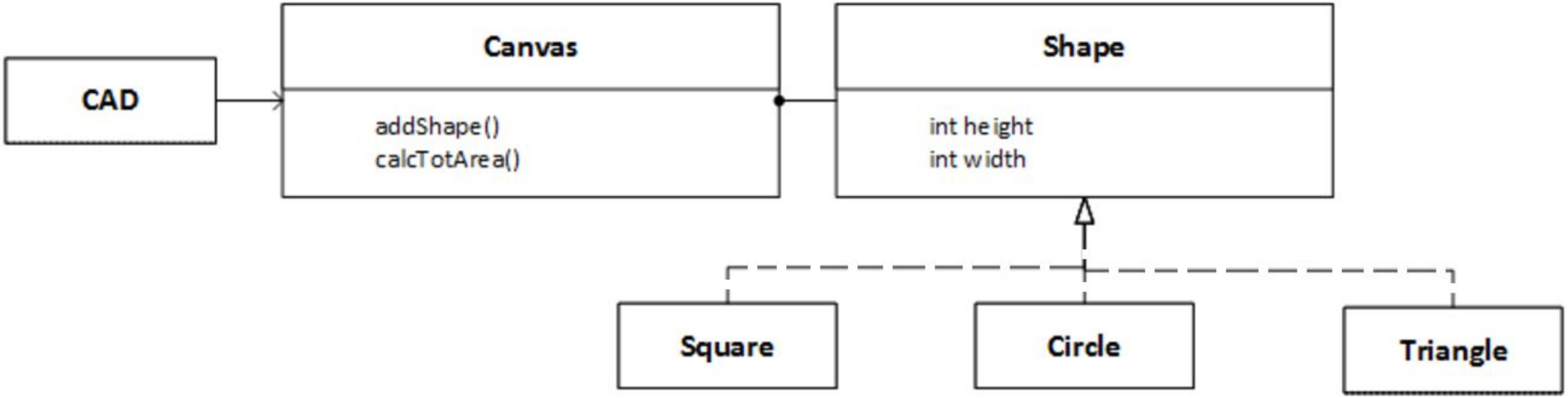
→ **Abstraction**

→ **Polymorphism + Information hiding + dynamic binding**

→ **Cohesion + loose coupling**

OCP - Open Closed Principle

EX: drawing program (CAD)



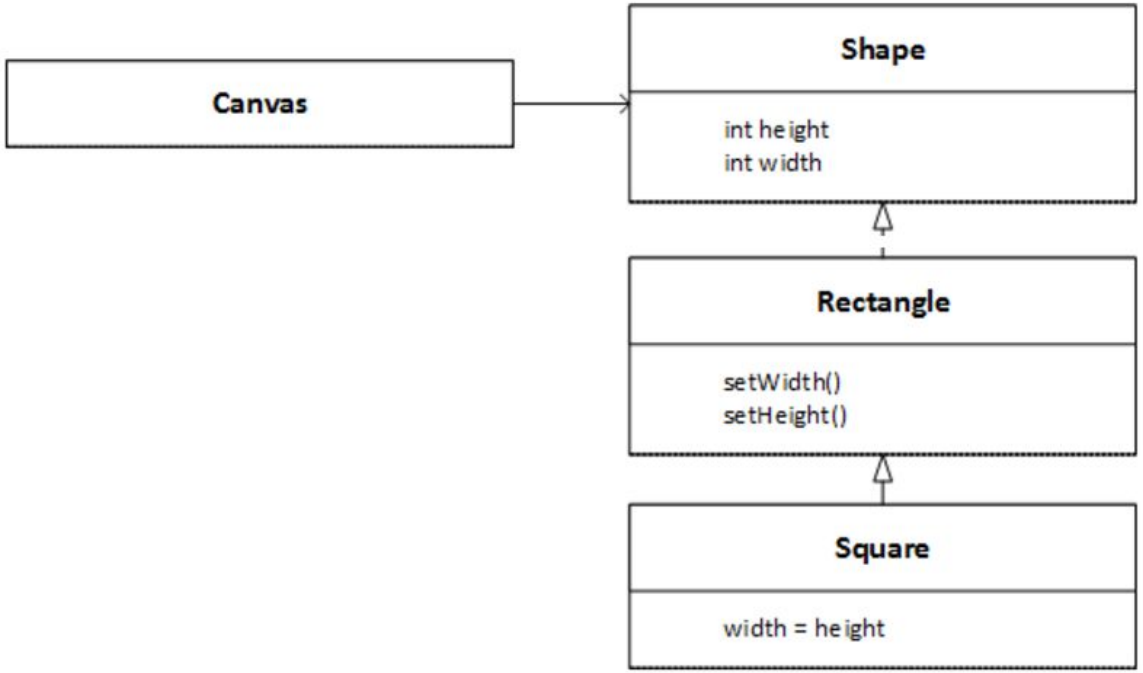
LSP - Liskov Substitution Principle

Subtypes must be substitutable by their base types

- **Abstraction → Polymorphism + Information hiding + Dynamic binding:** can lead to create hierarchies that do not conform to OCP

LSP - Liskov Substitution Principle

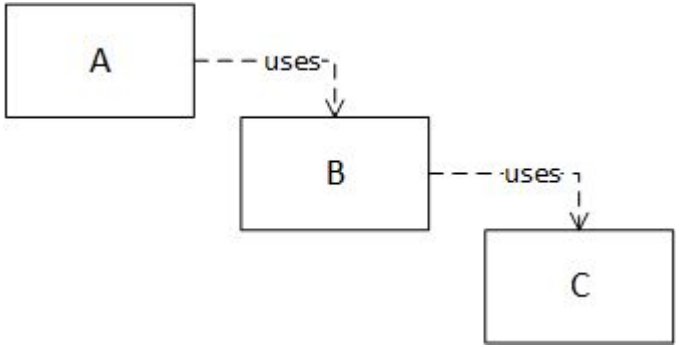
EX: BAD DESIGN



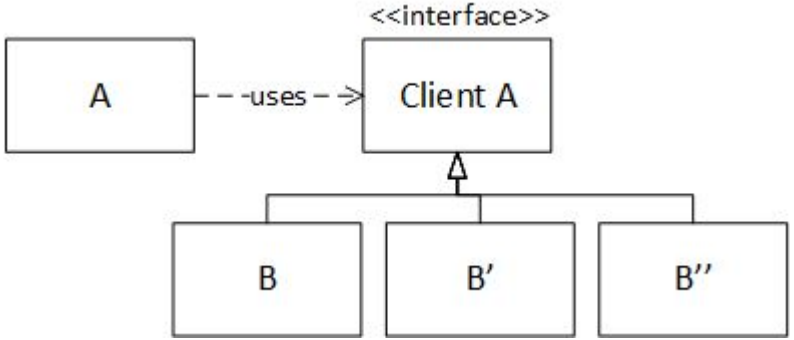
DIP - Dependency Inversion Principle

- *High level modules should not depend on low level modules. Both should depend on abstractions*
- *Abstractions should not depend on details. Details should depend on abstractions*

→ *The Hollywood principle: "Don't call us, we will call you!"*



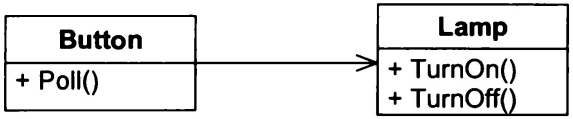
Top-down dependency: changes in C affect changes in B and in A



Bottom-up dependency: dependency levels are decoupled with an interface which inverts the dependency direction

DIP - Dependency Inversion Principle

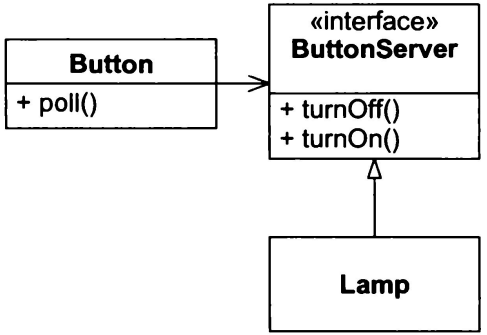
EX: bad



- **TOP-DOWN** dependency Button-Lamp: changes to Lamp imply changes to Button
- Button is **not reusable** (es. by a Motor)

→ I detect the services that A uses from B and put them in an interface. Then I can implement it in many ways

EX: good



- Inverted **BOTTOM-UP** dependency: Lamp now “depends”, it is not “depended on”

ISP - Interface Segregation Principle

Clients should not be forced to depend on methods they never use

- There should not be “fat” (non cohesive) interfaces → interfaces must be broken into groups of methods, s.t. each of them serves a different set of clients
- Leads to inadvertent coupling between classes. Coupling is the bane of reusability

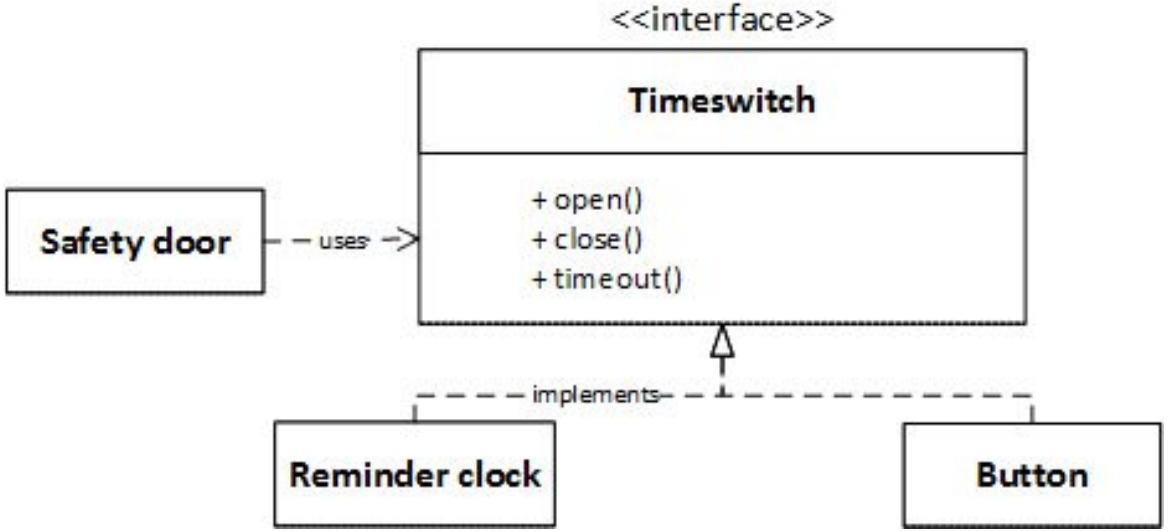
→ **high cohesion, loose coupling, information hiding**

… As in SRP, also interfaces have to implement a single service

ISP - Interface Segregation Principle

EX: design a switch which is opening on a specific timing

BAD DESIGN:

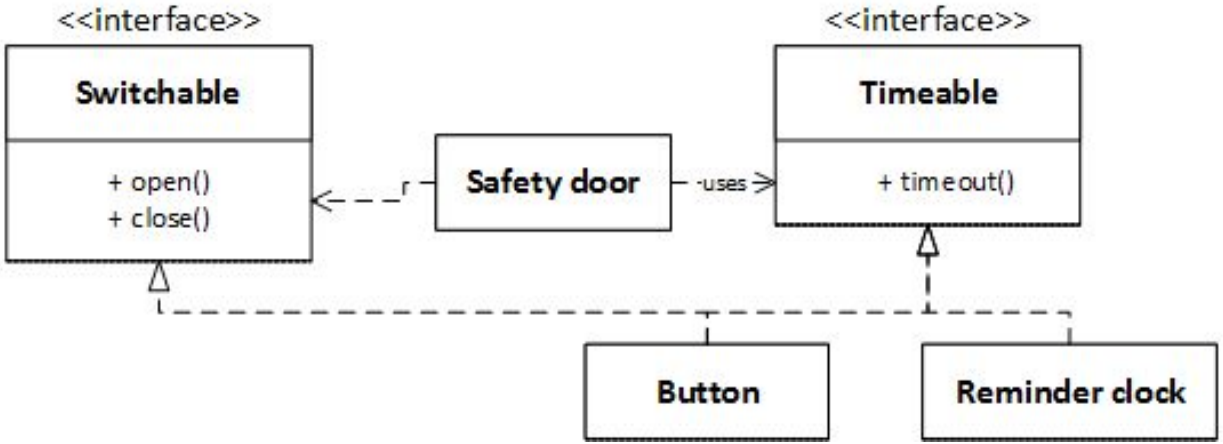


...I don't want to open/close the clock!!

ISP - Interface Segregation Principle

EX: design a switch which is opening on a specific timing

GOOD DESIGN:



I have split the interface in two, now the clock is no more “switchable”

SOLID principles

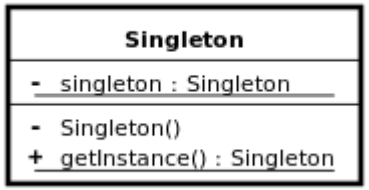
The design patterns

The design patterns: the recipes to enlightment

Standard solution	**Problem**
Singleton	Single instance of an object
Strategy	A service interface is implemented in many ways → DIP
Template	Common behaviour/algorythm that slightly changes according to the specific feature → overriding + DRY → DIP
Null object	Create a neutral object, to avoid to check “if x == null...”
Factory	Create different object implementations of the same types
Facade	Collect many functionalities or layers of services into a single class, to have a easier access
Observer/Observable	Automatic asyncronous decoupling: callback systems where if the observables change the state, the observer is automatically advised

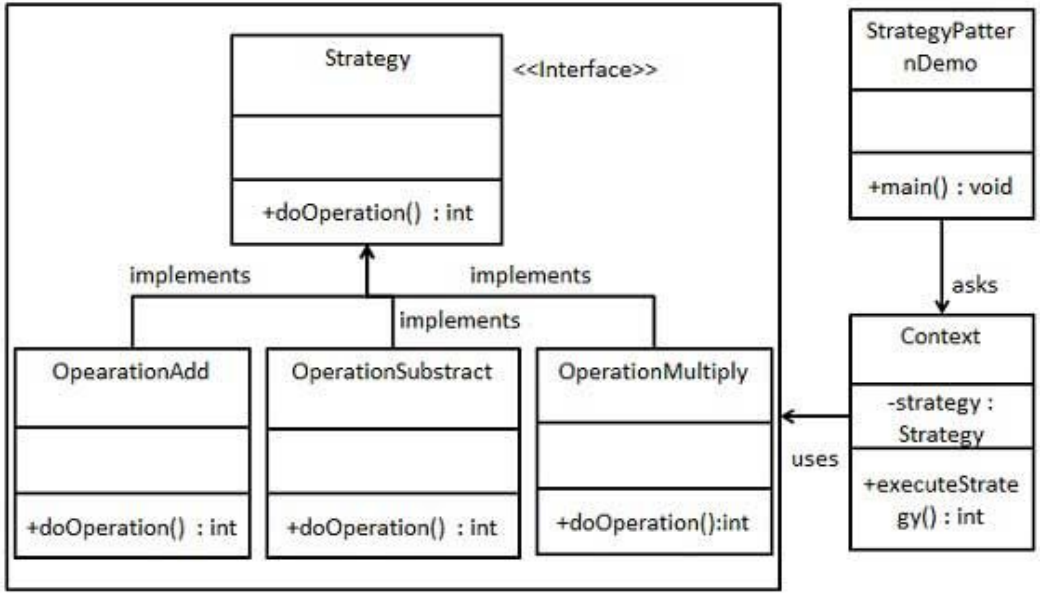
Singleton pattern

- **Single instance** of an object
- **Private constructor**
- **Public static getter** that returns always the same instance of the class



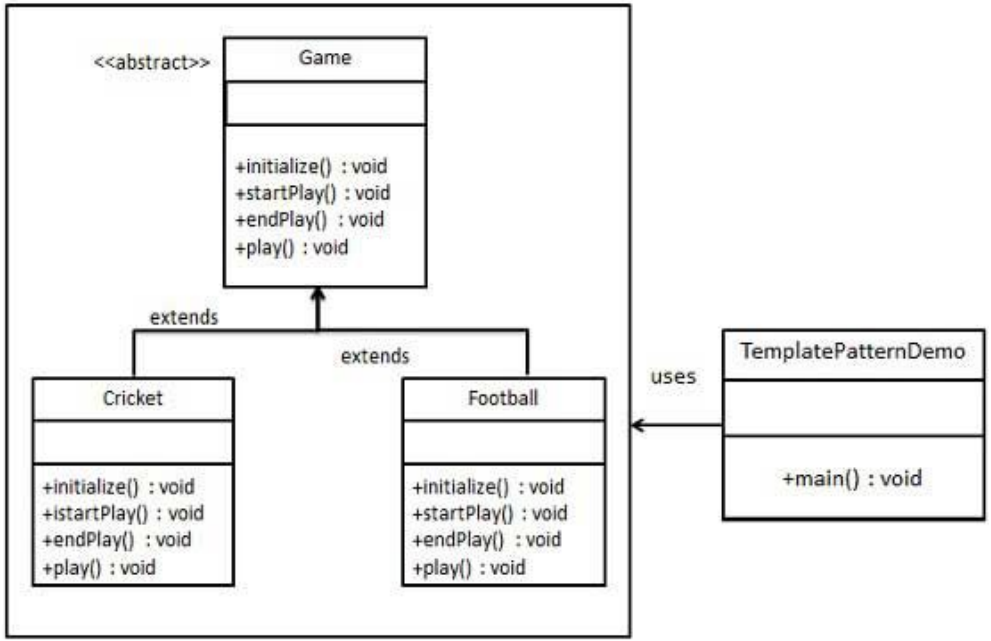

```java
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Strategy pattern



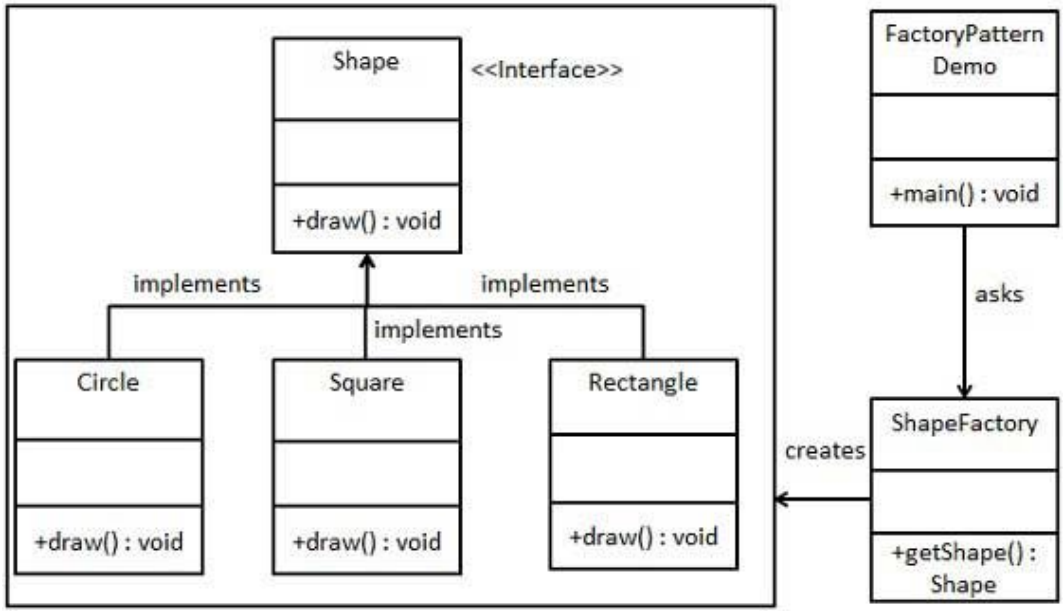
- Extraction of the services in the bottom classes and implementation of the interface → **DIP**
- The client does not have to make assumptions about what strategy (sub-class) is instantiated → **OCP + information hiding + dynamic binding**
- Keyword: **<<implements>>**

Template pattern



- An **abstract** class is implemented in specific ways → **DRY**
- The abstract behaviours are “overwritten” by the sub-classes behaviours → **overriding**
- In the parent abstract class there is the template of the algorythm/behaviour

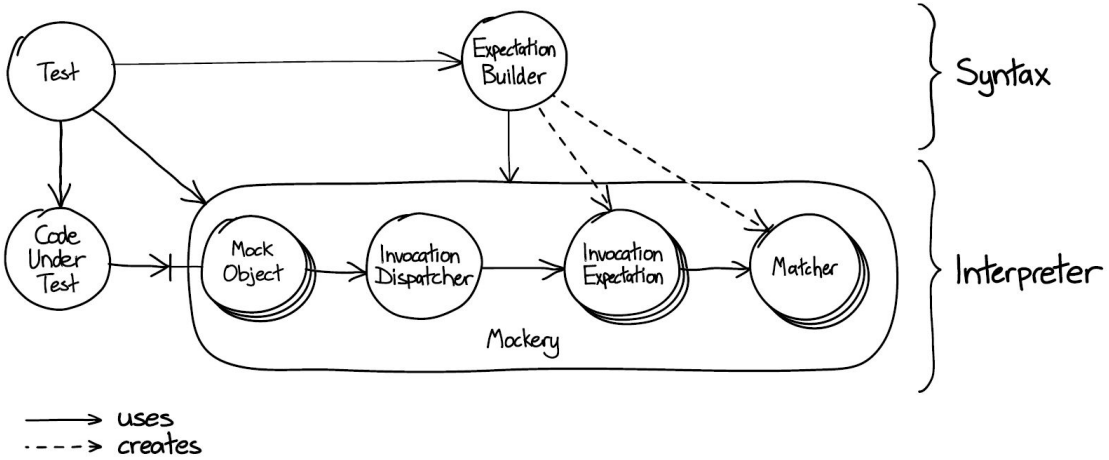
Factory pattern



- Creation of objects according to their type → **SRP + OCP**

Advanced TDD Concepts

Mockery



Mock objects are simulated objects that mimic the behavior of real objects in controlled ways. A programmer typically creates a mock object to test the behavior of some other object. Mocking an object is useful if:

- the object supplies non-deterministic results (e.g. the current time or the current temperature);
- it has states that are difficult to create or reproduce (e.g. a network error);
- it is slow (e.g. a complete database, which would have to be initialized before the test);
- it does not yet exist or may change behavior;
- it would have to include information and methods exclusively for testing purposes (and not for its actual task).

Mock objects meet the interface requirements of, and stand in for, more complex real ones; thus they allow programmers to write and unit-test functionality in one area without calling complex underlying or collaborating classes.

The FIRST Principles

FAST, ISOLATED/INDEPENDENT, REPEATABLE, SELF-VALIDATING and THOROUGH/TIMELY

Fast

- A developer should not hesitate to run the tests as they are slow.
- All of these including setup, the actual test and tear down should execute really fast (milliseconds) as you may have thousands of tests in your entire project.

The FIRST Principle

Isolated/Independent

- A test method should do the 3 As => Arrange, Act, Assert
- Arrange: The data used in a test should not depend on the environment in which the test is running. All the data needed for a test should be arranged as part of the test.
- Act: Invoke the actual method under test.
- Assert: A test method should test for a single logical outcome, implying that typically there
- should be only a single logical assert. A logical assert could have multiple physical asserts as
- long as all the asserts test the state of a single object. In a few cases, an action can update
- multiple objects.
- Avoid doing asserts in the Arrange part, let it throw exceptions and your test will still fail.
- No order-of-run dependency. They should pass or fail the same way in suite or when run individually.
- Do not do any more actions after the assert statement(s), preferably single logical assert.

The FIRST Principle

Repeatable

- A test method should NOT depend on any data in the environment/instance in which it is running.
- Deterministic results - should yield the same results every time and at every location where they run.
- No dependency on date/time or random functions output.
- Each test should setup or arrange its own data.
- What if a set of tests need some common data? Use Data Helper classes that can setup this data for re-usability.

Self-Validating

- No manual inspection required to check whether the test has passed or failed.

The FIRST Principle

Thorough and Timely

- Should cover every use case scenario and NOT just aim for 100% coverage.
- Should try to aim for Test Driven Development (TDD) so that code does not need re-factoring later.

Test Doubles (Gerard Meszaros)

Test Double is a generic term for any case where you replace a production object for testing purposes. There are various kinds of double:

- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
- **Mocks** are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

Patterns of TDD (Kent Beck)

Test n.: How do you test your software? Write an automated test.

Isolated Test: How should the running of tests affect each other? Not at all.

Test List: What should you test? Before you begin, write a list of all the tests you know you will have to write.

Test-First: When should you write your tests? Before you write the code that is to be tested.

Assert First: When should you write the asserts? Try writing them first.

Test Data: What data do you use for test-first tests? Use data that makes the tests easy to read and follow.

Evident Data: How do you represent the intent of the data? Include expected and actual results in the test itself, and try to make their relationship apparent.

TDD Implementation Strategies

Fake It ('Til You Make It)

What is your first implementation once you have a broken test? Return a constant. Once you have the test running, gradually transform the constant into an expression using variables.

Triangulate

How do you most conservatively drive abstraction with tests? Only abstract when you have two or more examples.

```
public void testSum() {
    assertEquals(4, plus(2, 2));
    assertEquals(7, plus(3,4));
}
```

TDD Implementation Strategies

Obvious Implementation

How do you implement simple operations? Just implement them.

One to Many

How do you implement an operation that works with collections of objects? Implement it without the collections first, then make it work with collections.

TDD Implementation Strategies

Regression Test

What's the first thing you do when a defect is reported? Write the smallest possible test that fails, and that once it runs, the defect will be repaired.

Explanation Test

How do you spread the use of automated testing? Ask for and give explanations in terms of tests.

TDD Implementation Strategies

Child Test

How do you get a test case running that turns out to be too big? Write a smaller test case that represents the broken part of the bigger test case. Get the smaller test case running. Reintroduce the larger test case.

Clean Check-in

How do you leave a programming session when you’re programming in a team? Leave all the tests running.

Using TDD Automation Tools

Maven

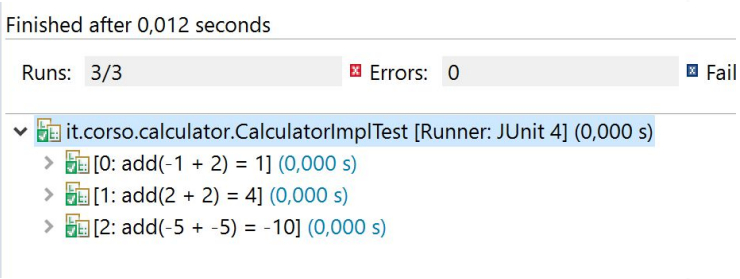
```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.15</version>
  <configuration>
      <!-- Sets the VM argument line used when unit tests are run. -->
      <argLine>${surefireArgLine}</argLine>
      <!-- Skips unit tests if the value of skip.unit.tests property is true -->
      <skipTests>${skip.unit.tests}</skipTests>
      <!-- Excludes integration tests when unit tests are run. -->
      <excludes>
          <exclude>**/IT*.java</exclude>
      </excludes>
  </configuration>
/plugin>
```

Surefire Plugin -

Execute tests in test folder during build

Parametrized Tests and Fixtures

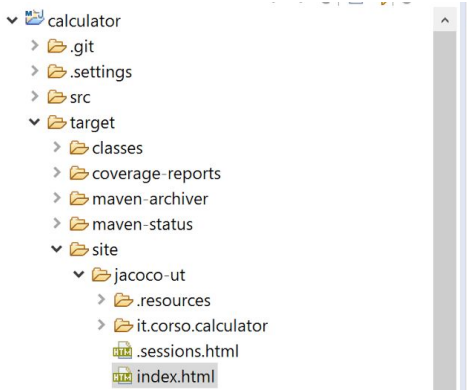
```java
@RunWith(Parameterized.class)
public class CalculatorImplTest {

    /**
     * Parametrized unit tests
     * 1) Create Fields
     * 2) Create Constructor for Injection of Parameters
     * 3) Create static method to load a Fixture and annotate as @Parameters
     * 4) Pass parameters to test method
     * 5) Declare the test runner as Parametrized
     */
    private int num1, num2, expected;

    public CalculatorImplTest(int num1, int num2, int expected) {
        this.num1 = num1;
        this.num2 = num2;
        this.expected = expected;
    }

    @Parameters( name = "{index}: add({0} + {1}) = {2}" )
    public static Collection<Integer[]> getFixtureData() {

        return Arrays.asList(new Integer[][] {
            {-1, 2, 1},
            {2, 2, 4},
            {-5, -5, -10}
        });

    }
```

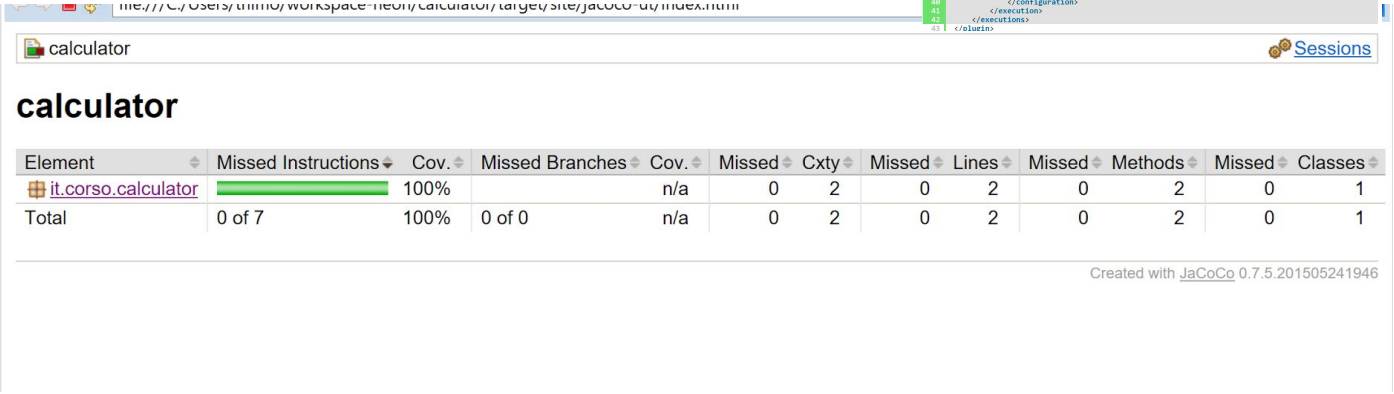


JaCoCo / Code Coverage

JaCoCo plugin and reports with Maven

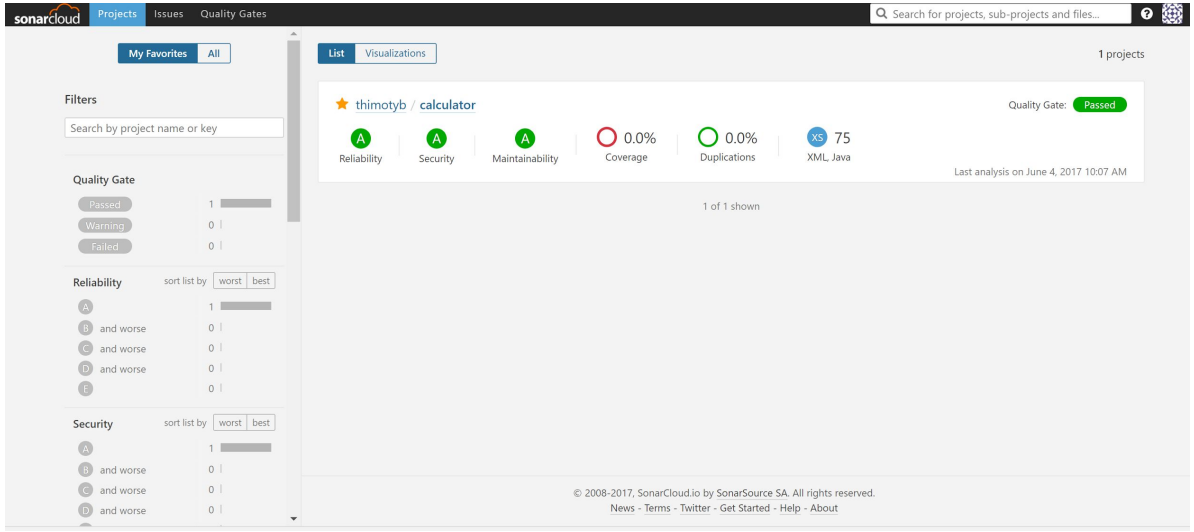
```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.7.5.201505241946</version>
    <executions>
        <!--
            Prepares the property pointing to the JaCoCo runtime agent which
            is passed as VM argument when Maven the Surefire plugin is executed.
        -->
        <execution>
            <id>pre-unit-test</id>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
            <configuration>
                <!-- Sets the path to the file which contains the execution data. -->
                <destFile>${project.build.directory}/coverage-reports/jacoco-ut.exec</destFile>
                <!--
                    Sets the name of the property containing the settings
                    for JaCoCo runtime agent.
                -->
                <propertyName>surefireArgLine</propertyName>
            </configuration>
        </execution>
        <!--
            Ensures that the code coverage report for unit tests is created after
            unit tests have been run.
        -->
        <execution>
            <id>post-unit-test</id>
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
            <configuration>
                <!-- Sets the path to the file which contains the execution data. -->
                <dataFile>${project.build.directory}/coverage-reports/jacoco-ut.exec</dataFile>
                <!-- Sets the output directory for the code coverage report. -->
                <outputDirectory>${project.reporting.outputDirectory}/jacoco-ut</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```





https://www.petrikainulainen.net/programming/maven/creating-code-coverage-reports-for-unit-and-integration-tests-with-the-jacoco-maven-plugin/

Sonarcloud Static Analysis



c:\apache-maven-3.3.9\bin\mvn clean org.jacoco:jacoco-maven-plugin:prepare-agent package sonar:sonar -Dsonar.host.url=https://sonarcloud.io -Dsonar.organization=thimotyb-github -Dsonar.login=144a6cee9564aa482a291d5f1aa04d297b1b0b68