# Testable code

Jean-Luc Delarbre

# Key rules for reusable code

- **Every implementation should derive from an interface**
  - Find the right abstractions
- **Factories (or builder)**
  - Only them are allowed to create object instances (use new)
  - Inject dependencies to object
  - Do not use static instances
- **(Immutability)**

# How to write testable code ?

- **Find a set of practicable rules to right testable code**
- **Help developers to easily write tests**

# Basics about test structure

- **Basic code architecture**
  - Setup
  - Test SUT (Software under test)
  - Verify
- **Styles of test**
  - London style (interaction style)
  - Chicago style (state based style)
- **Do not alter production code for tests by**
  - Adding specific methods for tests
  - Change visibility to public for some elements

# Rule 1: Constructor does not real work

- **Coupled design**
  - Unable to change collaborators
- **Violation SRP (Object has many responsabilities)**
  - Business purpose + application wiring
- **Difficult to test**
  - Force you to setup the whole environment of production code

# Violation of rule 1

- **Using “new” keyword**
  - Except for data
- **Using “static” keyword**
- **Logic (loop, if…) in constructor**

# Rule 2: Do not dig into collaborators

- **Violation principle of least surprise**
  - Lying API: seems to ask an object for one purpose but used for another. Intent not clear
- **Brittle tests**
  - Raise your number of dependencies ⇨ more reasons to change. You depends on the object you really need + another intermediate object (coupled with its changes)
- **Code hard to read and test**
  - Force to construct a long chain of elements
  - Hard to see what really matter / obscure test

# Violation of rule 2

- **Do not use passed object for themselves but only to access other objects**
- **Law of Demeter violation: chain of getter**
- **Suspicious objects: context, environment...**

# Rule 3: Avoid global state and singletons

- **Interact with people you are not suppose to know**
  - Statically call resources which as never been set as dependencies (through constructor or methods) ⇨ Lying API
- **Spooky action at distance, unexpected interaction**
  - Force to read implementation to know what’s happened
- **Untractable code with global state**
  - Anything interact with everything
- **Interdependent tests**
  - Force tear down between each test
  - Singleton almost untestable

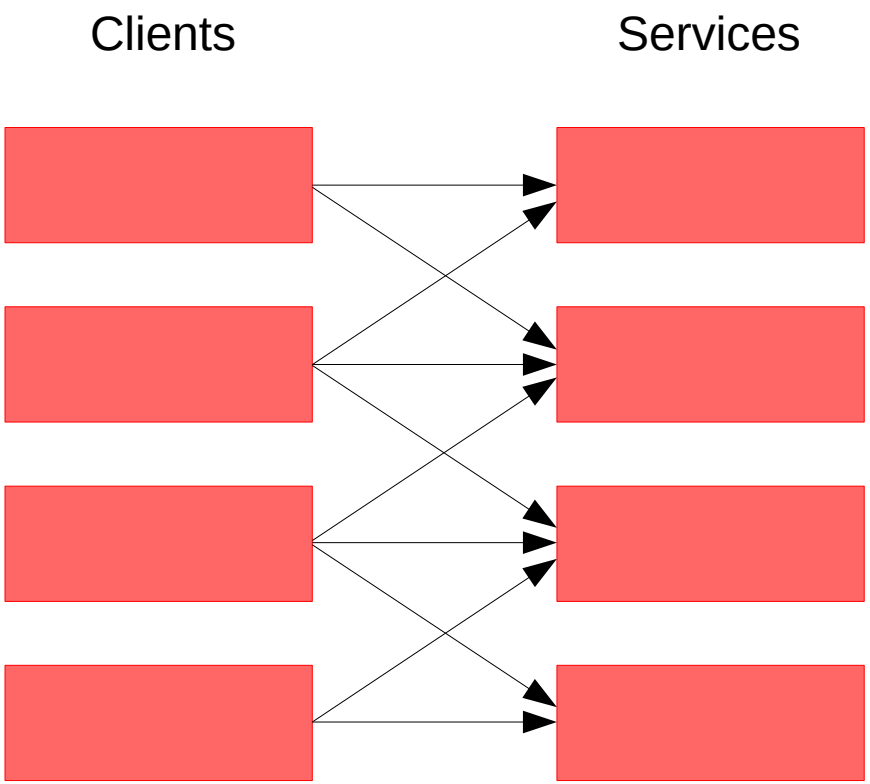
# Violation of rule 3

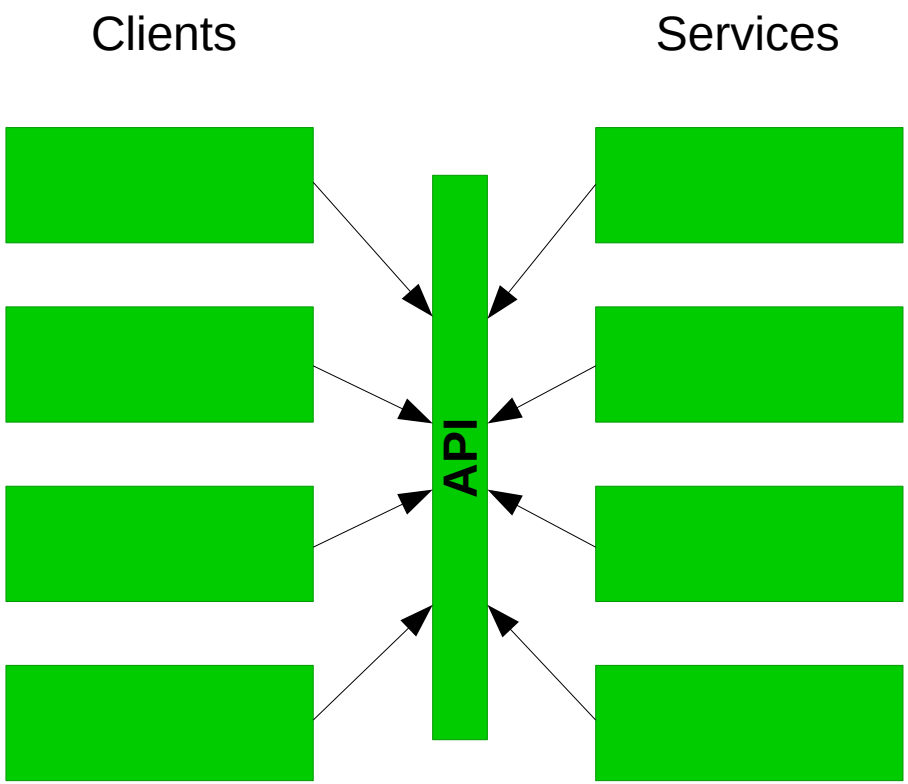
- Depend on global state
- Usage of singleton (as in GOF)
- Usage of static fields
- Usage of static methods (not private inner methods)

# Reusable and testable code: same rules / Code-Test complementarity

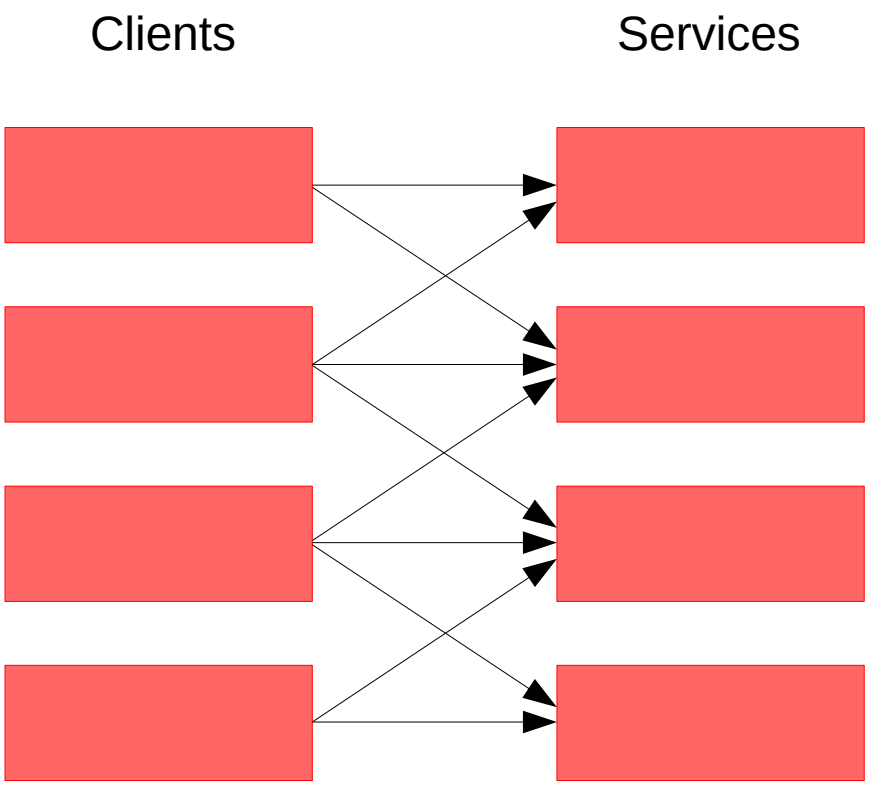
- **Tests is first case of code reuse**
  - At least one case of code reuse for tests
  - Ensure reusable and well design code
- **Test as documentation: explain usage and purpose of a software component**
  - Avoid obscure test
  - Clarify code API (how to use it)
  - Expose use cases (possible to express them in functional or acceptance tests)

# Design considerations

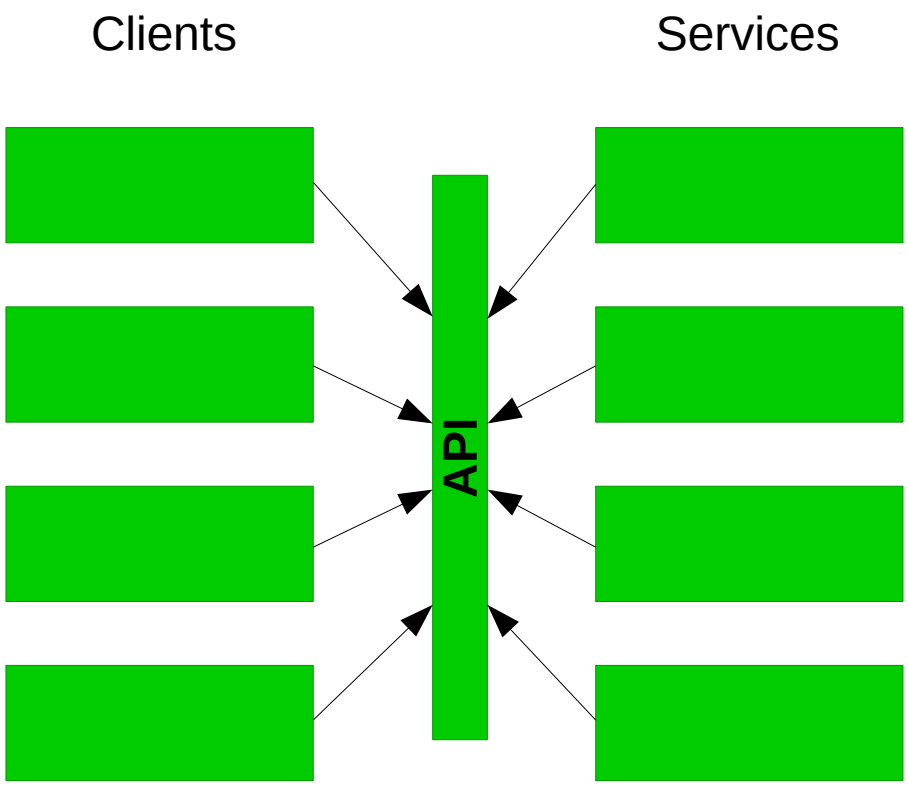


# Design considerations

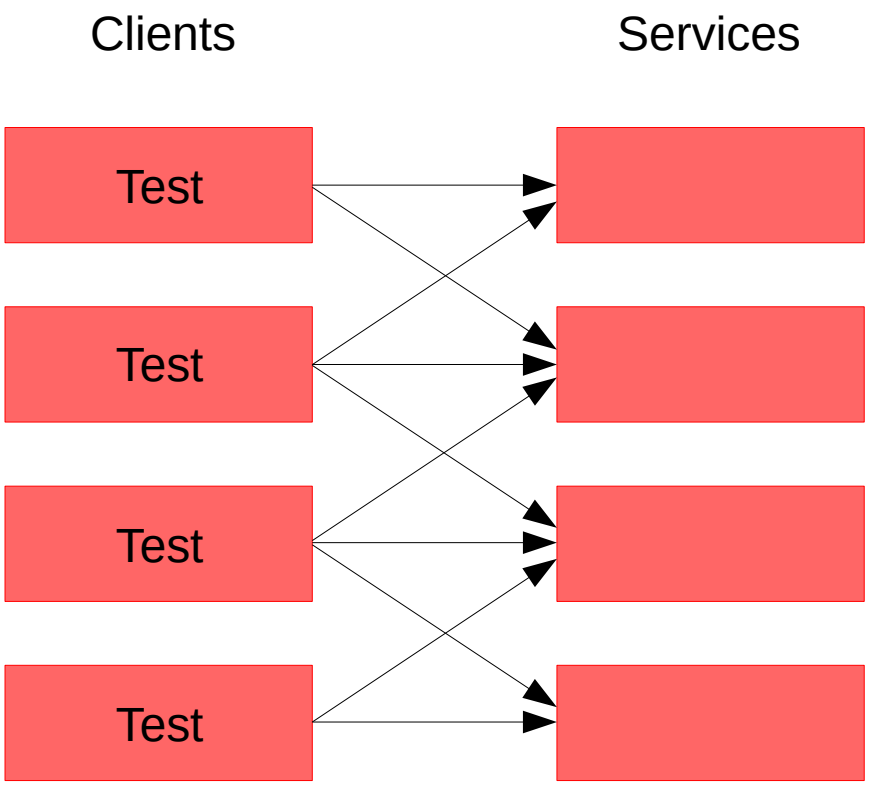



High Coupling

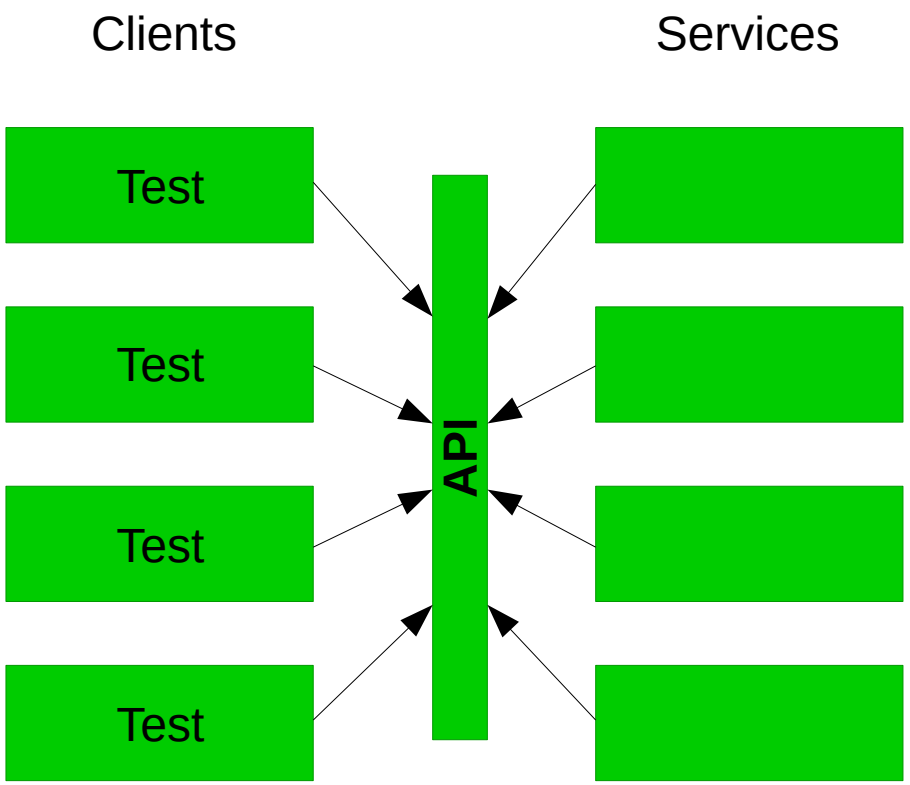



Low Coupling

# Design considerations




High Coupling




Low Coupling

# Application and tests design

- **Test packages should not be an image of implementation packages**
  - Implementation class ⇔ test class (one to one correspondence)
    - Fragile tests, change on every code change
    - Production code made rigid by tests (costs of changes are too expensive or worst unpredictable)
- **Need of an architectural vision**
  - Brittle tests: reveal importance of strategic decision
    - API definition (information hiding – Parnas 1971)
    - Tests are part of the system also suffer/show design flaw
  - Test a class (or set of class) against an API that has a business value.

# Tests purpose

- **Write better code with better architecture**
- **Also allow to prevent regression**

# Bugs taxonomy

| | Probability of occurrence | Bug detection | Fix Bug |
|---|---|---|---|
| **Logic** | **HIGH** | **DIFFICULT** | **DIFFICULT** |
| **Wiring** | **MEDIUM** | **MEDIUM** | **MEDIUM** |
| **HMI** | **MEDIUM** | **EASY** | **EASY** |

# Tests taxonomy

| | | Unit | Integration | Functional |
|---|---|---|---|---|
| | | **A separate part: A method, a class or a limited number of related classes** | **Many parts working together** | **End to end testing: Test a complete functionality** |
| **Isolation from Outside (DB...)** | **Mock, stub external components** | Test logic, run fast | Test logic and wiring run pretty fast | Test the whole stack, could run rather fast |
| | **Used real external component** | × | Test logic and wiring, run slowly | Test the whole stack, run slowly |

# Test code quality

- **No copy paste / duplication**
  - Unmaintainable tests
- **F.I.R.S.T Principles of Unit Testing**
  - Fast
  - Isolated / independent
  - Repeatable
  - Self-Validating
  - Thorough

**Bad tests quality:**

⇨ **Throw away test**

⇨ **Software rots**

# Facing complexity

| Coverage | Check |
|---|---|
| Function | Every functions covered |
| Decision/Branch | Every if/switch branches; every goto, break, continue, return, throw; every catch |
| Condition | Every condition (evaluated true and false):<br>If (A \|\| B) ⇨ 2 conditions A and B |
| MCDC (Modified condition / decision coverage) | Every decision and condition true and false, but also condition change the decision by itself<br>n conditions ⇨ $n+1$ tests |
| MCC (Multiple condition coverage) | Every possible combination for condition.<br>n conditions ⇨ $2^n$ tests |

# Facing complexity, MCDC in details

**if ((A || B) && (C || D)) {...}**

4 conditions ⇨

- All combinations:
  ***$2^4$ = 16***
- Reachable combinations (MCC):
  ***9***
- MCDC (yellow):
  ***5***

| | A | B | C | D | (A\|\|B)&&(C\|\|D) |
|---|---|---|---|---|---|
| 1 | F | F | F | F | F |
| 2 | F | F | F | V | F |
| 3 | F | F | V | _ | F |
| 4 | F | V | F | F | F |
| 5 | F | V | F | V | V |
| 6 | F | V | V | _ | V |
| 7 | V | _ | F | F | F |
| 8 | V | _ | F | V | V |
| 9 | V | _ | V | _ | V |

Find the best coverage compromise by combining unit, integration and functional tests.

# Next objective

- **Essence of software and SOLID principles**

# Reference

- http://misko.hevery.com/attachments/Guide-Writing%20Testable%20Code.pdf
- https://www.youtube.com/watch?v=XcT4yYu_TTs
- Extreme Programming Explained, Embrace Change – Kent Beck
- xUnit Test Patterns, Refactoring Test Code – Gerard Meszaros
- Object-Oriented Software Engineering: A Use Case Driven Approach - Ivar Jacobson
- https://blog.cleancoder.com/uncle-bob/2017/03/03/TDD-Harms-Architecture.html
- https://site.mockito.org/
  - https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html