Unit Test

“One original thought is worth a thousand mindless quotings”
- Diogenes

Outline

1. Introduction
2. Make Unit Tests Work for You
3. Practices

1. Introduction

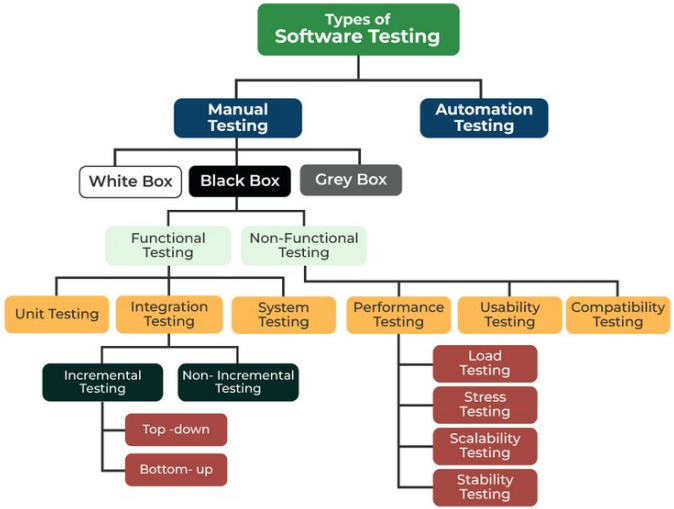
1.1. Problems

- Common issues In the process of software development:
  - A lot of bugs (from minor to major)
  - Afraid of fixing or refactoring old codes
  - Hard to be build up

→ **Testing**

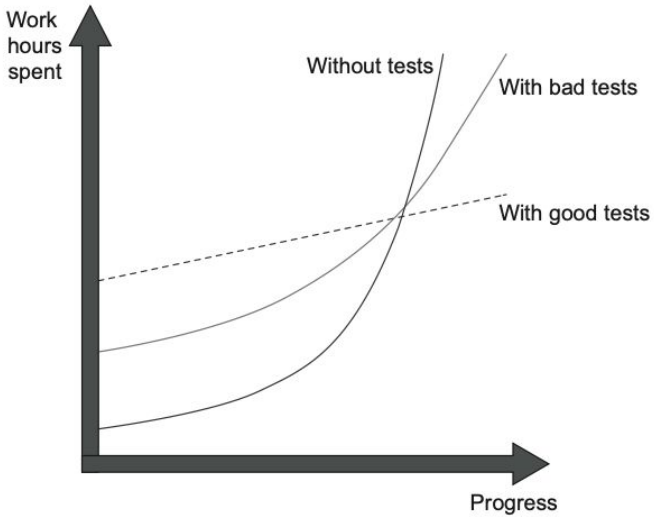
1.2. Types of Testing

- **Unit Testing**
- **Integration Testing**
- End-to-end Testing
- Regression Testing
  - Regression: bug
  - Ensures that new code changes do not negatively impact existing functionality.
- Smoke Testing:
  - A preliminary test to ensure that the basic functionalities work.
- Performance Testing
- Load Testing
- …



1.3. Goal of Unit Testing

- The goal is to enable **sustainable** growth of the software
- Unit Testing helps dev become more careful and focused on their code



1.4. Definition

- A unit test is an automated test that:
  - **Verifies a small piece (unit) of code**
  - **Does it quickly**
  - **Does it in isolation from other tests**

1.5. Test Accuracy

- **Every test has accuracy**
- Covid test kit
- If a guy passes the test (negative) and the guy does not have covid actually → true negative
- If a guy does not pass the test (positive) and the guy has covid actually → true positive
- If a guy passes the test (negative) but the guy has covid actually → false negative
- If a guy does not pass the test (positive) and the guy does not have covid actually → **false positive**

Table of error types		Functionality is	
		Correct	Broken
Test result	Test passes	Correct inference (true negatives)	Type II error (false negative)
	Test fails	Type I error (false positive)	Correct inference (true positives)

1.5. Test Accuracy

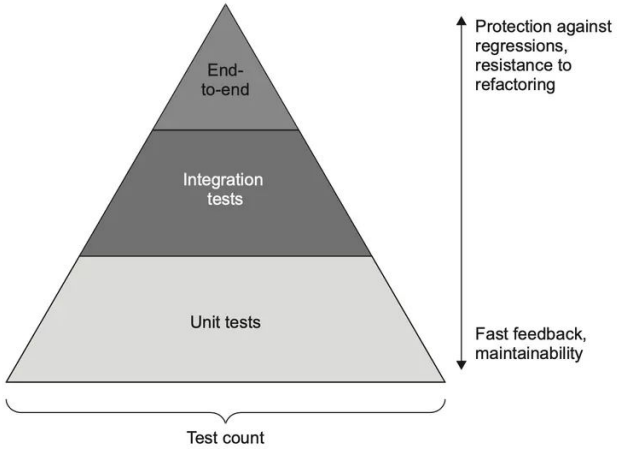
$$\text{Test accuracy} = \frac{\text{Signal (number of bugs found)}}{\text{Noise (number of false alarms raised)}}$$

- Signal:
  - High True Negative → Low False Negative
  - High True Positive
- Noise:
  - Low False Positive

2. Make Unit Tests Work for You

2.1. Four pillars of a good unit test

- **Fast feedback**
  - How quickly the test executes
- **Maintainability**
  - How hard it is to understand the test
  - How hard it is to run the test
- **Protection against regressions**
  - How good the test is at indicating the presence of bugs (regressions)
- **The accuracy of test remains when refactoring**

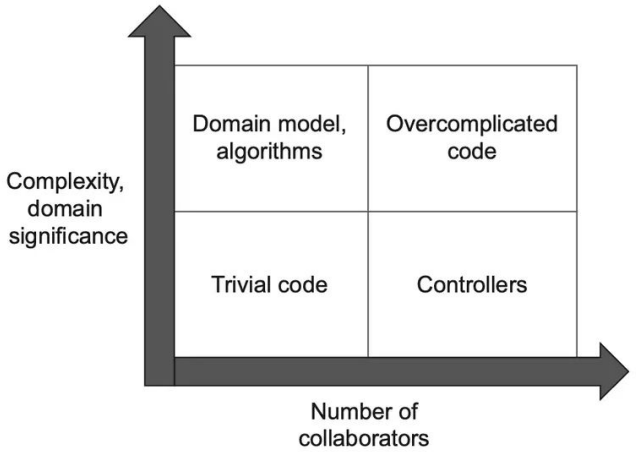


2.2. Coverage Metrics

- 2 types of Coverage Metrics:
  - Code Coverage: popular
  - Branch Coverage
- 2 types all has problem
- If code coverage is under the threshold (70%)
  → the quality of test suite and production code is not good
- If code coverage is over the threshold (70%)
  → can not say test suite is very good
- Coverage metric is good negative indicator
- **Should not imposed a threshold for dev**

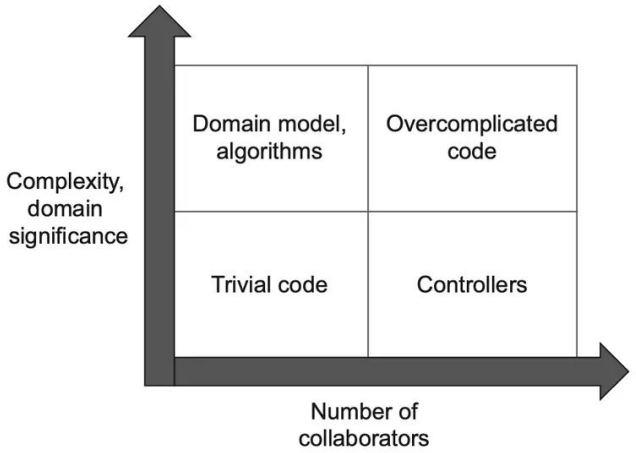
2.3. Four Types of Code

- All production code can be categorized along two dimensions:
  - Complexity or domain significance
    - Complexity: the number of decision-making (branching)
    - Domain significance shows how significant the code is for the problem domain
  - The number of dependencies



2.3. Four Types of Code

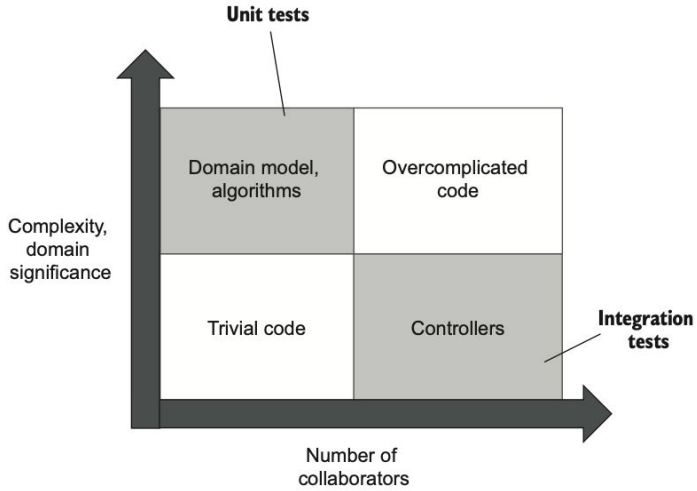
- **Domain model and algorithms**
- **Trivial code**
- **Controllers**
  - This code doesn’t do complex or business critical work by itself but coordinates the work of other components like domain classes and external applications.
- **Over-complicated code**
  - Such code scores highly on both metrics: it has a lot of collaborators, and it’s also complex or important.



2.3. Four Types of Code

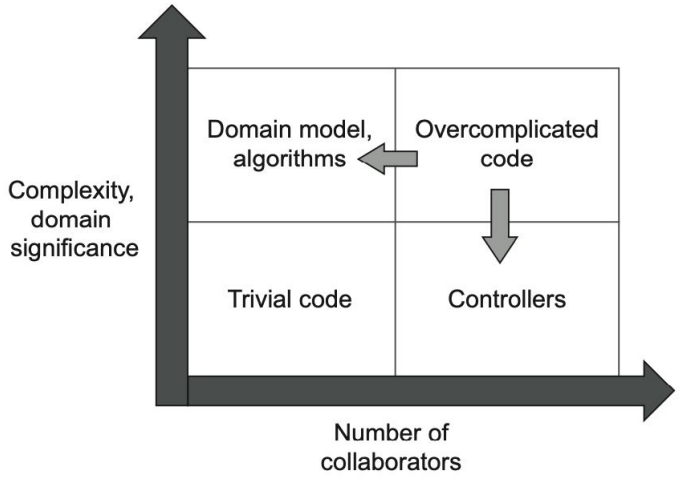
- **Unit tests for Domain model and algorithms**
- **Integration tests for Controllers**
- **Trivial Code no need to be tested**

How about Overcomplicated Code?



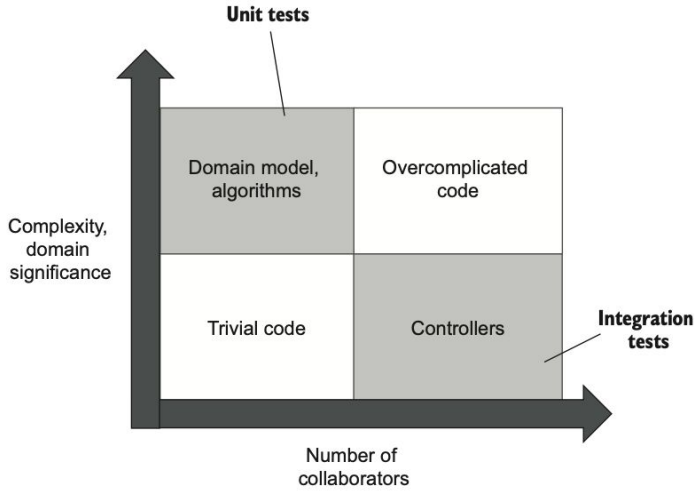
2.4. Handling with Over-complicated Code

- **Unit tests reflex to code**
- **Refactoring over-complicated code to Controller and Domain model, algorithms**



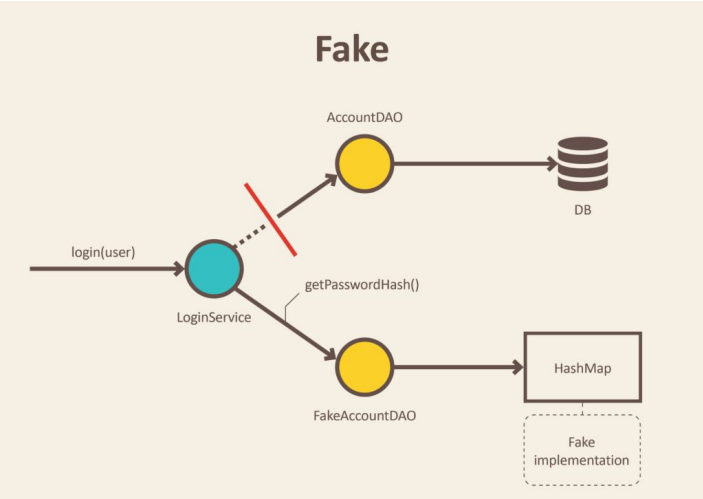
2.5. Integration Test

- **Integration test does not meet at least one** of three requirements (of a unit test):
  - Verifies a small piece (unit) of code
  - Does it quickly
  - Does it in isolation from other tests



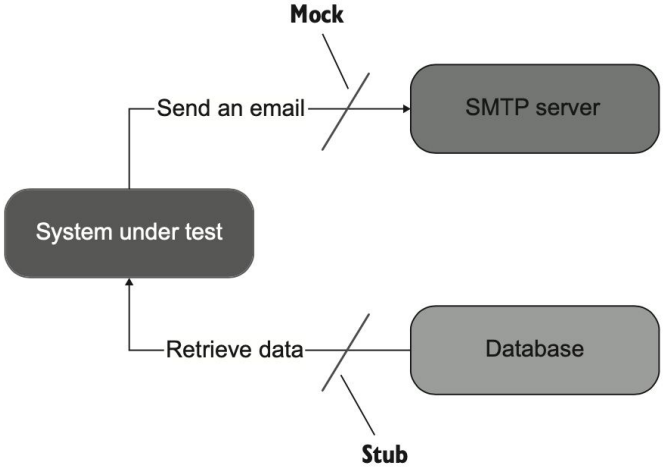
2.5. Test Double

- Test double is an overarching term that describes/simulates fake dependencies in tests



2.6. Mock vs Stub

- Test double is an overarching term that describes/simulates fake dependencies in tests
- **Mocks** help to emulate and examine **outcoming** interactions.
    - These interactions are calls the SUT makes to its dependencies to change their state.
    - Predefined behavior
- **Stubs** help to emulate **incoming** interactions.
    - These interactions are calls the SUT makes to its dependencies to get input data
    - Predefined values



3. Practices

3.1. Core Concepts

- An **edge case** is an issue that occurs at **an extreme** (maximum or minimum) operating parameter.
  - One is just number
  - Easy to see
- A **corner case** is when **multiple parameters** are simultaneously at extreme levels
  - Hard and time wasted to see all
- Example: an oven with two settings/parameters
  - Time setting
  - Heat setting

3.2.1. Best Practices / Unit Test

- First, make sure the happ case run.
- Catch all edge cases.
- Try your best to catch corner cases in a acceptable amount of time.
    - 10 corner cases take 10 hours
    - 7 corner cases take 2 hours only.
    - Trade off: will be mitigated by experience
- Let tester take care the left 3 corner cases.

3.2.1. Best Practices / Unit Test

- Unit tests are developed, executed and maintained separately from the production code
- Unit tests for Domain model and algorithms
- Naming: [MethodUnderTest]_[Scenario]_[ExpectedResult]

3.2.2. Best Practices / Mock

- Integration tests for Controllers
- Create interfaces for dependencies
- Mocks are for integration tests only
- Verifying the number of calls if needed
- Only mock types that you own

3.2.3. Best Practices

- Be careful with the end of a cord. ExampleL: [0, 99]
- Group related scenarios in one test
- Prioritize test cases
- Have to cover all edge cases
- Cover corner cases as possible. Do not waste too much time to cover all corner cases.
- With DDD, we can configure to include domain package, and exclude other packages. Then code coverage is on domain package only.

Recap

- Unit testing enables sustainable growth of the software
- The super high code coverage does not make sense. Let devs do unit tests naturally
- Unit tests on Domain model and algorithms. Integration tests on Controllers

Homework

- Write unit tests, integration tests for booking use case.

References

- https://viblo.asia/p/unit-testing-phan-1-r1QLxxydLAw
- Book: Unit Testing Principles, Practices, and Patterns (2019, Manning Publications) - Vladimir Khorikov

Thank you 🙏