# Principles of Software Design
# Software quality

Robert Lukoťka
lukotka@dcs.fmph.uniba.sk
www.dcs.fmph.uniba.sk/~lukotka

M-255

## Software quality

What is quality of software

- Software functional quality (External)
  - how well it complies with or conforms to design and functional requirements
  - fitness for purpose, how it compares to competitors in the marketplace
  - **degree to which the correct software was produced**
- Software structural quality (Internal quality)
  - how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability.
  - **what is under the hat**, lack of technical debt

## Software quality management

Software quality management:

- Quality assurance - Setting up processes and standards within the organization.
- Quality planning - Define the quality attributes associated with the output of the project and how those attributes should be assessed.
- Quality control - Reviews software at its various stages to ensure quality assurance processes and standards are being followed

To create quality software you need to:

- Measure attributes associated with quality
- Control and enforce them

This has to be an integral part of the development process.

## Structural quality - Example

How to enforce code readability?

## Structural quality - Example

How to enforce code readability?

- code reviews
- linter (especially important in languages with a lot of backward compatibility heritage)
- pair programming
- ...

Automated tools can be integrated into the tools used (e.g. git hooks).

## Functional quality - Verification and validation

- Verification - The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition.
- Validation - The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders.

You should verify and validate not only the whole product but all artefacts.

- Example: How to validate requirements?
    - Customer may read the requirements document (however, this is often insufficient)
    - Prototyping, Test case generation, ...

## Functional quality - Verification and validation

- Verification - The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition.
- Validation - The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders.

Verification

- Manual - preferred only in certain circumstances
- Automated - seems like a time wasting sometimes but actually saves a lot of time in projects that take longer than a few days.

## Software testing

*Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test.*

- Defect detection techniques (the numbers are probably not too precise, but you should check it out to see what the techniques are).

## Testing

### Testing pyramid

- Small-scale automated tests (**unit tests**) are fast - we can and should perform a lot of them.
- Medium-scale test (**integration tests**) - they check, if the smaller parts work well together. You do not need to catch each exceptional case as this should be done on the lowest level. They take a bit longer, but you need less of them. You should automate them too.
- High level tests (**system tests**) - a few test that check if the parts of the application are correctly put together. Even if you need just few of them, it is still a good idea to automate them.
- GUI tests - you want just to assure that each element is associated with the correct action. You should consider to automate these test (there are many tools to do this), but testing by humans has some merit here.

## Testing

Besides that you should perform tests for non-functional requirements when it makes sense (e.g. performance, resource consumption, etc). These test are outside the testing pyramid and may take quite long. You can run them e.g. overnight / weekend.

## Testing automation

- Automating the tests saves a lot of resources in long term projects.
- To benefit fully from the automated testing the tests should be created and improved consistently during the software development process.
  - You want to have confidence in your automated tests. That is, if the test passes, there is a good chance that you did not break something while performing changes.
  - Good tests should give you confidence to fight against software entropy - refactor when necessary.
  - If you do not do this, you might end with very fragile code "If it ain't broke, don't fix it". This approach guarantees that the software entropy takes over and the source may become unmaintainable over time.

## Testing automation

Perhaps the most important concern is to guarantee that your tests change as little as possible when you do refactoring.

- This requires not only good design of tests but also good design.

## Unit tests - dependencies between classes

O-O design is based on the cooperation between objects.

Dependencies:

- creates, destroys
- calls method
- modifies
- ...

## Unit tests

Class under test - class we want to test - what about its collaborations.

- Solitary - We strictly test separate classes. All other classes should be excluded from the testing.
- Sociable - We can use some classes closely related to the class under test (especially when the tests are still fast and there are no significant side effects present).

Even if we prefer sociable unit tests, we want to have the control on what to separate and what not.

## Unit tests - dependencies between classes

How the following dependencies affect our ability to perform unit test (of course, in languages like Python we can do almost anything, but this may introduce gaps into the testing)?

- creates, destroys
- calls method
- modifies
- ...

All these dependencies present a huge problem. But dependencies are important, aren't they?

## Dependency injection and dependency inversion

- A class should not create instances of other classes (if we do sociable testing it might make some exceptions of this rule for closely related classes) - dependency injection / dependency injection of a factory.
- A class should not depend on the implementation of the collaborator, just on the interface - dependency inversion

**Dependency injection** is a way to implement **dependency inversion**.

## OO design and object dependence

All collaborators (we want to separate) should be separated by an interface.

- creates, destroys - we give a factory (implementing a factory interface) as an argument in the constructor of the object
- calls method, modifies, . . . - we give the collaborator in the constructor or as a method parameter

We can inject the dependency either via constructor or as a method argument.

## Test doubles

What about the collaborators during the tests? They are behind an interface, thus we can create a new implementation of these classes - **test doubles**.

M. Fowler: Test doubles (read this)

## DI frameworks

- If DI is used a lot, the instance creation may become very complicated.
- However, testing is very often the main reason to do DI (other reasons is to gain flexibility, e.g. strategy pattern).
- Thus very often it is possible to indicate defaults for the dependencies injected.
    - There are subtle problems with this approach - this creates dependencies between the sources and affect compilation time negatively
- Other approach is to use DI (micro)frameworks - tools supporting DI.

## DI and functional programming

What if we do functional programming?

- DI is a notion of OO design - OO design is applicable for functional programming.
- If a function $f$ calls function $g$ which we want to separate from $f$, we do not use it directly. We should have $g$ injected - that is $f$ takes a function with the same signature as $g$ as a parameter (default may be set to $g$). This allows us to replace $g$ with a test double.

## Sociable versus solitary unit tests

I prefer sociable approach

- Advantage of solitary test:
  - they may be faster
  - failure precisely indicates what failed
- but:
  - may require a lot of detailed mocking
  - may make tests too coupled with implementation.
    This make cause that in the case of refactoring we need to rework the tests. This goes against one of the main reasons to white detailed unit tests. Another reason why this coupling is bad is that it is good to have two separate approaches to implementation and testing. If it is not the case it is possible that we do the same mistake twice.
- Solitary unit tests are an accepted approach - there are ways to mitigate these issues.

## When to write tests?

You can do it anytime you want

- After writing implementation

## When to write tests?

You can do it anytime you want

- After writing implementation - obvious choice, but not necessary the right one
  - You are happy you "finished your work". It is easy to write too few tests.
- While writing implementation - see Test Driven Development
- Between design and implementation.
- During design - grantees testable design, gives exact and clear specification to the programmer, may result in easier to use interfaces.

## Writing the tests during the implementation

Test driven development

- Three rules of TDD (I consider this a more hardcore version of TDD).
    1. You are not allowed to write any production code unless it is to make a failing unit test pass.
    2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
    3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.
- This guarantees that the tests will cover boundary and exceptional cases that were implemented

## Resources I

- Wikipedia - Software Quality Management
- M. Fowler: Test doubles
- Wikipedia - DI

## References I

- Wikipedia - Software Quality
- Wikipedia - Software Quality Management
- Wikipedia - Verification and Validation
- Wikipedia - Software Testing