1DV610 – Introduction to Software Quality

Daniel Toll – daniel.toll@lnu.se

Dr. Rüdiger Lincke – rudiger.lincke@lnu.se

HT 2016

Course Structure

- Course Introduction/Lecture 1 – Introduction to Software Quality
- Lecture 2 – Software Requirements and Testing
- Lecture 3 – Metrics & Heuristics – Size, Complexity, Coupling & Cohesion
- Lecture 4 – To be announced later…
- Lecture 5 –
- Lecture 6 -
- Lecture 7 -
- Lecture 8 -
- Lecture 9 -

Course Introduction (Daniel)

- Sort out some practical issues
  - Communication
    - Slack channel: #1dv610
    - E-mail: rudiger.lincke@lnu.se (Lectures 1-3), **daniel.toll@lnu.se** **(everything else ☺)**
    - Video lectures in Adobe Connect: https://connect.sunet.se/softwarequality
  - Course material
    - Course homepage: https://coursepress.lnu.se/kurs/introduktion-till-mjukvarukvalitet/
    - Literature: Clean Code, A Handbook of Agile Software Craftmanship, Robert C. Martin
    - Slides
  - Assignments/Exam
    - Please start today with working on Assignments A1 and A2 (as soon as they are available).
    - Obs! Hard deadlines for assignments.
    - Exam is online at end of course.
  - Questions or problems, e-mail to Daniel (daniel.toll@lnu.se).

Introduction to Software Quality

- What is Software Quality?
- Basic Software Quality
- Achieving Basic Software Quality
- Tests and Test Coverage
- Properties of Good Code
- Conclusions

Software Quality – Definition

Software Quality is the entirety of properties and attributes of a product or a process relating to their **fitness to fulfill certain requirements**.

(DIN 55350/11)

Software Quality – Definition cont.

- Very broad definition.
- Q/R are different for organizations, projects, teams, stakeholders, project phases, etc.
- Obs! Key is ”… **fitness to fulfill certain requirements.**”
- In particular, fitness to be made fit without too much hazle.
- Much theory and practice exists providing systematic approaches.
  - Metrics suits, quality models, quality assurance processes and standards, etc.
  - 4DV107 – Software Quality Course, provides some more extended insight.
- 1DV610 – **Introduction** to Software Quality, **provides foundations**.

Basic Software Quality

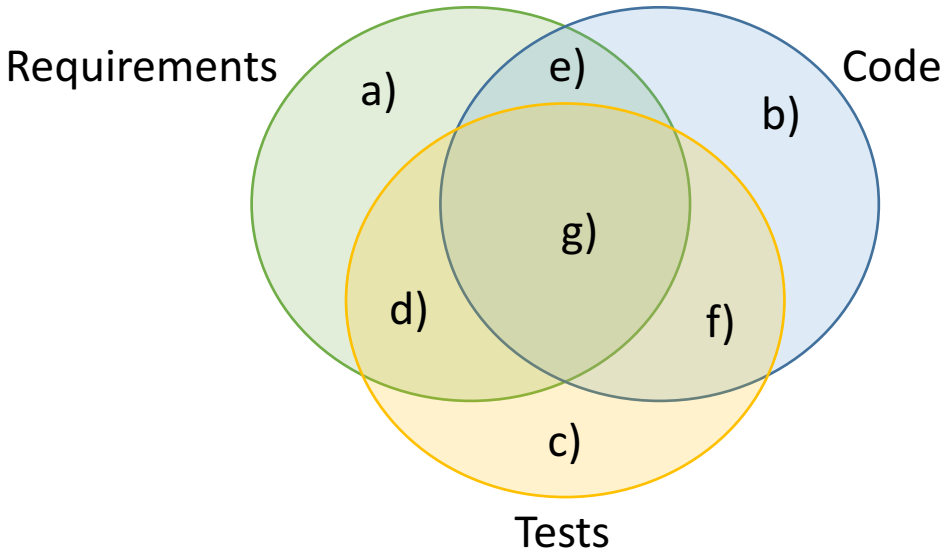
- Primary purpose of (Software) Quality is to **save time and money!**
- This usually includes development of **good quality systems** that:
  - comply with their requirements. I.e., as customer:
    - you don’t want to pay/wait for specification, development and testing of features you don’t need.
    - you want to get the *biggest bang for the buck*.
    - you want to take advantage of the 80:20 rule, i.e., 80% of value come from 20% of the features and thus cost (if you put the priorities right).
  - are easy to maintain and extend over time.
    - As customer you don’t want to pay for increasingly expensive changes due to inflexible/rotting code (incl. tests and specification).
    - As company/customer you don’t want to have cost due to long learning times from new staff.
    - **As developer you don’t want to be afraid doing changes and refactoring's to a (working) system.**

Achieving Basic Software Quality

- Requirements, tests and code must be **aligned** at all times.
- Code (incl. test code) must be **clean (code**), i.e.:
  - readable, understandable and changeable,
  - understandable and self-documenting,
  - well structured (small size, low complexity, low coupling, high cohesion),
  - (automatically) tested,
  - continuously improved/refactored,
  - include error handling.
- They must be easy to communicate between all stakeholders (biggest challenge).
- Note:
  - Requirements **will** change over time (known to unknown, abstract to detailed).
  - Code simply needs continuous improvement (refactoring) to transform working code into good code.
  - Once its working, you are far from being done ;)

Why Is Alignment So Important?

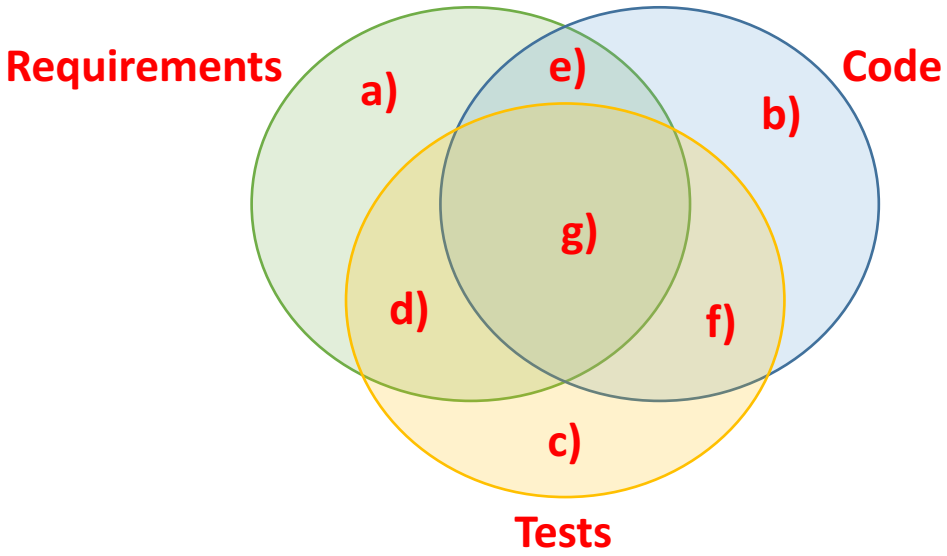
- **Misalignment increases cost!**
- a)+b)+c)+d)+e)+f)+g) = **Costs**
- g) **Value** (very good)
- a) Not implemented yet (ok?)
- d) About to be implemented, test cases exist (good?)
- e) implemented but not tested (bad!)
- c+f) implemented but not required (bad?, waste of time and money)
- b) implemented but neither required nor tested (very bad/evil, waste of time and money)



Why Is Alignment So Important? Cont.

a)+b)+c)+d)+e)+f)+g) = Costs

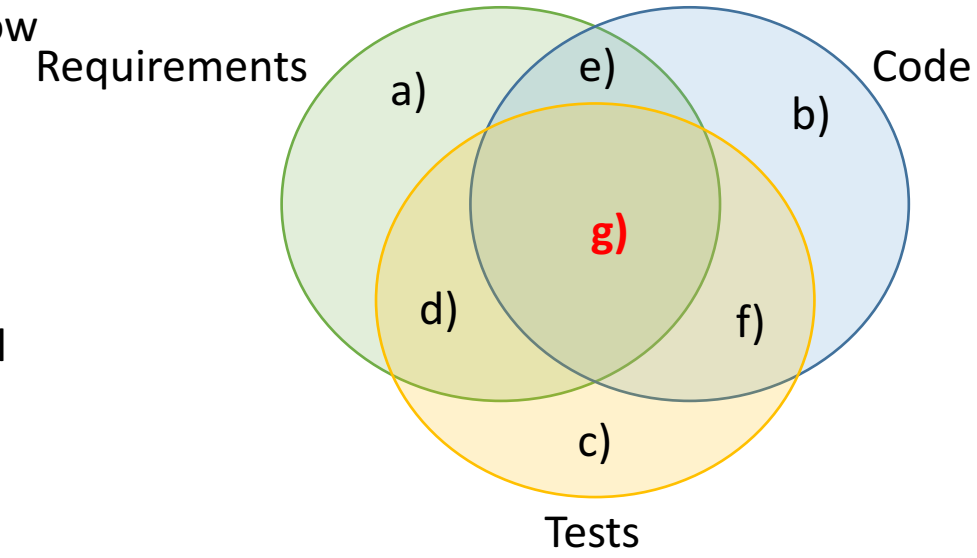
- To create any of this a)-g) takes time and thus costs money.
- The more complicated things are the more time it takes.
- The more people are involved the more time it takes.
- Unless a) is clear and well-defined, b)-g) will take more time than needed. Higher risk for b), c), f).
- Time for meetings, discussions, documentation, changing documentation, implementing and re-implementing, testing, etc.
- b)-f) must be minimized.
- Cost includes also future costs.



Why Is Alignment So Important? Cont.

g) Value (very good)

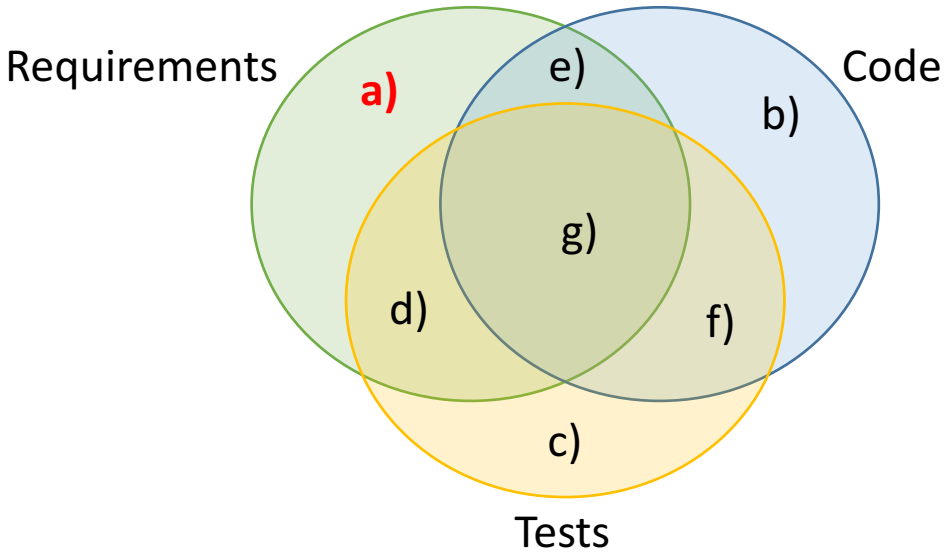
- Requirements describe the needs of a customer.
- Code describes the solution to the computer.
- Tests make sure that requirements and code are aligned (now and in future).
- One only has a **lasting** benefit once the requirements are implemented and tested assure that they stay that way.
- Requirements change and so their specification, the code and tests need to be adjusted.
- Misalignment due to changes to code and requirements will be uncovered by tests.
- Tests (incl. high test coverage) make sure that only the **minimal effort** is spent to implement requirements.
- High test coverage indicates high value (best use of resources).



Why Is Alignment So Important? Cont.

a) Not implemented yet (ok?)

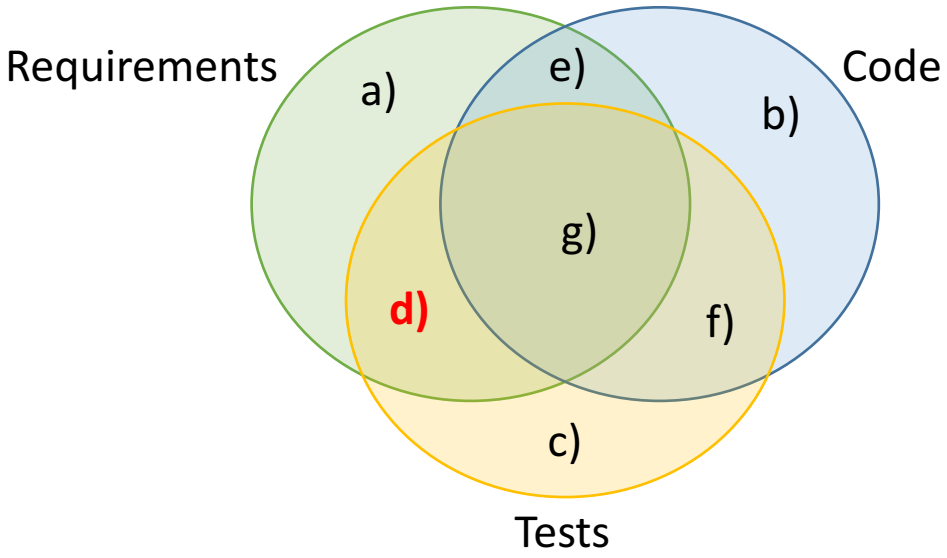
- Requirements exists on different levels.
- Some are abstract and high level, some are concrete and specific. Dep. on the stage in the process and project.
- Having too many requirements on a detailed level is not good.
- Having a empty backlog is not good neither.
- Specifying requirements takes time, in particular if one does it well.
- Specifying requirements is the first step in generating value.
- The right priority of requirements is paramount to reduce cost and create value.
- Often one needs to implement requirements and see a result to be able to generate more requirements iteratively and incrementally.
- Requirements need to be updated on-the-fly.
- Requirements should be detailed right before they are implemented (prior a sprint).



Why Is Alignment So Important? Cont.

d) About to be implemented, test cases exist (good?)

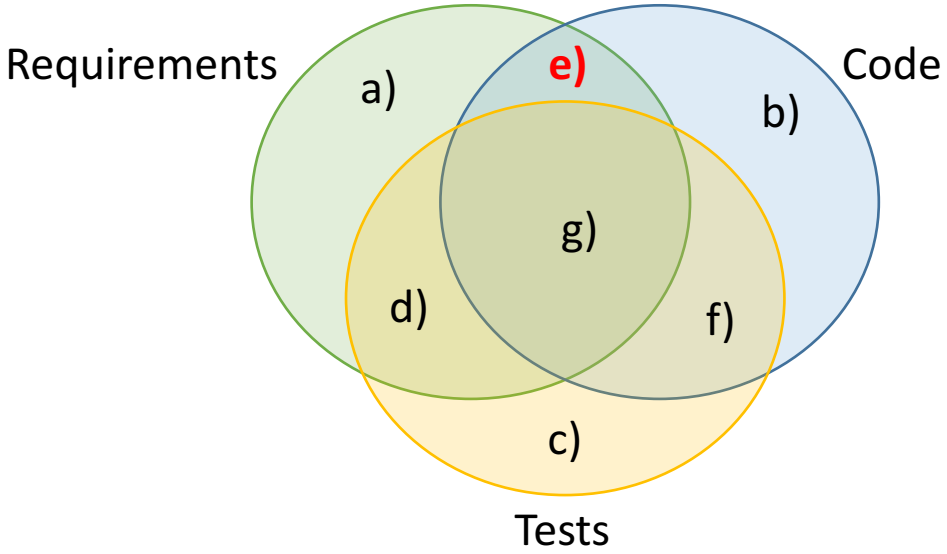
- In test and behavior driven development one **implements tests right before the code**. This is the **best practice**.
- Unclear requirements are uncovered and clarified.
- Code will be written in a testable and well structured way.
- One knows when he is done (test passes).
- One is not tempted to write more code than needed to pass a test (high test coverage).
- Tests are most detailed examples/requirements besides the code.
- Ones code breaks, tests tell at once.
- Tests need to be updated on-the-fly.
- Tests should be implemented right before the code is implemented (to minimize cost and increase value), most efficient, you are in the right state of mind.



Why Is Alignment So Important? Cont.

e) Implemented but not tested (bad!)

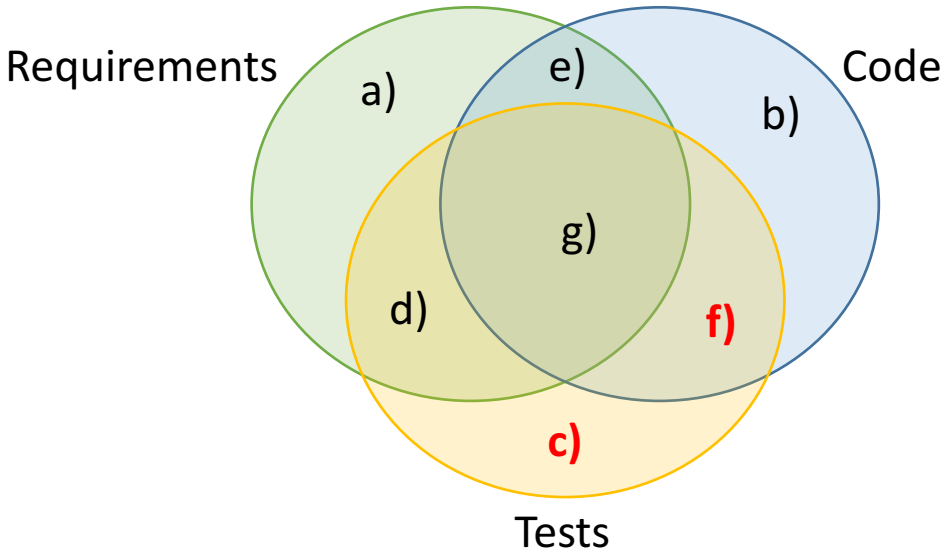
- Requirements have been implemented but are not tested.
- *This creates costs but only temporary value.*
- It is not clear if the full requirements are implemented.
- Current or future misalignment between requirements and code will be undetectable.
- Refactoring of code is impossible.
- Code will eventually rot.
- It cannot be assured that only the minimal amount of code has been written to satisfy a requirement.
- One could still implement tests later, but more difficult and less efficient (usually does not happen).



Why Is Alignment So Important? Cont.

c+f) Implemented but not required (bad)

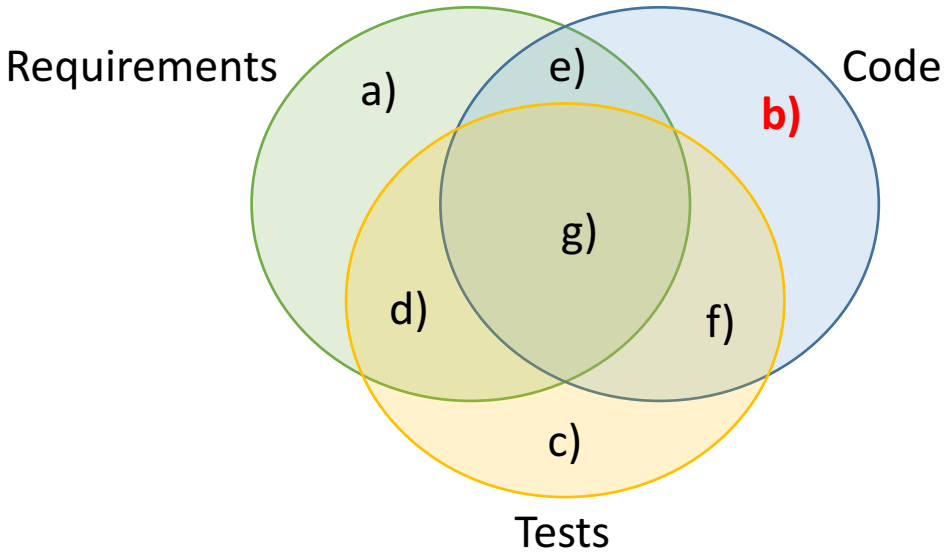
- Preparing for future requirements? Don’t do it!
- Do not prepare for future eventualities, just cost, no value.
- Waste of time and money. Code needs to be maintained, tested, build, etc.
- No alignment with requirements, thus no value, only cost.
- Might happen through misunderstood/changed requirements. Get rid of it (git will remember in case).
- f) is maybe not that bad since its tested at least, but still, don’t do it. Anyway, what requirements are the tests based on?



Why Is Alignment So Important? Cont.

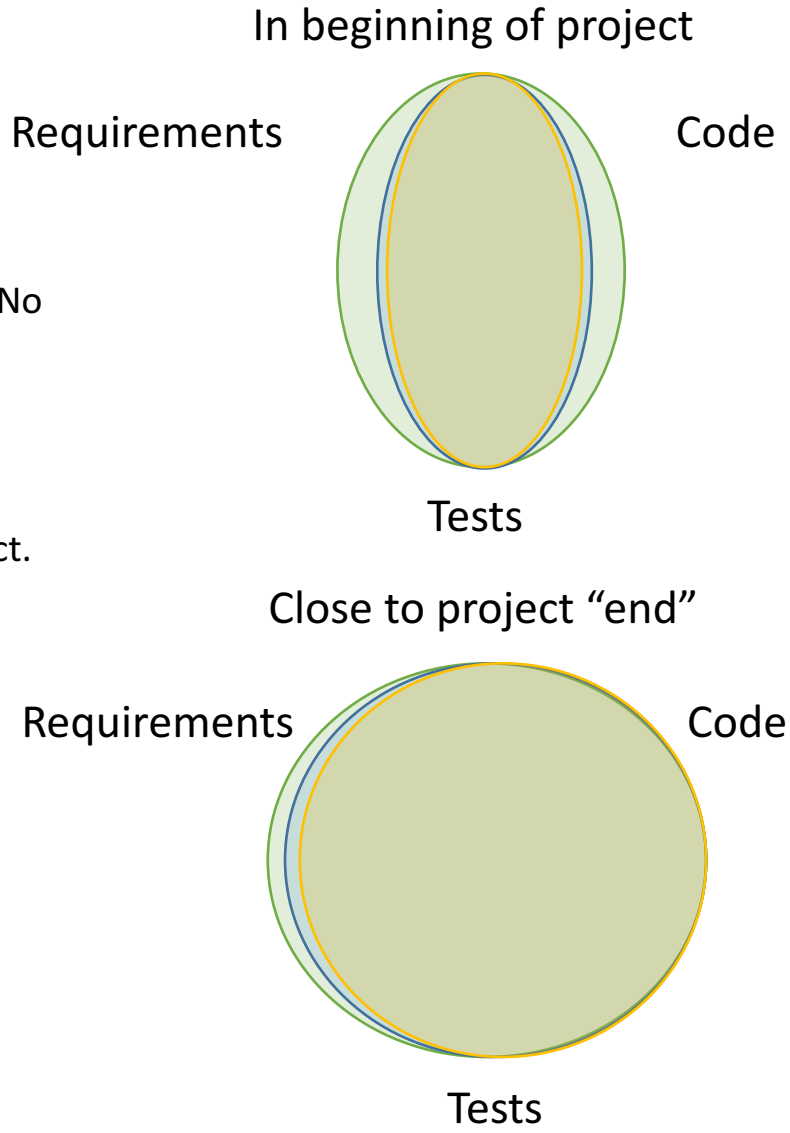
b) Implemented but neither required nor tested (very bad/evil)

- Pure evil. Waste of time and money. No value at all.
- Slows down system/development, contributes to confusion and rotting code.
- Adds to low test coverage and bad quality.
- You might think you could use it later, but you will not.
- Code cannot be refactored since no tests.



Ideal Situation

- Requirements, code and tests align (almost perfectly) throughout development.
- Agile development (iterative and incremental) allows to realize value fast. No large costs wait to be turned into value.
- Most of the requirements should/could be known on a high level from the beginning, others are discovered during the process.
- Requirements are slightly ahead of tests, which are slightly ahead of code.
- Almost all (relevant) requirements are implemented by “end” of the project.
- No time has been wasted on writing code/tests not being needed to fulfill requirements.
- Almost all code is tested (ca. 80-90% coverage) all the time.
- **Refactoring's or changes can be done any time without fear of breaking something.**
- At any point in time you know the system is working.
- No problem to add new requirements for future versions.



How To Achieve This?

- Well documented requirements in structured specifications.
- Well structured/clean code.
- Tests glue code and requirements together assuring consistency at all times.
- Traceability between requirements, tests and code.
- Right development models:
  - Agile development
  - Test Driven Development (TDD)
  - Behavior Driven Development (BDD)
  - Metrics (Test Coverage, Size, Complexity, Coupling & Cohesion)
- Follow a number of good practices and principles.

Basic Software Quality Summarized

- Main goal
  - Increase value
  - Minimize cost
  - Short- and long-term
- Reached by
  - Right and lasting functionality
  - No/minimal waist:
    - Requirements = Tests = Code
    - High test coverage (80-90%)
  - Good communication
  - **Good changeability**

- Changeable
  - Good structure
  - Understandable/(self-)documented code
  - Tested code (takes fear of changing)
- Change will happen
  - Iterative and incremental development
    - First make it work
    - Then improve, clean it up, keep it changeable in future
    - You won’t get it right from the start
    - Insight is a process
  - Changing requirements

Tests and Test Coverage

- Tests validate that code fulfills requirements.
- They are the **glue** between requirements and code.
- Without tests misalignment is unavoidable.
- Tests need to be automated.
  - Too many requirements/code to test.
  - Want to do it frequently.
- Test coverage shows how much of code contributes to solving requirements.

- High test coverage indicates low wait.
- Low test coverage indicates waist and/or missing tests.
- There are tools available calculating test coverage using different criteria:
  - Function coverage
  - Statement coverage
  - Branch coverage
  - Etc.

Properties of Good Code

- Explained in Clean Code book.
- Aid understanding, changing, testing, etc. of code.
- Includes:
  - Naming of methods and classes
  - Size of methods and classes
  - Following principles and patterns supporting changeability and good structure.
- In object-oriented programming and design, there are five (5) basic principles.

Five Basic Principles of OOPD

- The intention: make it more likely that a programmer will create a system that is **easy to maintain and extend over time**.
- Apply principles while working on software to remove code smells by causing the programmer to refactor the software's source code until it is both legible and extensible.
- Part of an overall strategy of agile and adaptive software development.
- Principles:
  - Single Responsible Principle
  - Open/Closed Principle
  - Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

Single Responsibility Principle (SRP)

- Every module or class should have responsibility over a single part of the functionality provided by the software.
- That responsibility should be entirely encapsulated by the class.
- All its services should be narrowly aligned with that responsibility.
- “A class should have only one reason to change.” - Robert C. Martin

SRP - Example

- As an example, consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change. These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.
- The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.
- The responsibility is defined as *a charge assigned to a unique actor to signify its accountabilities concerning a unique business task*.

Open/Closed Principle (OCP)

- The open/closed principle states “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”,
- that is, such an entity can allow its behavior to be extended without modifying its source code.
- The name open/closed principle has been used in two ways. Both ways use inheritance to resolve the apparent dilemma, but the goals, techniques, and results are different.

OCP - Example

- There are several problems with this function.
- First, it's large, and when new employee types are added, it will grow.
- Second, it very clearly does more than one thing.
- Third, it violates the Single Responsibility Principle (SRP) because there is more than one reason for it to change.
- Fourth, it violates the Open Closed Principle (OCP) because it must change whenever new types are added.
- But possibly the worst problem with this function is that there are an unlimited number of other functions that will have the same structure.

Listing 3-4
Payroll.java

```
public Money calculatePay(Employee e) throws
InvalidEmployeeType {
  switch (e.type) {
  case COMMISSIONED:
    return calculateCommissionedPay(e);
  case HOURLY:
    return calculateHourlyPay(e);
  case SALARIED:
    return calculateSalariedPay(e);
  default:
    throw new InvalidEmployeeType(e.type);
  }
}
```

Liskov Substitution Principle (LSP)

- Substitutability: in a computer program, if *S* is a subtype of *T*, then objects of type T may be replaced with objects of type S,
- i.e., an object of the type T may be substituted with its subtype object of the type S,
- without altering any of the desirable properties of that program (correctness, task performed, etc.).

LSP - Example

- A typical example that violates LSP is a Square class that derives from a Rectangle class, assuming getter and setter methods exist for both width and height.
- The Square class always assumes that the width is equal with the height.
- If a Square object is used in a context where a Rectangle is expected, unexpected behavior may occur because the dimensions of a Square cannot (or rather should not) be modified independently.
- This problem cannot be easily fixed: if we can modify the setter methods in the Square class so that they preserve the Square invariant (i.e. keep the dimensions equal), then these methods will weaken (violate) the postconditions for the Rectangle setters, which state that dimensions can be modified independently.
- Violations of LSP, like this one, may or may not be a problem in practice, depending on the postconditions or invariants that are actually expected by the code that uses classes violating LSP.
- Mutability is a key issue here. If Square and Rectangle had only getter methods (i.e. they were immutable objects), then no violation of LSP could occur.

Interface Segregation Principle (ISP)

- No client should be forced to depend on methods it does not use.
- ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
- Such shrunken interfaces are also called role interfaces.
- ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

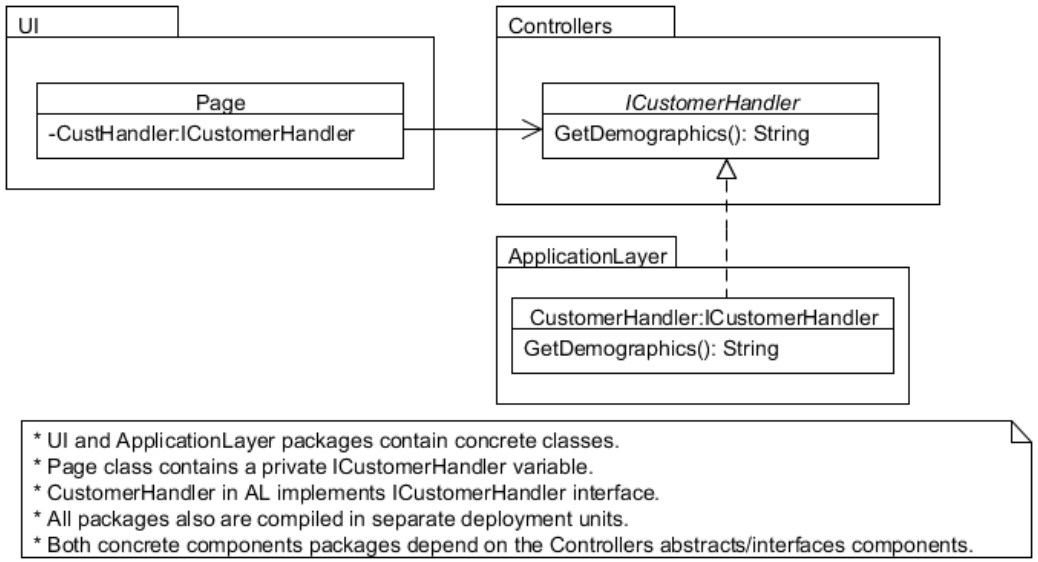
ISP - Example

- This example is about an interface for the User Interface for an ATM, that handles all requests such as a deposit request, or a withdrawal request, and how this interface needs to be segregated into individual and more specific interfaces.
- Read about it here: https://drive.google.com/file/d/0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi/view

Dependency Inversion Principle (DIP)

- Refers to a specific form of decoupling software modules.
- When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details.
- The principle states:
  - A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - B. Abstractions should not depend on details. Details should depend on abstractions.
- The principle inverts the way some people may think about object-oriented design, dictating that both high- and low-level objects must depend on the same abstraction.

DIP – Example



- UI and ApplicationLayer packages contains mainly concrete classes.
- Controllers contains abstracts/interface types.
- UI has an instance of ICustomerHandler.
- All packages are physically separated.
- In the ApplicationLayer there is a concrete implementation that Page class will use.
- Instances of this interface are created dynamically by a Factory (possibly in the same Controllers package).
- The concrete types, Page and CustomerHandler, don't depend of each other; both depend on ICustomerHandler.
- The direct effect is that the UI doesn't need to reference the ApplicationLayer or any concrete package that implements the ICustomerHandler.
- The concrete class will be loaded using reflection. At any moment the concrete implementation could be replaced by another concrete implementation without changing the UI class.
- Another interesting possibility is that the Page class implements an interface IPageViewer that could be passed as an argument to ICustHandler methods.
- Then the concrete implementation could communicate with UI without a concrete dependency. Again, both are linked by interfaces.

Other Principles

- Package principles
- Dry – Don’t repeat yourself
- GRASP - General responsibility assignment software patterns
- KISS – Keep it simple, Stupid
- YAGNI – You aren´t gonna need it
- Read about and apply them!
- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Conclusions

- Foundations to software quality are:
  - Automated tests with high test coverage
  - Well documented and up-to-date requirements (specification, code and tests)
  - Alignment of requirements, tests and code (no waist)
  - Clean code (incl. tests and specifications!)
  - Agile development for fast value realization and reduced risk
  - Adherence to principles and patterns.
- Without the above all other approaches to Software Quality cannot be applied
  - Refactoring's as result of
    - Changed requirements,
    - Bug fixes,
    - Code cleaning and improvement
    - Responding to quality metrics
  - cannot be applied without fear of side effects/breaking stuff.
  - You will never know if your system is still working.

Links and References

- https://en.wikipedia.org/wiki/Software_quality#Definitions
- https://en.wikipedia.org/wiki/Pareto_principle
- https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)