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 -
- Lecture 5 –
- Lecture 6 -
- Lecture 7 -
- Lecture 8 -
- Lecture 9 -

Software Requirements and Testing

- Requirements in General
- Agile take on Requirements and Testing
- Traceability
- Testing

Requirements in General

(Repitition)

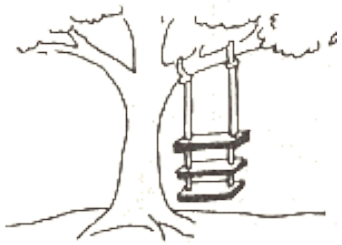
Software Requirements Definition

According to IEEE Software Engineering Terminology, software requirements are:

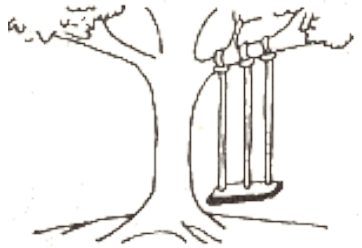
1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in 1 or 2.

Requirements Engineering and Testing

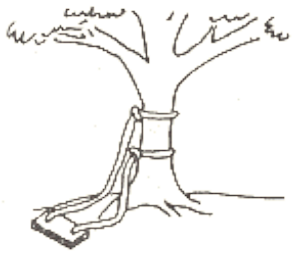
- IT projects usually require involvement of several individuals.
- Challenge: get everyone on the same page (key is communication).
- Example, build a swing ;):



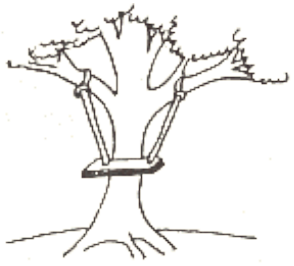
As proposed by the project sponsor



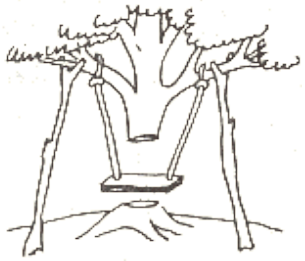
As specified in the project request



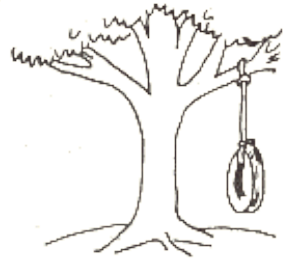
As designed by the senior analyst



As produced by the programmer



As installed at the users site

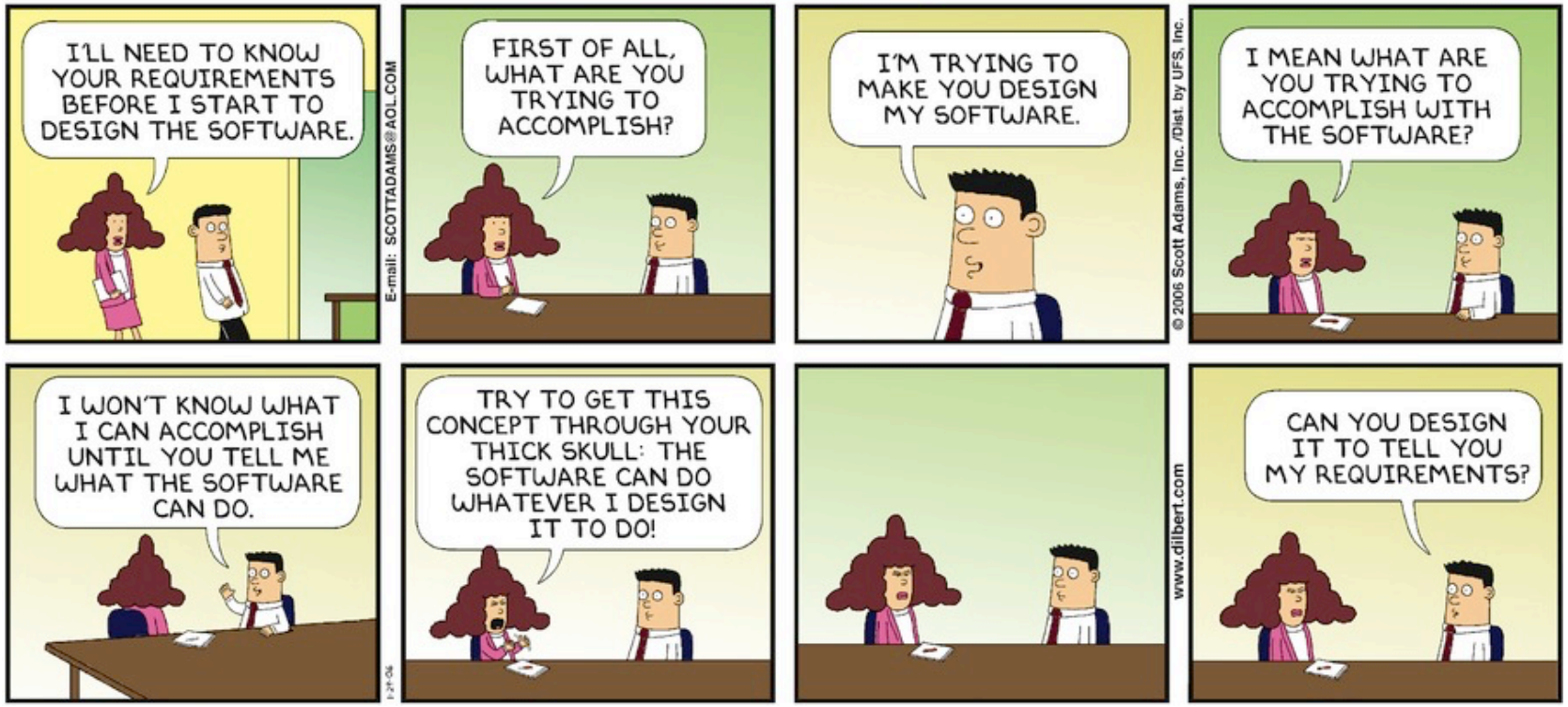


What the user wanted

Issues with Requirements

- Requirements are prone to issues of:
  - ambiguity,
  - incompleteness,
  - and inconsistency.
- Requirements are usually written as a means for communication between the different stakeholders (Requirements Specification).
- This means that the requirements should be easy to understand both for normal users and for developers.

Not easy to get requirements



Requirements Specification and Change

- A requirements specification involves representing and storing the collected requirements knowledge in a persistent and well-organized fashion that facilitates effective communication and change management.
- Examples: Use cases, user stories, functional requirements, and visual analysis models are popular choices for requirements specification.
- Requirements generally change with time.
- Once defined and approved, requirements should fall under change control.
- For many projects, requirements are altered before the system is complete.
- You simply never know everything from the beginning.
- Things change on the way.

Requirements change

Testing and Requirements

- Testing is the art of verifying requirements.
- This is a very extensive topic, but:
- If the requirements are the tests, a lot of testing is already done before implementation.

Requirements Discipline

Requirements discipline helps to deal with requirements in a structured way.

Goal

- Specify the solution, what shall the SWS do to solve our problem (under certain constraints).
- Ensure communication basis for people involved in Software Development Process (SDP).

Main Problem

- Find the right information
  - Elicitation techniques: Interviews, questionnaires, surveys, document studies, …
- Record information properly
  - Document templates: Software Requirements Specification, Use-Cases, Use-Case Model, …
- Deal with limitations and constrains
- Keep track of changes

Additional Aid

- Requirements engineering process
  - Systematic approach for
    - Collecting and
    - Specifying requirements
- Adaptation of (requirements) process required
  - Organization culture
  - Level of experience and ability
  - Other people involved in Software Development Process (SDP)
- Early discussion on
  - what constitutes a requirement
  - how do document it (document format)

Levels of Abstraction

- Different types of readers use req. in different ways
  - User requirement definition
    - Managers not interested in detail
    - System end users
  - System requirement definition
    - Developers concerned with implementation

- Example:
  - I need a User Management system vs.
  - I need to be able to reset my password.

Kinds of Requirements

- Functional requirements (not so critical)
  - What is the system supposed to do
  - Covered in **use-case model**, **use-cases, user stories, etc.**
  - Specify in-/output and behavior of system
- Non-functional requirements (critical)
  - Attributes of system and system environment
  - Covered in **supplementary specification**
  - **User interface and interaction**
- All together in **Software Requirements Specification** (SRS)

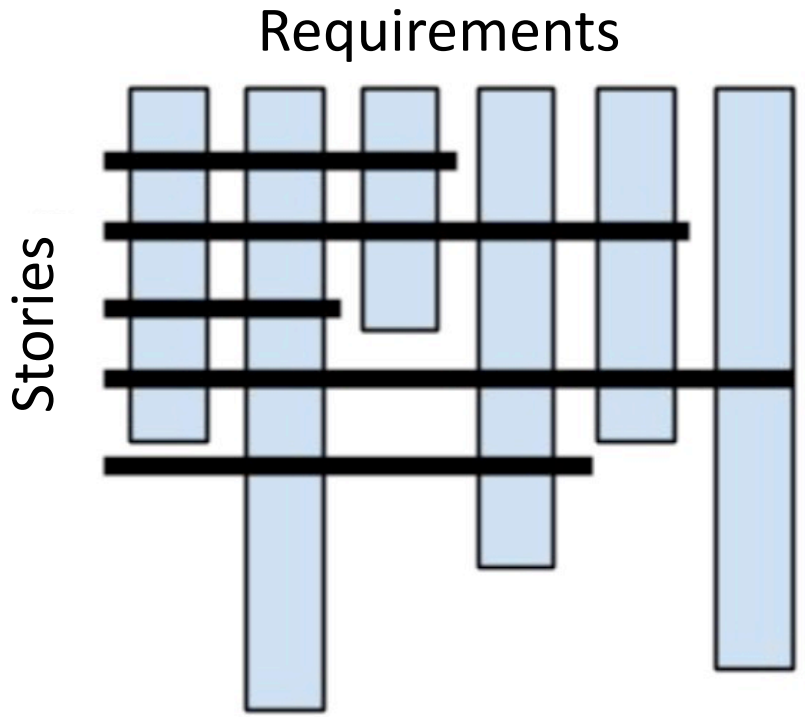
Agile take on Req. and Testing

User Stories and Sprints (Summary)

In agile projects requirements are often organized as epics, stories, scenarios.

- Product Backlog – Ordered list of requirements (features, bug fixes, non-functional requirements, what ever needs to be done).
- Sprints - A Sprint (or iteration) is the basic unit of development in Scrum. The Sprint is a timeboxed effort; that is, it is restricted to a specific duration.
- User Story - A description consisting of one or more sentences in the everyday or business language of the end user or user of a system that captures what a user does or needs to do as part of his or her job function.

Requirements vs. Stories



Behavior Driven Development (BDD)

- BDD is a software development process that emerged from test-driven development (TDD).
- Combines general techniques and principles of TDD with other ideas (domain-driven design, OOAD).
- Principles:
  - define a test set for the unit *first*;
  - make the tests fail;
  - then implement the unit;
  - finally verify that the implementation of the unit makes the tests succeed.

Roles and Responsibilities

- Product Owner (PO) – Defines requirements, connection to stakeholders, high level analysis, “knows all”, sets priorities, etc.
- Tester – Performs manual testing, implements higher level tests, explorative testing, test planning and strategy, CI/DevOps, etc.
- Developer – Develops solution (incl. architecture and design), works with requirements analysis on a lower level, implements automated tests (Unit, Regression, lower level), do estimates
- Analyst (usually split between PO, Developer, Tester)
- Team (PO, Tester, Developer)

Communication

- Requirements is about communication within team.
- The developer and tester needs to understand the problem that should be solved.
- The Product Owner is the expert and the link between the market and the team.
- Product Owner is expected to set right priorities.

Priorities

- Right priorities are important!
- Time To Market (TTM) is very important.
- Only the most important parts should be implemented.
- Don’t begin with the easiest features, begin with the most valuable ones.
- Less TTM → less risk
- Focus on Value!

Right Priorities (and Expectations)

Focus on Value

- Maximize Return On Investment (ROI).
- Focus on adding value for the customer, less risk on working stuff that does not benefit the customer (alignment).
- Knowing the motivation, why a feature is valuable helps to focus.
- Possible to split and prioritize work based on customer value, rather than based on UI views or domain entities.
- Best alignment between requirements, tests and code (cf. prev. lecture).

Motivation and Solutions

- A clear motivation allows all stakeholders to come up with different solutions. Without motivation peoples minds are locked down.
- PO gets to pick the solution with best ROI.
- Adds motivation and intent to requirements. Important for a common understanding and for avoiding implicit misunderstandings.
- Hard to write motivation for requirements with no real value.
- Motivation helps to communicate.

Scenarios and Examples

- Add examples which all stakeholders can agree on and use to discuss the details of a requirement, before implementation and test.
- Possible for any stakeholder to add new scenarios.
- Strongly supports communication of requirements.

Example Based on Jbehave & Stories

Name	Description	Extend	Tool	Example
Epic	Describes feature	multiple sprints	Jira/Confluence	Register time
Story	Describes one aspect/benefit of a feature	single sprint	Jira/Confluence	In order to invoice my customer As a carpenter I want to log time spent on an order
Scenario	Example of the aspect of the feature	single day	Git/Jbehave	Order canceled: Given a canceled order When I try to register time Then the canceled order should not be available
Step	Building block of example	one minute	git/Jbehave	Given a canceled order
StepImpl	Code to execute for step	one minute	git/Jbehave	@Given(“a canceled order”) void setupCanceledOrder() {...}

Epics

- Epics are large User Stories. A team often begins a Product Backlog with a set of Epics.
- An Epic could be: The website needs a robust authentication mechanism so users have secure access to their data.
- This would later be broken down into individual stories that allow login, forgot password and other authentication type Features.
- Epics are less defined than smaller stories. They appear later in the backlog and have estimates larger than can be accomplished in a single Sprint.
- It is sometimes not readily apparent when a story is an epic. We may start with what we we think is a normal story and later realize it is an epic. As we perform Release Planning these situations will become apparent.
- We need to split Epics into smaller stories that can be completed in a single Sprints.

Stories (Features)

- A User Story is defined as: a brief statement of intent that describes something the system needs to do for the user.
- It was first used in Extreme Programming, but was made most popular as part of Scrum.
- A story is written in the voice of the user and follows the pattern:

  As a <role> I want <activity, outcome> so that <business value>.

- It is critical to maintain this user centric perspective. These are normally business centric ideas and typically are not technical.
- A User Story also contains *Acceptance Criterion*.
- These are a number of statements that articulate tests for the story.
- This is a very good way of further describing what a user story does.

Scenarios

- Describes a certain aspect of a Story.
  - Main success scenario.
  - Variations.
  - Exceptions
- A user story has usually one or more scenarios.

Story and Scenarios (Example)

Story: Returns go to stock

In order to keep track of stock

As a store owner

I want to add items back to stock when they're returned.

Scenario 1: Refunded items should be returned to stock

Given that a customer previously bought a black sweater from me

And I have three black sweaters in stock.

When he returns the black sweater for a refund

Then I should have four black sweaters in stock.

Scenario 2: Replaced items should be returned to stock

Given that a customer previously bought a blue garment from me

And I have two blue garments in stock

And three black garments in stock.

When he returns the blue garment for a replacement in black

Then I should have three blue garments in stock

And two black garments in stock.

Examples

- A scenario can be exemplified by several examples.
- E.g., examples for input and expected output for each scenario.
- Define the most general case.
- But also include corner cases.
- Add exceptions and special cases to refine.
- In particular later when bugs or special situations are discovered.

Traceability

- In software development; the term traceability (or Requirements Traceability) refers to the ability to link product requirements back to stakeholders' rationales and forward to corresponding design artifacts, code, and test cases.
- Traceability supports numerous software engineering activities such as change impact analysis, compliance verification or traceback of code, regression test selection, and requirements validation.
- It is usually accomplished in the form of a matrix created for the verification and validation of the project.
- High level requirements need to be broken down to implementable tasks.
- Every task needs to be traced to its originating requirement.
- Every code needs to be traced to it originating task/requirement.
- Every change needs to be traced to its origin and implementation.

. and Testing . and Testing

Requirements Tools

- Word, Excel, Google Docs
- Confluence, Any Wiki
- Redmine, Youtrack, Jira, Trac, etc.
- Confluence and Jira have a good integration for traceability etc.

Testing

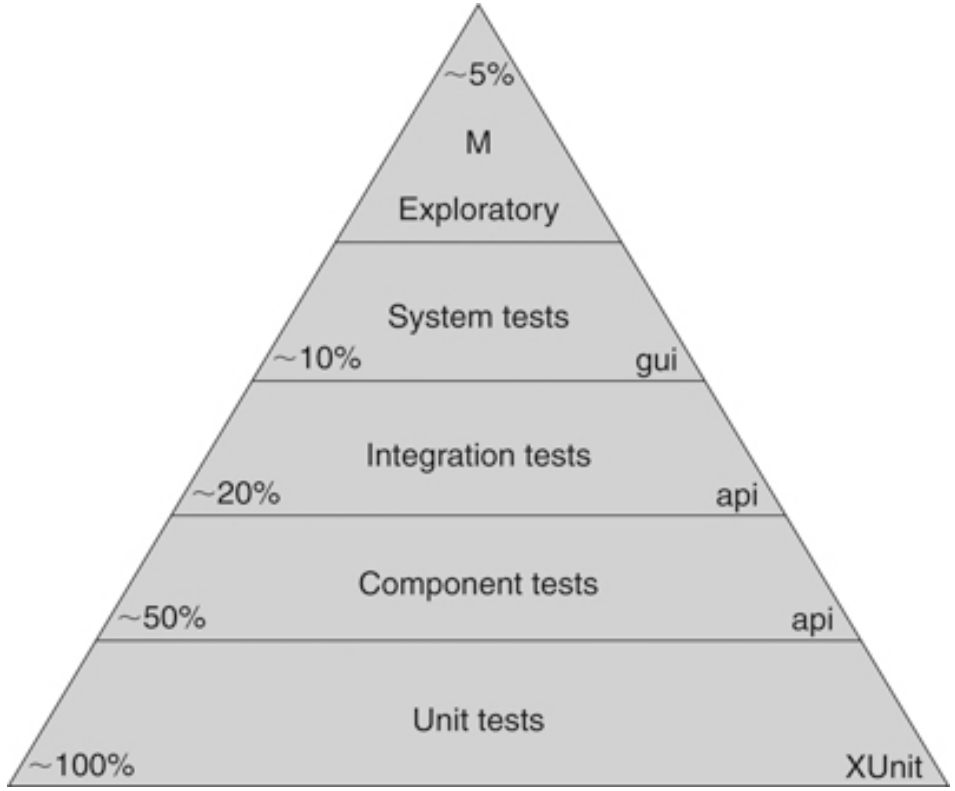
Testing

- Testing is the art of verifying requirements.
- If the requirements are the tests, a lot of testing is already done before implementation.
- Tester can focus on exploratory testing.
- Get the testers mindset and input at a very early stage.

Advantages of Automated Tests

- Adds regression tests, guarding existing functionality and requirements.
- Makes conflicting requirements impossible.
- High degree of confidence as a result of high code coverage.
- Impossible to add code not related to a requirement.
- Adds version control of requirements, supporting branching, merging and releasing just like the code.

Test Automation Pyramid



Unit Tests

- Verify the functionality of a specific section of code.
- Usually at the class level (OOP), and the minimal unit tests include the constructors and destructors.
- These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected.
- One function might have multiple tests, to catch corner cases or other branches in the code.
- Unit testing alone cannot verify the functionality of a piece of software, but rather is used to ensure that the building blocks of the software work independently from each other.

Component Tests

- Similar to unit tests but on component level.
- Verify the functionality of a specific section of code.
- These types of tests are usually written by developers as they work on code (white-box style, but also black-box), to ensure that the specific function is working as expected.
- One component might have multiple tests, to catch corner cases or other branches in the code.
- Component testing alone cannot verify the functionality of a piece of software, but rather is used to ensure that the building blocks of the software work independently from each other.

Integration Tests

- Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design.
- Software components may be integrated in an iterative way or all together ("big bang").
- Normally the former is considered a better practice since it allows interface issues to be located more quickly and fixed.
- Integration testing works to expose defects in the interfaces and interaction between integrated components (modules).
- Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

System Tests

- System testing, or end-to-end testing, tests a completely integrated system to verify that the system meets its requirements.
- For example, a system test might involve testing a logon interface, then creating and editing an entry, plus sending or printing results, followed by summary processing or deletion (or archiving) of entries, then logoff.

Exploratory Tests

- Exploratory testing seeks to find out how the software actually works, and to ask questions about how it will handle difficult and easy cases.
- The quality of the testing is dependent on the tester's skill of inventing test cases and finding defects.
- The more the tester knows about the product and different test methods, the better the testing will be.
- The main advantage of exploratory testing is that less preparation is needed, important bugs are found quickly, and at execution time, the approach tends to be more intellectually stimulating than execution of scripted tests.
- Disadvantages are that tests invented and performed on the fly can't be reviewed in advance (and by that prevent errors in code and test cases), and that it can be difficult to show exactly which tests have been run.

Testing Tools

- jbehave, selenium, helium, h2db to test end-to-end of the whole system.
- Junit (or other unit testing frameworks) for unit testing.
- Emma (or similar) for test coverage.
- Git for version control.
- Jenkins etc. for continuous integration.
- Coverage requirement of 90%
- Impossible to merge code if tests fails or coverage too low.

Process/Workflow

1. PO writes Epic (e.g., in Word, Confluence, etc.)
2. PO and team writes Stories (e.g., in Youtrack or Jira)
3. Tester writes Scenarios (e.g., in Jbehave, Junit, etc.)
4. Developer makes scenario pass

Summary

- Well handled Requirements Engineering is core.
- Testing validates that software meets requirements.
- Requirements change and good management is important.
- If requirements are the tests, a lot of testing is already done before implementation.
- Using BDD/TDD techniques, requirements and code can be glued together by automated tests (cf. previous lecture).
- Several testing techniques have to be combined (unit on lowest level to exploratory testing, highest).

Links

- http://agileforall.com/patterns-for-splitting-user-stories/
- http://jbehave.org
- http://emma.sourceforge.net
- https://jenkins.io
- https://en.wikipedia.org/wiki/Behavior-driven_development
- http://www.ryangreenhall.com/articles/bdd-by-example.html