

# ESOF Notes

MIEIC

January 2019

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Software engineering . . . . .	3
1.2	History . . . . .	4
1.2.1	Programming languages . . . . .	4
1.2.2	Important dates . . . . .	4
1.2.3	Famous software failures . . . . .	4
<b>2</b>	<b>Software processes</b>	<b>4</b>
2.1	Process Activities . . . . .	5
2.2	Software process models . . . . .	6
2.3	RUP - Rational Unified Process . . . . .	7
2.3.1	Best Practices . . . . .	7
2.3.2	Phases . . . . .	7
2.3.3	Disciplines . . . . .	7
2.3.4	Basic elements . . . . .	8
<b>3</b>	<b>Agile Methods</b>	<b>8</b>
3.1	Scrum . . . . .	8
3.1.1	Events . . . . .	8
3.1.2	Artifacts . . . . .	9
3.1.3	Roles . . . . .	9
3.1.4	Agile Estimation . . . . .	9
3.2	eXtreme Programming (XP) . . . . .	10
3.2.1	Core Values . . . . .	10
3.2.2	Practices . . . . .	10
<b>4</b>	<b>Requirements Engineering</b>	<b>11</b>
4.1	Requirements engineering process . . . . .	12
4.2	Requirements elicitation techniques . . . . .	13
4.3	System models in requirements engineering . . . . .	13
4.3.1	Use case model . . . . .	14
4.3.2	Domain Modeling . . . . .	14

<b>5</b>	<b>Arquitetural Design</b>	<b>14</b>
5.1	Basic Concepts . . . . .	14
5.1.1	Design Level . . . . .	14
5.1.2	Typical outputs . . . . .	15
5.1.3	Architectural design notations . . . . .	15
5.1.4	Non-functional requirements . . . . .	15
5.2	Architectural Views . . . . .	16
5.2.1	Civil engineering analogy . . . . .	16
5.2.2	4+1 view model of software architecture . . . . .	16
5.3	Component Diagrams . . . . .	17
5.4	Deployment Diagrams . . . . .	19
5.5	Package Diagrams . . . . .	19
5.6	Activity Diagrams . . . . .	20
5.7	Architecural Patterns . . . . .	21
5.7.1	Mode-view-controller (MCV) . . . . .	21
5.7.2	Pipes and filters (data flow) . . . . .	21
5.7.3	Layered architecture . . . . .	22
5.7.4	Repositories (data centric) . . . . .	22
5.7.5	Client-server and n-tier systems . . . . .	23
5.7.6	Design models and design views . . . . .	24
5.8	Process stages . . . . .	24
5.9	Common use cases of UML sequence diagram in detailed design . . . . .	24
5.10	Software Engineering Laws . . . . .	24
5.11	Lei N <sup>o</sup> 3 - Principio fundamental da Arquitetura de Software . . . . .	24
5.12	Lei n <sup>o</sup> 4 - Lei de Arquimedes da Arquitetura de Software . . . . .	25
<b>6</b>	<b>Design and Implementation</b>	<b>25</b>
6.1	Introduction . . . . .	25
6.1.1	Software design and implementation . . . . .	25
6.1.2	Design levels . . . . .	25
6.2	Object-oriented design using the UML . . . . .	25
6.2.1	Process stages . . . . .	25
6.2.2	Design models and design views . . . . .	26
6.2.3	Sequence diagrams (SD) . . . . .	26
6.2.4	State machine diagrams (SMD) . . . . .	26
6.2.5	Interface specification . . . . .	26
6.3	Design patterns . . . . .	26
6.3.1	Pattern elements . . . . .	27
6.4	Implementation issues . . . . .	27
6.4.1	Reuse . . . . .	27
6.4.2	Configuration management . . . . .	27
6.4.3	Host-target development . . . . .	28
6.4.4	Development platform tools . . . . .	28
6.4.5	Integrated development environments (IDEs) . . . . .	28
6.5	Key points (Design) . . . . .	29
6.6	Key points (Implementation) . . . . .	29

<b>7</b>	<b>Software Testing, Verification and Validation</b>	<b>29</b>
7.1	Part I – Software Reviews & Inspections . . . . .	29
7.1.1	Verification versus Validation . . . . .	29
7.1.2	Static and Dynamic V&V Techniques . . . . .	30
7.1.3	Software reviews and inspections . . . . .	31
7.1.4	Efficiency of defect removal methods . . . . .	31
7.1.5	Types of Reviews . . . . .	32
7.1.6	Review Best Practices . . . . .	32
7.1.7	Estimate Missed defects . . . . .	34
7.2	Part II – Software Testing . . . . .	34
7.2.1	Test Concepts . . . . .	34
7.2.2	Test Levels . . . . .	35
7.2.3	Test case design techniques . . . . .	37
7.2.4	Test automation tools . . . . .	38
7.2.5	Test management . . . . .	39
7.2.6	Testing best practices . . . . .	39
7.2.7	Useful Kahoots . . . . .	40
<b>8</b>	<b>Security in Software Development</b>	<b>40</b>
<b>9</b>	<b>Software Evolution</b>	<b>41</b>
<b>10</b>	<b>GitHub</b>	<b>41</b>
10.1	Importance of Issues and Pull Requests . . . . .	41
10.2	ChatOps . . . . .	41
10.3	Advice in Engineering . . . . .	41

# 1 Introduction

## 1.1 Software engineering

The application of a **systematic, disciplined, quantifiable** approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

### SWEBOK knowledge areas

- Software Requirements
- Software Design
- Software Construction
- Software Testing
- Software Maintenance

## 1.2 History

### 1.2.1 Programming languages

- 1956 - Fortran
- 1973 - C
- 1994 - Java

### 1.2.2 Important dates

- 1822 - First mechanical computer created by Charles Babbage
- 1911 - Foundation of IBM
- 1939 - First digital computer - the Atanasoft-Berry computer
- 1960s - SAGE software development process, and the Apollo on-board software MIT team led by Margaret Hamilton
- 1968 - First NATO Software Engineering Conference and Go To Statement Considered Harmful by Edgar Dijkstra
- 1975 - Foundation of Microsoft
- 1977 - Foundation of Oracle
- 1986 - Fred Brooks: No Silver Bullet
- 1995 - Creation of the Unified Modeling Language (UML)
- 2001 - Agile manifesto

### 1.2.3 Famous software failures

- Therac-25 (1985 - 1987)  
At least 5 patients died because of massive overdoses of radiation caused by a software error (race condition)
- Ariane 5 Explosion (1996)  
10 years of development and 7 billion lost in an explosion due to a software error (overflow)

## 2 Software processes

Software engineering - systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software – cost-effective, timely, predictable, high-quality

Software process - structured set of activities required to develop a software system.

### **Following process activities**

- Specification – defining what the system should do;
- Design and implementation – defining the organization of the system and implementing the system;
- Validation – checking that it does what the customer wants;
- Evolution – changing the system in response to changing customer needs.

### **Types of processes**

- Plan-driven: planned in advance and progress is measured against this plan
- Agile: planning is incremental
- Most approaches nowadays combine both methods

### **Why define processes?**

- Efficiency – helps to keep focus and structure
- Consistency - results likely to be similar
- Basis for Improvement - gathering data on your work -> room for improvement

## **2.1 Process Activities**

### **Software specification (or requirements engineering)**

- Requirements elicitation and analysis - What do the system stakeholders require or expect from the system?
- Requirements specification - Defining the requirements in detail
- Requirements validation - Checking the validity of the requirements

**Software design and implementation** Design - design a software structure that realises the specification

- Architectural design – overall structure
- Database design – system data structure
- Interface design – between system components
- Component (or detailed) design – design of each component individually

Implementation - translate the design into an exec. prog.

**Software verification and validation (testing)** the system conforms to its specification (verification) meets the requirements and customer needs (validation).

Testing

- Unit (or component) testing – individual component testing
- Integration testing – testing of interaction between components
- System testing – general testing (performance, usability, etc...)
- Acceptance testing – live testing with data

**Software evolution (or maintenance) – after development**

- Corrective – bug fixing
- Adaptive – adapt to new platforms, technologies
- Perfective – new functionalities

## 2.2 Software process models

Waterfall model (plan-driven) – Separate specification and development

- Inflexible partitioning of the project – hard to respond to changing requirements
- Used for large systems engineering projects where a system is developed at several sites – plan-driven aspect helps with coordination

Incremental development (& delivery) (agile or plan-driven) - Specification, development and validation are interleaved.

- Easier to adapt to changing requirements
- More feedback – reduced risk of failure
- Can be delivered staggered
- Needs constant refactoring (due to the multiple increments)
- Suboptimal reusability

Integration and configuration (agile or plan-driven) - The system is assembled from existing configurable components.

- Reduced costs and risks – less software developed from scratch
- Faster system delivery
- Needs requirement compromises to fit existing components

- Loss of control over the evolution of the used components

Software prototyping - Not actually a model but an approach to cope with uncertainty

- A prototype is an initial version of a system used to demonstrate concepts and try out design options – reduced uncertainty

## **2.3 RUP - Rational Unified Process**

### **2.3.1 Best Practices**

- Develop iteratively
- Manage requirements
- Use component architectures
- Model visually (UML)
- Continuously verify quality
- Manage change

### **2.3.2 Phases**

Each phase has several iterations, that walk through all disciplines.

- Inception  
Define the project scope (understand the problem)
- Elaboration  
Define the solution architecture (understand the solution)
- Construction  
Build the product
- Transition  
Transition the product into the end-users

### **2.3.3 Disciplines**

- Business modeling
- Requirements
- Analysis & design
- Implementation
- Test

- Deployment
- Configuration & change management
- Project management
- Environment

#### 2.3.4 Basic elements

- Role
- Activity
- Artifact
- Workflow & Workflow Detail

## 3 Agile Methods

### 3.1 Scrum

#### SCRUM FRAMEWORK

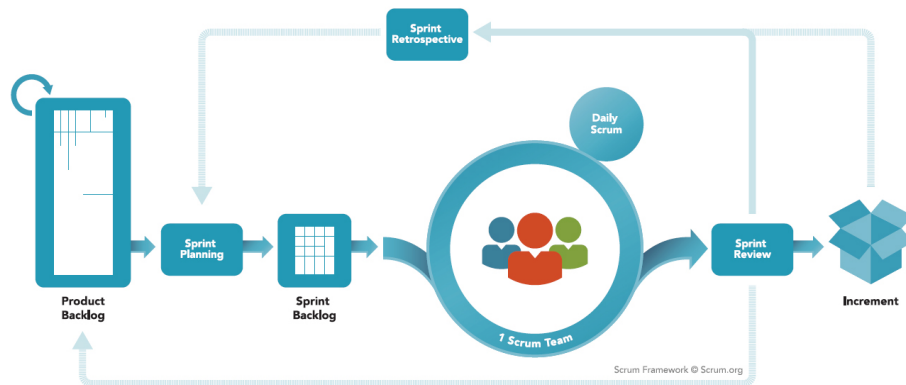


Figure 1: Scrum Overview

#### 3.1.1 Events

- Sprint planning meeting  
Review the features for the next Sprint



- Daily scrum  
Daily stand-up meeting for coordination and commitment among peers
- Sprint review  
The team presents what it accomplished during the sprint
- Sprint retrospective  
Team discusses what they'd like to start/stop/continue doing

### 3.1.2 Artifacts

- Product backlog  
A list of all desired work on the project
- Sprint backlog  
Shows list of tasks and estimates of work remaining (h)
- Sprint burndown chart  
Shows, during a sprint, the total work remaining per day

### 3.1.3 Roles

- Product Owner
  - Define the features of the product and priorities
  - Decide on release date and content
  - Accept or reject work results
- Scrum Master
  - Enact Scrum values and Practices
  - Remove impediments and external interferences
  - Ensure that the team is fully functional and productive
- Development Team
  - Does the work
  - Self-organizing
  - Typically 5-9 people, ideally full time and multifunctional

### 3.1.4 Agile Estimation

**User story** Describes something of value to the user or the system

Example

**As a** student, **I want to** indicate preferences for colleagues to share the same scholar timetable, **so that** I can be more productive in group works.

**Story points** Relative measure for expressing the “size” of a user story, Influenced by difficulty, risk, complexity, etc. Typically exponential.

**Team velocity** The number of story points implemented per Sprint

## 3.2 eXtreme Programming (XP)

Developed by Kent Beck.

### 3.2.1 Core Values

- Communication
- Simplicity
- Feedback
- Courage

### 3.2.2 Practices

- The Planning Game
  1. The customer comes up with a list of desired features, that are aggregated as user stories (similarly to Scrum).
  2. The developers sort them using story points, so as to know which are easier/harder to implement.
  3. Using this information and project velocity (total story points done per iteration), the customer prioritizes which features to implement.
- Small Releases
- System Metaphor
- Simple Design
- Test-driven Development
- Refactoring
- Pair Programming
- Collective Code Ownership
- Continuous Integration
- Sustainable Pace
- On-Site Customer
- Coding Standards

## 4 Requirements Engineering

**What is it?** The process of studying customer and user needs to arrive at a definition of system, hardware, or software requirements (i.e., a property that the software must have)

**Importance** Many defects can be traced to the (requirements) specification. These can be very hard to fix, since they are very structural to the project and, if discovered late, may be spread throughout the project

**Problems** (mis)communication & (mis)understanding

Evolving requirements – requirements creep: uncontrolled changes or continuous growth in a project's requirements

### Levels of software requirements

- Business requirements/needs - high-level objectives vision and scope document
- User requirements/needs- goals or tasks that the user must be able to perform with the product use case models
- System requirements – requirements for the system as a whole (HW/SW) system requirements specification document
- Software requirements – derived from system requirements software requirements specification (SRS) document

**Types of software requirements** Functional requirements (capabilities) - functions that the software is to execute

Nonfunctional requirements - act to constrain the solution

- Mostly quality requirements – Example: The maximum system down-time should be 8 hours per year
- Can also include development process requirements (such as programming languages, etc.)

Quality requirements - quality characteristics, sub-characteristics and metrics

### ISO/IEC 25010 standard

- Functionality suitability – degree to which provides functions that meet stated and implied needs
- Performance efficiency – performance relative to execution conditions

- Reliability – degree to which specified functions are performed under specified conditions for a specified period of time
- Usability - effectiveness, efficiency and satisfaction in a specified context of use
- Compatibility - degree to which information can be exchanged with other products, systems or components
- Maintainability – degree of effectiveness and efficiency during modification
- Portability - degree of effectiveness and efficiency during hardware or software transfer
- Security – degree of information and data protection

**Requirements Engineering and Agile processes** Requirements Engineering mainly goes against the agile methodology to not overplan before development

Solution: user stories - Lightweight way to record a software need, with just enough information

#### 4.1 Requirements engineering process

**Requirements elicitation (or discovery)** Interact with stakeholders and other sources (documents, existing systems, etc.) to collect/discover their requirements

##### Problems

- Problems of scope – ill-defined boundaries, unnecessary information
- Problems of understanding
- Problems of volatility - Requirements evolve over time.

**Requirements analysis (& negotiation)** Detect and resolve problems with the requirements

Checklist

- Completeness
- Consistency
- Unambiguity
- Verifiability
- Necessity
- Feasibility

**Requirements specification** Production a Software Requirements Specification (SRS) document, as well as other docs, Prototypes, Models, etc..

**Requirements validation** Demonstrate that the requirements define the system that the customer really wants. Attempts to prevent the costly errors (as said before) that can happen during requirements

## 4.2 Requirements elicitation techniques

**Interviews - Most widely used requirements elicitation technique**

- Open/unstructured - various issues are explored with stakeholders
- Closed/structured - based on a pre-determined list of questions
- Mixed

**Brainstorming - Useful to elicit new and innovative requirements**

- Moderator and 4-8 people
- Generate new ideas and discuss, review and organize them

Questionnaires (surveys) - Well-suited for confirming/prioritizing previously identified candidate requirements Set of questions

Goal analysis - Hierarchical decomposition of stakeholder goals to derive system requirements

Social Observation and Analysis - Requirements can be derived from the external observation of the routine way and tactics of work

Prototyping - initial/primitive version of a system (cheaper, faster to develop, limited in functionality)

- Throw-away prototypes - focus on requirements rather than implementation constraints
- Evolutionary prototypes - Appropriate for rapid, iterative, application development

## 4.3 System models in requirements engineering

A simplified representation of a system (as-is or to-be) from a certain perspective - tackle complexity through abstraction. In requirements engineering:

- Use case model - for organizing functional requirements
- Domain model - for organizing the vocabulary and information requirements

#### 4.3.1 Use case model

Use case diagram(s) + associated documentation

Purpose: show the system purpose and usefulness, capture functional requirements (through the use cases), specify the system context (actors)

Actors

- user role or external system

Use cases

- Functionality or service as perceived by users

Relationships

- Generalization
- Extension
- Include

#### 4.3.2 Domain Modeling

- Used to organize the vocabulary of the problem domain or to capture information requirements
- Represented through UML class diagrams
- Can use integrity constraints (or invariants) associated with classes to restrict valid object states

## 5 Architectural Design

### 5.1 Basic Concepts

Software Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

#### 5.1.1 Design Level

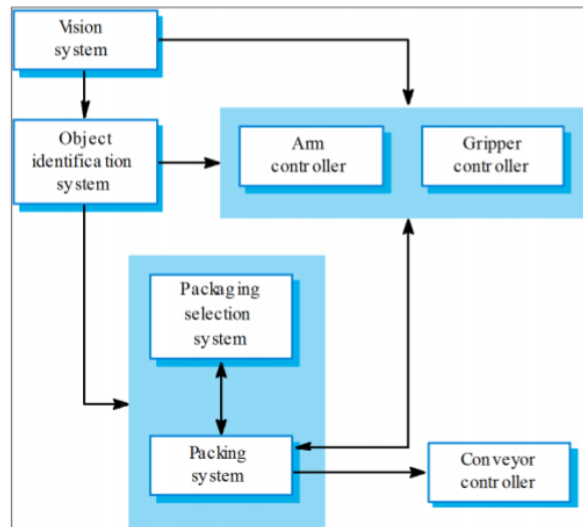
- High-level design or **architectural design**: partition the system into components.
- Detailed design (e.g., object-oriented design): partition each component into classes.
- Design of algorithms and data structures.

### 5.1.2 Typical outputs

"Requirements Capture - "Architectural Design (high-level design)" - "Detailed Design" | "Coding" | "Unit Testing" - "Integration and Testing"

### 5.1.3 Architectural design notations

- Block diagrams



- Informal, but simple and easy to understand.
- The most frequently used for documenting software architectures.
- Lacks semantics and detail.
- Architecture modeling languages (with UML)
  - Semi-formal
  - Multiple views
- Formal architecture description languages (ADLs)
  - Support automated analysis and simulation

### 5.1.4 Non-functional requirements

Architectural design decision are strongly influenced by non-functional requirements:

- Performance - Localize critical operations, minimize communications and levels of indirection

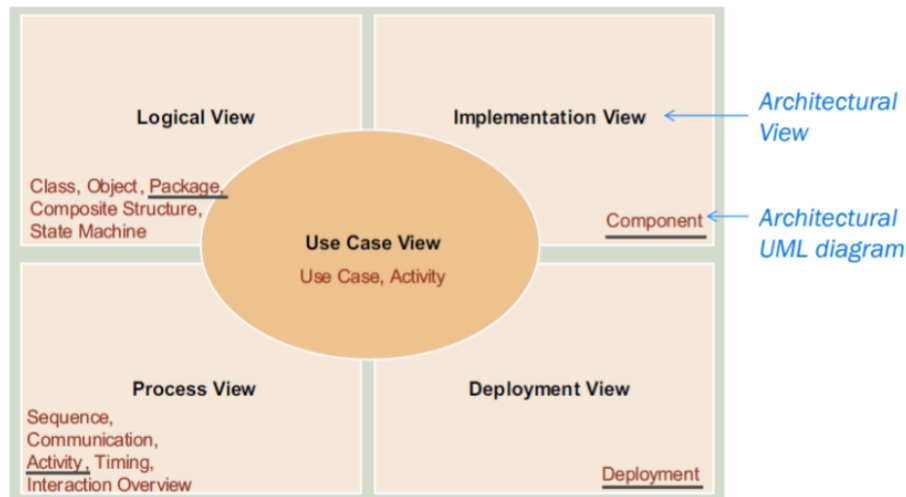
- Security - Use a layered architecture with critical assets in inner layers
- Safety - Localize safety-critical features in a small number of subsystems
- Availability - Include redundant components and mechanisms for fault tolerance
- Portability - Isolate platform dependencies in specific components
- Maintainability - Use fine-grained, loosely coupled, replaceable components.

## 5.2 Architectural Views





### 5.2.1 Civil engineering analogy

Architecture is best described by considering multiple views

### 5.2.2 4+1 view model of software architecture





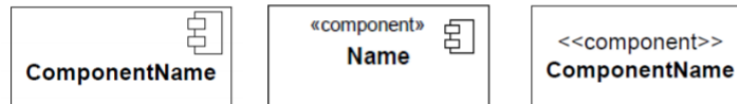
<b>Logical View:</b> <b>Package diagrams</b> (and others)  <p>Shows logical packages* and their relationships</p> <p>*division of responsibilities, independently of allocation to sw components or hw nodes</p>	<b>Implementation View:</b> <b>Component diagrams</b>  <p>Shows software components and dependencies among them</p>
<b>Process View:</b> <b>Activity diagrams</b> (and others)  <p>Shows processing steps, data/object stores, data/object-flows, and opportunities for parallelization</p>	<b>Deployment View:</b> <b>Deployment diagrams</b>  <p>Shows hardware nodes, communication relationships and software artifacts deployed on them</p>

Use Case View (+1): Relates the other views.

### 5.3 Component Diagrams

- Components

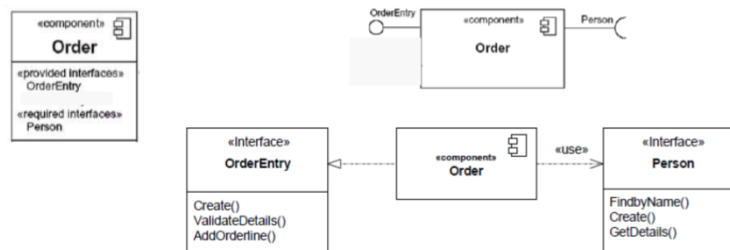
A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.



- Interfaces

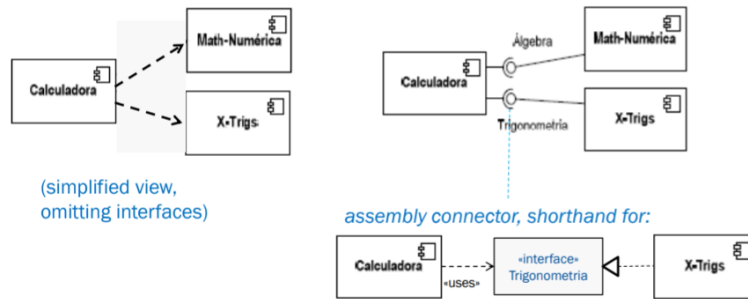
A component defines its behavior in terms of **Interfaces provided (realized)** and **Interfaces required (used)**

Components with the same interfaces are interchangeable.



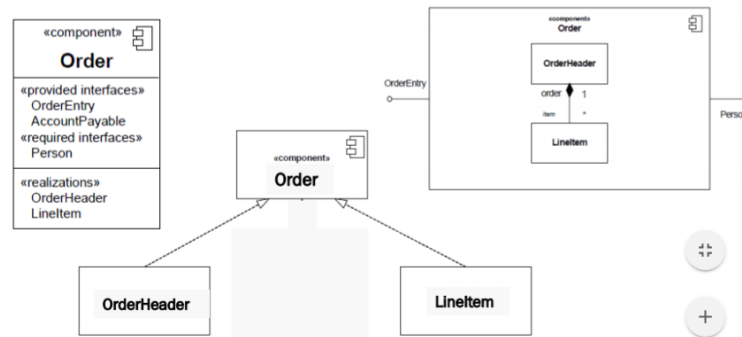
- Dependencies

To promote interchangeability, components should not depend directly on other components but rather on interfaces (that are implemented by other components)



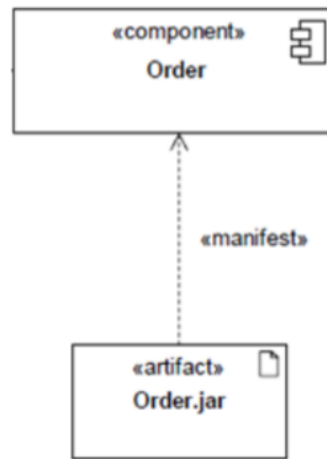
- Components and classes

The behavior of a component is usually realized by its internal classes



- Components and artifacts

Components manifest physically as artifacts (that may be deployed in hardware nodes)



## 5.4 Deployment Diagrams

- **Nodes**

Nodes are computational resources where artifacts may be deployed

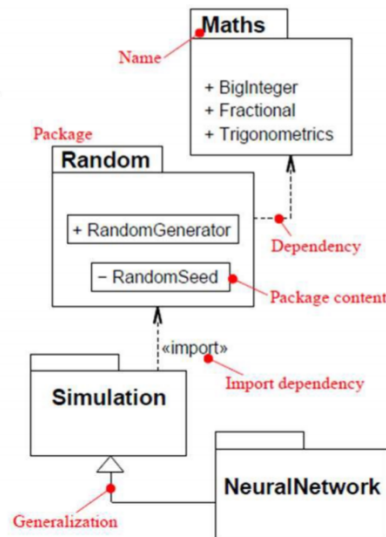
- **Artifacts**

Artifacts are physical information elements used or procedure by a software development process. (example: model files, source code files, executable files, scripts, etc).

## 5.5 Package Diagrams

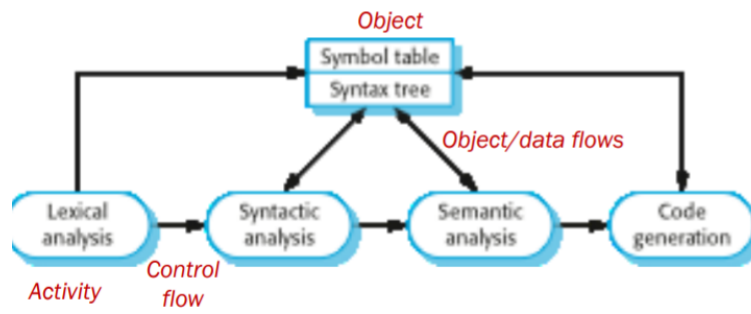
### Package diagrams

- Packages are a grouping mechanism in UML
- They may group elements of any type (even other packages)
- For the logical architecture, packages typically group classes
- May have stereotypes
  - «system»
  - «subsystem»
  - «layer», etc.



## 5.6 Activity Diagrams

### Compiler Architecture



## 5.7 Architectural Patterns

- Patterns are a means commonly used in software engineering of representing, sharing and reusing knowledge.
- A pattern describes a proven solution for a recurring problem in a context "Pattern = (Context, Problem, Solution)"
- An architectural pattern (or architectural style) is a stylized description of good architectural design practice, which has been tried and tested in different environments.

### 5.7.1 Mode-view-controller (MCV)

For interactive processing Separates presentation (**V**) and interaction (**C**) from the application data/state (**M**).

Example: Ruby on Rails.

#### When used:

- There are multiple ways to view and interact with data.
- Future requirements for interaction and data presentation are unknown.

#### Advantages:

- Allows the data and its representation to change independently.
- Supports presentation of the same data in different ways.
- Changes made in one data representation are shown in all of them.

#### Disadvantages:

- Code overhead for simple data model and interactions.

### 5.7.2 Pipes and filters (data flow)

- For batch processing

Organizes the system as a set of data processing components (filters), connected so that data flows between components for processing (as in a pipe).

Example: Compiler Architecture, Test Generation Tool.

#### When used:

- In data processing applications (both batch- and transaction based) where inputs are processed in separate stages to generate outputs.

#### Advantages:

- Easy to understand and supports transformation reuse.

- Workflow style matches the structure of many business processes.
- Evolution by adding transformations is straightforward.
- Can be implemented as either a sequential or concurrent system.

**Disadvantages:**

- Format for data transfer has to be agreed upon.
- Possible overhead in parsing/unparsing input/output data.
- Not really suitable for interactive systems

### 5.7.3 Layered architecture

For complex systems with functionalities at different levels of abstraction

Organizes the system into a set of layers, each of which groups related functionality and provides services to the layer above. (Strict, Relaxed)

Example: Three-Layered Services Application, CASE Tool.

**When used:**

- When building new facilities on top of existing systems.

**Advantages:**

- Supports the incremental development layer by layer.
- Lower layers provide isolation from system/platform specificities.

**Disadvantages (strict layering):**

- Providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers.
- Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

### 5.7.4 Repositories (data centric)

For accessing AND manipulating shared data by multiple subsystems

All data in a system is managed in a central repository that is accessible to all system components or subsystems. Components or subsystems do not interact directly, only through the repository. (Variants: passive, active)

Example: IDE.

**When used:**

- In systems in which large volumes of information are generated that have to be shared and/or stored for a long time.
- In data-driven systems where the inclusion of data in the repository triggers an action or tool (active repository).

**Advantages:**

- All data can be managed consistently as it is all in one place.
- Components can be independent (don't need to know each other).
- Changes made by one component can be propagated to all others.

**Disadvantages:**

- The repository is a single point of failure for the whole system.
- Possible inefficiency in having all communication through the repos.
- Distributing the repos. across several computers may be difficult.

### 5.7.5 Client-server and n-tier systems

For accessing shared data and resources from multiple locations

Asymmetrical distributed system in which clients request services from servers through a shared network or middleware.

N-tier systems are a generalization of client-server (2-tier) systems, in which servers may in turn act as clients. The tiers may also be implemented on a single computer.

Example: Film Library, Four-Tiered Web Application.

**When used:**

- When shared databases or other resources have to be accessed from a range of locations.
- Because servers can be replicated, may also be used when the load on a system is variable.

**Advantages:**

- Servers can be distributed and replicated across a network.
- General functionality (e.g., printing) can be available to all clients.

**Disadvantages:**

- Each service is a single point of failure so susceptible to denial of service attacks or server failure.

- Performance may be unpredictable because it depends on the network as well as the system.
- Possible management problems if servers are owned by different organizations.

#### 5.7.6 Design models and design views

An object-oriented design model may in general cover 4 inter-related design views, represented by means of appropriate UML diagrams. (**External, Internal, Static, Dynamic**)

### 5.8 Process stages

- There are a variety of different object-oriented design processes that depend on the organization using the process.
- Common activities in these processes include:
  - Define the system context and use cases;
  - Design the system architecture;
  - Identify the principal object classes in the system;
  - Develop design models;
  - Specify object interfaces (API).

### 5.9 Common use cases of UML sequence diagram in detailed design

- show interactions between the system and its environment
- show internal interactions between objects the system
- show a dynamic view of the system.

### 5.10 Software Engineering Laws

#### 5.11 Lei N<sup>o</sup>3 - Principio fundamental da Arquitetura de Software

Qualquer problema de estruturação de software resolve-se introduzindo níveis de indireção.

Corolário: Qualquer problema de desempenho resolve-se removendo níveis de indireção.



### 5.12 Lei nº4 - Lei de Arquimedes da Arquitetura de Software

Um sistema de software fundado numa má arquitetura afundar-se-á sob o peso do seu próprio sucesso.

## 6 Design and Implementation

### 6.1 Introduction

#### 6.1.1 Software design and implementation

- It is the stage in the software engineering process at which an executable software system is developed.
- Its activities are invariably inter-leaved.

#### 6.1.2 Design levels

- **High-level design or architectural design:** partition the system into components.
- **Detailed design:** partition each component (or small program) into classes.
- Design of algorithms and data structures

### 6.2 Object-oriented design using the UML

Structured object-oriented design processes involve developing a number of different system models.

Since they require a lot of effort for development and maintenance, these models are only cost-effective for large systems.

#### 6.2.1 Process stages

- Define the system context and use cases;
- Design the system architecture;
- Identify the principal object classes in the system;
- Develop design models;
- Specify object interfaces (API).

### 6.2.2 Design models and design views

An object-oriented design model may in general cover 4 inter-related design views.

		Static (or Structural)	Dynamic (or Behavioral)
refinement ↓	External	Class diagram (API, interfaces)	Sequence diagram (external interactions)
	Internal	Class diagram (all except private features)	Sequence diagram (inter-object interactions)
		Class diagram (private features per class)	State machine diagram (internal behavior per class)

### 6.2.3 Sequence diagrams (SD)

Sequence diagrams show the sequence of object interactions that take place. (usually in the context of a system use case).

### 6.2.4 State machine diagrams (SMD)

- State machine diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests;
- State machine diagrams are useful high-level models of a system or an object's run-time behavior;
- You don't usually need a state machine diagram for all of the objects in the system because most objects in a system are simple.

### 6.2.5 Interface specification

- Object interfaces have to be specified;
- Objects may have several interfaces which are viewpoints on the methods provide;
- The UML uses class diagrams for interface specification;

## 6.3 Design patterns

- Way of reusing abstract knowledge about a problem and its solution;
- A pattern is a description of the problem and the essence of its solution;
- It should be sufficiently abstract to be reused in other cases;
- Pattern descriptions usually make use of object-oriented characteristics,
- Any design problem may have an associated pattern that can be applied.

### 6.3.1 Pattern elements

- Name;
- Problem description;
- Solution description. (template for a design solution);
- Consequences (results).

## 6.4 Implementation issues

### 6.4.1 Reuse

Most modern software is constructed by reusing existing components or systems.

**Reuse levels:**

- **The abstraction level:** Don't reuse software directly but use knowledge of successful abstractions;
- **The object level:** Directly reuse objects from a library;
- **The component level:** Components are collections of objects and object classes that you reuse in application systems;
- **The system level:** reuse entire application systems.

**Reuse costs:**

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs;
- The costs of buying the reusable software;
- The costs of adapting and configuring the reusable software components or systems;
- The costs of integrating reusable software elements with each other and with the new code developed.

### 6.4.2 Configuration management

Configuration management is the name given to the general process of managing a changing software system.

Its aim is to support the system integration process.

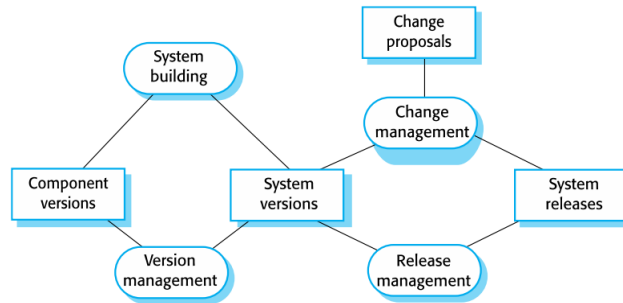
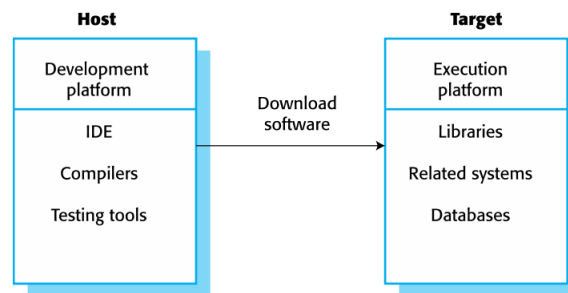


Figure 2: Configuration management tool interaction

### 6.4.3 Host-target development

Most software is developed on one computer (the host), but runs on a separate machine (the target).



### 6.4.4 Development platform tools

- Integrated compiler and syntax-directed editing system that allows you to create, edit and compile code;
- Language debugging system;
- Graphical editing tools;
- Testing tools;
- Project support tools that help you organize the code for different development projects.

### 6.4.5 Integrated development environments (IDEs)

- Software development tools are often grouped to create an integrated development environment (IDE);

- Is a set of tools that supports different aspects of software development;
- IDEs are created to support development in a specific programming language. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE.

## 6.5 Key points (Design)

Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.

The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.

A range of different models may be produced during an object oriented design process. These include static and dynamic models.

Component interfaces must be defined precisely so that other objects can use them.

## 6.6 Key points (Implementation)

When developing software, you should reuse existing software.

Configuration management is the process of managing changes to an evolving software system.

Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.

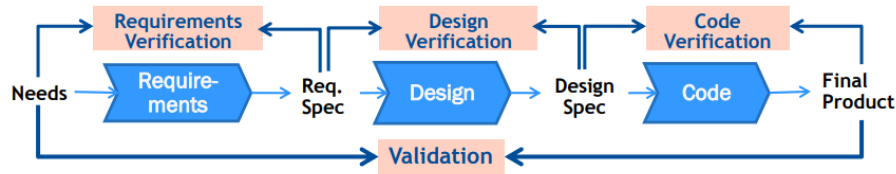
# 7 Software Testing, Verification and Validation

## 7.1 Part I – Software Reviews & Inspections

### 7.1.1 Verification versus Validation

- **Verification – are we building the product right**
  - Ensure (mainly through reviews) that intermediate work products and the final product are “well built”, i.e., conform to their specifications.
- **Validation – are we building the right product?**
  - Ensure (manly through tests) that the final product will fulfill its intended use in its intended environment.
  - Can also be applied to intermediate work products, as predictors of how well the final product will satisfy user needs.

Verification shows conformance with specification. Validation shows that the program meets the customer’s needs.



### 7.1.2 Static and Dynamic V&V Techniques

- **Static Techniques** — involve analyzing the static system representations to find problems and evaluate quality.
  - Reviews and inspections.
  - Automated static analysis (e.g., with lint).
  - Formal verification (e.g., with Dafny)
- **Dynamic Techniques** – involve executing the system and observing its behavior.
  - Software testing.
  - Simulation.

Static verification techniques involve examination and analysis of the program for error detection. Dynamic techniques involve executing the program for error detection.

They are complementary and not opposing techniques. Both should be used during the V&V process.

### 7.1.3 Software reviews and inspections

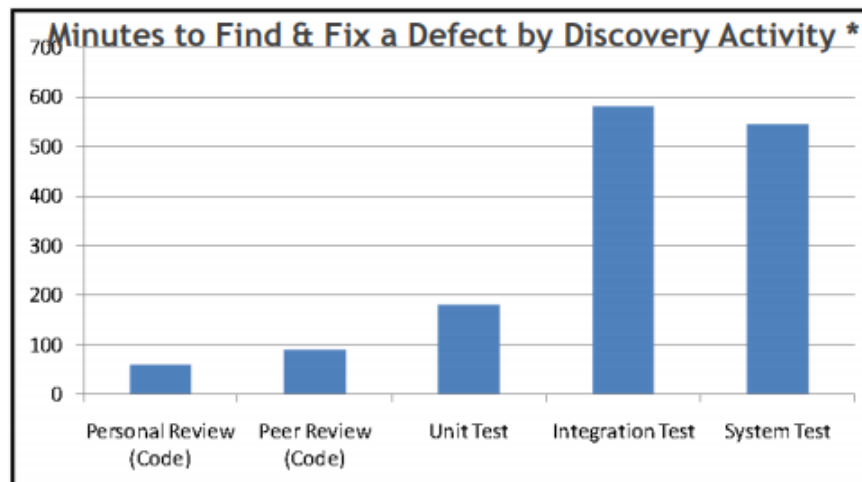
Analysis of static system representations to find problems.

- Manual analysis of requirements specs, design specs, code, etc.
- May be supplemented by tool-based static analysis.

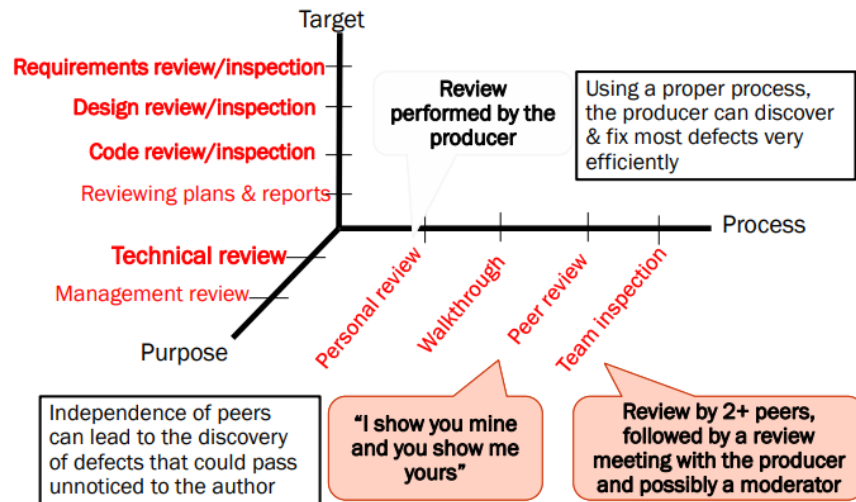
Advantages (as compared to testing):

- Can be applied to any artefact, and not only code
- Can be applied earlier (thus reducing impact and cost of errors)
- Fault localization (debugging) is immediate
- Allows evaluating internal quality attributes (e.g., maintainability)
- Usually more efficient and effective than testing in finding security vulnerabilities and checking exception handling
- Very effective in finding multiple defects
- Peer reviews promote knowledge sharing

### 7.1.4 Efficiency of defect removal methods



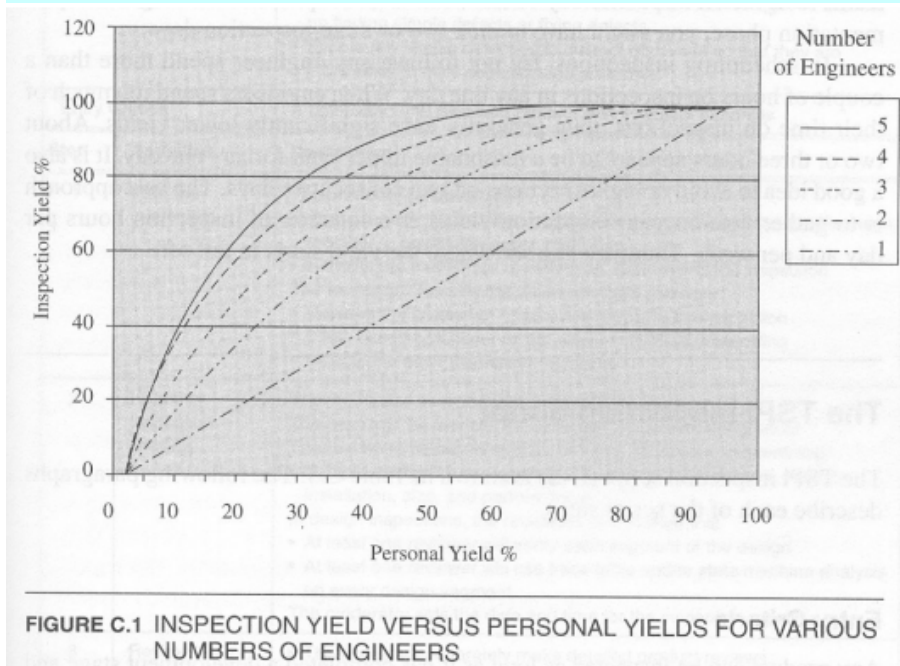
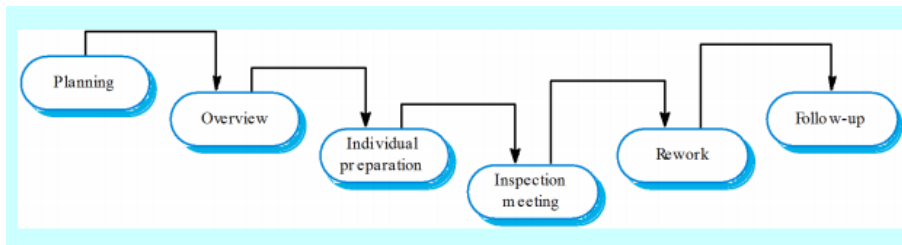
### 7.1.5 Types of Reviews



### 7.1.6 Review Best Practices

- **Use a checklist derived from historical defect data:**
  - Makes the review more effective and efficient, by focusing the attention on the most frequent and important problems.
  - CPersonal checklists make sense, because each person tends to repeat his/her own mistakes.
- **Take enough review time :** 200 LOC/hour is a recommended review rate by some authors(LOC-Lines of Code).
- **Take a break between developing and reviewing (in personal reviews).**
- **Combine personal reviews with peer reviews or team inspections:** Team inspections comprise individual reviews performed by 2+ peers (Individual preparation), followed by a meeting (Inspection meeting) with the producer and possibly a moderator.

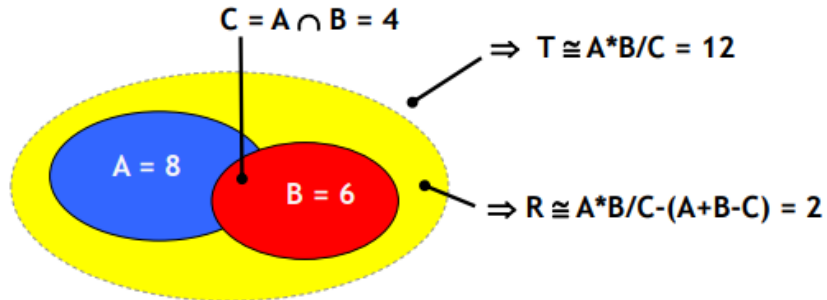




- **Measure the review process & use data to improve:** size, time spent, defects found, defects escaped (found later).

### 7.1.7 Estimate Missed defects

The capture-recapture method is used to estimate the total defects (T) and number of defects remaining (R) based on the degree of overlapping between defects detected by different inspectors (A, B).



In case of more than 2 inspectors, **A** refers to the inspector that found more unique defects, and **B** refers to the union of all other inspectors

## 7.2 Part II – Software Testing

### 7.2.1 Test Concepts

**Testing goals and limitations** Goals:

- exercise the software with defined test cases and observe its behaviour to discover defects.
- increase the confidence on the software correctness and to evaluate product quality.

Limitations:

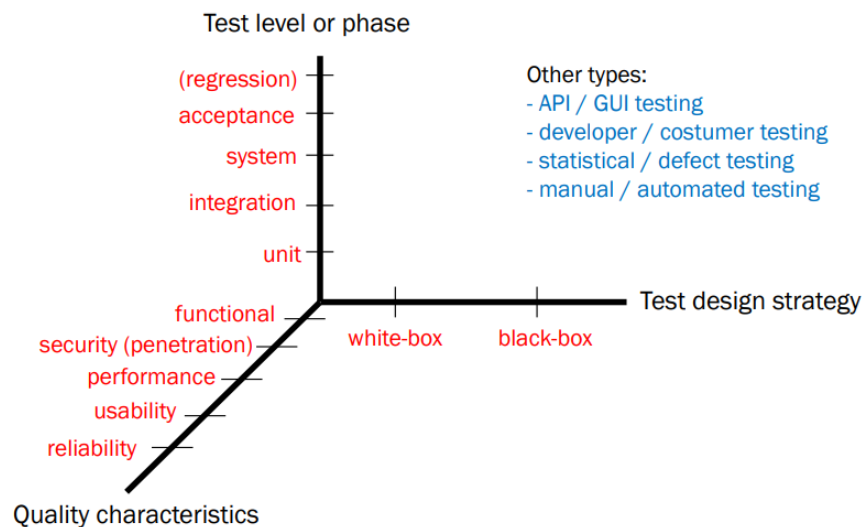
- Testing can show the presence of bugs, not their absence

#### Test Cases

- **Test Case:** A set of test inputs, execution conditions, and expected results developed to exercise a particular program path.
- **Test Script:** concrete definition of test steps / procedure( can be parameterized for reuse with multiple test data).

## Test Activities

- **Test Planning:** define the objectives of testing and the approach for meeting test objectives within constraints imposed by the context.
- **Test monitoring and control:** compare actual progress against the plan, and take actions necessary to meet the objectives of the test plan.
- **Test analysis:** identify testable features and test conditions.
- **Test design:** derive test cases.
- **Test implementation:** create automated scripts.
- **Test execution:** run test suites.
- **Test completion:** collect data from completed test activities.



## Test Types

### 7.2.2 Test Levels

#### Unit Testing/Component Testing/Module Testing

- Testing of individual hardware or software units or groups of related units.
- Detect functional (e.g., wrong calculations) and non-functional (e.g., memory leaks) defects in the unit.
- Usually API testing.

- Responsibility of the developer.
- Usually based on experience, specs and code.

### Integration Testing

- Software and/or hardware components are combined and tested to evaluate the interaction between them.
- Two levels of integration testing:
  - **Component integration testing:** interactions between components.
  - **System integration testing:** interactions between systems.
- Responsibility of an independent test team.
- Usually based on a system spec (technical/design spec).
- Detect defects that occur on the units' interfaces.
- For easier fault localisation, integrate incrementally/continuously.

### System Testing

- Conducted on a complete, integrated system to evaluate the system's compliance with specified requirements.
- Both functional behavior and quality requirements (performance, usability, reliability, security, etc.) are evaluated.
- Usually GUI testing.
- Responsibility of an independent test team.
- Usually based on requirements document.

### Acceptance Testing

- Determine whether or not a system satisfies its acceptance criteria.
- Enable a customer, user, or other authorized entity to determine whether or not to accept the system.
- Usually the responsibility of the customer.
- Based on a requirements document or contract.
- Check if customer requirements and expectations are met.

## Regression Testing

- Tests to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.
- Changes to software, to enhance it or fix bugs, are a very common source of defects.
- Not really a new test level, but just the repetition of testing at any level.

### 7.2.3 Test case design techniques

#### Design goals:

- Create a set of test cases (test suite) that are effective in validation and defect testing.
- A good test suite should have a small/manageable size and have a high probability of finding most of the defects.

**Design Strategies: Black-Box Testing:** Derivation of test cases based on some external specification.

- **Equivalence class partitioning:** partition the input domain into classes of equivalent behavior, separating classes of valid and invalid inputs, and select at least one test case from each class.
- **Boundary value analysis:** select test values at the boundaries of each partition (e.g., immediately below and above), besides typical values.
- **Decision table testing:** test all possible combinations of a set of conditions and actions (each combination corresponding to a business rule).
- **State transition testing:** derive test cases from a state-machine model of the system.
- **Use case testing:** derive test cases from a use case model of the system (with use cases possibly detailed with scenarios, pre/post-conditions, etc).

**White-box Testing:** Derivation of test cases according to program structure.

- **Using coverage analysis tools:** (e.g., EclEmma) to analyse code coverage achieved by black-box tests and design additional tests as needed.
- **Testing statement coverage:** Assure that all statements are exercised.
- **Decision/Branch coverage:** Assure that all decisions (if, while, for, etc.) take both values true and false.

#### 7.2.4 Test automation tools

- **Unit testing frameworks:** JUnit, NUnit.
- **Mock object frameworks:**
  - Facilitate simulating external components in unit testing.
  - EasyMock, jMock.
- **Test coverage analysis tools:**
  - Measure degree of code coverage achieved by the execution of a test suite.
  - Useful for white-box testing.
  - Eclemma, Clover.
- **Mutation testing tools:**
  - Evaluate the quality of a test suite by determining its ability to ‘kill’ mutants (with common fault types) of the program under test.
  - pitest, muJava.
- **Acceptance testing frameworks:**
  - Allows creating test cases by people without technical knowledge.
  - Cucumber, JBehave, Fitnesse.
- **Capture/replay tools (aka functional testing tools):**
  - Capture user interactions in scripts that can be edited and replayed.
  - Useful for GUI testing, particularly regression testing.
  - Selenium, IBM Rational Functional Tester.
- **Performance/load testing tools:**
  - Execute test suites simulating many users and measure system performance.
  - IBM Rational Performance Tester, Compuware QA Load.
- **Penetration testing tools:** Metasploit, ZAP.
- **Test case generation tools:**
  - Automatically generate test cases from models or code.
  - IParTeG (UML), EvoSuite (Java), Spec Explorer (Spec#), Conformiq (UML).

### 7.2.5 Test management

Tools:

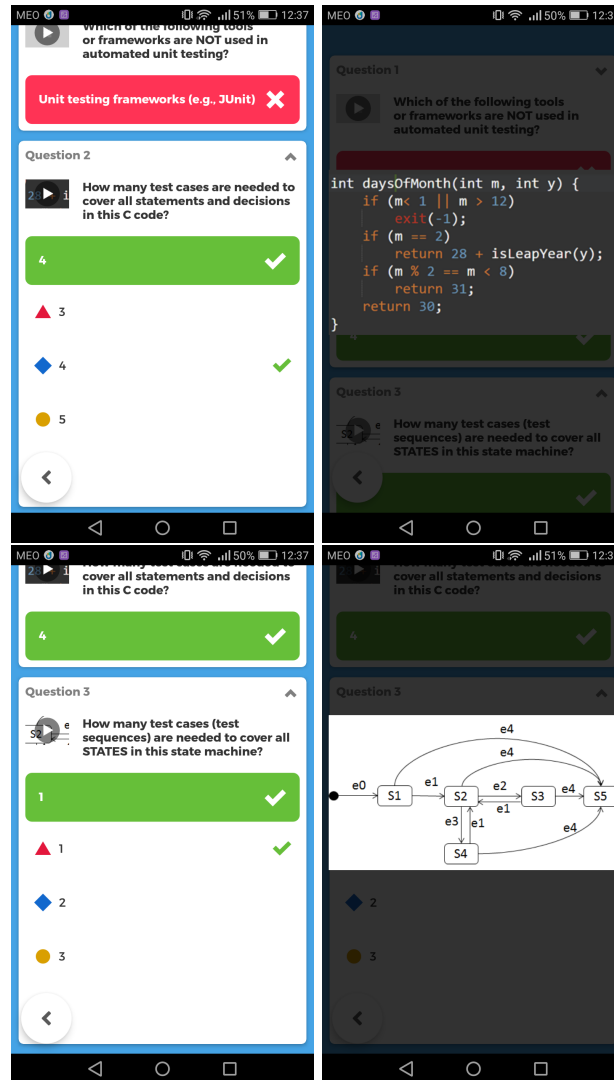
- Manage test information and status.
- Integrate with other management tools: requirements management, project management, bug tracking, configuration management.
- Integrate with test automation tools.
- TestLink.

Test Management charts are used to track progress of testing and bug fixing activities.

### 7.2.6 Testing best practices

- **Test as early as possible** The cost of finding and fixing bugs increases exponentially with time.
- **Automate the tests.**
- **Test first (write tests before the code):** Helps clarifying requirements and specifications since test cases are partial specifications of system behavior.
- **Black-box first:** Start by designing test cases based on the specification and then add any tests needed to ensure code coverage.

### 7.2.7 Useful Kahoots



## 8 Security in Software Development

Hello, here is some text without a meaning. This...



## 9 Software Evolution

Hello, here is some text without a meaning. This...

## 10 GitHub

### 10.1 Importance of Issues and Pull Requests

Before GitHub, open source project developers could receive feedback from multiple means, e.g. forums, e-mail, chat, etc. which means there wasn't a clear way to find and contribute to existing problems.

Issues and Pull Requests come to fix this, by centralizing everything. Take for example this year's third project, where students had to learn how to contribute to open-source projects on GitHub.

The Issues and Pull Requests tabs are incredibly useful since they aggregate the projects current and past issues and the decision making that goes into fixing them, making it easier to contribute to them. This way, it's also easier to find current problems, thus denying the possibility of duplicate threads.

### 10.2 ChatOps

Concept introduced by a GitHub employee. In an example shown in the talk, GitHub has a Slack workspace where they have deployed a bot that can do almost everything they need. From showing cat images to displaying informations about employees, repositories, pull requests, etc.

### 10.3 Advice in Engineering

- Shift left - perform testing earlier in the lifecycle
- Pattern importance
- Readable code & architecture
- Effective technical communication
- Re-use more / invent less
- Automate everything
- Deploy, measure, improve
- Early on, prioritise learning
- Find ways to constantly be challenged
- Understand selling