

ESOF Notes

MIEIC

January 2019

Contents

1	Software processes	2
2	Agile Methods	2
2.1	Scrum	3
2.1.1	Events	3
2.1.2	Artifacts	3
2.1.3	Roles	3
2.2	eXtreme Programming (XP)	3
2.2.1	Core Values	3
2.2.2	Practices	3
3	Requirements Engineering	4
4	Architectural Design	4
4.1	Basic Concepts	4
4.1.1	Design Level	4
4.1.2	Typical outputs	4
4.1.3	Architectural design notations	4
4.1.4	Non-functional requirements	5
4.2	Architectural Views	6
4.2.1	Civil engineering analogy	6
4.2.2	4+1 view model of software architecture	6
4.3	Component Diagrams	7
4.4	Deployment Diagrams	8
4.5	Package Diagrams	9
4.6	Activity Diagrams	9
4.7	Architectural Patterns	10
4.7.1	Mode-view-controller (MCV)	10
4.7.2	Pipes and filters (data flow)	10
4.7.3	Layered architecture	11
4.7.4	Repositories (data centric)	11
4.7.5	Client-server and n-tier systems	12

4.7.6	Design models and design views	13
4.8	Process stages	13
4.9	Common use cases of UML sequence diagram in detailed design .	13
4.10	Software Engineering Laws	13
4.11	Lei Nº3 - Principio fundamental da Arquitetura de Software . . .	13
4.12	Lei nº4 - Lei de Arquimedes da Arquitetura de Software	14
5	Design and Implementation	14
6	Software Testing, Verification and Validation	14
6.1	Part I – Software Reviews & Inspections	14
6.1.1	Verification versus Validation	14
6.1.2	Static and Dynamic V&V Techniques	14
6.1.3	Software reviews and inspections	16
6.1.4	Efficiency of defect removal methods	16
6.1.5	Types of Reviews	17
6.1.6	Review Best Practices	17
6.1.7	Estimate Missed defects	19
6.2	Part II – Software Testing	19
6.2.1	Test Concepts	19
6.2.2	Test Levels	20
6.2.3	Test case design techniques	22
6.2.4	Test automation tools	23
6.2.5	Test management	24
6.2.6	Testing best practices	24
6.2.7	Useful Kahoots	25
7	Security in Software Development	25
8	Software Evolution	26
9	GitHub	26

1 Software processes

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

2 Agile Methods

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

2.1 Scrum

2.1.1 Events

- Sprint planning meeting
- Daily scrum
- Sprint review
- Sprint retrospective

2.1.2 Artifacts

- Product backlog
- Sprint backlog
- Sprint burndown chart

2.1.3 Roles

- Product Owner
- Scrum Master
- Development Team

2.2 eXtreme Programming (XP)

Developed by Kent Beck.

2.2.1 Core Values

- Communication
- Simplicity
- Feedback
- Courage

2.2.2 Practices

- The Planning Game
- Small Releases
- System Metaphor
- Simple Design
- Test-driven Development

- Refactoring
- Pair Programming
- Collective Code Ownership
- Continuous Integration
- Sustainable Pace
- On-Site Customer
- Coding Standards

3 Requirements Engineering

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

4 Architectural Design

4.1 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.

4.1.1 Design Level

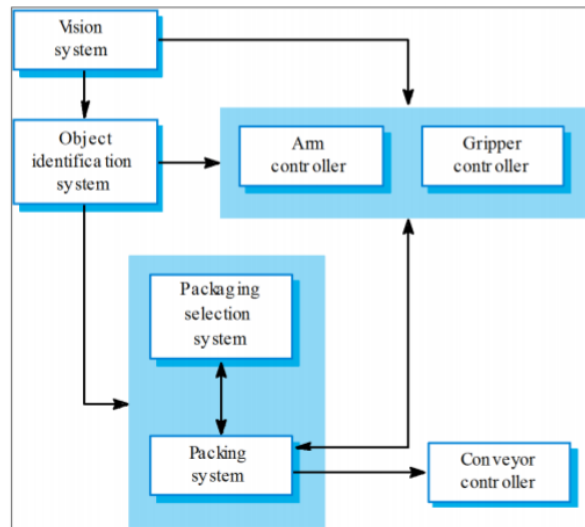
- High-level design or **architectural design**: partition the system into components.
- Detailed design (e.g., object-oriented design): partition each component into classes.
- Design of algorithms and data structures.

4.1.2 Typical outputs

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

4.1.3 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

4.1.4 Non-functional requirements

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

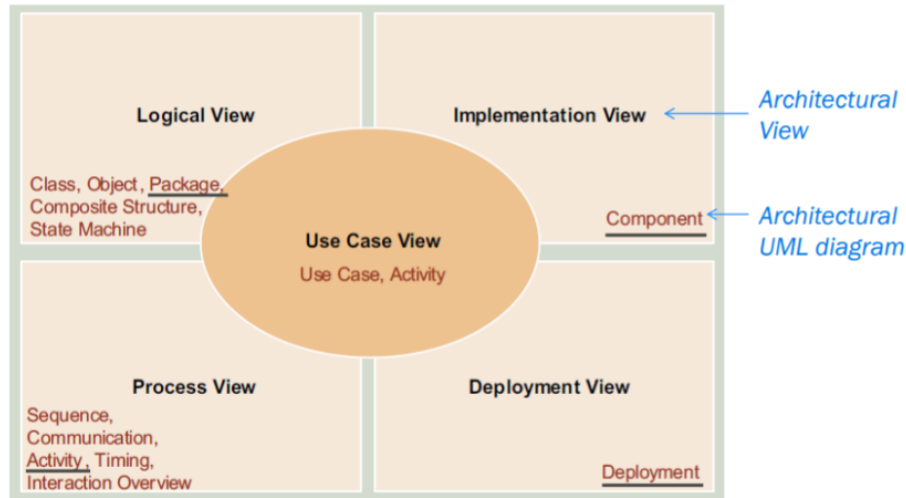
- Performance - Localize critical operations, minimize communications and levels of indirection
- Security - Use a layered architecture with critical assets in inner layers
- Safety - Localize safety-critical features in a small number of subsystems
- Availability - Include redundant components and mechanisms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





4.2 Architectural Views

4.2.1 Civil engineering analogy

Architecture is best described by considering multiple views

4.2.2 4+1 view model of software architecture



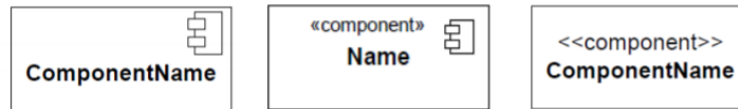
<p>Logical View: Package diagrams (and others) </p> <p>Shows logical packages* and their relationships</p> <p><small>*division of responsibilities, independently of allocation to sw components or hw nodes</small></p>	<p>Implementation View: Component diagrams </p> <p>Shows software components and dependencies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

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

4.3 Component Diagrams

- Components

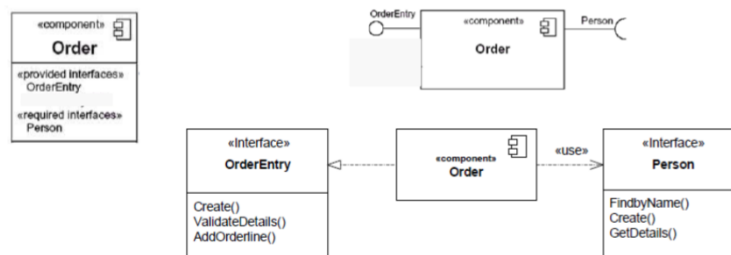
A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.



- Interfaces

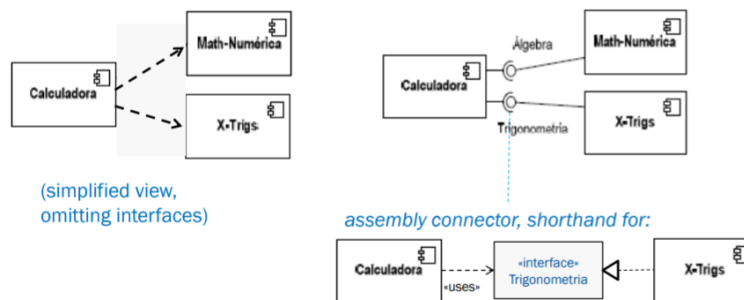
A component defines its behavior in terms of **Interfaces provided (realized)** and **Interfaces required (used)**

Components with the same interfaces are interchangeable.



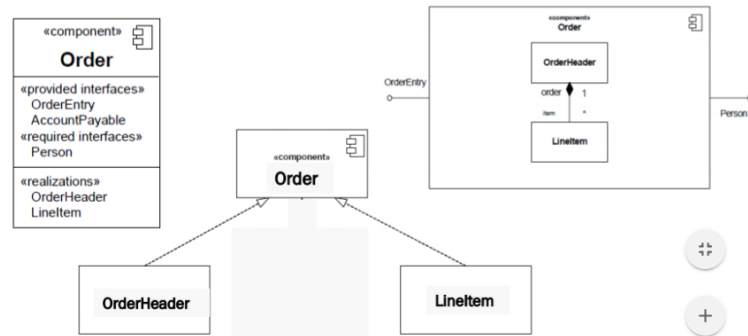
- Dependencies

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

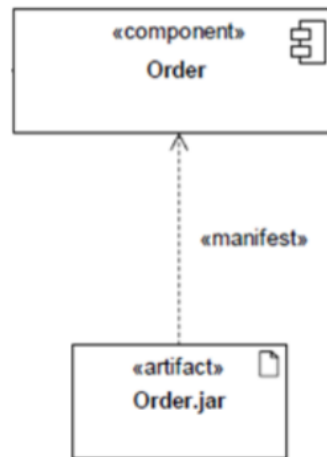


- Components and classes

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)



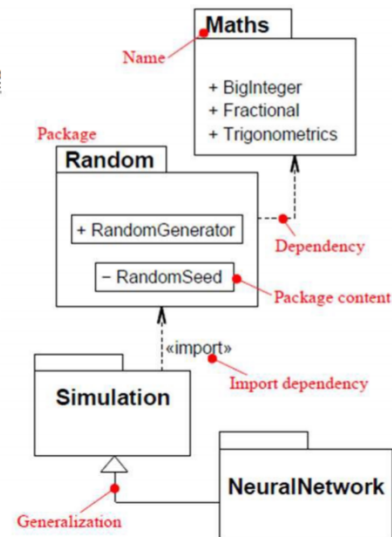
4.4 Deployment Diagrams

- **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).

4.5 Package Diagrams

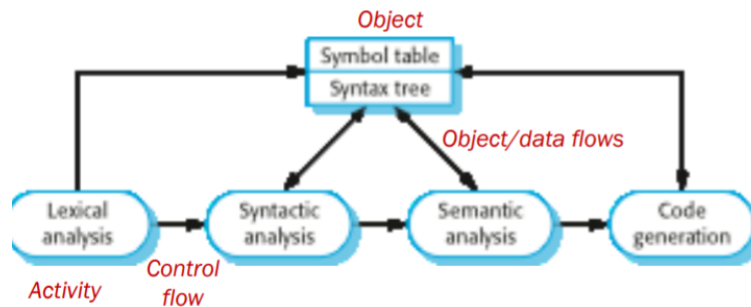
Package diagrams

- 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.



4.6 Activity Diagrams

Compiler Architecture



4.7 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.

4.7.1 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.

4.7.2 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

4.7.3 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.

4.7.4 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.

4.7.5 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.

4.7.6 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**)

4.8 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).

4.9 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.

4.10 Software Engineering Laws

4.11 Lei N^o3 - 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.

4.12 Lei nº4 - Lei de Arquimedes da Arquitetura de Software

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

5 Design and Implementation

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

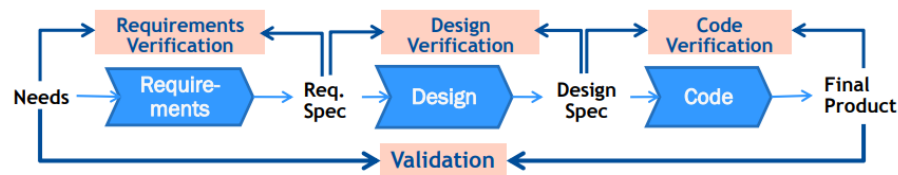
6 Software Testing, Verification and Validation

6.1 Part I – Software Reviews & Inspections

6.1.1 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.



6.1.2 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.

6.1.3 Software reviews and inspections

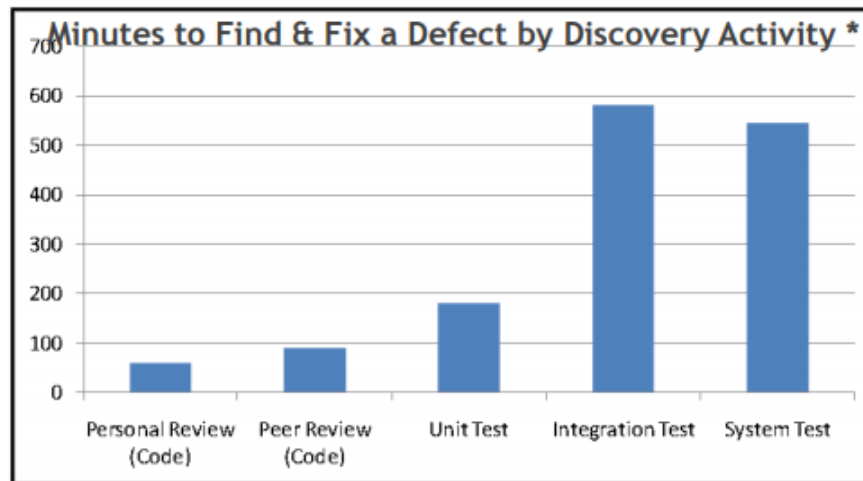
Analysis of static system representations to find problems.

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

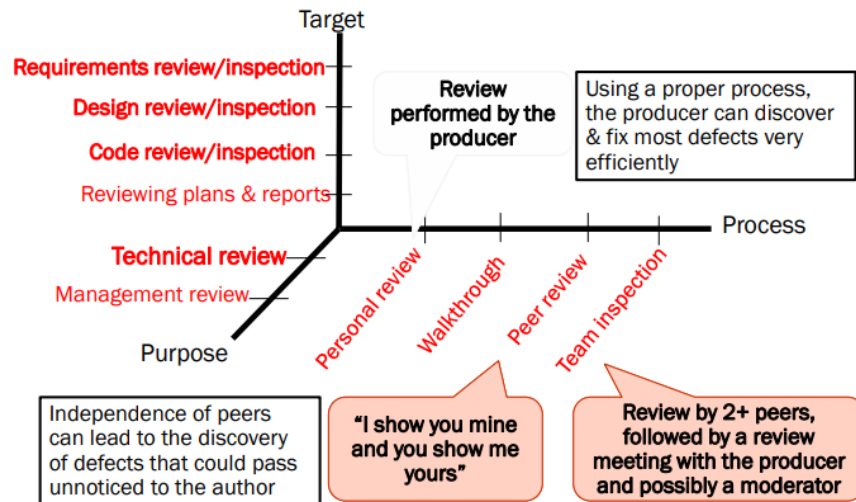
Advantages (as compared to testing):

- 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

6.1.4 Efficiency of defect removal methods

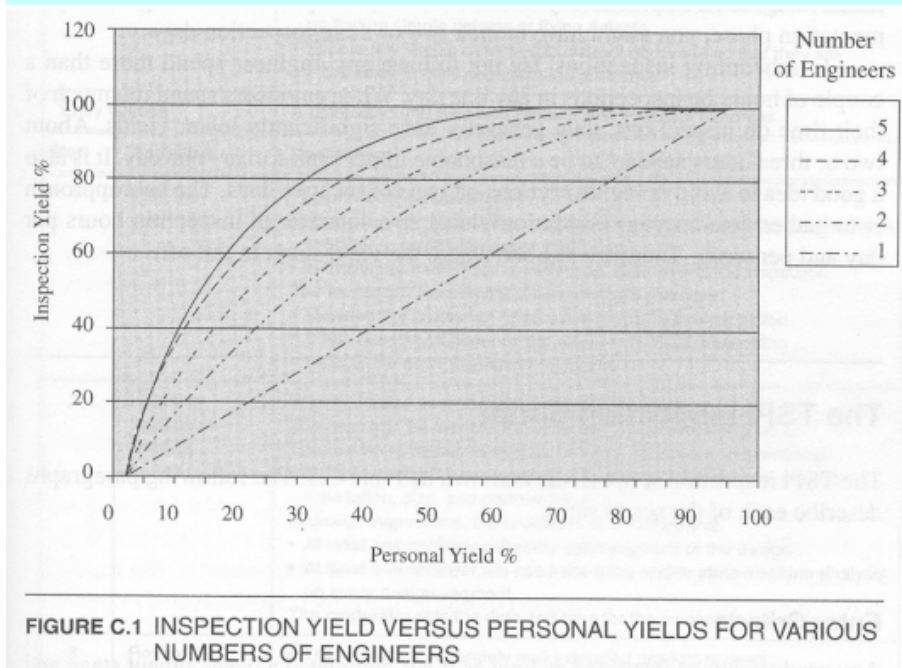
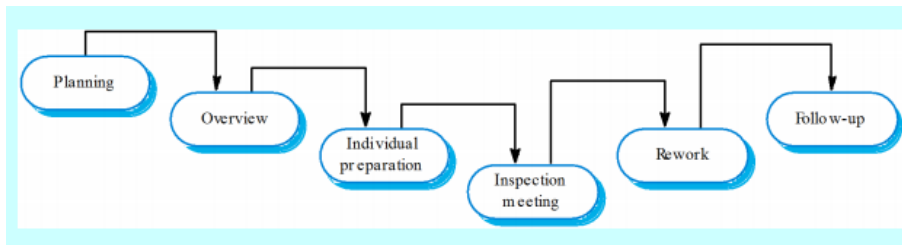


6.1.5 Types of Reviews



6.1.6 Review Best Practices

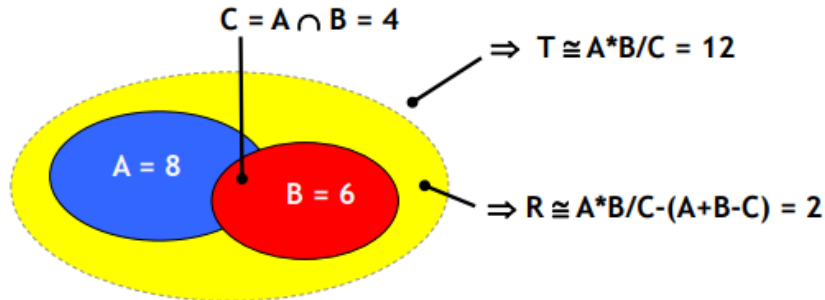
- **Use a checklist derived from historical defect data:**
 - Makes the review more effective and efficient, by focusing the attention on the most frequent and important problems.
 - CPersonal checklists make sense, because each person tends to repeat his/her own mistakes.
- **Take enough review time :** 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).

6.1.7 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

6.2 Part II – Software Testing

6.2.1 Test Concepts

6.2.1.1 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

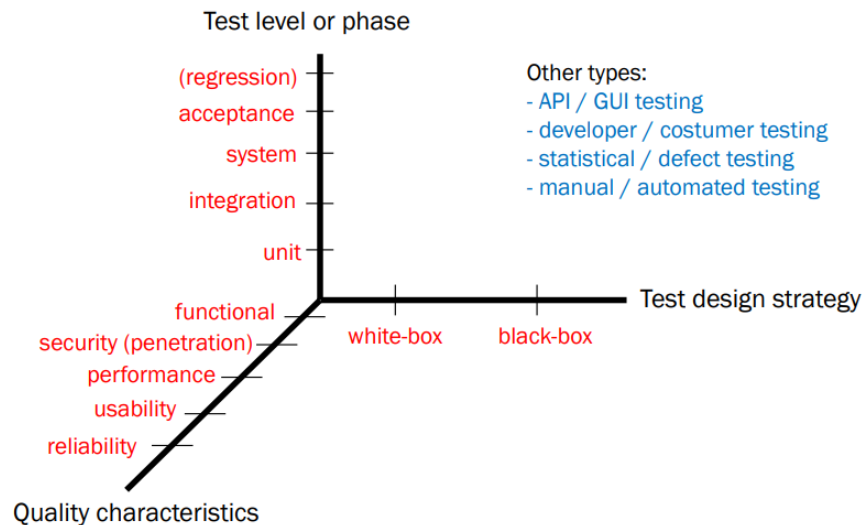
6.2.1.2 Test Cases

- **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).

6.2.1.3 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.

6.2.1.4 Test Types



6.2.2 Test Levels

6.2.2.1 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.

6.2.2.2 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.

6.2.2.3 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.

6.2.2.4 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.

6.2.2.5 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.

6.2.3 Test case design techniques

6.2.3.1 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.

6.2.3.2 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.

6.2.4 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).

6.2.5 Test management

Tools:

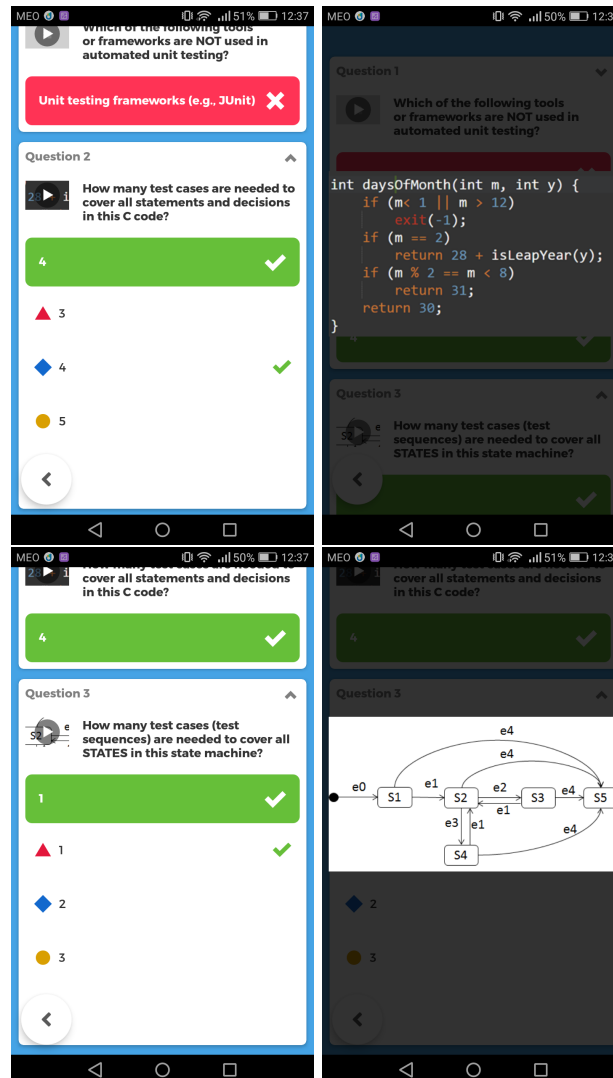
- Manage test information and status.
- Integrate with other management tools: requirements management, project management, bug tracking, configuration management.
- Integrate with test automation tools.
- TestLink.

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

6.2.6 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.

6.2.7 Useful Kahoots



7 Security in Software Development

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

8 Software Evolution

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

9 GitHub

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