Requirements

Requirements are the most critical aspect of the software development cycle.

Requirements Engineering is: the process of establishing the services that the customer requires from a system, and the constraints under which it operates and is developed.

User requirements, statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

System requirements, a structured and detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor. Written for systems and software developers.

Functional requirements, statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. May also state what the system should not do. (NE?)

Non-functional requirements, constraints on the services or functions offered by the system such as timing constraints, constraints on the development process,... Often apply to the system as a whole rather than individual features or services. (NASIL?)

Domain requirements; Constraints on the system from its operational domain. If not satisfied, the system may be unworkable. Requirements are expressed in the language of the application domain, and this is often not understood by engineers developing the system.

Process requirements; Requirements that are specified mandating a particular IDE, programming language, development method or standard.

Specification of Requirements

Unambigious, they should not be interpreted in different ways by developers and users.

Complete, they should include descriptions of all facilities required.

Consistent, there should be no conflicts or contradictions in the descriptions of the system facilities.

Requirement Engineering Process

The processes used for RE vary widely depending on the application domain, the people involved and the organization developing the requirements.

However, there are a number of generic activities common to all processes:

Requirements elicitation and analysis, Requirements specification,

Requirements validation, Requirements management.

1. Requirements Elicitation & Analysis

Technical staff works with customers to find out about:

- **the application domain,**
- **the services that the system should provide,**
- **the system's operational constraints.**

May involve end-users, managers, engineers... These are called **stakeholders**.

Problems of Requirements Elicitation & Analysis

Stakeholders don't know what they really want.

Stakeholders express requirements in their own terms.

Different stakeholders may have conflicting requirements.

Organizational and political factors may influence the system requirements.

The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

2 Requirements specification

Requirements specification is the process of writing the user and system requirements in a **requirements document.**

User requirements have to be understandable by end-users and customers who do not have a technical background.

System requirements are more detailed requirements and may include more technical information.

The requirements may be part of a contract for the system development

It is therefore important that these are as complete as possible.

Scenarios (or user stories)

Scenarios are real-life examples of how a system can be used.

They should include;

A description of the **starting situation,** A description of the **normal flow of events,**

A description of **what can go wrong,** Information about other concurrent activities,

A description of **the state when the scenario finishes.**

Use Cases

Use-cases are a scenario based technique in UML (Unified Modeling Language), which identify actors in an interaction and describe **externally observable interaction between the system and its actors.**

3. Requirement Validation

Concerned with demonstrating that the **requirements define the system that the customer really wants.**

Requirements error costs are high so validation is very important.

Fixing a requirements error after delivery may cost **up to 100 times** the cost of fixing an implementation error.

Checklist For Requirements Validation

Validity. Does the system provide the functions which best support the customer's needs?

Consistency. Are there any requirements conflicts?

Completeness. Are all functions required by the customer included?

Realism. Can the requirements be implemented given available budget and technology?

Verifiability. Can the requirements be verified (e.g. tested)?

Requirements Validation Techniques

Requirements review; Systematic manual analysis of the requirements.

Prototyping; Using an executable model of the system to check requirements.

Test-case generation; Developing tests for requirements to check test-ability.

Checklist For Requirements Review

Verifiability; Is the requirement realistically testable?

Comprehensibility; Is the requirement properly understood?

Trace-ability; Is the origin of the requirement clearly stated?

Adaptability; Can the requirement be changed without a large impact on other requirements?

4. Requirements management

Requirements management is the process of managing changing requirements during the requirements engineering process and system development.

New requirements emerge as a system is being developed and after it has gone into use.

We keep track of individual requirements and maintain **links between dependent requirements** so that you can assess the impact of requirements changes..

Requirements change management

Problem analysis and change specification; During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to change request or who may respond with a more specific requirements change proposal, or decide to withdraw the request.

Change analysis and costing; The effect of the proposed change is assessed using trace-ability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the

requirements change.

Change implementation; Requirements document and, where necessary, system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

System Modeling

System modeling is the process of developing abstract models of a system, with each model presenting a different **view** or **perspective** of that system.

System modeling has now come to mean representing a system using some kind of **graphical notation,** which is now almost always based on notations in the **Unified Modeling Language (UML).**

System modelling helps the **analyst** to understand the functionality of the system and models are used to communicate with customers.

Existing and planned system models

Models of the existing system are used during requirements engineering, re-engineering and reverse-engineering .

They help analyst clarify what existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.

Re-engineering: Systematic starting over and reinventing the way a firm, a software, or a business process, work (code re-factoring in our context)

Reverse engineering: taking apart an object to see how it works in order to duplicate or enhance the object. The practice, taken from older industries, is now frequently used on computer hardware and software.

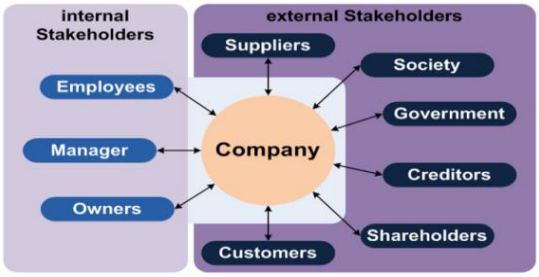
Re-engineering **!=** Reverse engineering

Models of the new system are used during requirements engineering to help explain proposed requirements to other system stakeholders.

Engineers use these models to discuss design proposals and to document system for implementation.

In a **model-driven engineering process,** it is possible to generate a complete or partial system implementation from system model.

Stakeholders of a Company



System Perspective

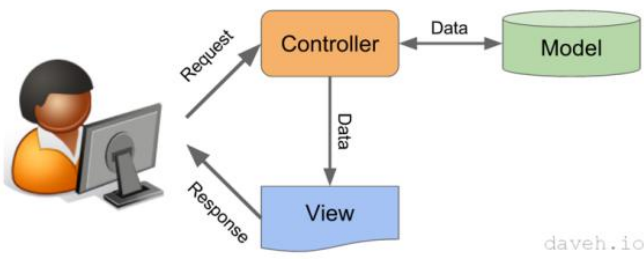
External perspective, where you model context or environment of the system.

Interaction perspective, where you model interactions between a system and its users and/or environment, or between components of a system.

Structural perspective, where you model organization of a system or structure of data that is processed by system.

Behavioral perspective, where you model dynamic behavior of system and how it responds to events.

***M**odel**V**iew**C**ontroller*



Model-driven engineering (MDE) is an approach to software development where models rather than programs are principal outputs of development process.

System Models - 1) Interaction models (Use Case Diagram)

Modeling **user interaction** is important as it helps to identify user requirements.

Modeling **system-to-system interaction** highlights the communication problems that may arise.

Modeling **component interaction** helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

System Models - 2) Structural models (Class-Component-Deployment Diagram)

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.

You create structural models of a system when you are discussing and designing the **system architecture.**

Structural models may be **static models,** which show the structure of the system design, or **dynamic models,** which show the organization of the system when it is executing.

Generalization

Generalization is an everyday technique that we use to **manage complexity.**

Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (trees, animals, cars, houses, etc.) and learn the characteristics of these classes.

This allows us to infer that different members of these classes have some common characteristics. Example; Çam, Kavak, Köknar -> Ağaç.

System Models - 3) Behavioral models (Activity-Use Case-State Machine Diagram)

Behavioral models are models of dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.

You can think of these stimuli as being of two types:

Data: Some data arrives that has to be processed by system.

Events: Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

Data-driven modeling

Many business systems are **data-processing systems** that are primarily driven by data. They are controlled by data input to the system, with relatively little external event processing.

Data-driven models show the sequence of actions involved in processing input data and generating an associated output.

They are particularly useful during analysis of requirements as they can be used to show **end-to-end processing in a system**.

Event-driven modeling

Real-time systems are often event-driven, with minimal data processing.

For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone.

Event-driven modeling shows how a system responds to **external** and **internal events.**

It is based on the assumption that a system has a finite number of states and that **events (stimuli)** may cause a transition from one state to another.

State-machine modeling

These model the behaviour of the system in response to external and internal events.

They show the system's responses to stimuli so are often used for modelling real-time systems. State machine models show **system states** as **nodes** and events as arcs between these nodes. When an event occurs, the system moves from one state to another. State-charts are an integral part of the UML and are used to represent state machine models.

Model-Driven Engineering

Model-driven engineering (MDE) is an approach to software development where **models** rather than programs are principal **outputs of development process.**

Programs that execute on a hardware/software platform are then **generated automatically** from models.

Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Usage of Model-Driven Engineering (MDE)

MDE is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.

Pros

Allows systems to be considered at higher levels of abstraction

Generating code automatically means that it is cheaper to adapt systems to new platforms.

Cons

Models for abstraction and not necessarily right for implementation.

Savings from generating code may be outweighed by costs of developing translators for new platforms.

Model-Driven Architecture (MDA)

Model-driven architecture (MDA) was the precursor of more general model-driven engineering

MDA is a **model-focused approach** to software design and implementation that uses a subset of UML models to describe a system.

Models at different levels of abstraction are created. From a high-level, **platform independent model,** it is possible, in principle, to generate a working program without manual intervention.

Process Models

Context models simply show the other systems in the environment, not how the **system under development (SuD)** is used in that environment.

Process models reveal how **SuD** is used in broader business processes.

UML activity diagrams may be used to define business process models.

Software Architecture

The design process for identifying the major system components making up a system and their control and communication is **architectural design.**

An early stage of the design process - Represents the link between specification and design processes.

Often carried out **in parallel** with some specification activities.

The output of this design process is a description of the **system/software architecture.**

Software Design Activities

Architectural design, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships and how they are distributed.

Interface design, interfaces between system components.

Component design, you take each system component and design how it will operate.

Database design, you design the system data structures and how these are to be represented in a database.

Architectural Abstraction

Architecture in the small is concerned with the architecture of individual programs.

At this level, we are concerned with the way that an individual program is decomposed into components.

Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components.

These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Architectural representations

Simple, informal **block diagrams** showing entities and relationships are the **most frequently used** method for documenting software architectures;

- criticized to be very abstract because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- However, useful for communication with stakeholders and for project planning.

Architecture and system characteristics

Performance; Localize critical operations and minimize communications.

Security; Use a layered architecture with critical assets in the inner layers.

Safety; Localize safety-critical features in a small number of sub-systems.

Availability; Include redundant components and mechanisms for fault tolerance.

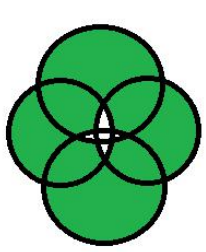
Maintainability; Use fine-grain, replaceable components.

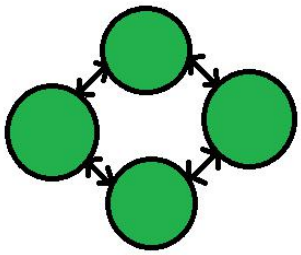
Cohesion & Coupling

Cohesion is the degree to which the elements in a design unit (package, class etc.) are logically related, or "belong together".

Coupling is the degree to which the elements in a design are connected.

<table>
<tr><td rowspan="3">class A
Check();
Validate();
Send();</td><td>class B
Check();</td></tr>
<tr><td>class C
Validate();</td></tr>
<tr><td>class D
Send();</td></tr>
<tr><td>LOW COHESION</td><td>HIGH COHESION</td></tr>
</table>





Tight coupling:
1. More Interdependency
2. More coordination
3. More information flow

Loose coupling:
1. Less Interdependency
2. Less coordination
3. Less information flow

Cohesion & Coupling - A good architecture :

Maximizes the **cohesion** of each module

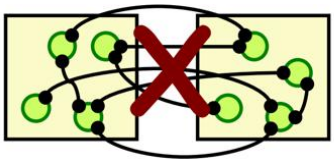
Goal: the contents of each module are strongly inter-related

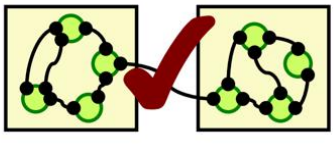
High cohesion means the sub-components really do belong together

Minimizes **coupling** between modules:

Goal: modules don't need to know much about one another to interact

Low coupling makes future change easier.





Architectural Views

Each **architectural model** only shows one view or perspective of the system.

It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network.

4+1 View Model of Software Architecture

A logical view, which shows the key abstractions in the system as objects or object classes. *(Class, Object)*

A process view, which shows how, at run-time, the system is composed of interacting processes. *(Sequence)*

A development view, which shows how the software is decomposed for development. *(Component, Package)*

A physical view, which shows the system hardware and how software components are distributed across the processors in the system. *(Deployment)*

Related **use cases or scenarios** (+1)

Architectural Pattern

In software engineering, a pattern is a general repeatable solution to a commonly occurring problem in software engineering.

Patterns are a means of representing, sharing and reusing knowledge.

An **architectural pattern** is a stylized description of good design practice, which has been tried and tested in different environments.

Patterns should include information about when they are and when they are not useful. Patterns may be represented using tabular and graphical descriptions.

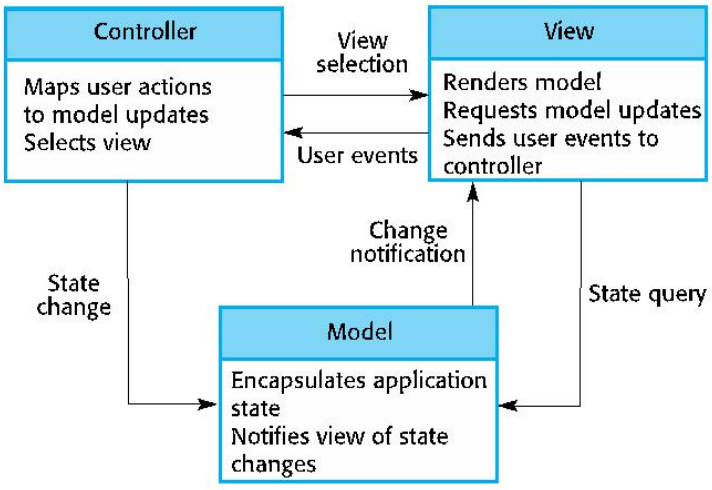
Review of several Architectural patterns:

1. Model-View-Controller (MVC) pattern

Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. **Model component** manages data and associated operations on that data. **View component** defines and manages how data is presented to user. **Controller component** manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to View and Model.

Advantages; Allows data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.

Disadvantages; Can involve additional code and code complexity when data model and interactions are simple.



2. Layered architecture pattern

Used to model the interfacing of sub-systems.

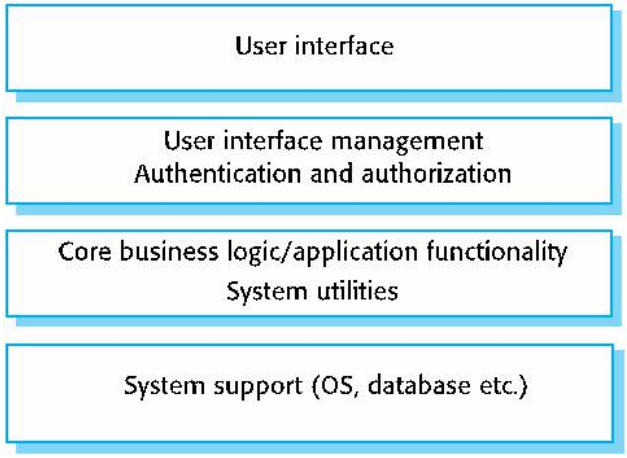
Organizes the system into a **set of layers** (or abstract machines) each of which provide a set of services.

Supports the **incremental development** of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.

However, often artificial to structure systems in this way.

Advantages; Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.

Disadvantages; In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.



3. Repository architecture pattern

Sub-systems must exchange data. This may be done in two ways:

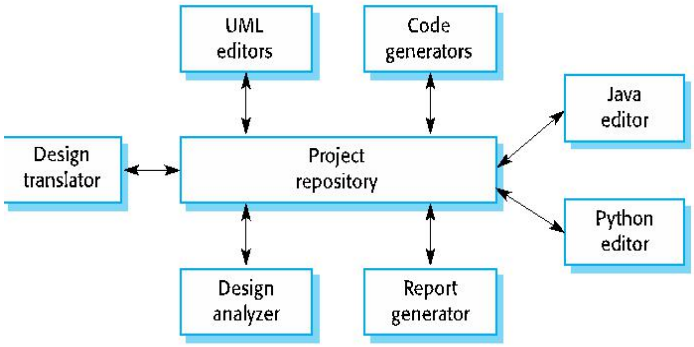
- **Shared data** is held in a **central database** or repository and may be accessed by all sub-systems;
- Each sub-system maintains its own database and passes data explicitly to other sub-systems.

When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an efficient data sharing mechanism.

Advantages; Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.

Disadvantages; The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

A repository architecture for an IDE



4. Client-server architecture pattern

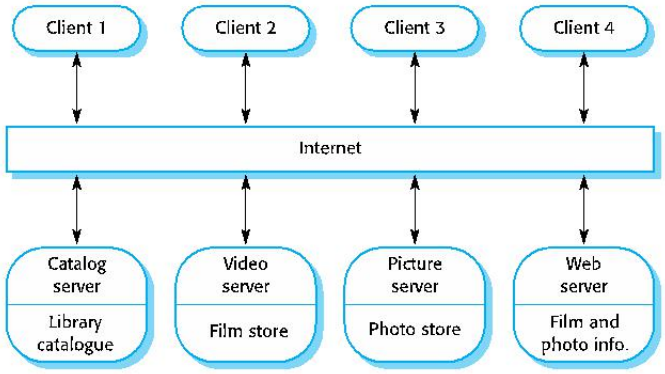
Distributed system model which shows how data and processing is distributed across a range of components.

- Can be implemented on a single computer.

Set of **stand-alone servers** which provide specific services such as printing, data management, etc.

Set of clients which call on these services.

Network which allows clients to access servers.



Advantages; Servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.

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. May be management problems if servers are owned by different organizations.

5. Pipe and filter architecture pattern

Functional transformations process their inputs to **produce outputs.**

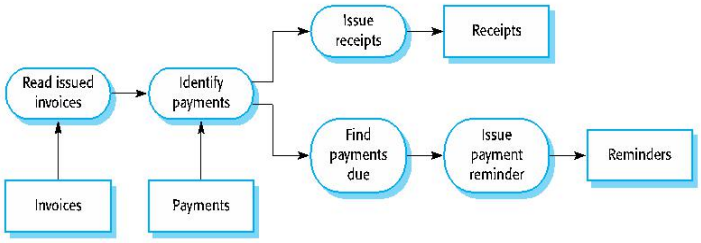
May be referred to as a pipe and filter model **(as in UNIX shell).**

Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.

Not really suitable for interactive systems.

Advantages; Easy to understand and supports transformation reuse. Work-flow 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; The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.



Compiler components

A lexical analyzer, which takes input language tokens and converts them to an internal form.

A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.

A syntax analyzer, which checks the syntax of the language being translated.

A syntax tree, which is an internal structure representing the program being compiled.

A semantic analyzer that uses information from syntax tree and symbol table to check the semantic correctness of the input language text.

A code generator that 'walks' the syntax tree and generates abstract machine code.

Application Types

Data processing applications, data driven applications that process data in batches without explicit user intervention during the processing.

Transaction processing applications, data-centered applications that process user requests and update information in a system database.

Event processing systems, applications where system actions depend on interpreting events from the system's environment.

Language processing systems, applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.

Design Patterns

A design pattern is a 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 different settings.

Pattern descriptions usually make use of object-oriented characteristics such as inheritance and **polymorphism.**

Implementation Issues

1. Reuse: Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.

Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.

An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse Levels

The abstraction level; At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.

The object level; At this level, you directly reuse objects from a library rather than writing the code yourself.

The component level; Components are collections of objects and object classes that you reuse in application systems.

The system level; At this level, you reuse entire application systems.

Reuse Cost

The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.

Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.

The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.

The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Do the Reuse only when: **Costs < Benefits**

2. Configuration management: During the development process, you have to keep track of the many different versions of each software component in a configuration management system.

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

The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

3. Host-target development: Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on

one computer (the host system) and execute it on a separate computer (the target system).

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

Integrated development environments (IDEs)

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

An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.

IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

Open-Source Development

Open-source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process

Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.

Open-source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open-Source Systems; There are many major open-source products, e.g., **the Linux operating system** which is widely used as a server system and, increasingly, as a desktop environment.

Other important Open-source products are **Java, the Apache web server** and **the MySQL database management system.**

Definitions

SW Project Management: The discipline of managing projects to achieve quality within time constraints and budget.

SW Quality (Engineering): The discipline of specifying, assuring, monitoring, and controlling the quality of software products.

SW Verification: Did we build the product right?

Ensures that the product satisfies or matches the original design (low-level checking). This is usually done through white-box testing.

SW Validation: Did we built the right product?

Checks that the product design satisfies or fits the intended usage (high-level checking). This is usually done through black-box testing.

SW Testing: Techniques to execute programs with the intent of finding as many defects as possible and/or gaining sufficient confidence in the software system under test.

SW Inspections: Techniques aimed at systematically verifying software artifacts (design documents, code, tests, etc) with the intent of finding as many defects as possible, as early as possible.

What SW testing is and is NOT

Testing is to **execute** a program with the intent of finding as many defects as possible and/or gaining sufficient confidence in the software system under test

Testing **is not** cut-and-fit improvement of a user interface or requirements elicitation by prototyping.

Testing **is not** the verification of an analysis or design model by syntax checkers or simulation.

Testing **is not** the scrutiny of documentation or code by humans in inspections, reviews, or walkthroughs.

Testing **is not** static analysis of code using a language translator or code checker.

Testing **is not** the use of dynamic analyzers to identify memory leaks or similar problems.

Testing **is not** debugging, although successful testing should lead to debugging.

Testing Definition

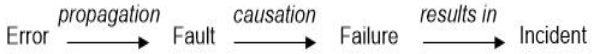
Error: The very root cause. Errors are committed by people.

Fault: A fault is the result of a human error in the software documentation, code, etc.

Failure: A failure occurs when a fault executes.

Incident: Consequences of failures. Failure occurrence may or may not be apparent to the user.

Defect: any of the above, usually a fault, a.k.a. bug.



A test case is set of inputs and the expected outputs for a unit/module/system under test.

A test suite (test set) is set of test cases (at least one).

Types of Testing Activities

Test-case Design: Exploratory (Human-based); Design test values based on domain knowledge of the program and human knowledge of testing, exploratory testing.

Test-case Design: Criteria-based; Design test values to satisfy coverage criteria, covering all lines of code.

Test Automation; Embed test values into executable scripts.

Test Execution; Run tests on the software and record the results.

Test Evaluation; Evaluate results of testing, report to developers.

Test Levels / Test Organization

Since we can have defects in different level and stages of the SW process, we need **testing at different levels and stages;**

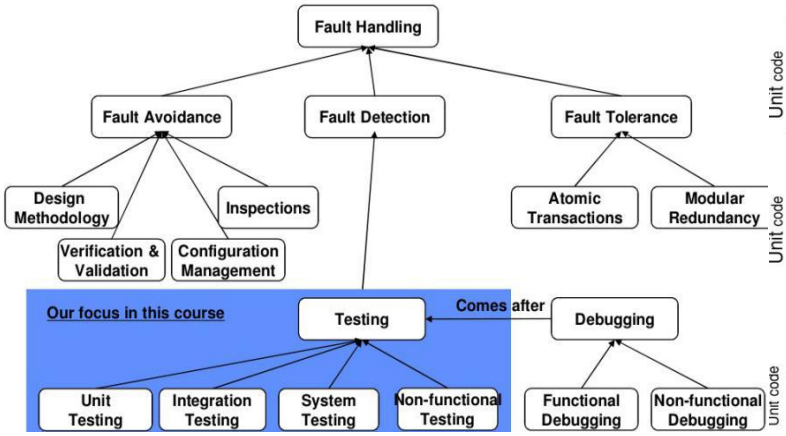
- Unit testing (also called module or component testing)
- Integration testing
- System testing
- Non-functional (e.g., performance) testing
- Acceptance testing
- Installation testing

Manual Testing;

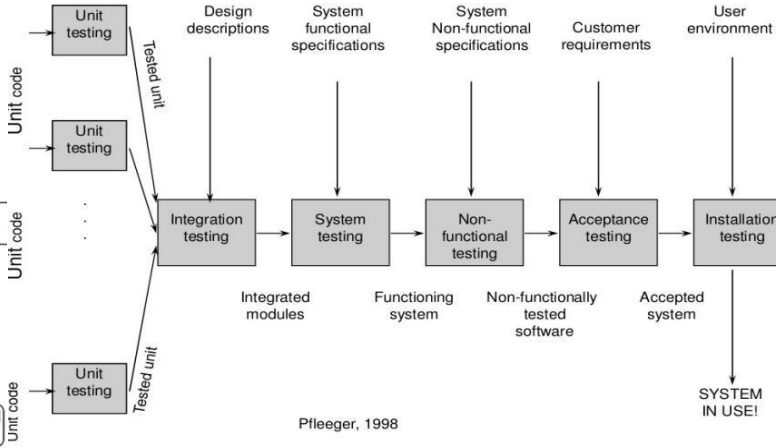
- Clever test-case design
- Interaction with system inspiration for new tests
- Human oracle
- Single test case execution
- Limited data

Automated Testing

- Specs, models, & code used to derive test cases
- Automated oracle needed
- Test execution easily repeatable
- Massive input data possible



Test Organization (just one typical test process)



Pfleeger, 1998