**Software Engineering**

**Technical Debt -** A poor, hard to understand "hacky" implementation which will have to be repaid with interest later.

**SDLC**

* **Requirements -** What we are going to build.
* **Design -** how we ae going to build, how we create front end, how we integrate different technologies together, how we use servers etc.
* **Implementation -** we actually build, code, implement
* **Verification -** we perform test, what we build is actually what we want to build, map with what.
* **Maintenance -** it's in cycle, when we want to solve any bug, we want to add something other features, functionality etc.

**EX.**

**Building a form:**

**Requirements –**

* Collect email address and message
* Send to and store in a database
* Prevent user from bad data input

**Design -**

* Use HTML and CSS to build the framework
* Use JS for verification of input
* Use jQuery and MySQL for contacting backend

**Implementation –**

* Code and documentation the work

**Verification –**

* Does the form collect information?
* Does the form send that info to a database
* Does the form prevent bad user input

**Maintenance –**

* Create lifecycle plan, fix any bugs

Requirement Definition – A way of figuring out the exact specifications of what software should do, finding out the goals of the system

In the context of the Software Development Life Cycle (SDLC), “time upfront” typically refers to the initial phases of the project where significant planning and analysis occur.

* **Specification –** What’s need to be done to meet the requirements. Specifications provide detailed descriptions of how the requirements will be met. They include technical details, guidelines, and constraints for the design, development, and implementation.

Let’s consider an example of a project to develop an online bookstore.

1. **Functional Requirement**: The system should allow users to search for books by title, author, or ISBN.
2. **Functional Specification**:
   1. The search functionality will use a RESTful API to query the database.
   2. The search results will be displayed in a paginated format, with 10 results per page.

**Functional Requirements**

Functional requirements describe the specific behaviours or functions of a system. They define **what the system should do**. Here are some examples:

User Authentication, Search Functionality

**Non-Functional Requirements**

Non-functional requirements define the **quality attributes** of a system. They describe **how the system performs** a function rather than what the system does. ex.

Performance, Usability, Reliability

**Product Requirements**

These requirements specify the features and functionalities of the software product itself. They describe what the system should do to meet the needs of the users and stakeholders.

* **Example**: The system should allow users to create, edit, and delete their profiles.

**Organizational Requirements**

These requirements are derived from the policies, procedures, and standards of the organization developing the software. They ensure that the software aligns with the organization’s goals and practices.

* **Example**: The software must comply with the organization’s data security policies and use the approved technology stack.

**External Requirements**

These requirements are influenced by external factors such as regulatory standards, legal constraints, and market conditions. They ensure that the software adheres to external rules and expectations.

* **Example**: The system must comply with GDPR regulations for data protection and privacy.

**WRSPM Model**

The WRSPM model is a framework used in requirements engineering to bridge the gap between the environment and the system being developed. It stands for World, Requirements, Specifications, Program, and Machine. Here’s a brief overview of each component:

1. **World (W):** Represents the environment in which the system operates. It includes all the assumptions and facts about the domain.
2. **Requirements ®:** Describes what the system should achieve in terms of its impact on the environment. These are the goals and needs of the stakeholders.
3. **Specifications (S):** Provides a detailed description of how the system will meet the requirements. It includes technical details and constraints.
4. **Program (P):** The actual code or software that implements the specifications.
5. **Machine (M):** The hardware or platform on which the program runs.

**Variables-**

* **eh (environmental hidden)**: These are phenomena in the environment that are not visible to the system. They occur in the environment but are not monitored or controlled by the system.

**Example**: The internal state of a patient’s body that is not directly measured by the monitoring system.

* **ev (environmental visible)**: These are phenomena in the environment that are visible to the system. They can be monitored or controlled by the system.

**Example**: The heart rate of a patient as measured by a sensor.

* **sh (system hidden)**: These are phenomena within the system that are not visible to the environment. They are internal to the system and not directly observable by external entities.

**Example**: The internal processing logic of the monitoring system that determines when to trigger an alarm.

* **sv (system visible)**: These are phenomena within the system that are visible to the environment. They represent the outputs or actions of the system that can be observed by external entities.

**Example**: The alarm sound triggered by the monitoring system when a patient’s heart rate falls below a certain threshold

**DESIGN**

**Architecture**

* Is the very top level of design
* Architects are the link between idea and reality
* Architecture is something that cannot be fixed once implemented
* They take your idea and build the plan to convert idea into reality

Architecture mistakes are almost impossible to fix once coding has begun

Good architecture always allows for faster and smarter task allocation and development

System architecture is all about breaking up larger system into smaller focused systems

**Subsystem:**

A subsystem is a self-contained system within a larger system. It operates independently and can function on its own.

**Module:**

A module is a smaller, self-contained unit within a subsystem or system. It provides specific functionality and is not typically considered an independent system.

**Design**

**Activity:** Working to design the software

**Product:**  A document or set of documents detailing the design of software

**Stages of design:**

* Break larger problem into smaller problems
* Understand each problem
* Identify potential solutions
* Describe solution abstract
* Repeat until everything is abstracted
* Component structure design
* Data structure design
* Algorithm design

**Modularity**

* **Coupling:** How well modules work together, Measuring the strength of connection between module or subsystem
* **Cohesion:** How well a module meets a singly designed goal

Goals of modularity

* Abstraction
* Decomposability
* Composability
* Module Understandability
* Continuity
* Protectability

Information Hiding – Hide complexity in a black box

Data Encapsulation - Hide implementation details from the user. Only private set of tools to manipulate the data

**Tight coupling** in software engineering refers to a scenario where classes or modules are highly dependent on each other. This means that changes in one module are likely to require changes in the other, making the system less flexible and harder to maintain.

* Content coupling: When one module modifies or relies on the internal working of another module
* Common coupling: when several modules have access to and manipulate the global data
* External Coupling: When several modules have direct access to the same external i/o

**Medium coupling** refers to a level of dependency between software modules that is neither too tight nor too loose. It strikes a balance where modules are somewhat dependent on each other but still maintain a degree of independence.

* Control coupling: When data is passed that influences the internal logic of another module
* Data-structure Coupling: When multiple modules share the same data-structure

**Loose coupling** is a design principle that aims to reduce the dependencies between software modules. In a loosely coupled system, modules interact through well-defined interfaces and are minimally dependent on each other’s internal implementations. This makes the system more flexible, easier to maintain, and more scalable.

* Data coupling: when two modules share or pass the same data
* Message coupling: When messages or commands are passed between modules
* No coupling: No communication whatsoever

**STRONG COHESION AND LOOSE COUPLING IS GOOD**

**Cohesion:** A module’s ability to work toward a set and well-defined purpose

* **Weak Cohesion:** Weak cohesion occurs when the elements of a module are loosely related and perform a variety of unrelated tasks. This type of cohesion is undesirable because it makes the module harder to understand, maintain, and reuse.
  + Coincidental cohesion
  + temporal cohesion
  + logical cohesion
* **Medium Cohesion**: Medium cohesion is when the elements of a module are somewhat related but still perform a range of tasks that are not entirely focused on a single purpose. This level of cohesion is better than weak cohesion but still not ideal. An example of medium cohesion is a class that handles user authentication and also manages user profiles.
  + Procedural Cohesion
  + Communication cohesion
  + Sequential cohesion
* **Strong Cohesion:** Strong cohesion is the most desirable level of cohesion. It occurs when the elements of a module are highly related and work together to achieve a single, well-defined purpose. Modules with strong cohesion are easier to understand, maintain, and reuse. An example of strong cohesion is a class that handles all aspects of user authentication, such as logging in, logging out, and password management.
  + Functional Cohesion
  + Object Cohesion
* Deployment should be built with the idea of retreat. If something goes wrong, how can we revert?
* Rollback: reverting the system back to a previous working state.

1. Bug: Bugs can be both an error, and a deviation from expected behaviour
2. Failure: The event by which the code deviates from expected behaviour
3. Error: The part of code that leads to the failure
4. Fault: What the outcome actually was

Verification: Are we building the things right ?- does the software work compare to given specifications

Validation: Are we building the right thing ?- does the software work compared to what the user needs ?

**Verification** is the process of evaluating software to ensure that it meets the specified requirements and design specifications. It answers the question, “Are we building the product right?” Verification involves activities such as reviews, inspections, and walkthroughs to check that the software correctly implements the design and requirements.

**Validation** is the process of evaluating software to ensure that it meets the user’s needs and expectations. It answers the question, “Are we building the right product?” Validation involves testing the software in real-world scenarios to ensure it performs as intended and satisfies the end-users’ requirements.

**White Box Testing** (also known as clear box or glass box testing) involves testing the internal structures or workings of an application. The tester has knowledge of the internal code, algorithms, and logic. This type of testing is typically performed by developers and focuses on verifying the flow of inputs and outputs through the code, ensuring that all paths, branches, and conditions are tested.

**Key Characteristics**:

* **Code Coverage**: Ensures that all code paths are tested.
* **Detailed Testing**: Involves testing internal structures, such as loops, branches, and conditions.
* **Tools**: Often uses tools like code analyzers and debuggers.
* [**Examples**: Unit testing, integration testing, and code reviews1](https://www.baeldung.com/cs/testing-white-box-vs-black-box).

**Black Box Testing** (also known as behavioral or functional testing) involves testing the software without any knowledge of the internal workings. The tester focuses on the inputs and outputs of the software, ensuring that it behaves as expected according to the requirements and specifications. This type of testing is typically performed by testers and end-users.

**Key Characteristics**:

* **Functional Testing**: Focuses on testing the functionality of the software.
* **User Perspective**: Tests the software from the end-user’s point of view.
* **Tools**: Uses tools like test management software and automated testing frameworks.
* [**Examples**: Functional testing, system testing, and acceptance testing](https://www.softwaretestinghelp.com/black-box-vs-white-box-testing/)

**Scrum**

**Scrum** is a framework used to implement Agile methodology. It provides a structured approach to managing and completing complex projects by breaking them down into smaller, manageable pieces. Scrum involves specific roles (such as Scrum Master, Product Owner, and Development Team), ceremonies (like Sprint Planning, Daily Stand-ups, Sprint Review, and Sprint Retrospective), and artifacts (such as Product Backlog, Sprint Backlog, and Increment). [The main goal of Scrum is to enhance collaboration, flexibility, and continuous improvement within the team1](https://www.indeed.com/career-advice/career-development/scrum-vs-sprint)[2](https://insightspotter.com/what-is-the-difference-between-sprint-and-scrum/).

**Sprint**

A **Sprint** is a time-boxed iteration within the Scrum framework, typically lasting between one to four weeks. During a Sprint, the team works on a set of tasks from the Sprint Backlog, aiming to deliver a potentially shippable product increment by the end of the Sprint. [Sprints provide a structured timeline for the team to focus on specific goals, allowing for regular assessment and adaptation based on feedback and progress1](https://www.indeed.com/career-advice/career-development/scrum-vs-sprint)[3](https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/Sprint-vs-Scrum-Whats-the-difference).

**Key Differences**

* **Scope**: Scrum is the overarching framework that guides the entire project, while a Sprint is a specific, time-boxed period within Scrum where work is executed.
* **Purpose**: Scrum focuses on the overall process and structure of project management, whereas a Sprint focuses on delivering a specific set of tasks within a defined timeframe.
* [**Components**: Scrum includes roles, ceremonies, and artifacts, while a Sprint is primarily concerned with the execution of tasks and delivery of product increments](https://insightspotter.com/what-is-the-difference-between-sprint-and-scrum/)

**Product Backlog**

* **Definition**: The Product Backlog is a dynamic, prioritized list of all the features, enhancements, bug fixes, and requirements needed for the product. It represents everything that could be done to improve the product.
* **Ownership**: Managed by the Product Owner, who is responsible for prioritizing the items based on their value to the product and stakeholders.
* **Scope**: Covers the entire project and includes long-term goals and future enhancements.

**Sprint Backlog**

* **Definition**: The Sprint Backlog is a subset of the Product Backlog, consisting of items selected for implementation in the current Sprint. It includes tasks that the development team commits to completing during the Sprint.
* **Ownership**: Managed by the Development Team, who selects the items and breaks them down into actionable tasks.
* **Scope**: Focuses on the short-term goals of the current Sprint, typically lasting 1-4 weeks.

**Lean Startup**

Experiment -> Metrix -> Assumption

Build -> measure -> learn

**Product Owner**

* **Role**: Acts as the customer’s representative, ensuring the product meets the stakeholders’ needs.
* **Responsibilities**:
  + Defines and prioritizes the product backlog.
  + Communicates the product vision to the team.
  + Balances the needs of stakeholders and the development team.
* **Skills**: Strong leadership, communication, and understanding of agile principles.
* **Certification**: Popular certification includes PSPO (Professional Scrum Product Owner).

**Scrum Master**

* **Role**: Facilitates the Scrum process, ensuring the team follows Scrum principles.
* **Responsibilities**:
  + Guides and coaches, the team on Scrum practices.
  + Removes obstacles that hinder the team’s progress.
  + Leads daily Scrum meetings and sprint planning.

**Development Team**

* **Role**: Responsible for delivering a potentially releasable increment of the product at the end of each sprint.
* **Responsibilities**:
  + **Self-organizing**: The team decides how to accomplish the work set out in the sprint backlog.
  + **Cross-functional**: Members have all the skills necessary to create the product increment.
  + **Collaborative**: Works closely with the Product Owner and Scrum Master to ensure alignment and progress.
  + **Quality-focused**: Ensures the product increment meets the Definition of Done.