**SOLID Design Principles**

* SOLID principles are the five essential building blocks of OOPS
* Acronym introduced by Michael Feathers but concept given by Robert C Martin
* By applying these principles, one can right better-quality code that have high cohesion, loose coupling and strong encapsulation. The code is easy to maintain and extend over time
* SOLID stands for
  + Single Responsibility principle (SRP)

**There should be never more than one reason for a class to change**

An object/class should have only one responsibility and it should be completely encapsulated by the class. Here responsibility means a reason to change. If there are more than one reason to change, the functionality should be split into two classes and each will handle its own responsibility.

When a class has more than one responsibility, those responsibilities become coupled and this results in difficulties in refactoring when new responsibilities emerge.

* + Open-Closed principle (OCP)

**Software entities (classes, modules, functions etc.) should be open for extension but closed for modification**

An entity should allow its behaviour to be extended but never by modifying its source code. Modifying behaviour comes by adding new code and never by modifying the old code.

By applying this one should get loose coupling, improves readability and reduces the risk of breaking existing functionality.

* + Liscov Substitution principle (LSP)

**Subtypes must be substitutable for their base types**

Objects should be replaceable by instances of their subtypes without affecting the functioning of your system from a client’s point of view. This is very closely related to the OCP.

* + Interface Segregation principle (ISP)

**Classes that implement interfaces should not be forced to implement methods they do not use**

Interfaces should be more specific and should be defined by the client who will use it. Client should not be forced to depend on the interface they do not use.

It is intended to keep a system decoupled and thus it becomes easier to refactor, change and deploy.

* + Dependency Inversion principle (DIP)

**High-level modules should not depend on low level modules rather both should depend on abstraction. Abstraction should not depend on details; rather detail should depend on abstraction**

This principle is primarily concerned with reducing dependencies among code modules. If the implementation details depend on higher-level abstractions it means the system is coupled correctly.

* While developing a software, two concepts are really important

* + Cohesion (different parts of the system work together to yield better results than if each part would be working individually)
  + Coupling (degree of dependence of a software entity)

* The ultimate goal of software solution development should be high cohesion and low coupling.

**Technical and Functional Debt**

* Technical debt represents the behaviour of code maintenance and scalability over time
* Whenever you are forced to leave some parts of your code with a ‘To Do’, ‘Refactor’, or ‘Review Later’ mark, it’s an additional piece of debt you acquire. This way you get some Cash, to buy some time. You meet the deadline, you launch the product on time and you have a prototype to show for an investment round. But you can’t forget about the cutbacks made in code quality, because it’s your debt. It’s something you owe, it’s something you need to pay back.
* If you ignore it and continue to build up on it, your product will progressively become less stable and less scalable and its maintenance cost will skyrocket. To keep the analogy with the real world, it’s like taking a new loan to pay back the one you took earlier, repeatedly.
* Function debt - the way I see it is that in order to deliver it on time, the functionality is implemented partially. The functionality itself suffers from cutbacks. Let’s say that we need to deliver a simple form with a few fields and file input. As we are tight on time and that we don’t want to introduce a Technical Debt, we decide to ‘simplify’ the functionality. We eliminate the ‘drag & drop’ for the file attachment, support for multiple files, and the fields’ verifications’ on legitimacy of the introduced data.
* Our application works perfectly fine, code is clean (the parts that are present don’t need to be modified), and we deliver it on time. It is a miracle. Well, it’s not. By doing this, we keep invalid data in our database and the user experience is not good enough. Our code is spotless, and does exactly what we expect. We don’t have Technical Debt. Our debt is Functional.
* Later on, we will have to come back to this form, and add new parts, integrate a validation service, and change the UI. As time passes by, all developers will not recall exactly what is missing. They will need time to study what has to be done, and they will need to deal with the toxic data that we allowed in our database. The time invested here is much higher than the time needed to have the full functionality done in the first place. It’s the fee we pay, the pay-off of our dept. Its effects are less harmful than the Technical Debt’s effects, yet are still needed to be watched and taken care off.

**Estimation**

* Estimation are of the following three types:

* + Analogous estimation

Uses a similar past project experience to estimate duration or cost

* + Parametric estimation

A type of estimation technique where we develop techniques/algorithms to evaluate different activities. This technique differs from project to project. Function point analysis and Delphi technique falls in this category

* + Three-point estimation

Revolves around three parameters as:

* + - * Optimistic Outcome (outcome in the most ideal situation)
      * Pessimistic Outcome (outcome in the worst-case scenario)
      * Most Likely Outcome (outcome in the most likely scenario)

* Function point is a unit of measurement to express the amount of business functionality, an information system provides to a user. FPs measures software size
* There are two types of functions
  + Data functions (2 types – Internal Logical Files and External Logical Files)
  + Transaction functions (3 types – EI, EO and EQ)

* ILF is totally in boundaries of the application. These are direct or derived data from application or project
* ELF is the data that has to be taken from outside of the scope of application. Ex. Any application taking data from an API
* EI (External Inputs) – Transaction function in which data goes into the application from outside the boundary
* EO (External Outputs) – Transaction function in which data comes out of the system
* EQ (External Inquiries) – Transaction function with both input and output components that result in data retrieval

**Agile estimation and story points**

* Story point
  + A story point is an abstract measure of effort required to implement a user story
  + In simple terms, it is a number that tells the team about the difficulty level of the story
  + Difficulty could be related to complexities, risks, and efforts involved
  + Story point sizing can be thru
    - GP series: 1, 2, 4, 8, 16 …
    - T-shirt sizing: small, medium, large, x-large etc.
    - Fibonacci seq: 1, 2, 3, 5, 8, 13, 21 …
  + Baseline story is a story whose complexity/size are clear to all members. Story points then can be assigned in reference to this
  + Story points and hours-based estimation should not be related to each other, those are different