SOFTWARE DESIGN ARCHITECTURE AND PRINCIPLES

Assoc. Prof. Dr. Mehmet Akif Çifçi

What is Software Architecture?

- In short, software architecture can be defined as "the highest level concepts that define the system". From the program language you choose to the database, flexibility, security and hardware requirements, you have to design a whole software project before you start coding. The people who design the infrastructure of a project are software architects. The software architect designs the software project from start to finish, steps in where necessary, and ensures that the plan is not deviated from throughout the software process.

Software Architecture Features

- Functionality: Refers to the level of performance of the software according to its intended use.
- Reliability: Refers to the ability of the product to deliver the desired functionality under given conditions.
- Self-Reliance: Refers to the ability to perform optimally despite the interruption of one of the individual services.

Software Architecture Features

- Usability: Refers to the extent to which the software product can be easily used.
- Performance: An estimate based on processing speed, response time, resource utilization, output and productivity.
- Supportability: Refers to the ease with which programming developers can port software from one platform to another with no or minimal changes.

What are the Templates Used in Software Architecture?

- Model-View-Controller (MVC) is the most well-known and frequently used template. In a sense, we can say that it is the default and suitable for all types of software.

- The Pipe and Filter template is usually used for the complier and the purpose of the nu architectural template is to break down the process into parts.

What are the Templates Used in Software Architecture?

- Service-Oriented-Architecture (SOA), which is the most preferred template in the software field in recent years, basically aims to make many systems work harmoniously on the system.

- There are also more comprehensive and relatively less preferred software architecture templates such as Multitier Architecture and Implict Invocation.

What are Software Design Principles?

- Software Design Principles are a set of guidelines that help developers to design a good system. These principles are a list of approaches, styles, philosophies and best practices introduced by some of the well-known software engineers and authors in the software industry.
- Software design principles are guidelines for developing software that is accessible, readable, maintainable, flexible, testable and reusable. These principles make software understandable, orderly, organized and scalable. They also ensure that software is long-lasting and designed holistically.

Purpose of Software Design Principles

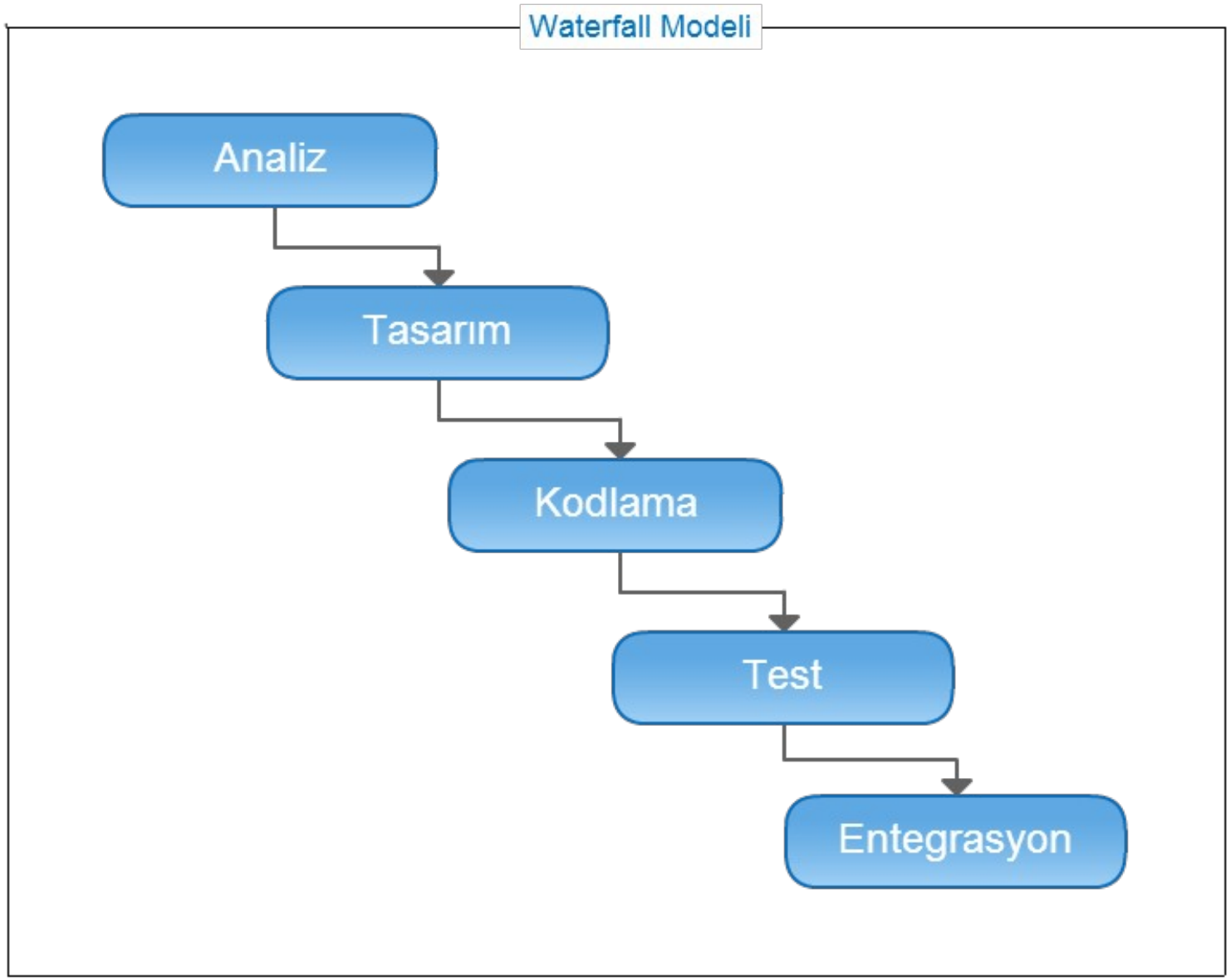
- Explain what design is and how various types of design deal with different aspects of the product.
- To present design as a problem solving activity, to reveal the role of abstraction and modeling in design.
- Identify the place of design in the software life cycle.
- To examine design methods in software engineering.

Importance of Software Architecture and Design

- Maintainability: Good software architecture and design makes code easier to understand, easier to maintain and easier to find bugs. This increases the maintainability of the software.
- Flexibility: A good design provides flexibility to adapt to changes in the system. When components are independent and loosely connected to each other, changes have less impact on the system.

Importance of Software Architecture and Design

- Scalability: A good software architecture ensures scalability as system requirements change. It can easily adapt to new requirements or increased load demands.
- Testability: A good design makes software easier to test. Modular structures allow components to be isolated and tested independently.
- Reusability: A good design encourages the reuse of components. This leads to less code duplication and speeds up the development process.


Waterfall Modeli
Analiz
Tasarım
Kodlama
Test
Entegrasyon

What are Software Design Principles?

- DRY (Don't' Repeat Yourself)
- KISS (Keeping It Simple, Stupid)
- YAGNI (You Aren't Gonna Need It)
- SoC (Seperation of Concerns)
- SOLID

DRY (Don't' Repeat Yourself)

- One of the fundamental principles of software development, "Don't Repeat Yourself", or DRY for short, refers to the principle that a code base should be defined in only one place. This principle aims both to manage code more effectively in software engineering and to facilitate maintenance and development processes by minimizing code repetition.

DRY (Don't' Repeat Yourself)

- The basic philosophy of the DRY principle is that blocks of code containing the same logic or functionality are written once, kept in one place and used as a reference in other areas. This not only makes the code more understandable, but also allows updates to be made in only one place if any changes are made.
- When writing a code, if there are repetitive expressions, they should be methodized and made reusable, and repetitive blocks should be avoided.

KISS (Keeping It Simple, Stupid)

- In the dynamic world of software development, where complexity can easily spiral out of control, simplicity emerges as a fundamental principle that promotes clarity, maintainability and overall effectiveness. One such principle is "Keeping It Simple, Stupid" or KISS. Let's explore the essence of KISS and understand why simplicity is not just a choice, but the cornerstone of successful software development.

KISS (Keeping It Simple, Stupid)

- In essence, KISS suggests that systems work better when kept simple rather than unnecessarily complex. This principle encourages developers to avoid unnecessary details, complex designs or confusing solutions that can make understanding difficult and increase the likelihood of bugs.
- The principle of simplicity and simplicity. It says that a problem that can be solved in an easy way should not be complicated by unnecessary expressions. It also emphasizes emphasizing the features that are really needed.

YAGNI (You Aren't Gonna Need It)

- In software development, the YAGNI principle advises developers to add only those features that meet current needs. Adding features beyond the requirements can make the code complex and difficult to maintain. By focusing only on meeting current needs, the YAGNI principle aims to make software more readable, simple and maintainable.

YAGNI (You Aren't Gonna Need It)

- In addition to reducing unnecessary complexity, this principle provides advantages such as fast development, less maintenance and system flexibility. When combined with the principles of continuous communication, requirements analysis and clean code writing, the YAGNI principle contributes to more effective and successful software projects.
- It is important not to include things that are not needed, thinking that they will be needed in the future. Features developed without demand bring extra testing effort, documentation and maintenance challenges.

YAGNI (You Aren't Gonna Need It)

- In addition to reducing unnecessary complexity, this principle provides advantages such as fast development, less maintenance and system flexibility. When combined with the principles of continuous communication, requirements analysis and clean code writing, the YAGNI principle contributes to more effective and successful software projects.
- It is important not to include things that are not needed, thinking that they will be needed in the future. Features developed without demand bring extra testing effort, documentation and maintenance challenges.

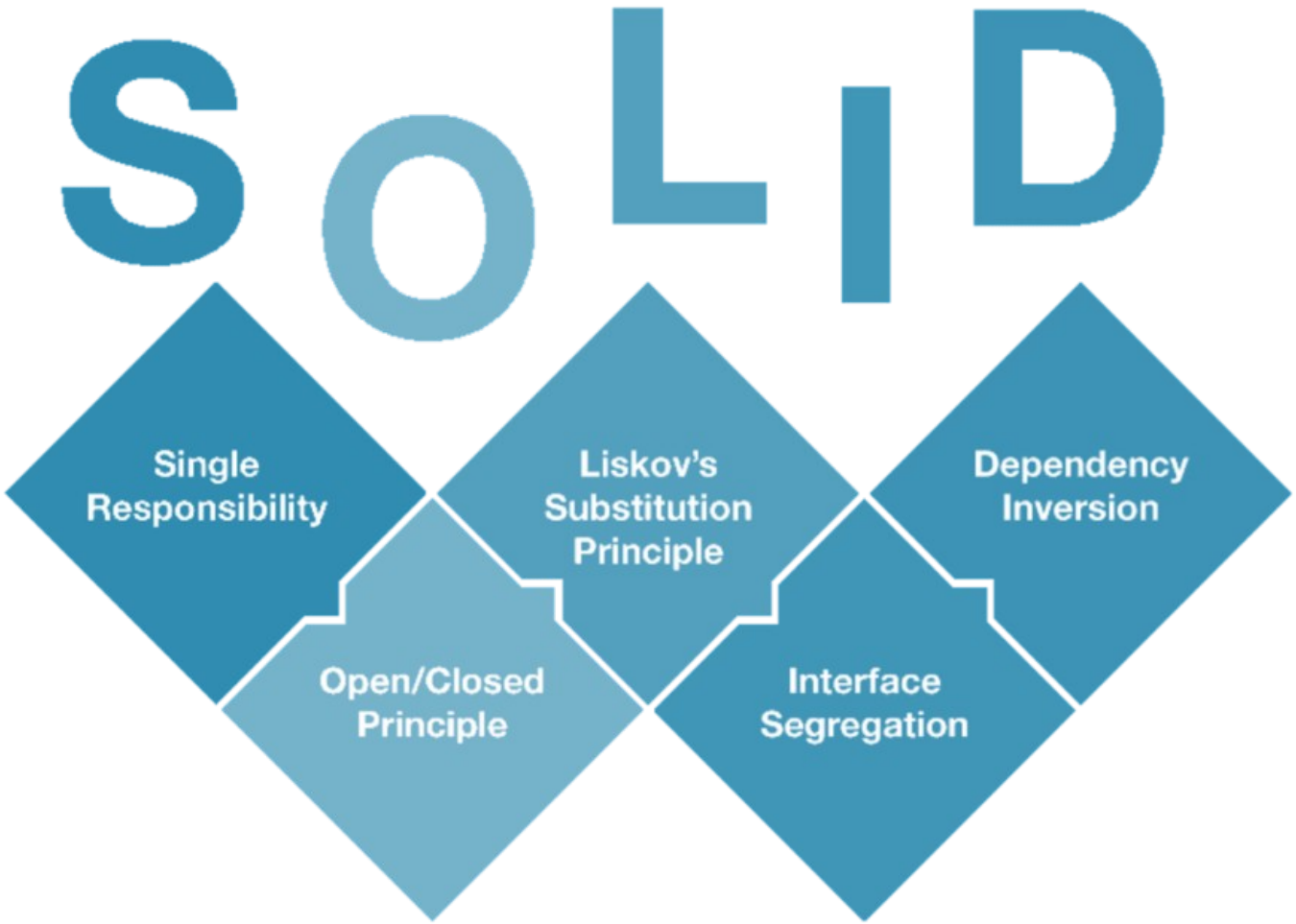
SoC (Seperation of Concerns)

- It is one of the important principles of Aspect Oriented Programming. It can be defined as managing parts with different functions by keeping them separate from each other. The architectural design pattern mvc is actually one of the best examples of this principle. Data and representation are abstracted and separated from each other because they have different functions.

- Thanks to this structure, when a development is to be made, it can be developed and managed without affecting other structures.

SOLID

- S - Single Responsibility
- O - Open Closed (Open-Closed Principle)
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle


SOLID
Single Responsibility
Open/Closed Principle
Liskov's Substitution Principle
Interface Segregation
Dependency Inversion

Single Responsibility Principle

- According to the Single Responsibility Principle, basically every class or module should have a single responsibility. In other words, it means that in order to change/update a class or module, only one reason should be changed, namely the responsibility it assumes. It aims for an easier to read and more maintainable code structure.

Open Closed (Open-Close Principle)

- The Open-Closed Principle states that a class or module should be open to development but closed to change. In short, it can be thought of as defining a new function in a new class derived from that class instead of updating the existing class when a new function is to be added. It aims for a more flexible and maintainable code structure.

Liskov Substitution Principle

- According to the Liskov Substitution Principle, lower level classes must be substitutable for higher level classes. If we can replace an object from a subclass with an object from a superclass and the program does not break in any way, we can say that it complies with the Liskov Substitution Principle.

Interface Segregation Principle

- The Interface Segregation Principle states that an Interface should provide only the methods that the classes using that Interface need. This means that a class should not need to define a feature that it does not need, so it is better to create multiple specialized Interfaces that focus on different functions.

Dependency Inversion Principle

- According to the Dependency Inversion Principle, higher level classes or modules should not depend on lower level classes or modules, but both should depend on abstractions. These dependencies are usually established through Interfaces and Abstract classes. This principle helps to create a code structure that is more flexible and easier to maintain and test.