1. **Principles of class design- SRP, OSP, LSP, ISP, DIP**

* Single Responsibility Principle
* Open-Closed Principle
* Liskov Substitution Principle (LSP)
* Interface Segregation Principle (ISP)
* Dependency Inversion Principle (DIP)

The SOLID principles ensure that OOP applications are readable, testable, scalable, and maintainable. It also ensures that the software is modular, easy to understand, debug, and refactor

**Single Responsibility Principle**

* “*a* class *should have only one reason to change*” which means every class should have a single responsibility or single job or single purpose
* To achieve the goal of the single responsibility principle, we should implement a separate class that performs a single functionality only.
* **MY INFERENCE**: The example I saw, had a Bank class which had many methods including withdrawAmount, getLoadInfo, sendNotification, etc. This class has many responsibilities. Instead break each responsibility into separate classes.

**Open-Closed Principle**

* “*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*” which means you should be able to extend a class behaviour, without modifying it.
* **MY** **INFERENCE**: The example I saw had a SendNotification class. One may want to implement email notification initially and may want to include SMS notification in the future. Instead of modifying the original code, just extent the base class and implement the new functionality.

**Liskov Substitution Principle (LSP)**

* “*Derived or child classes must be substitutable for their base or parent classes”.*
* **MY** **INFERENCE**: The example I saw had a SocialMedia interface which was implemented by Facebook class, Whatsapp Class, etc. SocialMedia interface has all methods including chatWithFriend, publishPost, etc. Facebook class implements all methods and hence it can be substituted for the base class. But Whatsapp class cannot implement all methods as it has no feature to publish posts. To overcome this, separate the interface into multiple interfaces and implement the appropriate interfaces according to our necessity.

**Interface Segregation Principle (ISP)**

* similar to the single responsibility principle
* “*do not force any client to implement an interface which is irrelevant to them*”
* **MY** **INFERENCE**: The same example as above …we should not force Whatsapp to implement publishPost method…instead we should segregate interface and client should implement the interfaces according to their necessity.

**Dependency Inversion Principle (DIP)**

* we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module but both should depend on the abstraction

1. **Cohesion and Coupling [context of modules and context of classes]**

**Cohesion** refers to what the class (or module) can do. Low cohesion would mean that the class does a great variety of actions - it is broad, unfocused on what it should do. High cohesion means that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.

**Coupling** refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing something major in one class should not affect the other. High coupling would make it difficult to change and maintain your code

Good software design has **high cohesion** and **low coupling**.

1. **Relationship between classes - Association, composition, Aggregation**

Association: one object is related to another to use functionality and service provided by that object. Association can be one-one, one-many, many-one, many-many.

* composition and aggregation are two subsets of association
* An aggregation is a special form of association, and composition is the special form of aggregation

Aggregation:

* represents has a relationship
* more specific than an association
* binary association, i.e., it only involves two classes
* kind of relationship in which the child is independent of its parent

Composition:

* exists between similar objects
* depicts dependency between a composite (parent) and its parts (children), which means that if the composite is discarded, so will its parts get deleted

1. **Factory Design pattern**

* also known as **Virtual Constructor**
* subclasses are responsible to create the instance of the class
* define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate
* promotes loose coupling
* **MY INFERENCE**: In spring, we use IOC to create instances for objects. Using factory design pattern, subclasses are responsible for creating instances based on the requirement.

1. **Multi Module Maven Project**

* A Spring Boot project that contains nested maven projects
* a **multi-module project** is built from a parent pom that manages a group of submodules
* The parent maven project must contain the packaging type **pom** that makes the project as an aggregator
* The **pom.xml** file of the parent project consists the list of all **modules, common dependencies,** and **properties** that are inherited by the child projects
* The parent pom is located in the project's root directory. The child modules are actual Spring Boot projects that inherit the maven properties from the parent project
* The parent POM defines the **Group ID, Artifact ID, version**, and **packaging**
* the parent **POM** defines the packaging pom instead of packaging jar. The packaging pom refers to other Maven projects.

**Directory Structure**

* parent project has the pom file at the bottom of the directory
* within the parent, modules can be created and these modules will have their own pom files.
* In parent pom, packaging should be changed to pom and modules tag need to be included
* In child pom, packaging should be jar

Looked up the example in javatpoint website.