SOLID Design Principles

Design Patterns in Java

What are design principles?

- Design principles are guidelines, to be applied when designing classes/applications, which help us achieve a software design that is easy to extend, understand, maintain and test.

- These are not RULES and so they don't **_guarantee_** that the result will _always_ satisfy above requirements. However, by following them it's much easier to reach those goals.

- In practical world, it's not always possible to adhere to these guidelines 100%. Reasons could be performance, legacy code, integrations. But as designer/programmer we should try to follow them where it's practical.

- Like design patterns, these principles teach you what has worked for others and you're free to choose, modify and mix them together to come up with a design that is best suited for your problem.

- Remember there is no silver bullet for designing software. All we have are some guidelines based on others experience. But this is what makes software design exciting. We can draw on these principles & patterns and come up with our own unique designs

SOLID Principles

- SOLID is an acronym are a group of 5 principles. If we take first letter of names of these 5 principles we word SOLID.

- S – Single responsibility principle

- O – Open/Closed principle

- L – Liskov's substitution principle

- I – Interface segregation principle

- D – Dependency inversion

- These are NOT the only principles for software design but they are few of the most important ones.

- Their order is not important when applying these guidelines and they are not dependant on each other.

Single Responsibility Principle

- This principle can be summarized as: every class that we write should have only one responsibility/purpose.

- This principle is based on cohesion principle. Remember? High Cohesion & Low Coupling!

- A single responsibility here means that class provides a very focused functionality or it addresses a specific concern of our desired functionality. In other words a class should have one & only one reason to change.

- Let's say we've a code that creates & sends a message to a remote service listening on a port. Now this code can change for few reasons: the communication protocol/parameters may change or the message format changes or if the content of the message changes.

- The Single Responsibility principle says that we have three separate responsibilities here, and should be in three separate classes or modules. This makes our classes more easy to write, test, change & review.

Open/Closed Principle

- Open/closed principle states that any software entity (method, class, interface, module etc.) should be open for extension but closed for modification. Sounds intentionally cryptic, isn't it? ☺

- The principle is fairly simple: We should be able to modify behavior of software entity *easily* without needing to change the original code.

- Open => Open for extension (change/modify behavior); Closed => original entity's source code is sealed from changes.

- So in Java it means using inheritance to modify the "behavior". We can then substitute/swap implementation used by client code with a derived class and get different/extended behavior.

- Abstraction is the key! If client code is using interface/abstract class references it's easy to swap out implementations without requiring changes to client code.

Liskov's Substitution Principle (LSP)

- This principle states that: we should be able to use sub-class or child class object wherever base class object is expected and this should NOT alter desired properties of the program.

- The principle appears quite simple on surface. But note that, we are not **_only_** talking about simple type based substitution where we need to follow some rules for overriding a method. This is also about behavioral subtyping.

- To put simply, this principle ensures that child class method does not change what was expected from a base class method. Child class can use different logic/algorithm but it should not deviate from the overall expectation.

- For example, if a base class has print method which accepts a string and prints a string to console, then a child class method can override it and print to a file. We're changing where the printing occurs but not changing the "printing" part of the expectation.

Liskov's Substitution Principle (LSP)

- A great confusion can happen about what are the expectations? Who defines then? Where? There are multiple places for this: a method name itself, method level comments or an external document (design document) can serve as source of expectation.

- For existing code where none of the above are present then the base class method code and its unit tests will set expectation.

- For example in the print example on last page, if the method was called printToConsole then the method name is setting expectation which is: print a string to *console* and so a child class method printing to file will violate the expectation and thus the Liskov Substitution principle.

Liskov's Substitution Principle (LSP)

- Another common way this principle is broken is by providing more than the expected behavior. For example a print method in child class also taking some action based on the message in addition to printing it.

- So overall takeaway is: Other people/code expect certain behavior from a method, when overriding such method do not deviate from this behavior as then our child class is not truly a drop-in replacement of base class.

- This principle is important because it forces us to write child classes that are truly portable. If we come up with a more efficient way to do something, all existing code which was using old base class would be able to take advantage of it if we follow Liskov's substitution principle.

Interface Segregation Principle (ISP)

- This principle states that: client code should not be forced to implement methods that are not needed by it.

- This principles suggests breaking down large interfaces into smaller & more specific interfaces. These smaller interfaces are also called as role interfaces.

- Although it's an easy principle to understand, it's also one which gets broken fairly often. Interfaces for UI, interfaces for web services often have methods that do lots of different things, breaking the interface segregation principle.

- In worst case scenario you end up with implementations that provide a "dummy" or empty implementation or methods that throw exceptions like UnsupportedOperationException. These are clear indicators that you need to enforce the ISP to offending interface.

Dependency Inversion Principle

- This principle states that code should depend on abstractions & not on concrete implementations.

- Robert Martin's description of this principle has two parts:

  - High-level modules should not depend ***directly*** on low-level modules. Both should depend on abstractions.

  - Abstractions should not depend upon details. Details should depend upon abstractions.

- In essence this principle is about trying to avoid direct, tight coupling between modules/classes. This allows us to compose the system using different implementations without requiring changes to modules.

- Word module here can be applied to our regular Java modules and even to classes.

- High level and low level modules here are referring to type of functionality the modules provide.

- A high level module will implement "business logic" and will map to use cases for the application.

Dependency Inversion Principle

- A low level module provides functionality that is basic and not tied to any one use case. For example a database connection code, a sorting and searching code or file & network handling code etc.

- As per this principle, our business logic classes should not directly be tied to concrete classes providing database handling or network code etc. Instead these classes should use abstractions to talk to low level classes e.g. Using interfaces. (High-level modules should not depend **_directly_** on low-level modules.)

- And our low level classes also must confirm to the interfaces they implement. For example if a new low level class for handling networking with UDP is written it must implement the methods of interface already used by high level classes. It should not require high level classes to use methods specific to this new class. (Abstractions should not depend upon details. Details should depend upon abstractions.)

Dependency Inversion Principle

- Dependency Inversion simply states that someone else should give me dependencies that I need. I don't care who or how I'm given these dependencies. It is a principle which states *what* should happen but NOT *how* it happens.

- Dependency Injection is a specific pattern with which we provide dependencies. Injection is *HOW* we implement the Dependency Inversion principle.

- So to summarize, Dependency inversion is **_NOT_** same as dependency injection! Dependency injection is about control over wiring of dependencies, who creates necessary dependency objects, who supplies them, how & when.