1 Design Principle

Design principles are guidelines to assist in your object-oriented design.

1.1 Encapsulate what varies

Identify the aspects of your application that vary and separate them from what stays the same.

- Look for code that changes with every new requirements. We can see which behavior needs to be pulled out and separated from the stuff that does not change.
- Alter or extend the code that varies without affecting code that doesn't vary.
- Basis of almost every design pattern. All patterns tend to provide ome way to let some part of your system vary independently on the other parts.
- Pay attention to how each pattern makes use of this principle.

1.2 Favor composition over inheritance

Identify the aspects of your application that vary and separate them from what stays the same. Also know as has-a is better than is-a.

- Inheritance is a powerfull technique that avoid code duplication. However it is a technique that can bet easily overused. It is also a technique that can lead to designs that are far too rigged and not extensible.
- Is-a is a relationship of inheritance e.g. dog is a animal.
- Has-a is a relationship of composition e.g. dog has a owner. Compisition can be a powerfull alternative to inheritance. How can we switch from Is-a to Has-a:
  - Instead of a CoffeeWithButter is-a Coffee.
  - what about a Coffee Has-a condiment?
- We can add any number of condiments easily at runtime.
- Implementing new condiments by adding a new class.

- No code duplication.
- Avoid class explosion.
- Instead of inheriting behavior, we can compose our objects with new behaviors.
- Composition often gives us more flexibility, even allows behavior changes at runtime.
- Composition is a common techniques used in design patterns.

1.3 Loose Coupling

Components should be independent, relying on knowledge of other component as little as possible.

- When two classes interact, changes in one class should not force changes on the other one.
- Loose coupling reduces the dependency between components. Keeping designs loosely coupled helps us to build object-oriented systems that can handle change well.
- Simple technique to achieve loose coupling: abstracting away a concrete type into an interface. This is a technique your are going to see reoccur over and over again in good design.
- A good example of loose couple is the Observer pattern.

1.4 Program to Interfaces, Not Implementations

Where possible, components should use abstract classes or interfaces instead of a specific implementation.

- "program to an Interface" really means "program to a super/abstract type".
- The point is to be able to exploit polymorphism by programming to a super type so that your actual runtime object isn't locked into you code.
- The "new" operator only work on concrete types. Doesn't that violates this principle? Here we can make use of creational design patterns.

- Encourages us to use interfaces/abstract classes when possible, rather than concrete classes.
- This principle frees classes from knowledge of concrete types.
- Improve extensibility and maintainability.

1.5 Single Responsiblity Principle

Classes should have only one reason to change.

- Look at the change in your class: Are part os it changing while other parts are not?
- Change only matters if it really happens. So apply Single Responisiblity only when the need is real, or you are just creating complexity.

1.6 Open/Close Principle

Object-Oriented designs should be open for extension, but closed for modification.

- An example of this is using composition to allow behavior extension of a class, while protecting the existing (proven) code of the class from modification.
- A design pattern that make use of this principle is the strategy pattern. In the duck example we are able to add new duck behaviors without changing the duck class.
- Improve maintainability and extensibility of a design.

1.7 Liskov's Substitution Principle

You should always be able to substitute subtypes for their base class.

- Using Inheritance the wrong way, can lead to undesired behavior when substituting the baseclass with the subsclass.
  [scale=0.2]$\text{liskov}_{substitution_{principle}}$ $Example is the Rectangle and Square class. When Square effect is not in the rectangle superclass. i.e. a Square class cannot substitude the Rectangle class.$
- Don't assume that the IsA principle will always result in hierarchies that adhere to this principle. You will need to consider how substitutable a base class is for it's supertype as well.

- To adhere to this principle, we can use techniques like design by contract.

1.8 Interface Segregation Principle

Classes should not be forces to depend on methods that they don't use.

- Cohesion: How string are the relationship between an interface's methods. i.e. We say a class is highly cohesive if their methods are related.
- Think able the client that uses the interface. Is the client going to use all the method in the interface? if yes then the interface has high cohesion.
- Highly cohesive interfaces leads to software that is easier to maintain and easier to extend.

- Segregate Interfaces needed to keep them focussed and cohesive. i.e. you free the clients from methods they are not intrested in.
[scale=0.2]interface$_s egregation_p rinciple$

1.9 Dependency Inversion Principle

High-level modules should not depend on low-level modules

- Typical Object-Oriented Thinking: We take a problem, and we factor it into a high-level set of components (the policy-makers) that depend on low-level components, who are carrying out the real work. The problem with this approach is that is tightly couples our high-level components to our low-level ones.
- A symptom of tight coupling is that it's difficult to reuse the high-level components with a different implementation of the low-level component.
- Dependency Inversion Principle:
    1. High-level components should not depend on low-level components. Both should depend on abstractions.
    2. Abstractions should not depend on details. Details should depend on abstractions.

- What is high-level policy? It is the abstraction that underlies the application. i.e. In our example, it is a remote control that can control anything.
  [scale=0.2]dependency$_{i}nversion_{p}rinciple This principle frees our high-level components from being dependent on the details of the low-level components.$

- It helps design software that is reusable and resilient against changes.

- Abstraction allow details to remain isolated from each other.