OOP Design Principles

Sang Shin
"Code with Passion!"

Topics

- What and Why Design Principles?
- S.O.L.I.D. design principles
- Other design principles

What and Why Design Principles?

What is and Why OO Design Principles?

- Represent a set of guidelines that helps us to avoid having a bad design (Robert Martin)
- Characteristics of bad design
  - Rigidity - It is hard to change because every change affects too many parts of the system
  - Fragility - When you make a change, unexpected parts of the system break
  - Non-reusability (immobility) - It is hard to reuse in another application because it cannot be disentangled from the current application
- Characteristics of good design
  - Clean and modular (opposite to non-reusability)
  - Highly cohesive (opposite to rigidity)
  - Loosely coupled (opposite to fragility)

Benefits of OO Design Principles

- Extensibility
  - New feature can be easily added without breaking other parts of the system
- Testability
  - Testing verifies the system does what it is expected to do
- Code reuse'ability
  - Software component can be usable in many applications
- Runtime change'ability (Flexibility)
  - Behavior of a software component can be changed during runtime without modifying the existing code

S.O.L.I.D. Design Principles

S.O.L.I.D. Design Principles

- Single Responsibility Principle (SRP)
- Open Closed Design Principle
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle (SRP)

- Motivation
  - If you put more than one functionality in one class/method in Java, it introduce coupling between two functionality
- What is it?
  - There should not be more than one reason for a class/method to change, or a class should always handle single functionality
  - Do not mix responsibilities (concerns) into a single class
- Benefit
  - Less coupling
  - Higher change'ability

Lab:

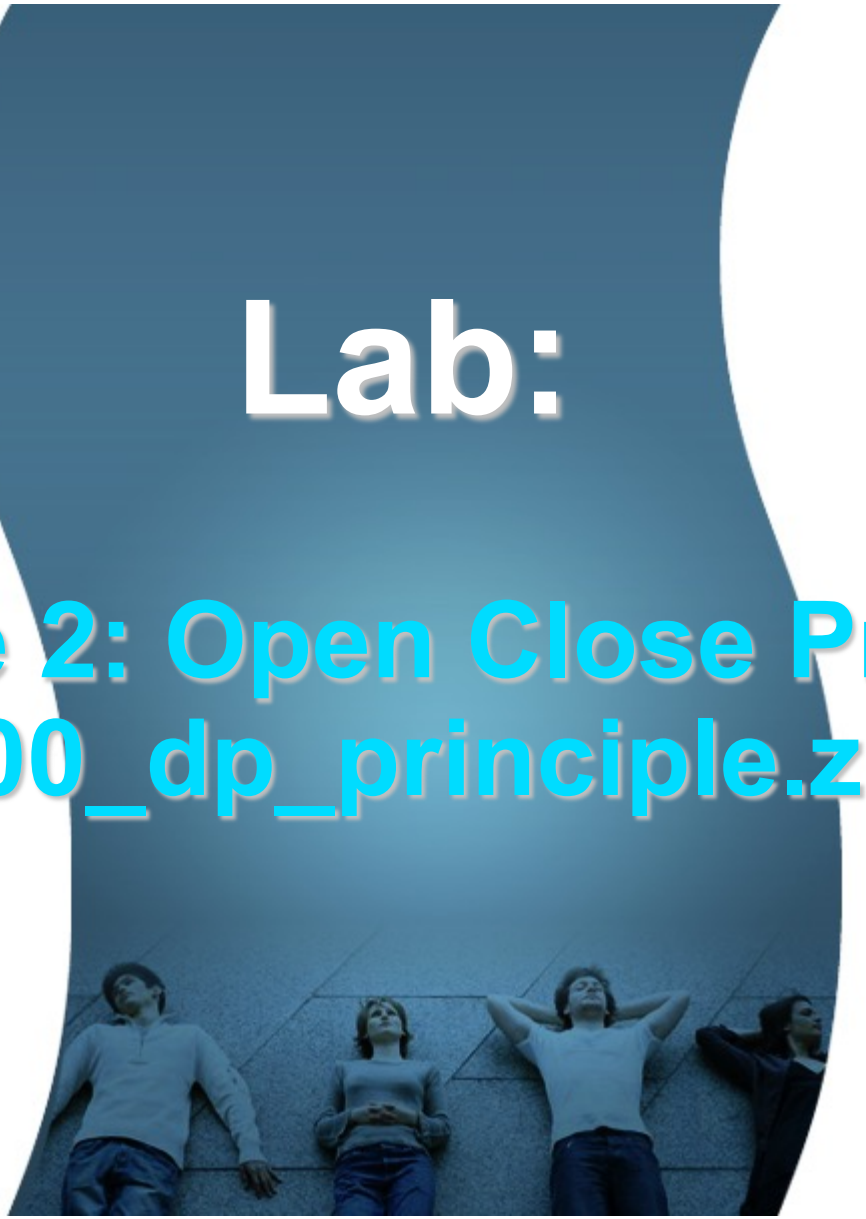
Exercise 1: Single Responsibility 9000_dp_principle.zip

Open Closed Design Principle

- Motivation
  - Software is always changing
  - Change should be able to be made with minimum impact to the rest of the application
- What is it?
  - Classes should be Open for extension (for new functionality) but closed for modification
  - Design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged
- Benefit
  - The existing code, which has been already tested, does not need to be changed

Lab:

Exercise 2: Open Close Principle 9000_dp_principle.zip



Liskov Substitution Principle(LSP) Principle

- Motivation
  - It is easy to inherit a parent class and override the existing features
- What is it?
  - Derived classes must be usable through the base class, without the need for the user to know the difference
  - Inheritance should behave in the way that a derived type should add features, and not violate any of the superclass' existing features
- Example
  - In Java, if you see code where “instanceof”, it indicates LSP is violated

Lab:

Exercise 3: LSP Principle 9000_dp_principle.zip

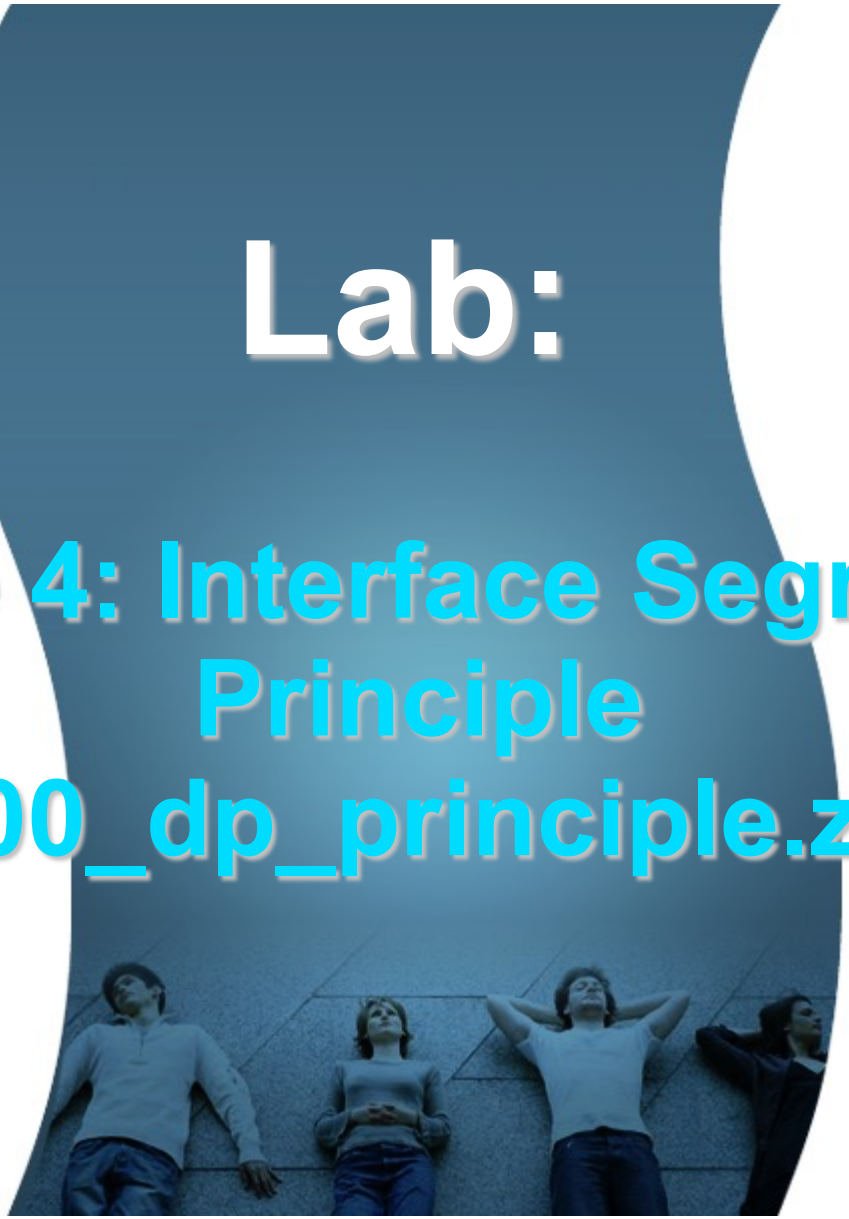
Interface Segregation Principle

- Motivation
  - Clients should not be forced to implement interfaces they don't use.

- What is it?
  - Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule

Lab:

Exercise 4: Interface Segregation Principle
9000_dp_principle.zip



Dependency Inversion Principle

- What is it?
  - High-level modules should not depend on low-level modules. Both should depend on abstractions
  - Example: When concrete class A (higher-level) uses concrete class B (lower-level), the direction of dependency is from A to B. When B changes, A needs to be changed (at compile time). By introducing an abstraction C (maybe Java interface in Java program) between A and B, now the direction of dependency is now from B to C, which inverses the dependency
- Benefit
  - De-coupling of code between modules

Dependency Inversion Principle Tips

- All member variables in a class must be interfaces or abstract types
- Classes must connect only through interface/abstract
- No class should derive from a concrete class
- No method should override an implemented method
- All variable instantiation requires the implementation of a Creational pattern as the Factory Method or the Factory pattern, or the more elaborated use of a Dependency Injection framework

Lab:

Exercise 5: Dependency Inversion Principle
9000_dp_principle.zip

Other Design Principles

Other Popular Design Principles

- Encapsulate what changes
- Don’t Repeat Yourself (DRY)
- Don't look for things. Ask for things. (Dependency Injection)
- Favor Composition over Inheritance
- Program to the Interface not to the Implementation
- Delegation Principle

Encapsulate What Changes

- What is it?
  - Encapsulate the code that is expected to change in the future
- Benefit
  - Better testability because existing code does not need to be changed
  - Better maintainability because you go to a single place for change
- Examples
  - Factory pattern: encapsulates object creation, provides flexibility to introduce new way of creating object later with no impact on existing code

Lab:

Exercise 6: Encapsulate what changes 9000_dp_principle.zip

Don't Repeat Yourself (DRY)

- What is it?
  - Don't write duplicate code
  - Use Abstraction instead in order to abstract common things in one place
- Benefit
  - Increased maintainability of the code
- Examples
  - If/Else and Switch statements tend to duplicate

Lab:

Exercise 7: DRY Principle
9000_dp_principle.zip

Don't look for things. Ask for things.

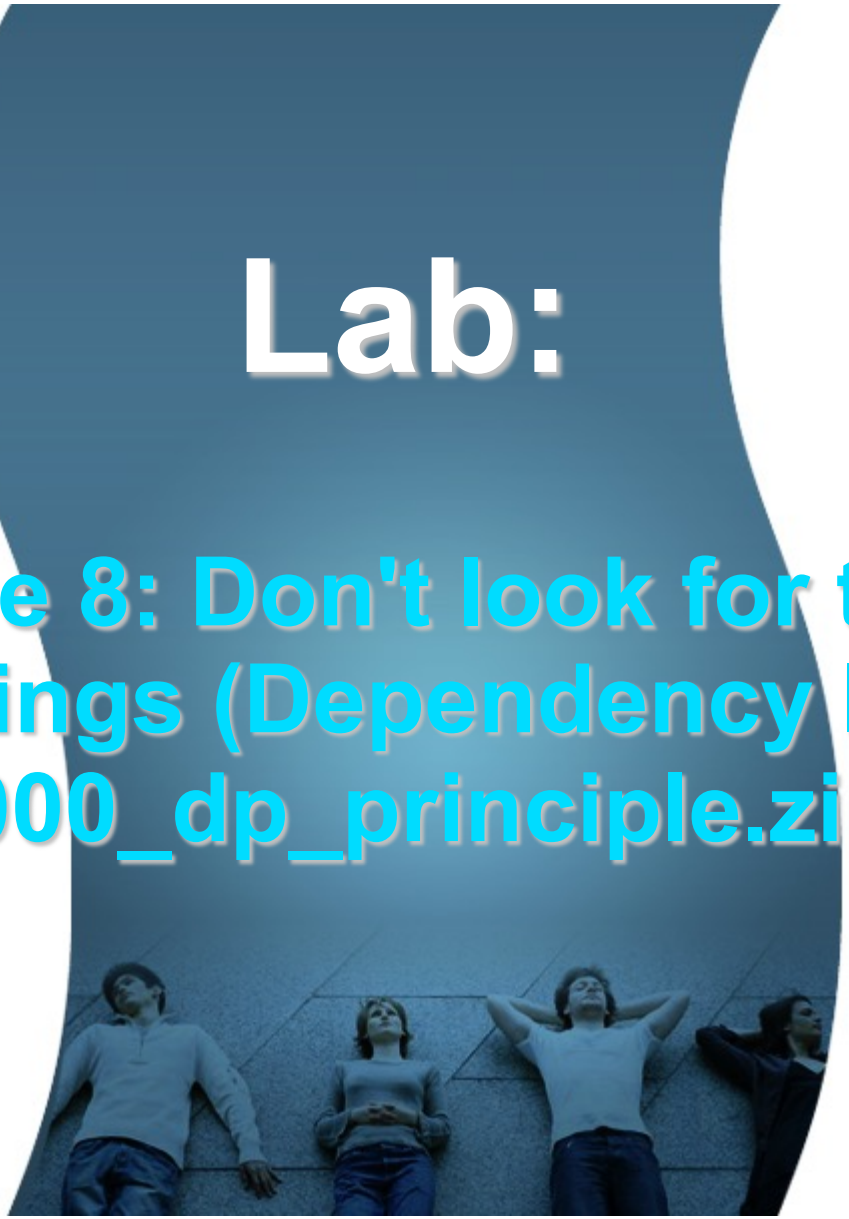
- What is it?
  - Do not construct (using *new* keyword) or look for a dependency object. Instead get it injected
  - This is called “dependency injection”

- Benefit
  - Dependency resolution is separated from business logic
  - Unit testing will be easier via mocking the dependency
  - Dependency resolution can be done via Dependency Injection framework (such as Spring DI)

Lab:

Exercise 8: Don't look for things. Ask for things (Dependency Injection)

9000_dp_principle.zip



Favor Composition over Inheritance

- What is it?
  - Favor composition over inheritance for encapsulating changing behavior
- Benefit
  - Composition provides higher flexibility and other benefits (see the following slides for more details)

Composition vs Inheritance (1)

- Higher flexibility
  - With Inheritance, you must choose which class you are extending at compile time, and it cannot be changed at runtime
  - With Composition, you just define an abstraction which you want to use, which can hold different implementation during runtime
- Code reuse
  - Through Inheritance, you can only extend one class, which means you code can only reuse just one class, not more than one
- Unit testing
  - When you design your class using Inheritance, you must have parent class in order to test child class. There is no way you can provide mock implementation of parent class.
  - When you design classes using Composition, they are easier to test because you can supply mock implementation of the classes you are using (dependencies)

Composition vs Inheritance (2)

- Encapsulation
  - Though both Inheritance and Composition allows code reuse, Inheritance breaks encapsulation because in case of Inheritance, sub class is dependent upon super class behavior. If parent classes changes its behavior, then child class is also get affected
- Final class
  - Composition allows code reuse even from final classes, which is not possible using Inheritance because you cannot extend final class in Java, which is necessary for Inheritance to reuse code.

When to use Inheritance?

- Both classes (super class and subclass) are in the same logical domain
- The subclass is a proper subtype of super class
- The super class's implementation is necessary or appropriate for the subclass
- The enhancements made by the subclass is “additive”

source: https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

When to use Inheritance over composition for changing behavior?

- When the changing behavior varies completely upon sub-types
  - *HourlyEmployee* and *SalariedEmployee* classes are subclasses of Employee class and computation of monthly payment varies based on sub-type only

- When the changing behavior depends on the fields of the subclass
  - Computation of monthly payment uses “hoursWorked” and “hourlyRate” fields of *HourlyEmployee* subtype and “*monthlySalary*” field for *SalariedEmployee* subtype

Program to Interface not Implementation

- What is it?
  - Always program to the interface and not to implementation
  - Use interface type on variables, return types of method or argument type of methods
- Benefit
  - Flexible code which can work with any new implementation of the interface

Delegation Principle

- What is it?
  - Don't do all stuff by yourself, delegate it to respective class
- Benefit
  - No duplication of code
  - Easy to modify behavior
- Examples at code
  - equals() and hashCode() method in Java Object class

Code with Passion!