

S.O.L.I.D.

The Benefits and Potential of Using SOLID Principles

- S** Single Responsibility
- O** Open/Closed
- L** Liskov substitution
- I** Interface Segregation
- D** Dependency Inversion



SoftUni Team
Technical Trainers



SoftUni

Software University
<https://softuni.bg>

sli.do

#java-advanced

Table of Contents

1. **S.O.L.I.D.** Principles
2. **Single** Responsibility
3. **Open / Closed**
4. **Liskov** Substitution
5. **Interface** Segregation
6. **Dependency** Inversion





SOLID

SOLID Principles

- **S** – Single responsibility principle – class should only have one responsibility
- **O** – Open–closed principle – open for extension, but closed for modification
- **L** – Liskov substitution principle – objects should be replaceable with instances of their subtypes without altering the correctness of that program

- **I** – Interface segregation principle – many specific interfaces are better than one general interface
- **D** – Dependency inversion principle – one should depend upon abstractions, not concretions

Single
responsibility



Liskov
substitution



Open-Closed
Principle

Liskov
substitution



Dependency
inversion



Interface
segregation





Single Responsibility

Single Responsibility Principle

- A class should **have only one responsibility**
 - Reduces **dependency** complexity
 - Each additional responsibility is an **axis to change the class**



```
public class HeroSettings {  
    public static void changeName(Hero hero) {  
        // Grant option to change  
    }  
}
```


Single Responsibility Principle

- Still, classes **can have multiple methods**
 - Each method should have a **single functionality** part of the class responsibility

```
public class HeroSettings {  
    public static void changeName(Hero hero) {  
        // Grant option to change name  
    }  
    public static void selectRole(Hero hero) {  
        // Grant option to select role  
    }  
}
```





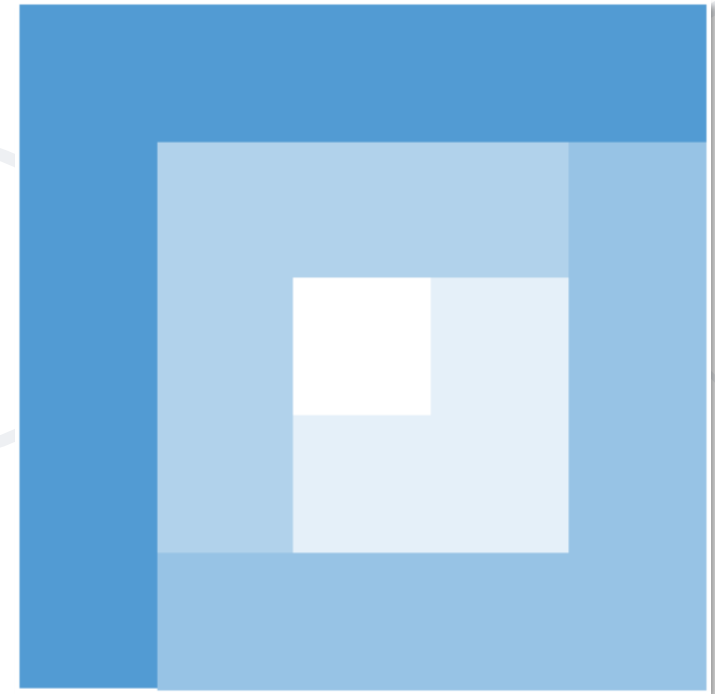
Open / Closed

What is Open/Closed?

- Software entities (classes, modules, functions, etc.) should be
 - **open** for **extension**
 - **closed** for **modification**
- **Design** the code in a way that **new** functionality can be added with **minimum changes** in the **existing** code



- Implementation takes **future growth** into **consideration**
- New or **modified functionality** affects little or not at all the internal structure and data flow of the system



- Software reusability refers to **design features** of a software **element** that enhance its **suitability** for **reuse**
- Modularity
- Low coupling
- High cohesion
- Coupling and Cohesion



- **Cascading changes** through modules
- Each change **requires re-testing**
- Logic **depends** on conditional statements



- Inheritance / Abstraction
- Inheritance / Template Method pattern
- Composition / Strategy patterns





Liskov Substitution

What is Liskov Substitution?

- Derived types must be **completely substitutable** for their base types
- Reference to the base class can be replaced with a derived class without affecting the functionality of the program module
- Derived classes extend without replacing the functionality of old classes



- OOP Inheritance

Student IS-A Person

- Plus LSP

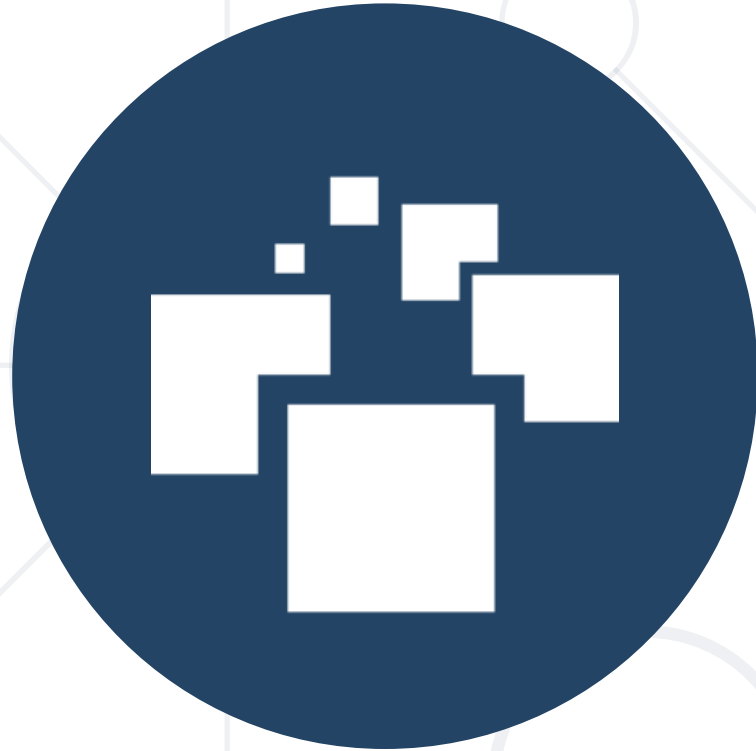
Student IS-SUBSTITUTED-FOR Person

OCP vs LSP

- Liskov Substitution Principle is just an **extension** of the Open-Closed Principle
- We must make sure that new derived classes are extending the base classes **without changing** their **behavior**



- Violations
 - Type Checking
 - Overridden methods say "I am not implemented"
 - Base class depends on its subtypes
- Solutions
 - Refactoring in the **base class**



Interface Segregation

ISP – Interface Segregation Principle

- Clients should **not** be **forced** to **depend** on methods they do **not use**
- Segregate interfaces
 - Prefer **small, cohesive** interfaces
 - Divide "**fat**" interfaces into "**role**" interfaces

- Classes whose interfaces are not cohesive have "fat" interfaces

```
public interface Worker {  
    void work();  
    void sleep();  
}
```

Class **Employee** is
OK

```
public class Robot implements Worker {  
    public void work() {}  
    public void sleep() {  
        throw new UnsupportedOperationException();  
    }  
}
```

- Having "fat" interfaces:
 - Classes have methods they do not use
 - Increased **coupling**
 - Reduced flexibility
 - Reduced maintainability

- Solutions to broken ISP
 - **Small** interfaces
 - **Cohesive** interfaces
 - Let the client **define** interfaces – "role" interfaces

- Small and Cohesive "Role" Interfaces

```
public interface Worker {  
    void work();  
}
```

```
public interface Sleeper {  
    void sleep();  
}
```

```
public class Robot implements Worker {  
    void work() {  
        // Do some work...  
    }  
}
```



Dependency Inversion

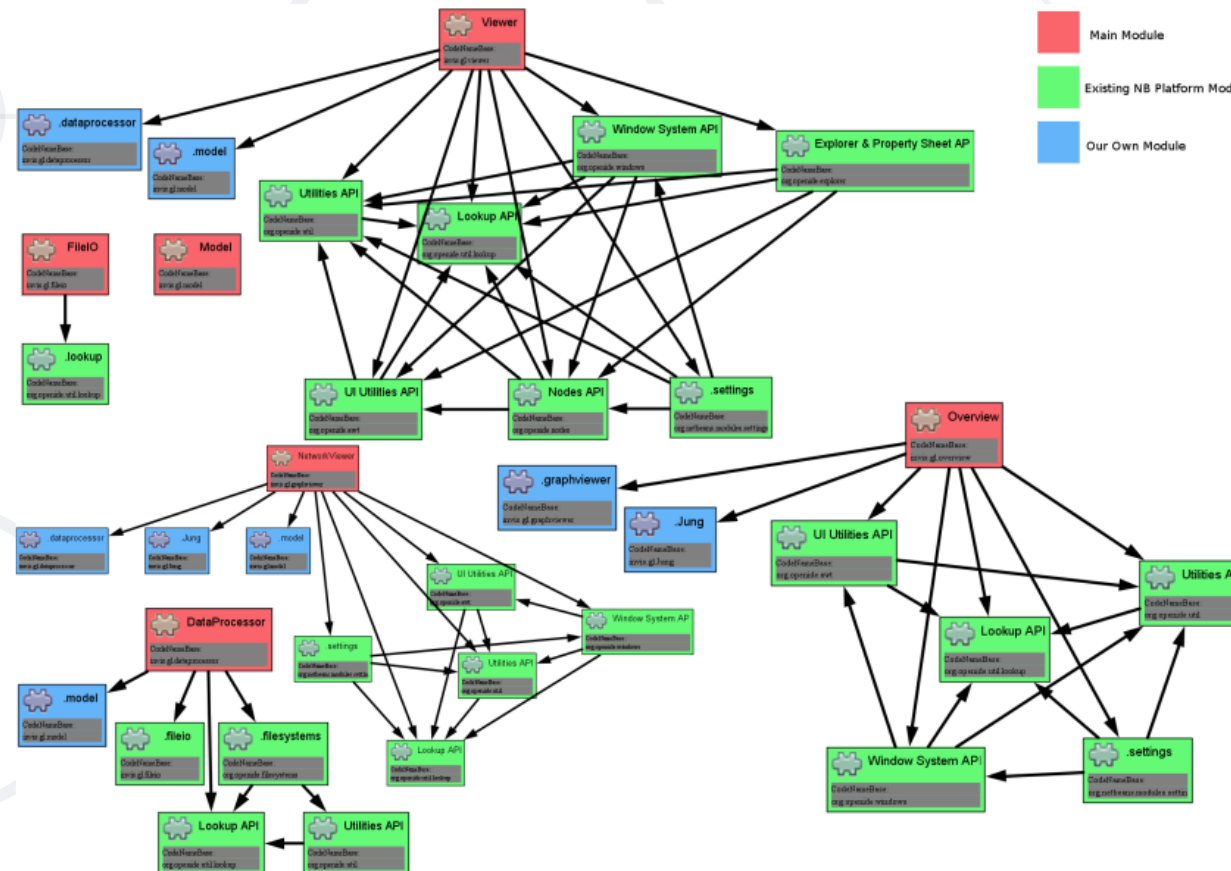
Dependency Inversion Principle (DIP)

- High-level modules should not depend on low-level modules
 - Both should **depend** on **abstractions**
- Abstractions should not depend on details
- Details should depend on abstractions
- Goal: **decoupling between modules** through abstractions

-

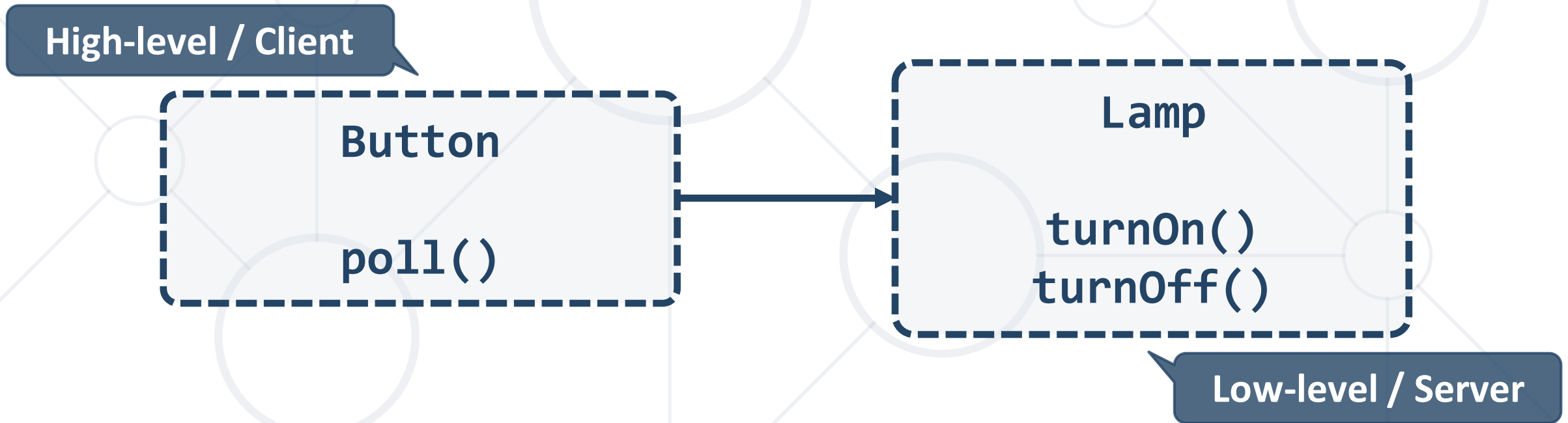
Dependencies and Coupling

- The goal is to **depend on abstractions**



The Problem

- Button → Lamp Example – **Robert Martin**
- Button **depends on** Lamp



Dependency Inversion Solution

- Find the abstraction independent of details



- A **dependency** is an external component / system:
 - Framework
 - Third party library
 - Database
 - File system
 - Email
 - Web service
 - System resource (e.g. clock)
 - Configuration
 - The **new** keyword
 - Static method
 - Global function
 - Random generator
 - System.in / System.out

- **Constructor injection** - dependencies are passed through **constructors**
 - **Pros**
 - Classes **self-documenting** requirements
 - Works well without a container
 - Always **valid state**
 - **Cons**
 - Many parameters
 - Some methods may not need everything



Constructor Injection – Example

```
public class Copy {  
    private Reader reader;  
    private Writer writer;  
    public Copy(Reader reader, Writer writer) {  
        this.reader = reader;  
        this.writer = writer;  
    }  
    public void copyAll() {}  
}
```

- **Setter Injection** - dependencies are passed through **setters**
 - **Pros**
 - Can be changed anytime
 - Very **flexible**
 - **Cons**
 - Possible **invalid state** of the object
 - Less intuitive

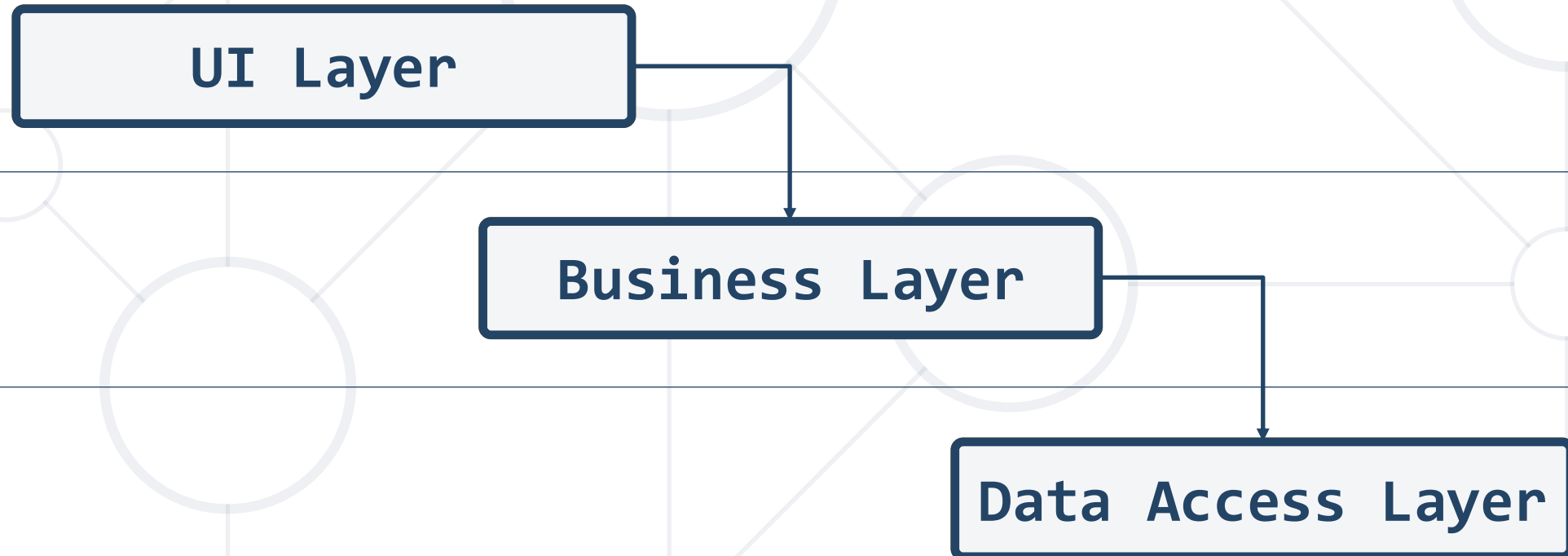
Setter Injection – Example

```
public class Copy {  
    private Reader reader;  
    private Writer writer;  
    public void setReader(Reader reader) {}  
    public void setWriter(Writer writer) {}  
    public void copyAll() {}  
}
```

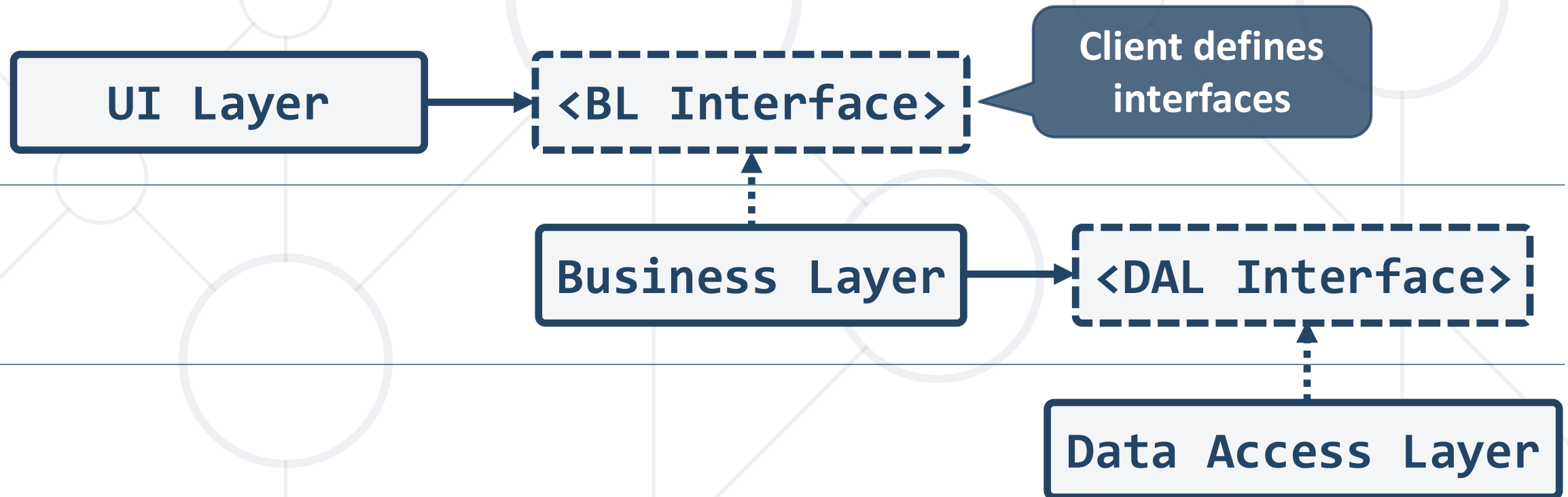
- **Parameter injection** - dependencies are passed through **method parameters**
 - **Pros**
 - No change in rest of the class
 - Very flexible
 - **Cons**
 - Many parameters
 - Breaks the method signature

```
public class Copy {  
    public void copyAll(Reader reader, Writer writer) {}  
}
```

- Traditional programming
 - **High-level** modules use **low-level** modules

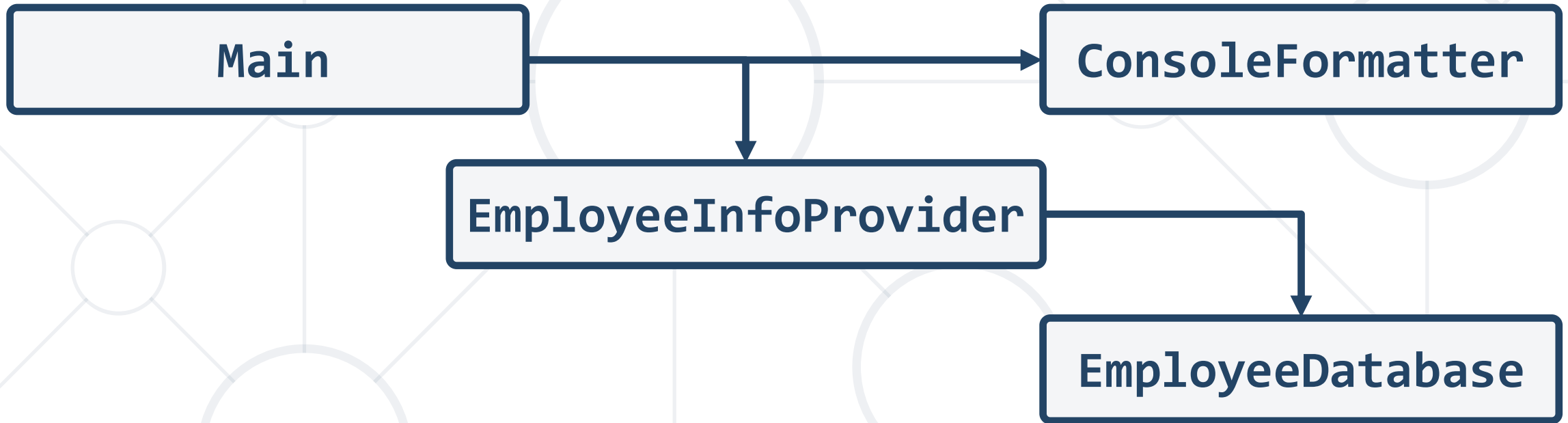


- Dependency Inversion Layering
 - **High** and **low-level** modules **depend on abstractions**



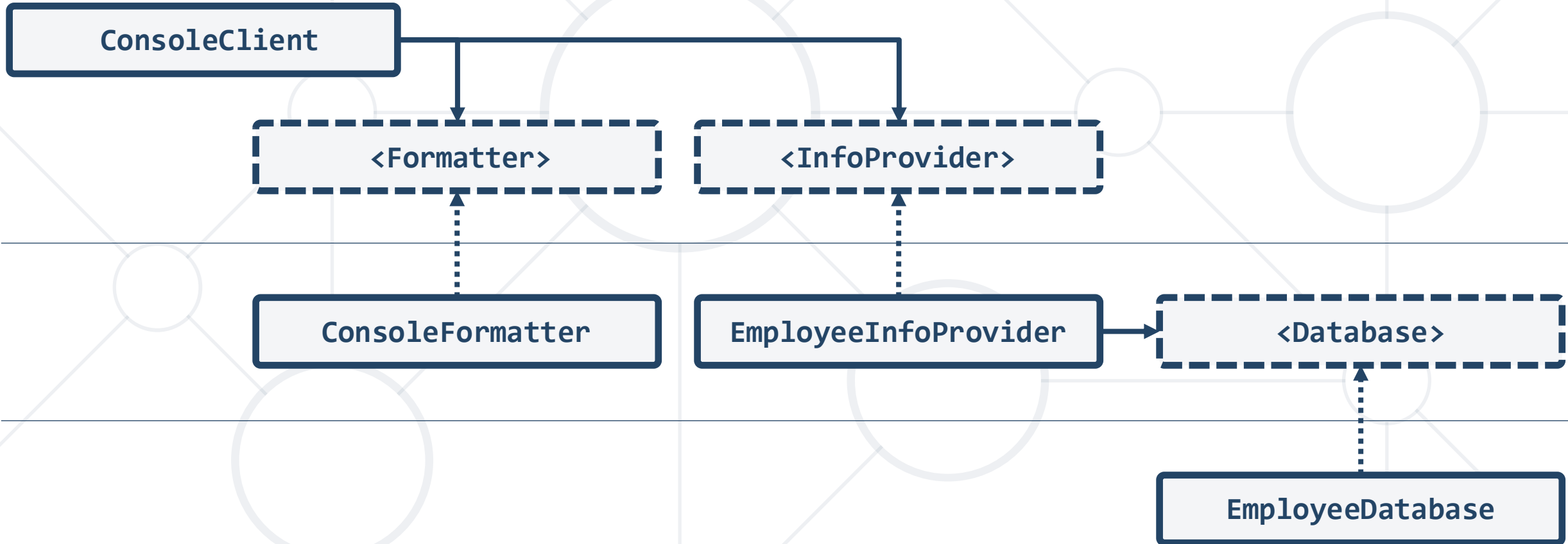
Problem: Employee Info

- You are given some classes



- Refactor the code so that it conforms to DIP

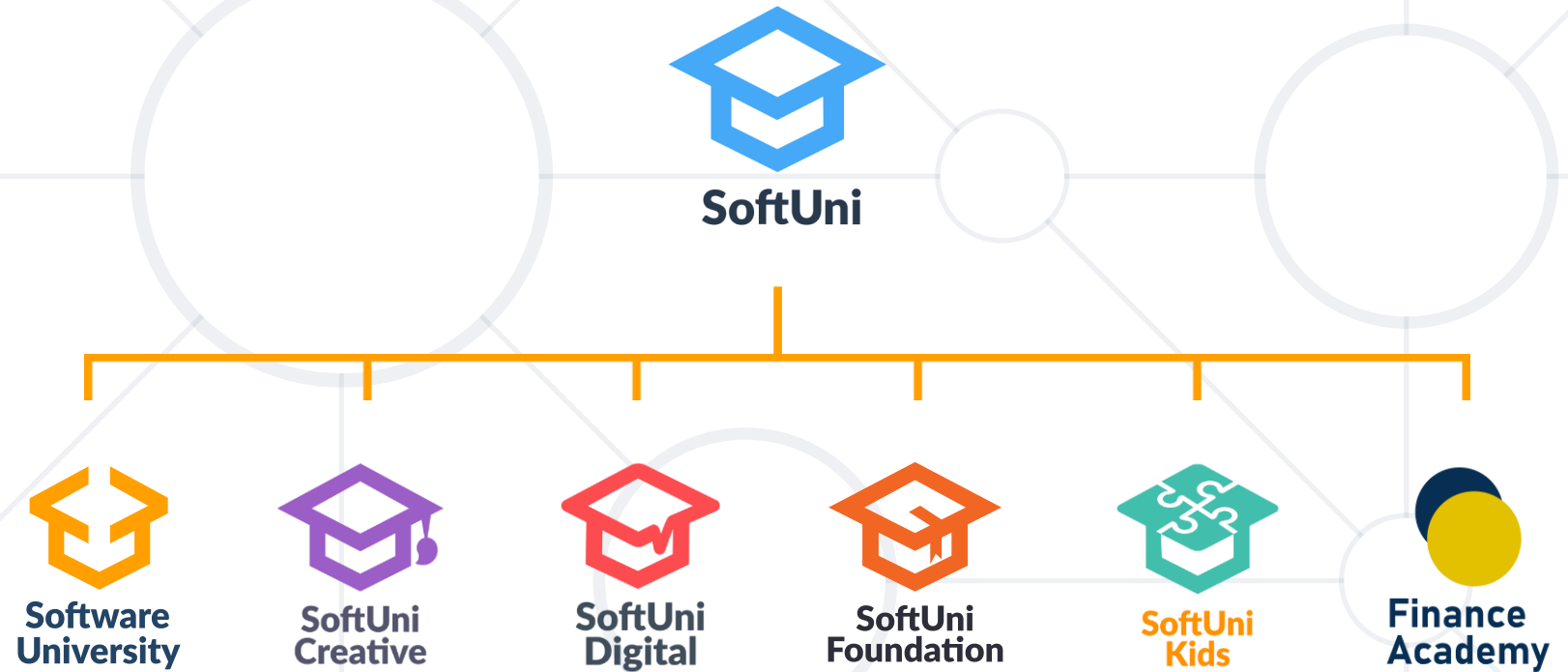
Solution: Employee Info



- **SOLID** principles make the software:
 - Understandable
 - Flexible
 - Maintainable



Questions?



SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

