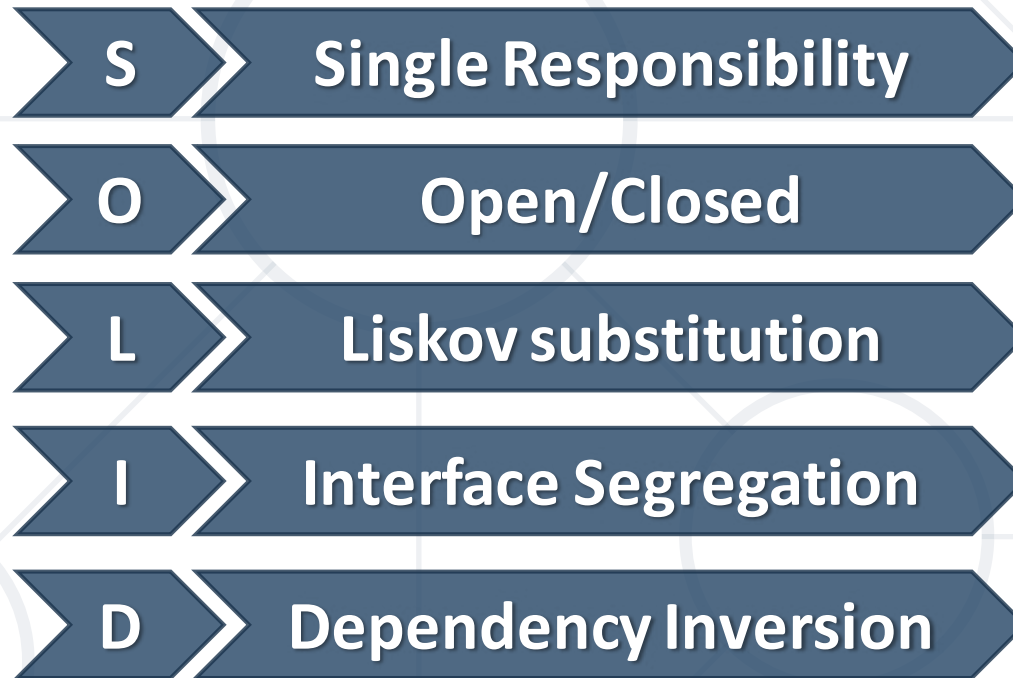


# S.O.L.I.D.

The benefits and potential of using SOLID principles



**SoftUni Team**  
Technical Trainers



# Table of Contents

- Single Responsibility
  - A Class Should Have Only One Reason to Change
- Open / Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion



# Have a Question?

sli.do

**#java-advanced**

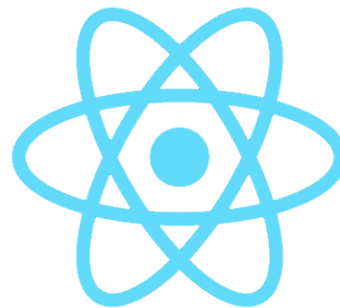


**SOLID**

**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



S.O.L.I.D




# Single Responsibility

A Class Should Have Only One Reason to Change

# Single Responsibility Principle

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

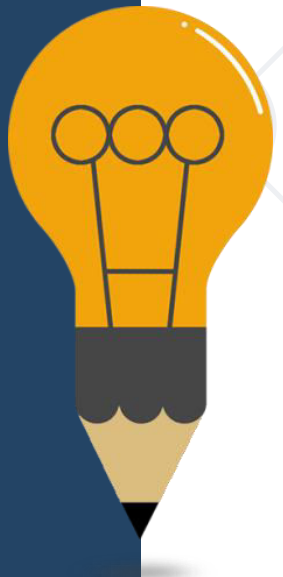


```
public class HeroSettings {  
    public static void changeName(Hero hero) {  
        // Grant option to change  
    }  
}
```



# Single Responsibility Principle

- Still classes **can have multiple methods**
  - Each method should have **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  
    }  
}
```

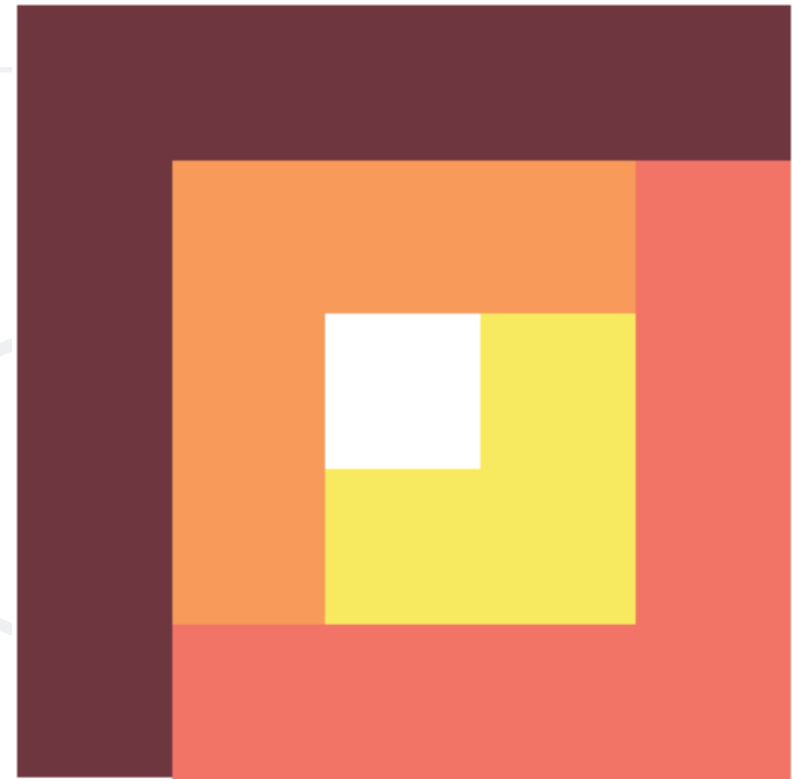


**Open / Closed**

# What is Open/Closed?

*"Software entities (classes, modules, functions, etc.) should be **open for extension**, but closed for **modification**."*

- Implementation takes **future growth** into consideration
- Extensible system is one whose internal **structure** and **data flow** are **minimally or not affected** by new or modified functionality



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



# Open/Closed Principle (OCP)

- **Design and writing of the code** should be done in a way that **new functionality should be added with minimum changes in the existing code**
- Changes to source code are not required

- 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 just **extend without replacing** the functionality of old classes

- OOP Inheritance

Student **IS-A** Person

- Plus LSP

Student **IS-SUBSTITUTED-FOR** Person

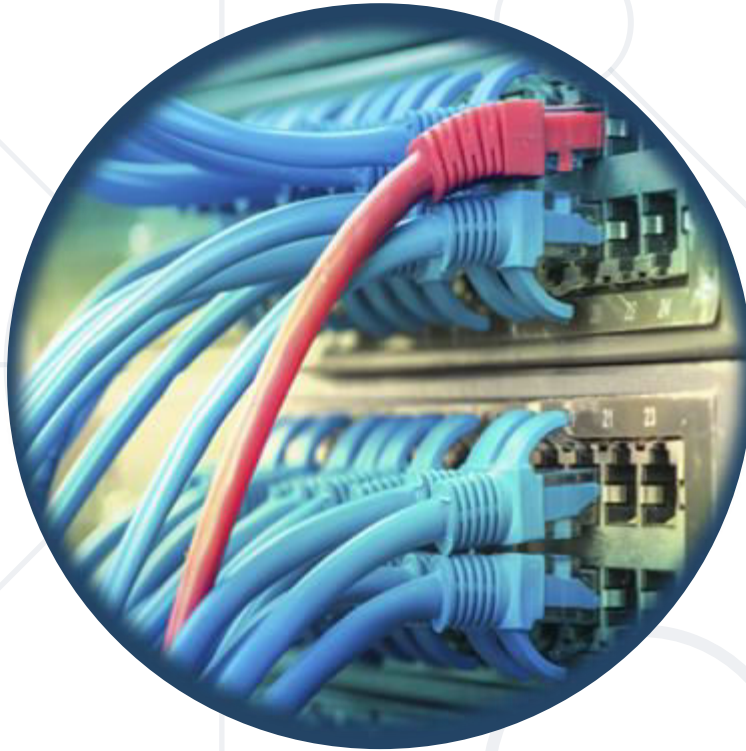
"Liskov Substitution Principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior."

- Type Checking
- Overridden methods say "I am not implemented"
- Base class depends on its subtypes



- "Tell, Don't Ask"
- Refactoring in the **base class**





# **Interface Segregation**

## **Clients Require Cohesive Interfaces**

*"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();  
    }  
}
```

# "Fat" Interfaces

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

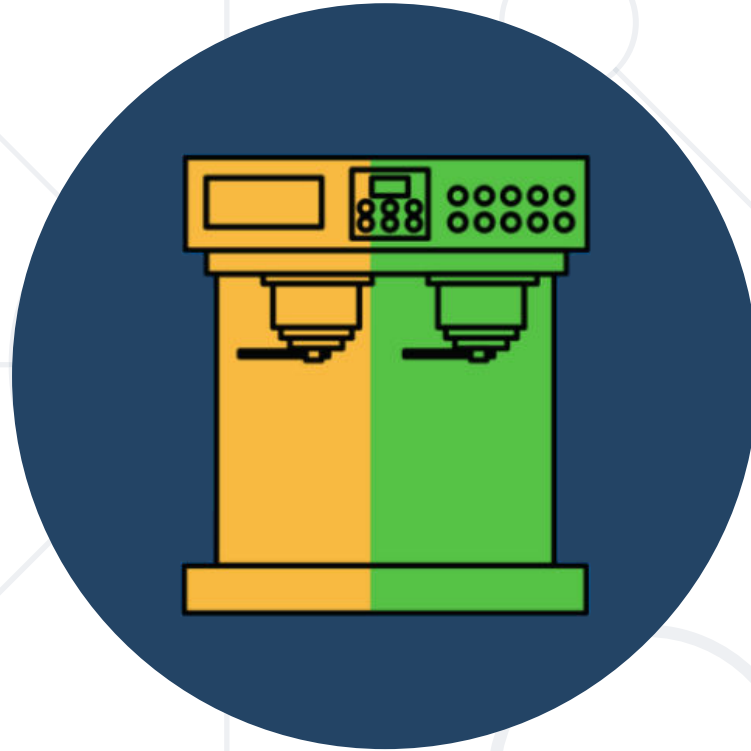
- Solutions to broken ISP
  - **Small** interfaces
  - **Cohesive** interfaces
  - Let the **client define interfaces** – "**role**" interfaces

- 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

## Flip Dependencies

# Dependency Inversion Principle (DIP)

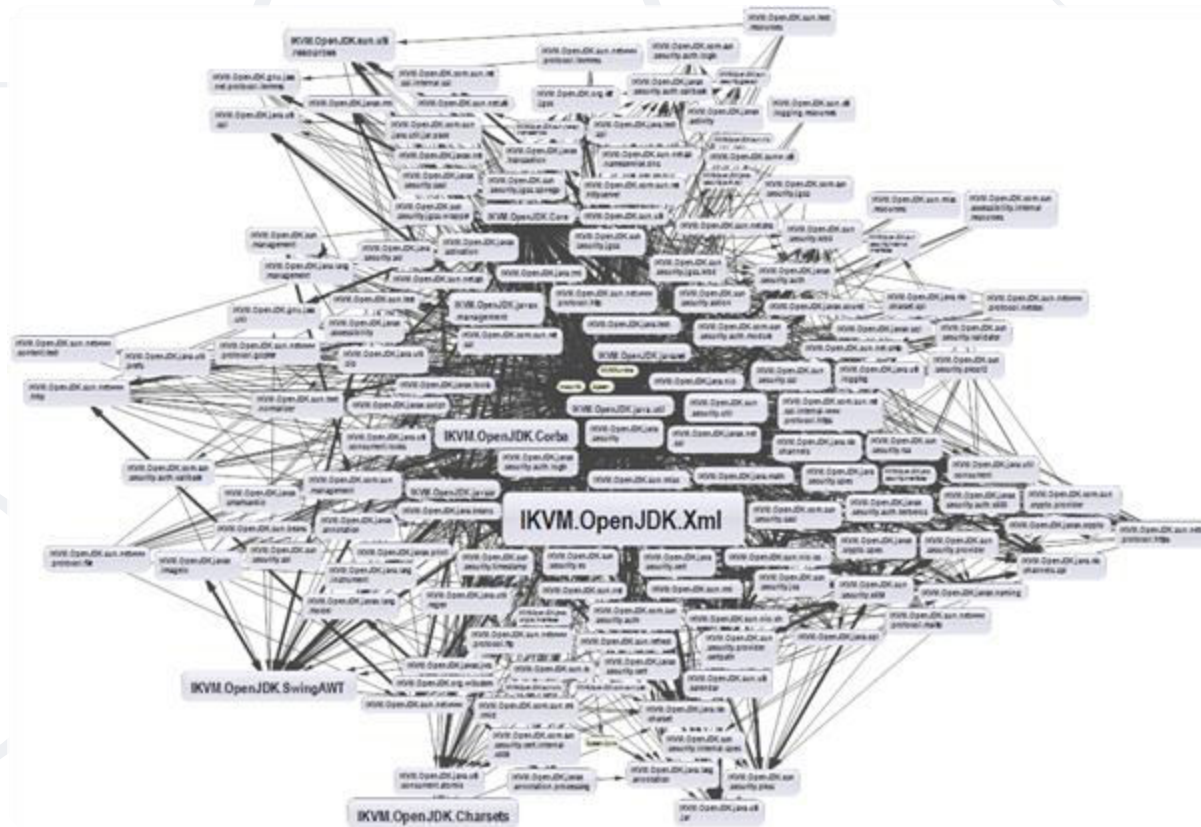
*"**Dependency Inversion Principle** says that high-level modules should not depend on low-level modules. Both should depend on **abstractions**."*

*"Abstractions should not depend on details. Details should **depend on abstractions**."*

- Goal: **decoupling between modules** through abstractions

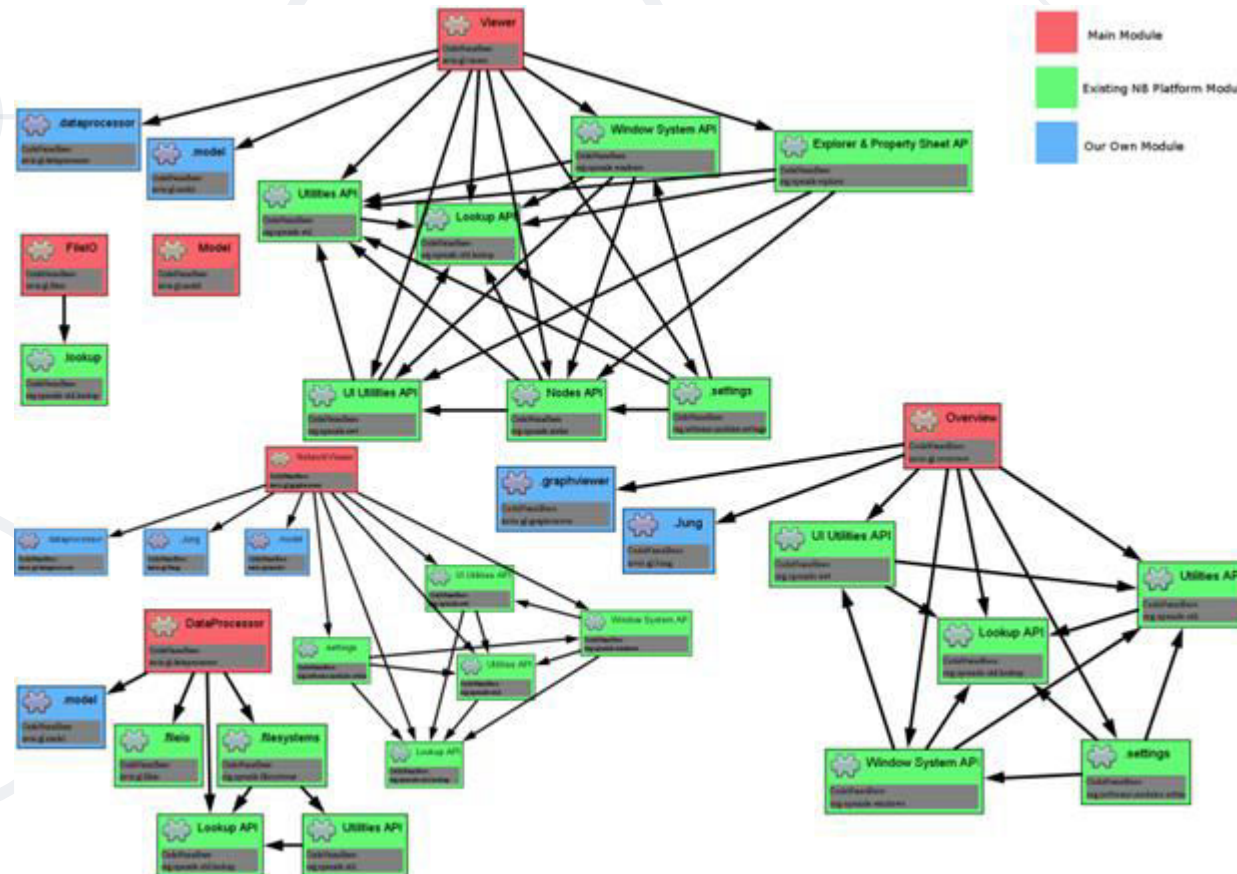
# Dependencies and Coupling (1)

- What happens when modules **depend directly** on **other modules**



# Dependencies and Coupling (2)

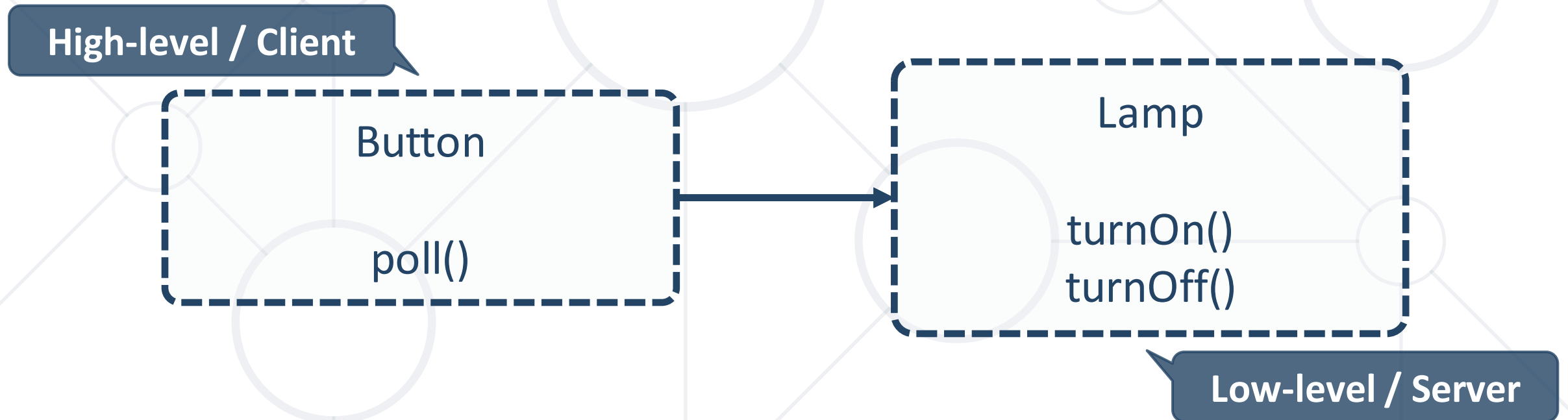
- The goal is to **depend on abstractions**





# The Problem

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



# Dependency Inversion Solution

- Find the abstraction independent of details



- A **dependency** is any 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 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() {}  
}
```

# How to DIP? (3)

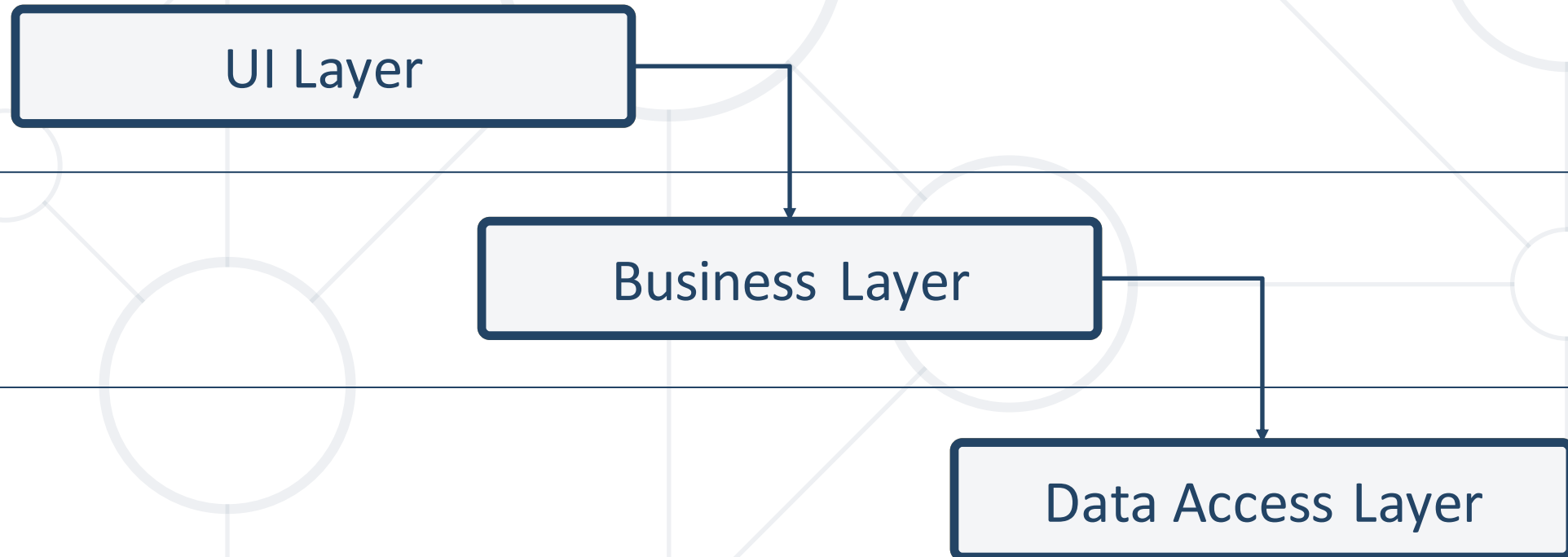
- **Parameter injection** - dependencies are passed through **method parameters**
  - **Pros**
    - No change in rest of the class
    - Very flexible
  - **Cons**
    - Many parameters
    - Breaks the method signature

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

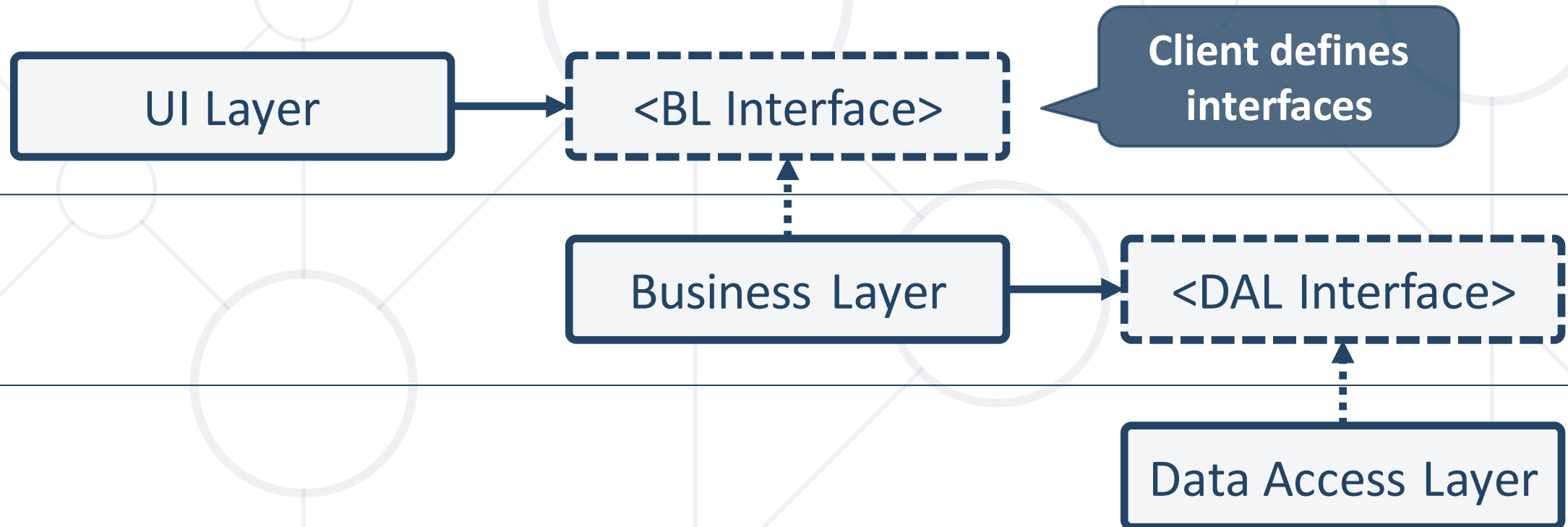


# Layering (1)

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

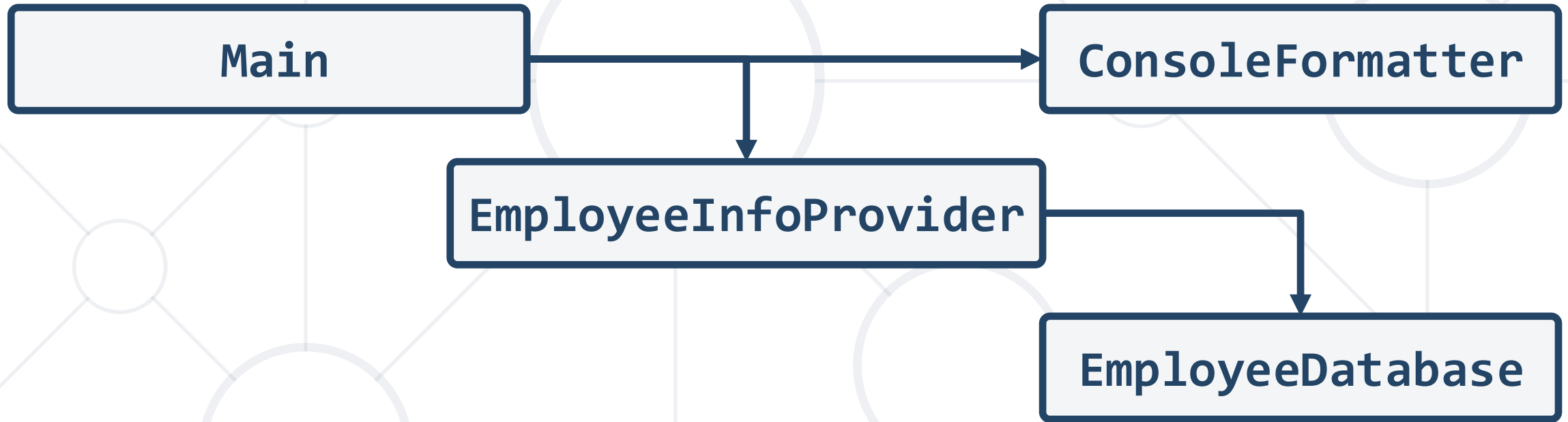


- Dependency Inversion Layering
  - **High** and **low-level** modules **depend on abstractions**



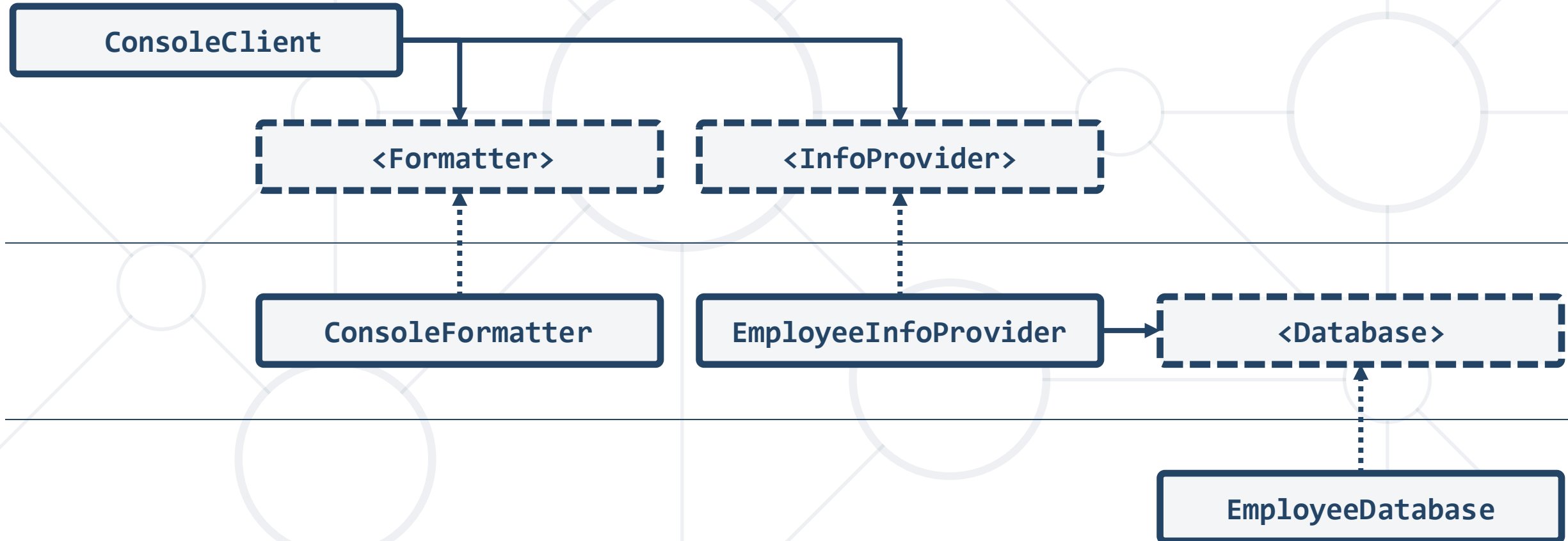
# Problem: Employee Info

- You are given some classes



- Refactor the code so that it conforms to DIP

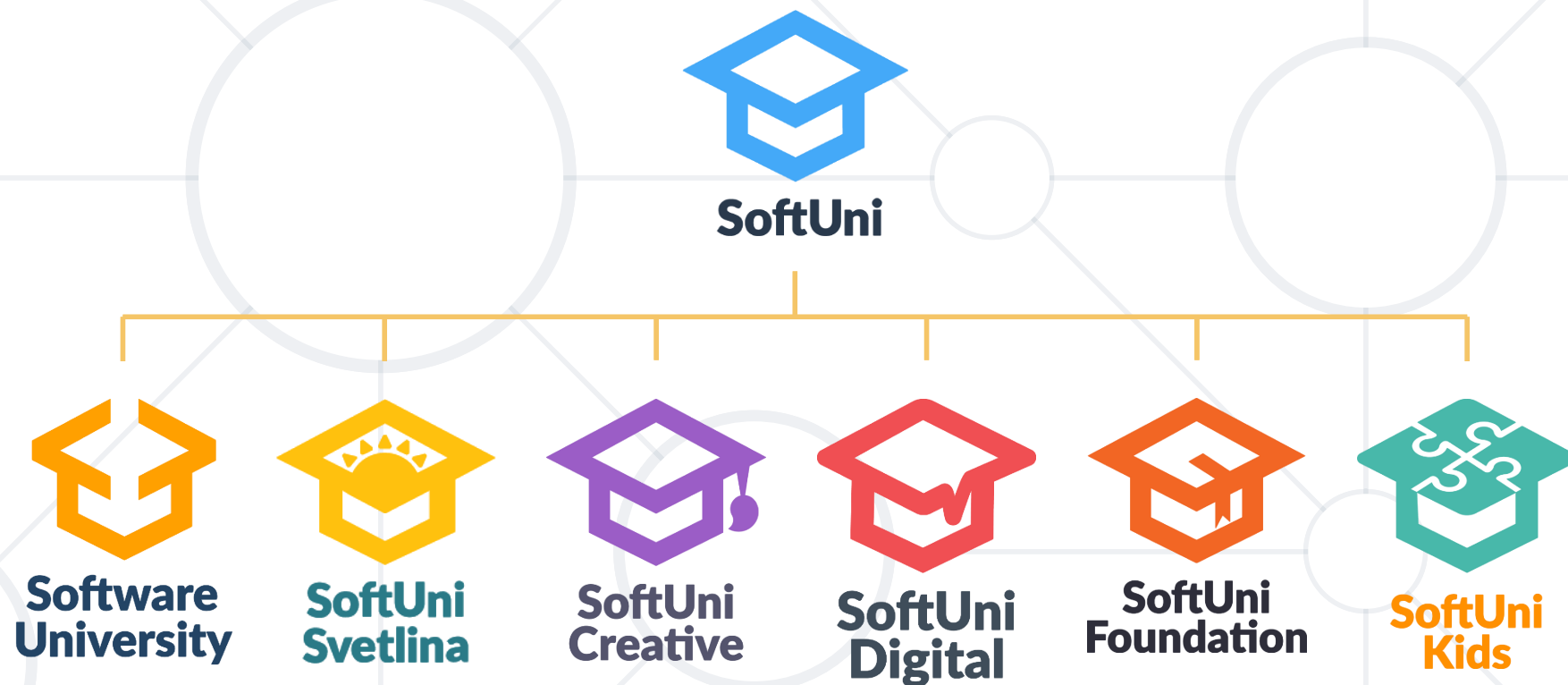
# Solution: Employee Info



- **SOLID** principle make software more:
  - Understandable
  - Flexible
  - Maintainable



# Questions?



# SoftUni Diamond Partners



**XS**software



**SBTech**  
*we know sports*



telenor



**SoftwareGroup**  
*doing it right*

**NETPEAK**



**SmartIT**



**Postbank**

*Решения за твоето утре*

**SUPER  
HOSTING  
.BG**

**INDEAVR**

*Serving the high achievers*



**INFRAGISTICS®**



**STEMO®**  
*Computer Systems & Software*



# SoftUni Organizational Partners



OneBit  
SOFTWARE



WORLD  
OF  
MYTHS



# Trainings @ Software University (SoftUni)

- Software University – High-Quality Education and Employment Opportunities
  - [softuni.bg](http://softuni.bg)
- Software University Foundation
  - <http://softuni.foundation/>
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

