Software Design Principles

Parinya Ekparinya

Parinya.Ek@kmitl.ac.th

Software Architecture and Design

Outline

- Design Smells
- The SOLID Principles
  - The **S**ingle-Responsibility Principle
  - The **O**pen–Closed Principle
  - The **L**iskov Substitution Principle
  - The **I**nterface Segregation Principle
  - The **D**ependency Inversion Principle

Design Smells

"Design smells are the odors of rotting software."

– Robert C. Martin

- Rigidity
- Fragility
- Immobility
- Viscosity
- Needless complexity
- Needless repetition
- Opacity

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Rigidity

- (noun) inability to be changed or adapted. – Oxford Languages
- Rigidity is the tendency for software to be difficult to change, even in simple ways.
- A design is rigid if a single change causes a cascade of subsequent changes in dependent modules.
- The more modules that must be changed, the more rigid the design.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Fragility

- (noun) the quality of being easily broken or damaged. – Oxford Languages
- Fragility is the tendency of a program to break in many places when a single change is made.
- Often, the new problems are in areas that have no conceptual relationship with the area that was changed.
- Fixing those problems leads to even more problems, and the development team begins to resemble a dog chasing its tail.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Immobility

- (noun) the state of not moving; motionlessness. – Oxford Languages
- Immobility is the inability to reuse software from other projects or from parts of the same project.
- A design is immobile when it contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great.
- So immobile software components is simply rewritten instead of reused.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Viscosity

- (noun) the state of being thick, sticky, and semifluid in consistency, due to internal friction. – Oxford Languages
- Viscosity comes in two forms: viscosity of the software and viscosity of the environment.
- When faced with a change, developers usually find more than one way to make that change. Some of the ways preserve the design; others do not (i.e., they are hacks). When the design-preserving methods are more difficult to use than the hacks, the viscosity of the design is high.
- Viscosity of environment comes about when the development environment is slow and inefficient. For example, if compile times are very long, developers will be tempted to make changes that don't force large recompiles, even though those changes don't preserve the design.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Needless complexity

- A design smells of needless complexity when it contains elements that aren't currently useful.
- This frequently happens when developers anticipate changes to the requirements and put facilities in the software to deal with those potential changes.
- At first, this may seem like a good thing to do. Unfortunately, the effect is often just the opposite. By preparing for many contingencies, the design becomes littered with constructs that are never used.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Needless repetition

- All too often, software systems are built on dozens or hundreds of repeated code elements.
- When the same code appears over and over again, in slightly different forms, the developers are missing an abstraction.
- Finding all the repetition and eliminating it with an appropriate abstraction may not be high on their priority list, but it would go a long way toward making the system easier to understand and maintain.
- Bugs found in such a repeating unit have to be fixed in every repetition. However, since each repetition is slightly different from every other, the fix is not always the same.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Opacity

- (noun) the condition of lacking transparency or translucence; opaqueness. – Oxford Languages
- Opacity is the tendency of a module to be difficult to understand.
- Code can be written in a clear and expressive manner, or it can be written in an opaque and convoluted manner.
- Code that evolves over time tends to become more and more opaque with age.
- A constant effort to keep the code clear and expressive is required in order to keep opacity to a minimum.

"When I Wrote It, Only God and I Knew the Meaning;

Now God Alone Knows."

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Why Software Rots

- Designs degrade because requirements change in ways that the initial design did not anticipate.
- Often, these changes need to be made quickly and may be made by developers who are not familiar with the original design philosophy.

"Change is inevitable; change is constant."

- We must somehow find a way to make our designs resilient to such changes and use practices that protect them from rotting.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

The SOLID Principles

“On the one hand, if the bricks aren’t well made, the architecture of the building doesn’t matter much.

On the other hand, you can make a substantial mess with well-made bricks.”

Martin, R. C., Grenning, J., & Brown, S. (2018). *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall.

The SOLID Principles

- SOLID is a mnemonic acronym for five principles for object-oriented class design.
- The SOLID principles provide guidance on how to arrange functions and data structures into classes, and how those classes should be interconnected.
- The goal of the principles is the creation of mid-level software structures that:
  - Tolerate change
  - Are easy to understand
  - Are the basis of components that can be used in many software systems
- The term "mid-level" refers to the fact that these principles are applied by programmers working at the module level.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

The Single-Responsibility Principle

The Single-Responsibility Principle (SRP)

- Originally, Robert C. Martin formulated and described the SRP as follows:

  "A module should have one, and only one, reason to change."

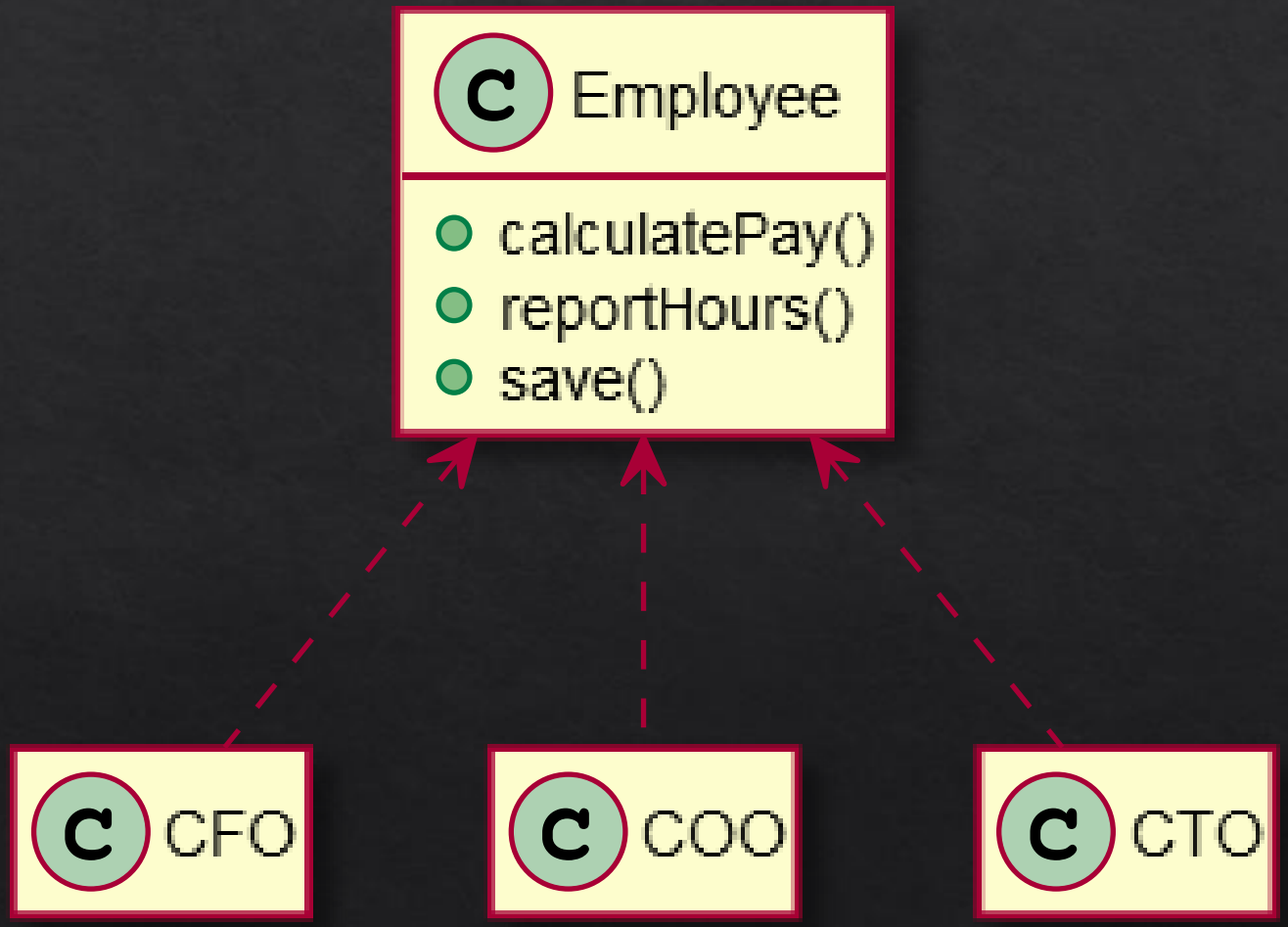
- In his 2018 book, the principle was rephrased to:

  "A module should be responsible to one, and only one, actor."

- What do we mean by the word "module"? The simplest definition is just a source file. Most of the time that definition works fine. Other times, a module is just a cohesive set of functions and data structures.

Martin, R. C., Grenning, J., & Brown, S. (2018). *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall.

Bad Example: SRP Violation (1)



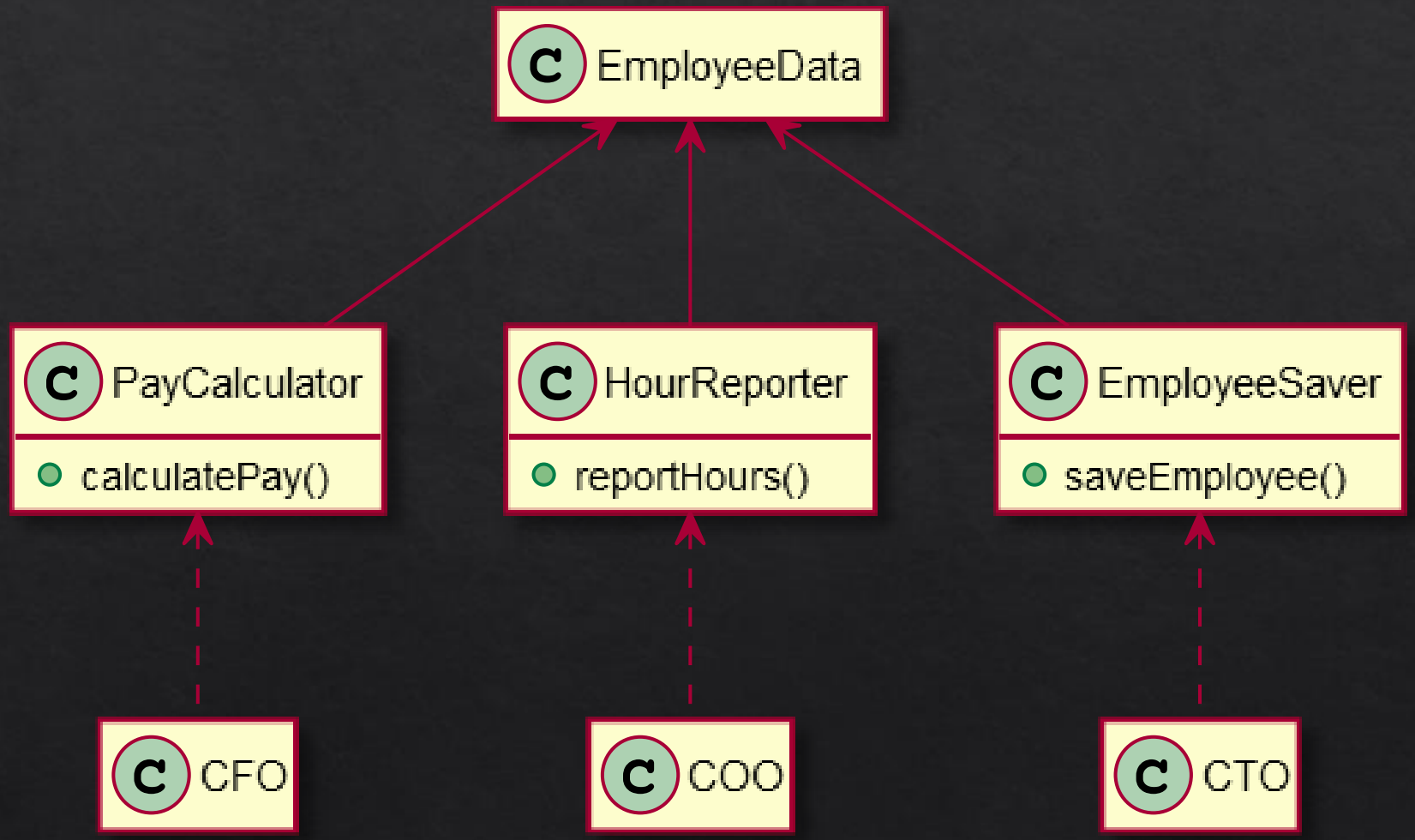
Martin, R. C., Grenning, J., & Brown, S. (2018). *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall.

Bad Example: SRP Violation (2)

- The Employee class violates the SRP because those three methods are responsible to three very different actors.
  - The calculatePay() method is specified by the accounting department, which reports to the CFO.
  - The reportHours() method is specified and used by the human resources department, which reports to the COO.
  - The save() method is specified by the database administrators (DBAs), who report to the CTO.

Martin, R. C., Grenning, J., & Brown, S. (2018). *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall.

Good Example: Fixing SRP Violation



Martin, R. C., Grenning, J., & Brown, S. (2018). *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall.

The Open–Closed Principle

The Open–Closed Principle (OCP)

- In 1988, Bertrand Meyer described the OCP as follows:

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

- In other words, the behavior of a software entity ought to be extendible, without having to modify that entity.
- Of all the principles of object-oriented design, this is arguably the most important.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

OCP Example

- https://reflectoring.io/open-closed-principle-explained/

The Liskov Substitution Principle

The Liskov Substitution Principle (LSP)

- The principle was initially introduced by Barbara Liskov in 1988.
- Barbara Liskov and Jeannette Wing described the principle in a 1994 paper as follows:

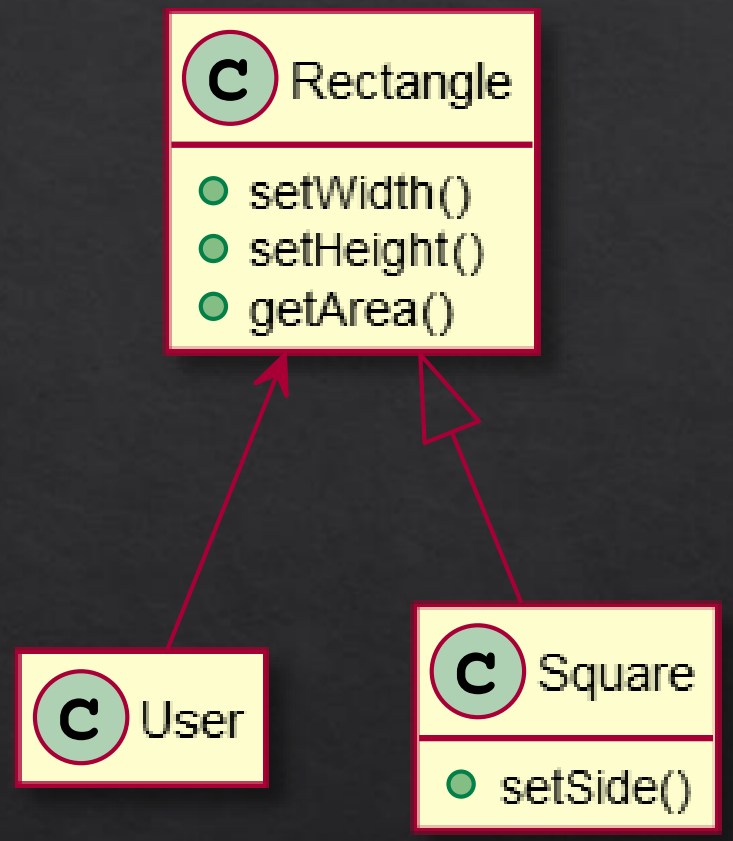
"Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T."

- In other words, subclasses should be substitutable for their base classes.
- Technically, it's called behavioral subtyping

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Bad Example: LSP Violation (1)

- In this example, Square is not a proper subtype of Rectangle because the height and width of the Rectangle are independently mutable.
- In contrast, the height and width of the Square must change together.
- Since the User believes it is communicating with a Rectangle, it could easily get confused.



Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Bad Example: LSP Violation (2)

The following code shows why:

```
# User's expectation

Rectangle reg = …
reg.setWidth(4);
reg.setHeight(3);
assert reg.getArea() == 12;
```

```
# Overridden methods in class Square

public void setWidth(int v) {
    this.width = v;
    this.height = v;
}
public void setHeight(int v) {
    this.width = v;
    this.height = v;
}
```

If the … code produced a Square, then the assertion would fail.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

The Interface Segregation Principle

The Interface Segregation Principle (ISP)

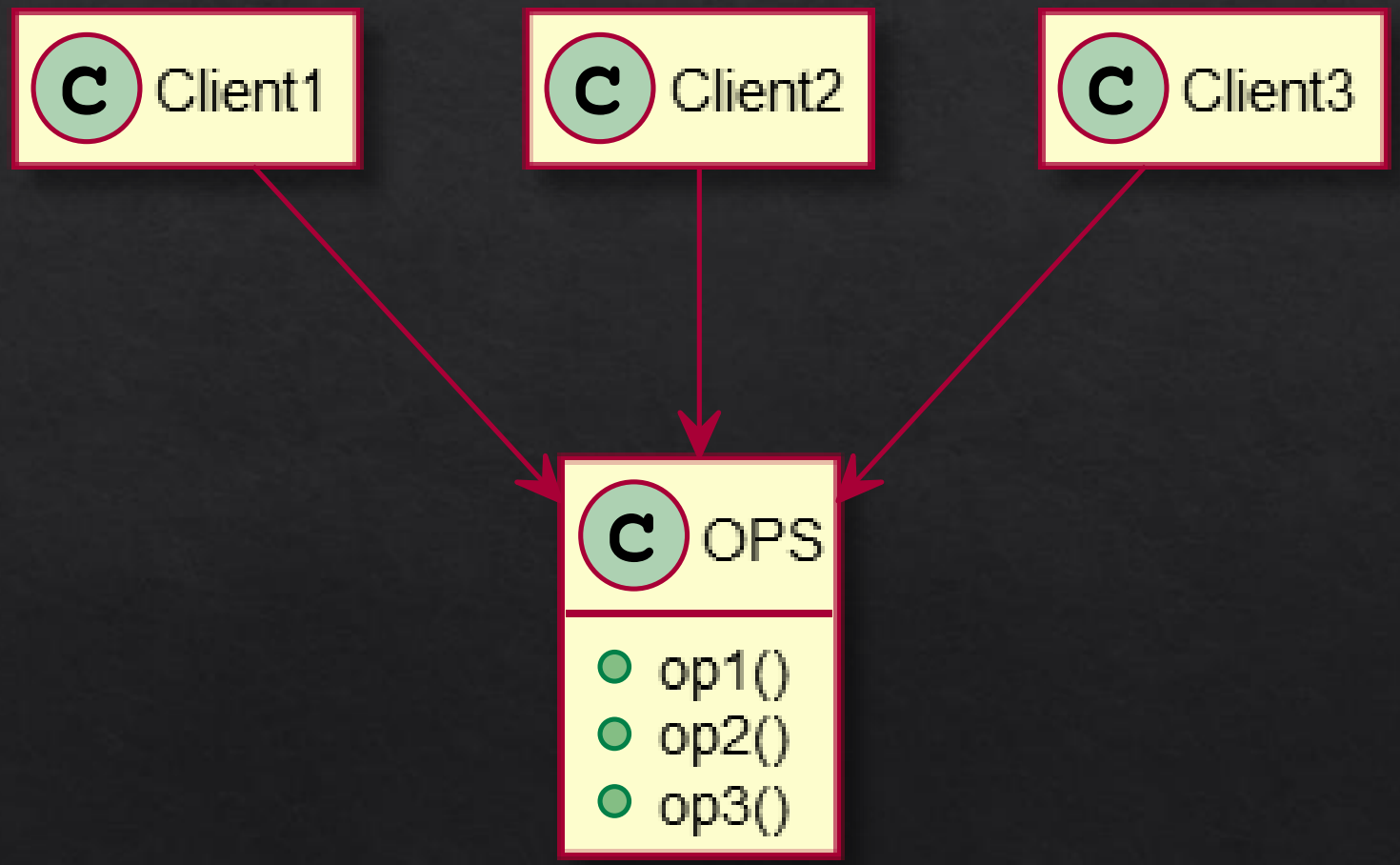
- The ISP is first used and formulated by Robert C. Martin. It can be described as follows:

  "Many client specific interfaces are better than one general purpose interface."

- If you have a class that has several clients, create specific interfaces for each client and multiply inherit them into the class.
- The ISP does not recommend that every class that uses a service have its own special interface class that the service must inherit from.
- Rather, clients should be categorized by their type, and interfaces for each type of client should be created.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Bad Example: ISP Violation (1)



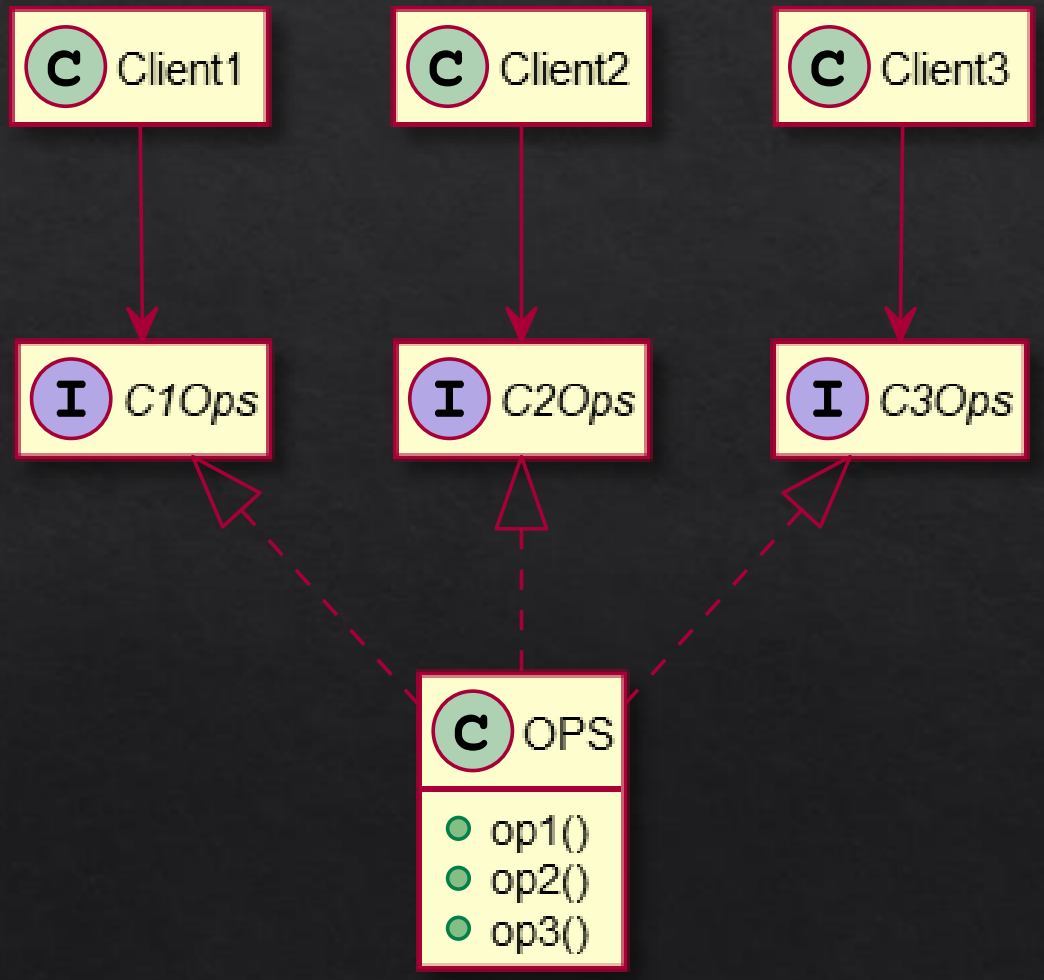
Martin, R. C., Grenning, J., & Brown, S. (2018). *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall.

Bad Example: ISP Violation (2)

- Now imagine that OPS is a class written in a language like Java. Clearly, in that case, the source code of Client1 will inadvertently depend on op2 and op3, even though it doesn't call them.
- This dependence means that a change to the source code of op2 in OPS will force Client1 to be recompiled and redeployed, even though nothing that it cared about has actually changed.
- This problem can be resolved by segregating the operations into interfaces.

Martin, R. C., Grenning, J., & Brown, S. (2018). *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall.

Good Example: Fixing ISP Violation



Martin, R. C., Grenning, J., & Brown, S. (2018). *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall.

The Dependency Inversion Principle

The Dependency Inversion Principle (DIP)

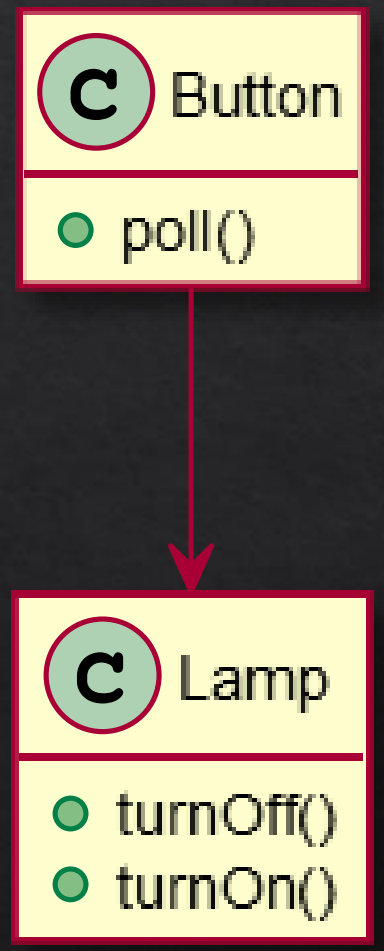
- It is the high-level modules that contain the important policy decisions and business models of an application.
- However, traditional procedural methods tend to create software structures in which high-level modules depend on low-level modules.
- In his 2003 book, Robert C. Martin described the DIP as follows:

A. "High-level modules should not depend on low-level modules. Both should depend on abstractions."

B. "Abstractions should not depend upon details. Details should depend upon abstractions."

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Bad Example: DIP Violation (1)




```
public class Button {
    private Lamp lamp;
    public void poll() {
        if (/*some condition*/) {
            lamp.turnOn();
        }
    }
}
```

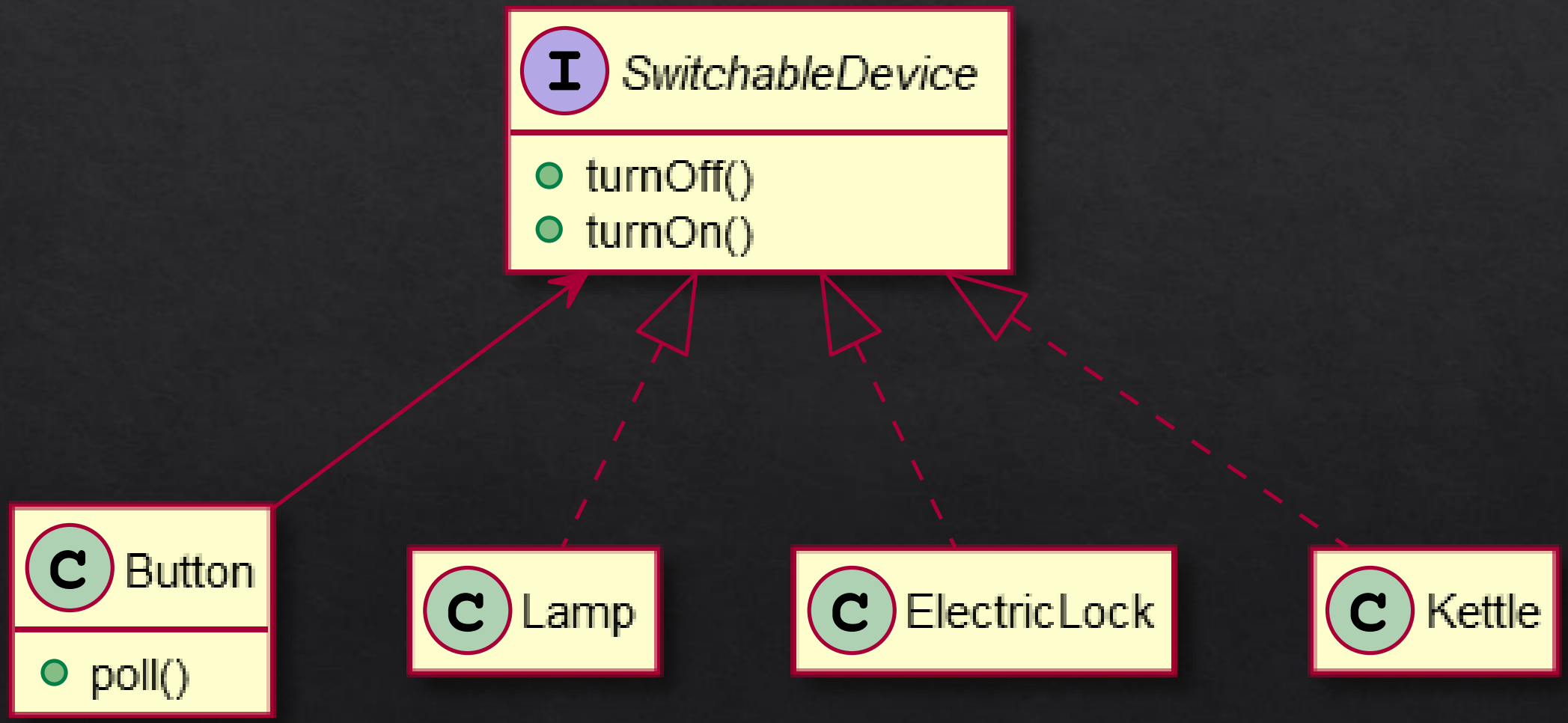
Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Bad Example: DIP Violation (2)

- This solution violates DIP.
- The high-level policy of the application has not been separated from the low-level implementation.
- The abstractions have not been separated from the details.
- Without such a separation, the high-level policy automatically depends on the low-level modules, and the abstractions automatically depend on the details.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Good Example: Fixing DIP Violation



Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Dependence on Abstractions

- A naive interpretation of DIP is the simple heuristic: "Depend on abstractions."
  - No variable should hold a reference to a concrete class.
  - No class should derive from a concrete class.
  - No method should override an implemented method of any of its base classes.
- Certainly, this heuristic is usually violated at least once in every program. Somebody has to create the instances of the concrete classes, and such a module will depend on them.
- If a concrete class is not going to change very much, and no other similar derivatives are going to be created, it does very little harm to depend on it.

Martin, R. C., Martin, M., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Prentice Hall.

Recommended Readings

SOLID examples in Java (**STRONGLY RECOMMENDED**):

- Introduction to SOLID Design Principles for Java Developers - DZone Java

SOLID examples in C#:

- SOLID Principles in C# with Examples - Dot Net Tutorials

SOLID examples in PHP:

- The SOLID Principles - Envato Tuts+ Code Tutorials (tutsplus.com)

SOLID real-world examples in Java:

- SOLID Principles with Examples – HowToDoInJava