COM S 362
Object-Oriented Analysis & Design

SOLID
Interface Segregation and
Dependency Inversion

Reading

Robert C. Martin. Clean Architecture, 2018.

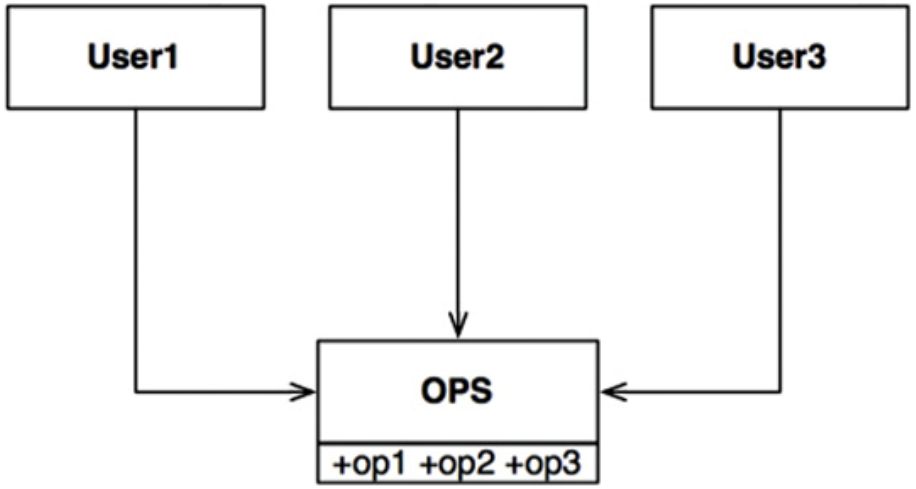
- Ch. 10 and 11, pp. 83-91

SOLID

- SOLID
  - **S**ingle Responsibility Principle
  - **O**pen-Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle
- Design principles assembled by Robert Martin (“Uncle Bob”), although some originally developed by others
- Goal: make mid-level (module level) software structures that
  - Tolerate change
  - Are easy to understand
  - Are reusable in many software systems

Interface Segregation: Motivation

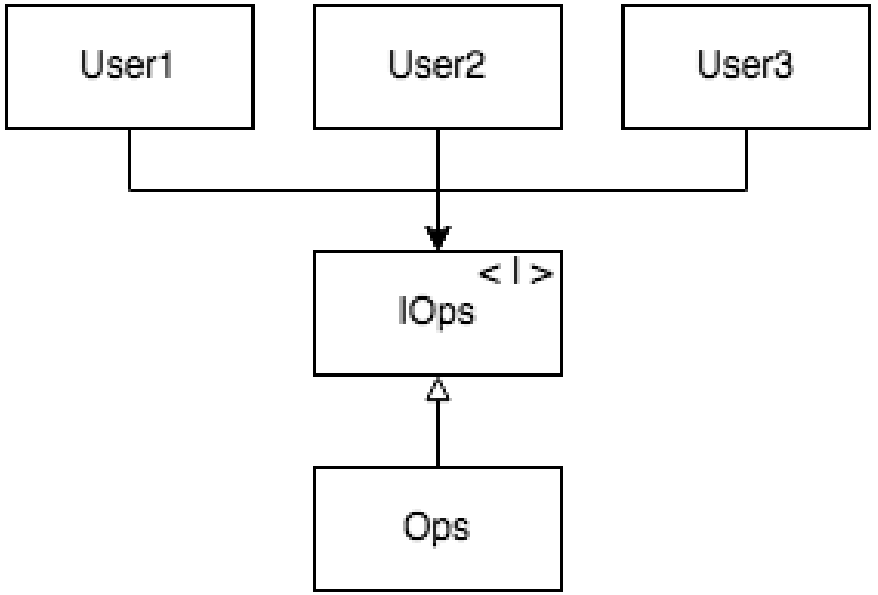
- Suppose there are several users of a class and the users call non-overlapping subsets of the class's operations



- What is the consequence of a change to OPS?
  - User1 has a dependency on op2 and op3
  - A change to OPS requires a recompile of all users

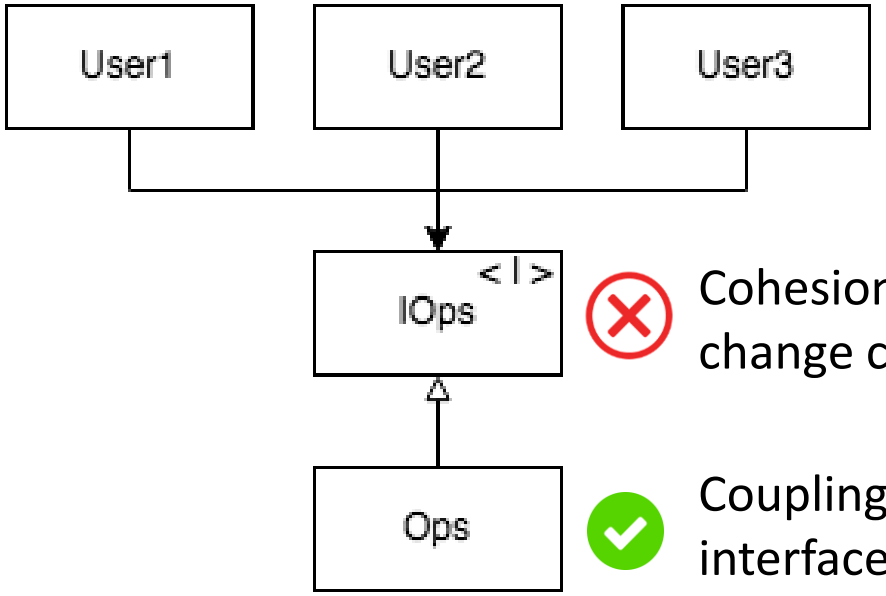
Reduce Dependency with Interface

- How can we reduce the impact of change on OPS?
- A first step would be to make the users dependent only on an interface and not the concrete class
- Does this solve the dependency problem?



Reduce Dependency with Interface

- How can we reduce the impact of change on OPS?
- A first step would be to make the users dependent only on an interface and not the concrete class
- Does this solve the dependency problem?

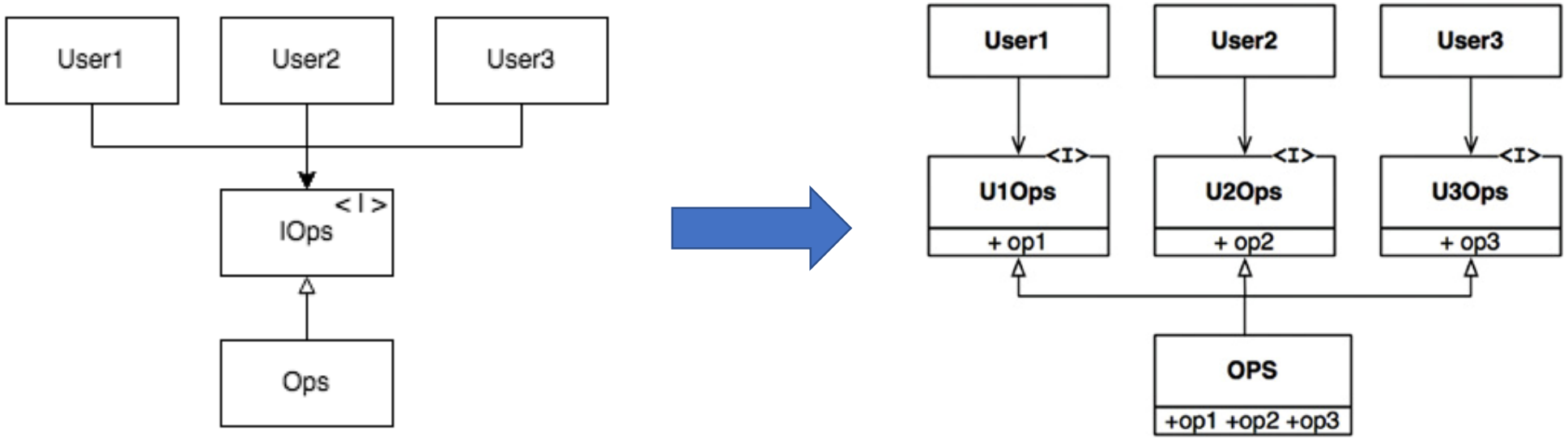


Cohesion in interface is low, interface change causes recompile of all users.

Coupling to OPS is now low due to added interface.

The Interface Segregation Principle

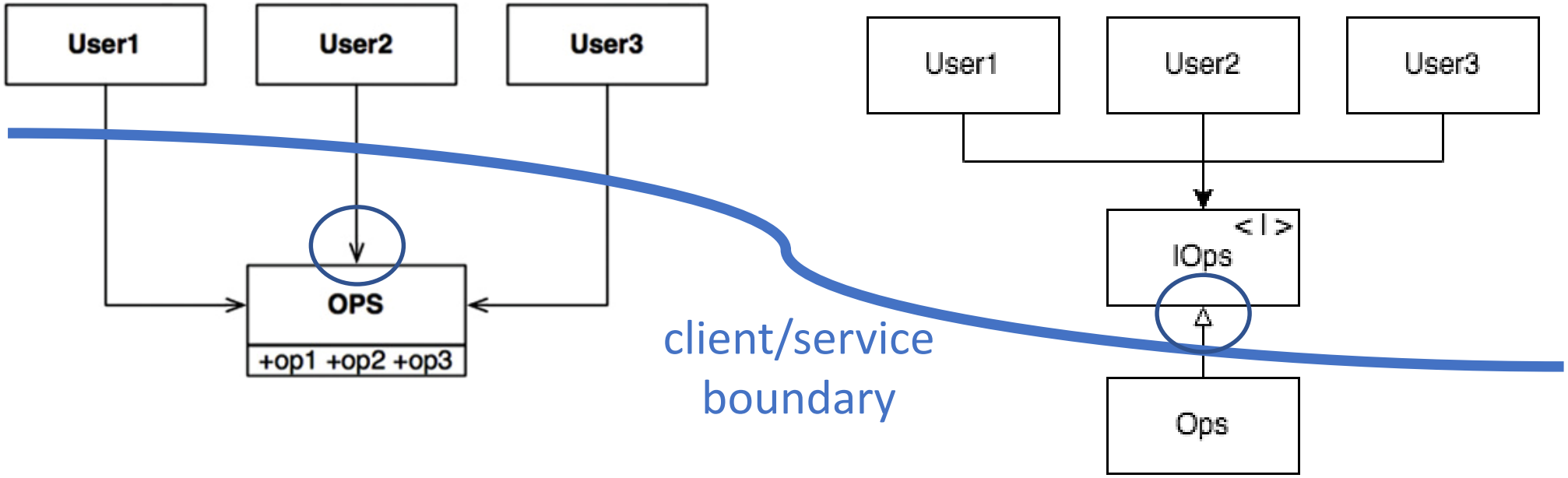
- Problem: Big interfaces are not cohesive
- Solution: Create a separate interface for each of the non-overlapping subset of interface uses



- Interface segregation principle: “Clients should not be forced to depend on interfaces that they do not use.”

Dependency Inversion Example

- In previous example we did something important



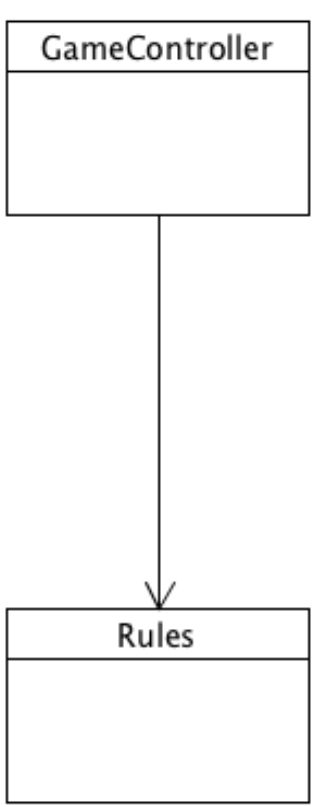
- Example 1 (left): users (clients) depends on service
- Example 2 (right): users have no dependency on service, only on an interface
- Across the client/service boundary line, we have *inverted the dependency* from pointing down to up

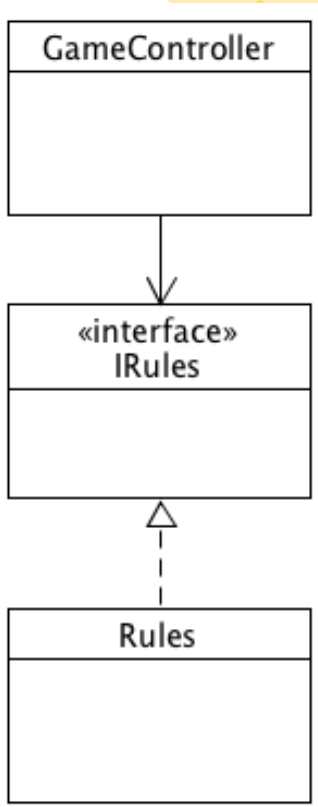
The Dependency Inversion Principle

- Observation: abstractions (interfaces) tend to be stable, concrete classes tend to be volatile
- We want to depend on things that are stable, not volatile
- **Dependency Inversion Principle:** the most flexible systems are those in which dependencies refer only to abstractions (e.g., interfaces and abstract classes), not to concretions (concrete classes).

Dependency Inversion Refactoring Steps

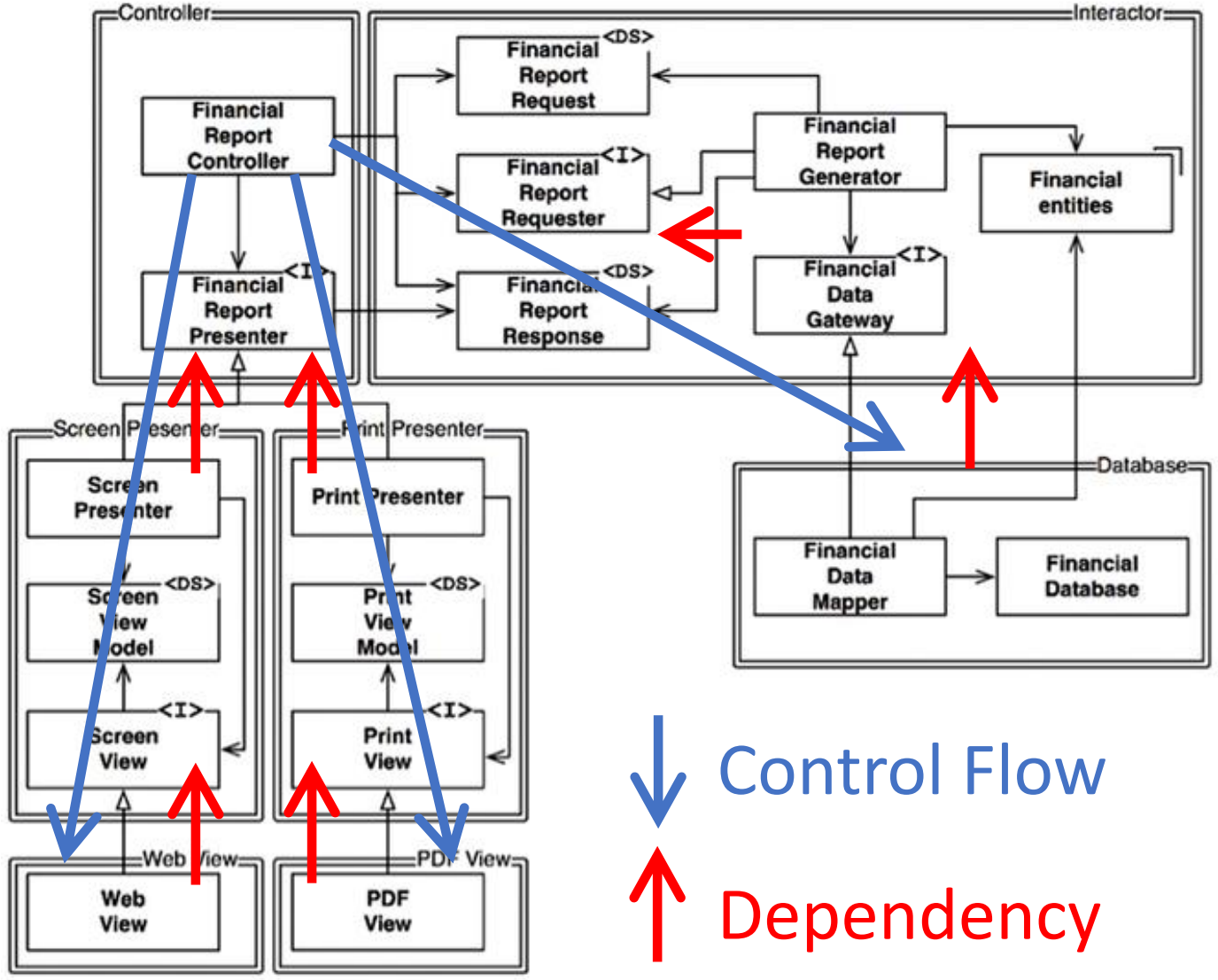
1. Create an interface
2. Make the service class implement the interface
3. Change all service class references in the client class to reference the interface





Dependency Inversion In Practice

- It is not realistic to completely remove every class decency
- Where should or focus be?
- Identify components of the system and hierarchy of abstractions/the flow of control
- Dependencies across component boundaries should point up (to higher abstractions/against the flow of control)



Relation to Open-Closed Principle

- When we study patterns, we will see many examples of dependency inversion, it is most visible organizing principle in architecture
- The use of interfaces to invert dependencies plays a major role in achieving an open-close design
- Each interface becomes an *extension point,* onto which we can extend the software without requiring modification of higher-level software components