CSCB07 - Software Design

SOLID Design

Fall 2023

Non-functional Requirements

- Quality attributes that a software system must meet in addition to its functional requirements
- Examples include
  - Modifiability (the ease with which changes can be made)
  - Reusability
  - Testability
  - Portability
- Conforming to SOLID design principles helps improve the overall quality of a software system

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

What is SOLID?

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

Single Responsibility Principle (SRP)

A class should have only one reason to change

- If you can think of more than one motive for changing a class, then that class has more than one responsibility
- If a class has more than one responsibility, then the responsibilities become coupled

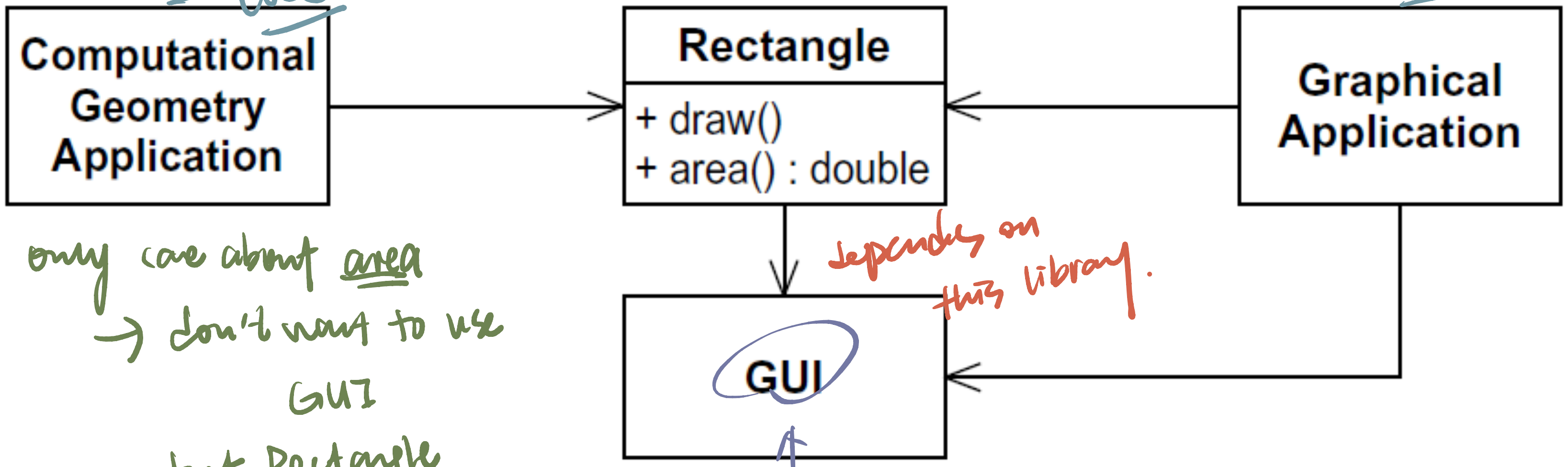
Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

Single Responsibility Principle (SRP)

Violating the SRP (example)

our code → client code.

two responsibility



client code

client code

only care about area → don't want to use GUI but Rectangle depend on GUI

depends on this library.

library of graphics.

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

Single Responsibility Principle (SRP)

Conforming to the SRP (example)

Main job of the class

depend

library

import

class rectangle {

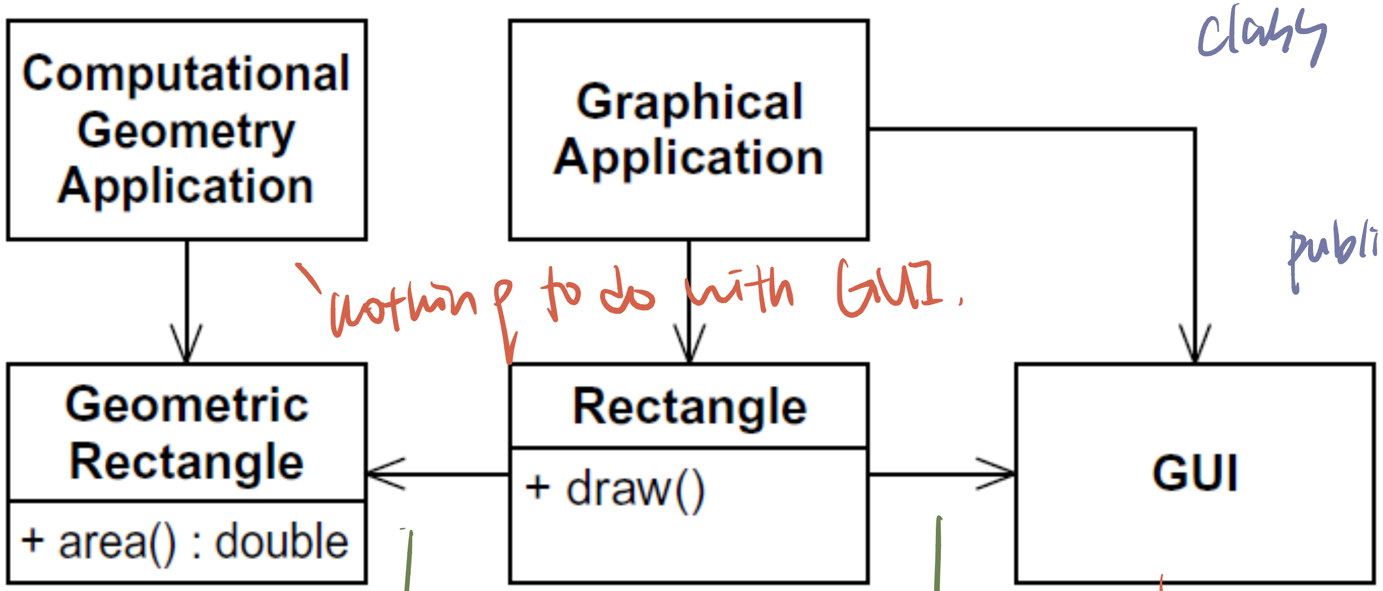
=

public void draw( ){

GeometricRect r=_;

double d=r.area();}

}



Computational Geometry Application

Graphical Application

Geometric Rectangle

+ area() : double

Rectangle

+ draw()

GUI

nothing to do with GUI.

depends

depends.

un

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

depend on expectation whether we want to change the feature

The Open/Closed Principle (OCP)

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

add features

change

without changing existing code

- When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity.
  - ➢ If the Open/Closed principle is applied well, then further changes of that kind are achieved by adding new code, not by changing old code that already works.
- In Java, it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors
  - ➢ The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes.

abstract / interface.

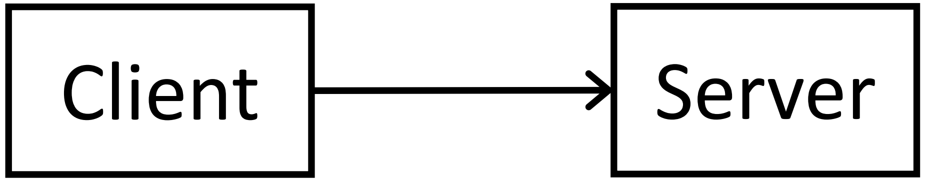
* extension → inheritance

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

The Open/Closed Principle (OCP)

Violating the OCP (example)

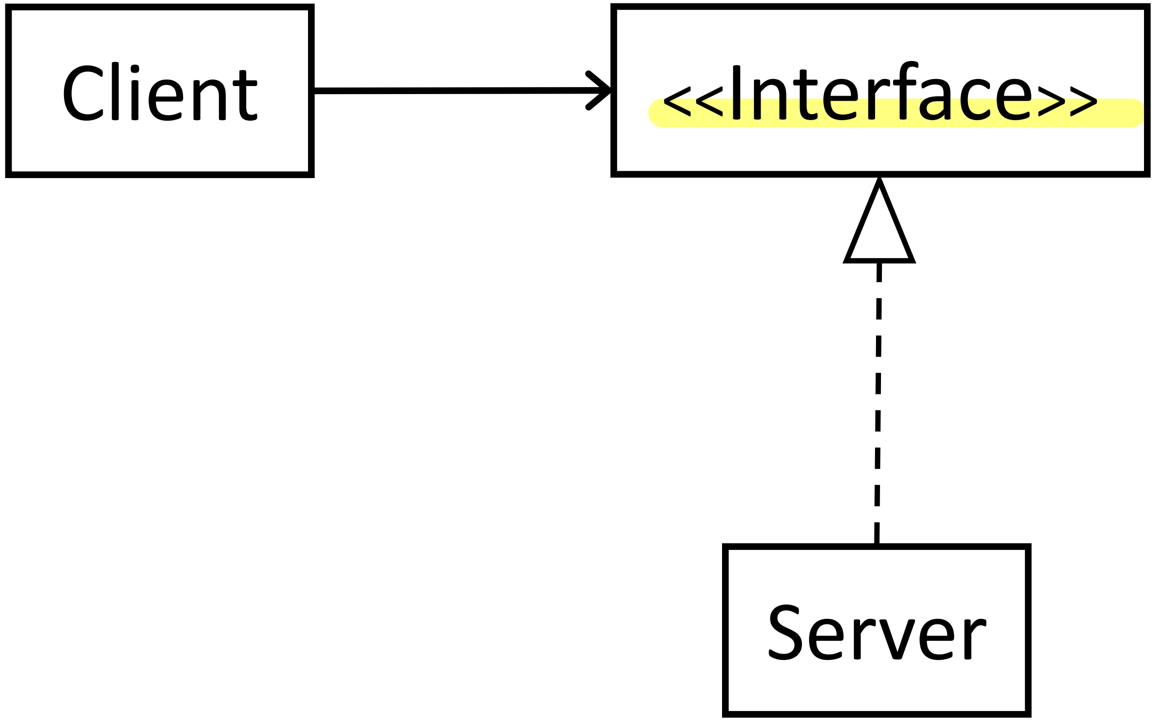
- Both classes are concrete
- The **Client** uses the **Server** class



Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

The Open/Closed Principle (OCP)

Conforming to the OCP (example)



Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

inheritance

The Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

- Formally: *Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.*
- Counter-example: *“If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction”*

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

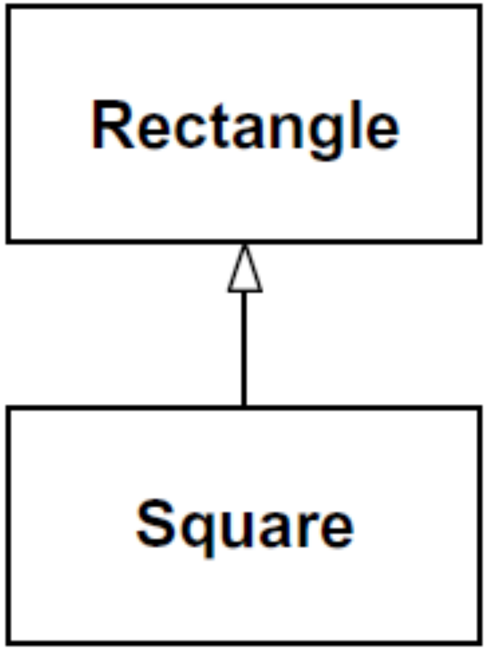
The Liskov Substitution Principle (LSP)

Violating the LSP (example)

Issues

- Inheriting **height** and **width**
- Overriding **setHeight** and **setWidth**
- Conflicting assumptions. For example:

```
void testRectangleArea(Rectangle r){
        r.setWidth(5);
        r.setHeight(4);
        assertEquals(r.computeArea(), 20);
}
```



Rectangle

Square

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

The Liskov Substitution Principle (LSP)

- Implication: A model, viewed in isolation, cannot be meaningfully validated.
  - ➢ The validity of a model can only be expressed in terms of its clients.
  - ➢ One must view the design in terms of the reasonable assumptions made by the users of that design.

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

The Interface Segregation Principle (ISP)

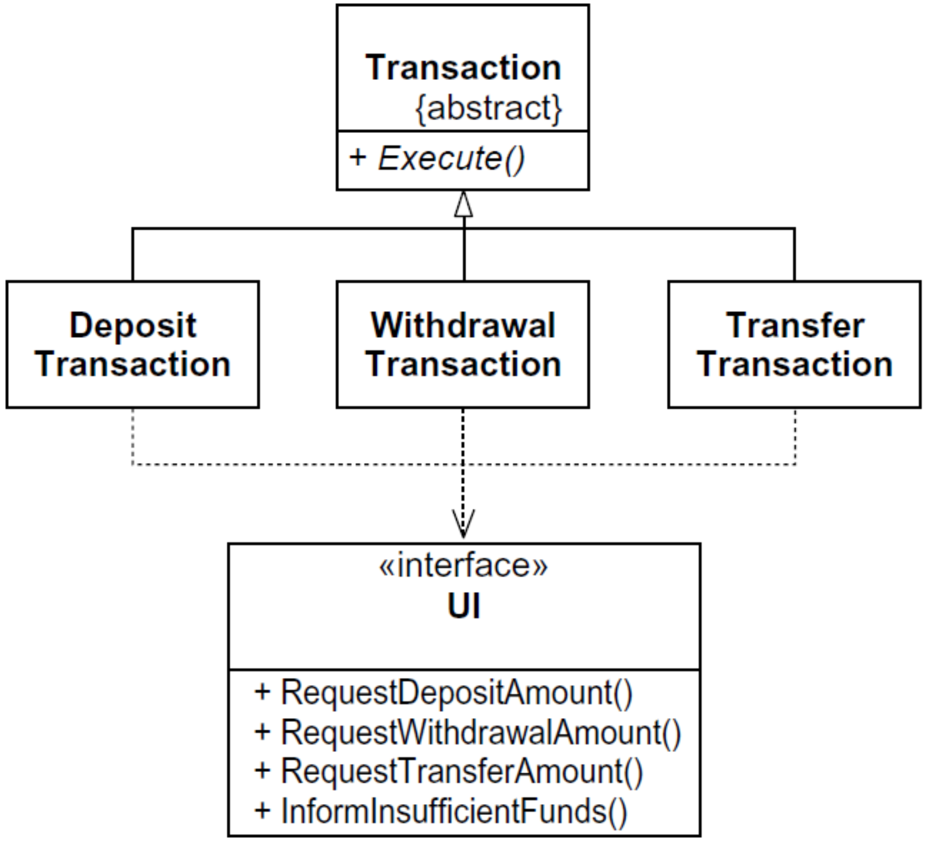
Clients should not be forced to depend on methods that they do not use.

- This principle deals with classes whose interfaces are not cohesive. That is, the interfaces of the class can be broken up into groups of methods where each group serves a different set of clients.
- When clients are forced to depend on methods that they don't use, then those clients are subject to changes to those methods.

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

The Interface Segregation Principle (ISP)

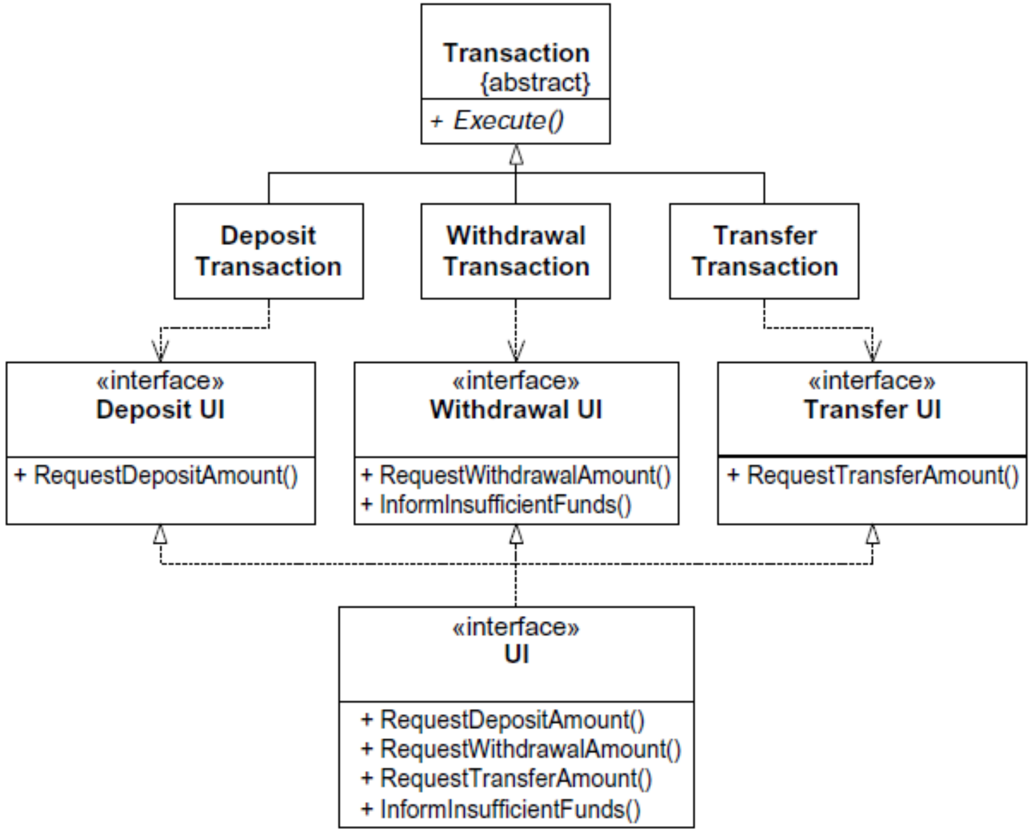
Violating the ISP (example)



Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

The Interface Segregation Principle (ISP)

Conforming to the ISP (example)



Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

The Dependency-Inversion Principle (DIP)

A. High-level modules should not depend on low-level modules. Both should depend on abstractions. → should not depend on concrete class

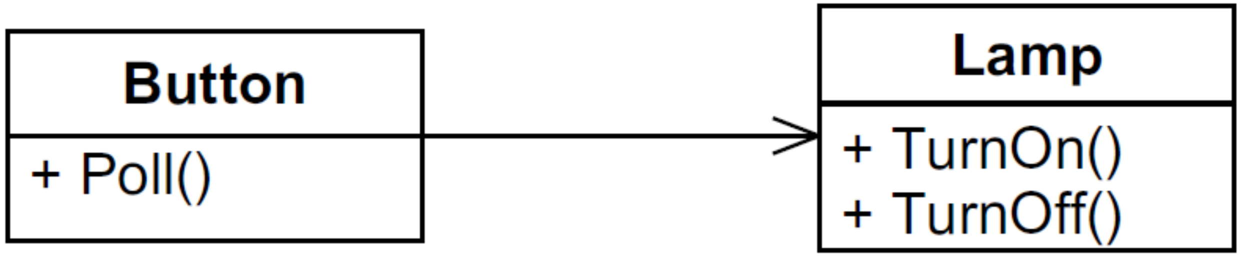
B. Abstractions should not depend on details. Details should depend on abstractions.

- The modules that contain the high-level business rules should take precedence over, and be independent of, the modules that contain the implementation details.
- When high-level modules depend on low-level modules, it becomes very difficult to reuse those high-level modules in different contexts.

Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

The Dependency-Inversion Principle (DIP)

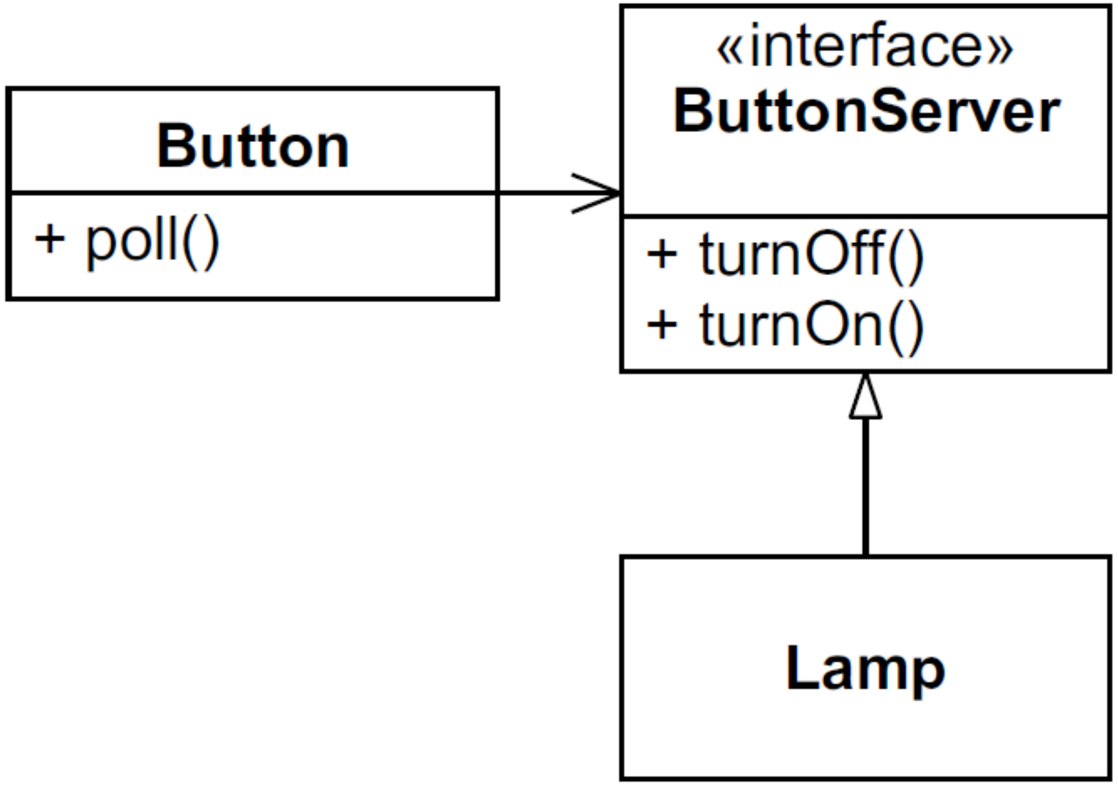
Violating the DIP (example)



Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.

The Dependency-Inversion Principle (DIP)

Conforming to the DIP (example)



Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, First Edition, © 2003 Pearson Education, Inc.