Software Architecture

Lecture 10
The SOLID Software Design Principles

Néstor Cataño
Innopolis University

Spring 2016

Outline

Outline

Software Design Principles

Design

- are guidelines that help people to avoid bad design practices of **OO** programs.
- are due to Robert C. Martin (**Book**: **Agile Principles, Patterns and Practices in C♯**)

The SOLID Design Principles

SOLID

Single Responsibility Principle

Open-Closed Principle

LSP (Liskov Substitution Principle)

Interface Segregation Principle

Dependency Integration principle

Outline

The Single Responsibility Principle

SRP

A class should have only one reason to change

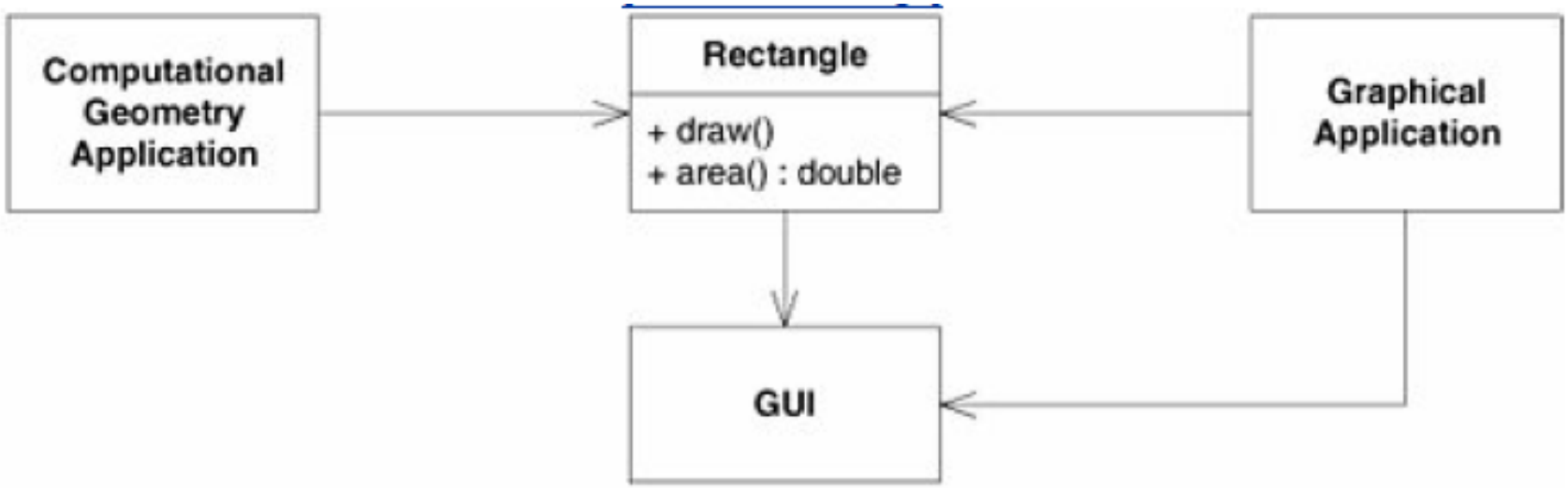
The Single Responsibility Principle

Why is it important to separate two responsibilities into separate classes?

- each class is an axis of change
- if requirements change, the change will effect a chain of classes
- if a class has more than a responsibility, then it will have more than one reason to change

The Single Responsibility Principle

More Than One Responsibility



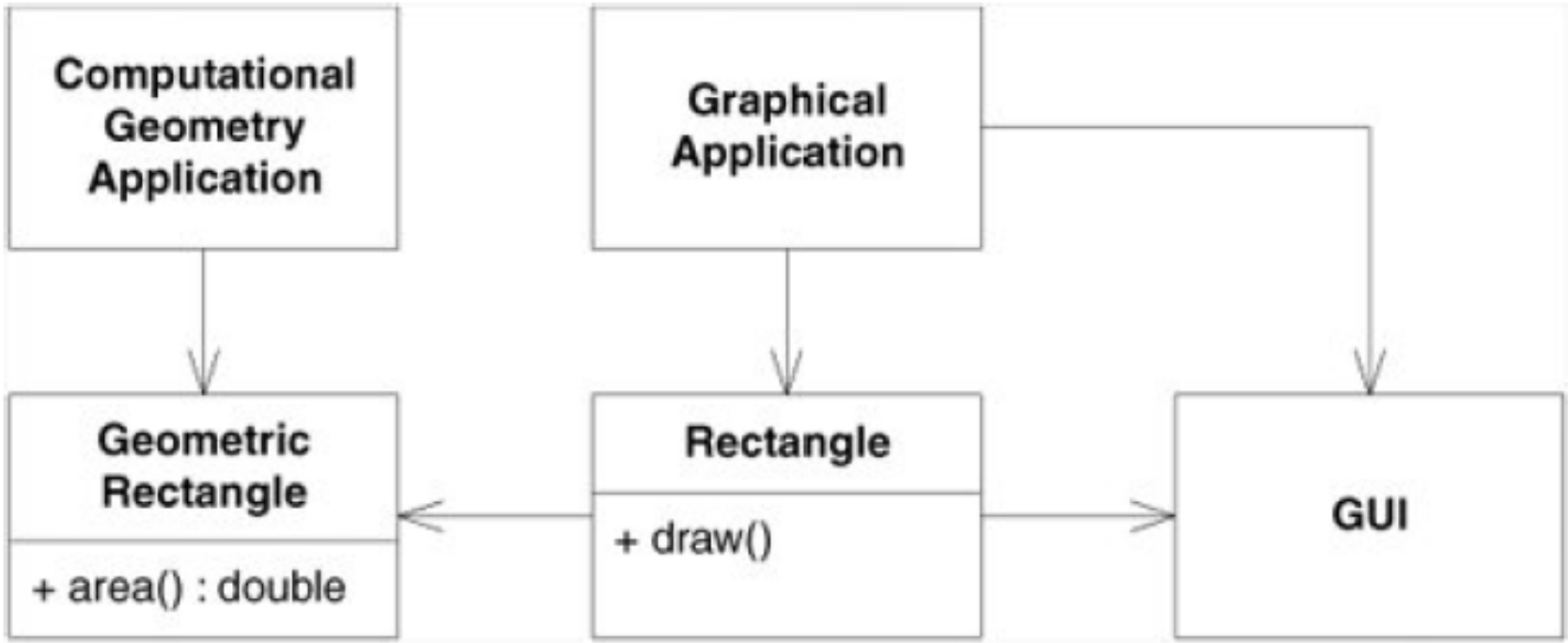
The Single Responsibility Principle

SRP's Violation

- Class `Rectangle` has two responsibilities
  - to provide a mathematical model of the geometry of the rectangle
  - to render the rectangle to the **GUI**.
- **Problem 1:** A change in the `GraphicalApplication` that may cause the `Rectangle` to change may force to rebuild the `ComputationalGeometryApplication`.
- **Problem 2:** The **GUI** must be included in the computational geometry application.

The Single Responsibility Principle

Separated Responsibilities



Example 2: Modem Interface

SRP Violation

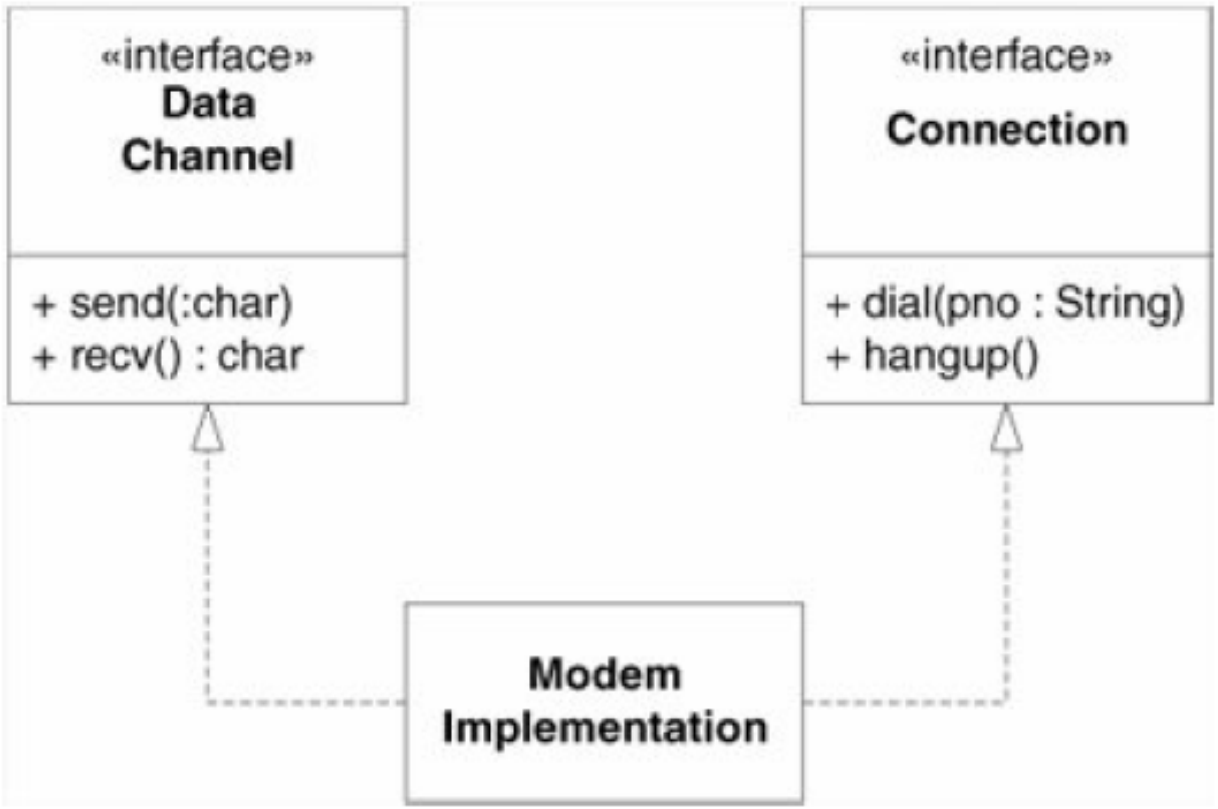
```
public interface Modem
{
  public void Dial(string pno);
  public void Hangup();
  public void Send(char c);
  public char Recv();
}
```

Two Responsabilities:

1. Connection management
2. Data Communication

Example 2: Modem Interface

Separated Modem Responsibilities



Example 3: Employee

Coupled Persistence

Two Mixed-Up Responsibilities

1. Business rules
2. Persistence

Outline

The Open/Closed Principle

OCP

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

Bertrand Meyer's Original Definition

Open (for Extension)

- A module is said to be **open** if it's still available for extension. For example, it should be possible to expand its set of operations or add field to its data structures.

Closed (for Modification)

- A module is said to be **closed** if it's available for use by other modules. You can compile it, store it in a library, and make it available to clients.

OCP

OCP

How can a module be modified without changing its source code?

OCP

OCP

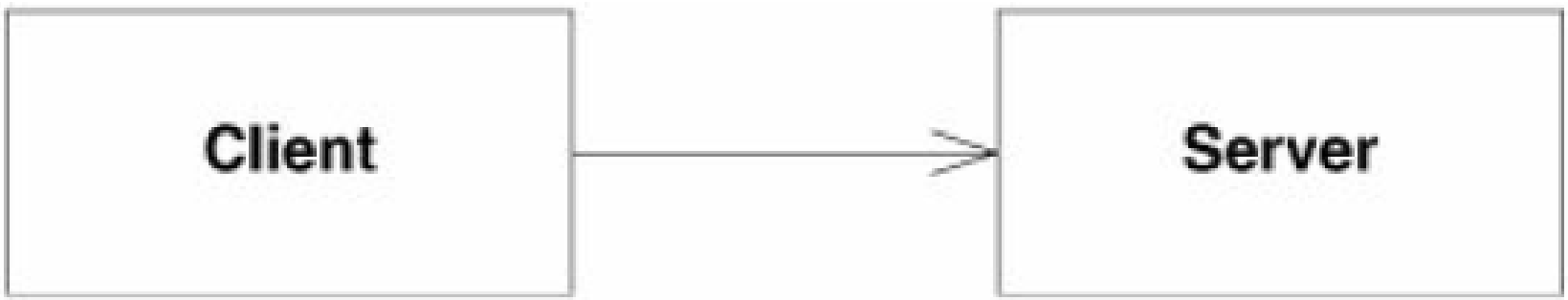
How can a module be modified without changing its source code?

Meyer's Answer: **Inheritance**

Today's Answer: **Abstraction**

The Open/Closed Principle

`Client` is not Open (for Extension) and Closed (for Modification)



The Open/Closed Principle

`Client` is not **Open**

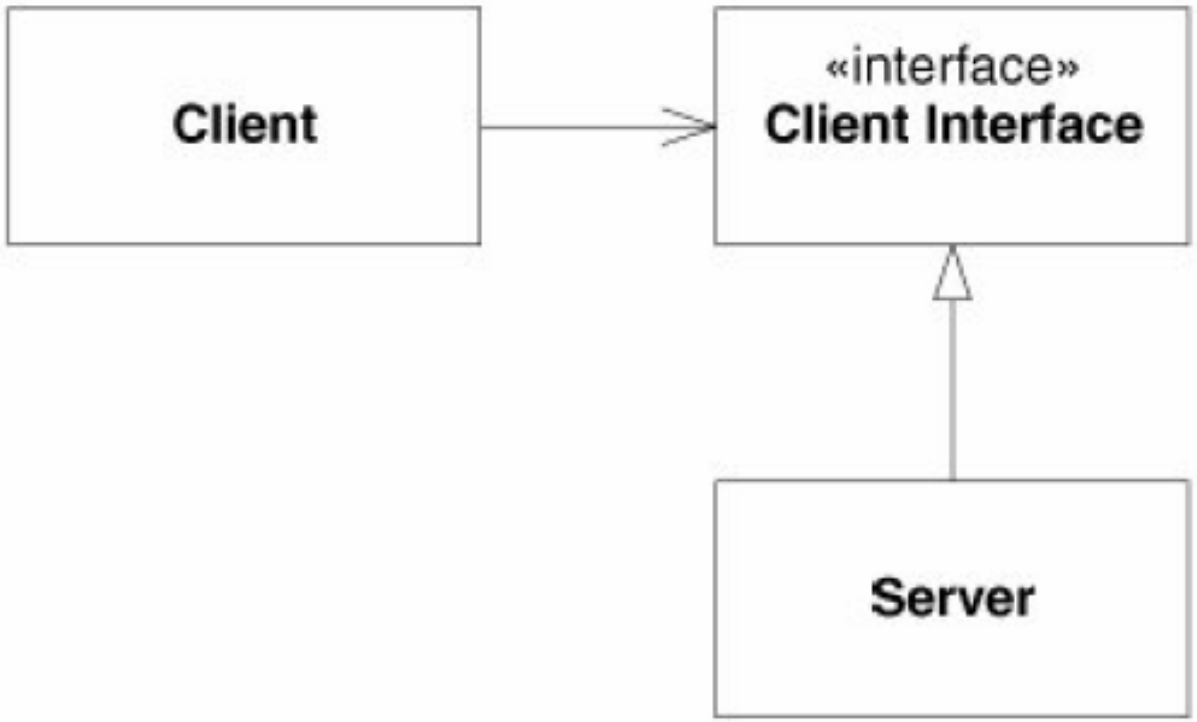
- if we want for a `Client` object to use a different `Server` object, then the `Client` class must be changed to refer to the new `Server` class

`Client` is not **Closed**

- if `Server` changes then `Client` must be recompiled!

The Open/Closed Principle

`STRATEGY` Pattern: `Client` is both Open and Closed



The Open/Closed Principle

`STRATEGY` Pattern: `Client` is both Open and Closed

- if we want `Client` objects to use a different server class, a new derivative of the `ClientInterface` class can be created. The `Client` class can remain unchanged.

The Open/Closed Principle

`STRATEGY` Pattern: `Client` is both Open and Closed

- if we want `Client` objects to use a different server class, a new derivative of the `ClientInterface` class can be created. The `Client` class can remain unchanged.
- the behavior specified in **Client** can be **extended** and **modified** by creating new subtypes of `ClientInterface`.

The STRATEGY Pattern

ClientInterface

```
public interface ClientInterface {
  public abstract void someMethod();
}
```

Server

```
public class Server implements ClientInterface {
  public void someMethod() {
    System.out.println("client method");
  }
}
```

The STRATEGY Pattern

Client

```
public class Client {
  private ClientInterface c;
  public void clientMethod() {
    c.someMethod();
  }
}
```

OCP: Example on Shapes

class ShapeType

```
public enum ShapeType {
  square, circle;
}
```

class Shape

```
public class Shape {
  ShapeType type;
}
```

OCP: Example on Shapes

class Square

```
public abstract class Square {
  int type;
  double side;
  Point topLeft;

  public static void Draw(Shape sp) { ... }
}
```

OCP: Example on Shapes

class Circle

```
public abstract class Circle {
  int type;
  double radius;
  Point center;

  public static void Draw(Shape sp) { ... }
}
```

Violating OCP: Shapes

Does `DrawAllShapes` Conform to **OCP**?

```
public static void DrawAllShapes(Shape[] shapes) {
 for(Shape sp:  shapes) {
  switch(sp.type) {
   case square :
    Square.Draw(sp);
   case circle :
    Circle.Draw(sp);
  }
 }
}
```

The Open/Closed Principle

Violating OCP

- `DrawAllShapes` is not **closed** against adding a new `Shape`

The Open/Closed Principle

Violating OCP

- `DrawAllShapes` is not **closed** against adding a new `Shape`
  - to extend this function to be able to draw a list of shapes that includes a new `Triangle`, one would need to modify `DrawAllShapes`

The Open/Closed Principle

Violating OCP

- `DrawAllShapes` is not **closed** against adding a new `Shape`
  - to extend this function to be able to draw a list of shapes that includes a new `Triangle`, one would need to modify `DrawAllShapes`
  - and one would need to modify `ShapeType`, and hence all the classes need to be re-compiled

Conforming to the Open/Closed Principle

class Shape

```
public abstract class Shape {
  public abstract void Draw();
}
```

Conforming to the Open/Closed Principle

class Square

```
public abstract class Square extends Shape {
  double side;
  Point topLeft;

  public void Draw() {
    // draw square
  }
}
```

Conforming to the Open/Closed Principle

class Circle

```
public abstract class Circle extends Shape {
  double radius;
  Point center;

  public void Draw() {
    // draw circle
  }
}
```

Conforming to the Open/Closed Principle

DrawAllShapes

```
public static void drawAllShapes(Shape[] shapes) {
  for(Shape sp: shapes) {
    sp.Draw();
  }
}
```

Outline

The Liskov Substitution Principle

LSP: Motivation

We want to define a proper notion of **sub-typing** whereby each time we take a program **P** and replace every object of type **T** by a subtype object of type **S**, then the program behaves the same: it enjoyes the same properties as the original program had, e.g. correctness.

The Liskov Substitution Principle

LSP

Any type **T** must be **substitutable by** any of its sub-types **S**

The Liskov Substitution Principle

Barbara Liskov

- If for each object **o2** of type **S** there is an object **o1** of type **T** such that for all programs **P** defined in terms of **T**, the behavior of **P** is **unchanged** when **o1** is substituted by **o2**, then **S** is a subtype of **T**

Violation of LSP Causing Violation of OCP

```
public enum ShapeType {
  square, circle;
}
public class Shape {
  ShapeType type;

  Shape(ShapeType t) { type = t; }

  static void DrawShape(Shape sp) {
    if(sp.type == ShapeType.square)
      ((Square) sp).Draw();
    else if (sp.type == ShapeType.circle)
      ((Circle) sp).Draw();
  }
}
```

Violation of LSP Causing Violation of OCP

```
public abstract class Square extends Shape {
  double side;
  Point topLeft;

  public Square(ShapeType t) { super(t); }

  public void Draw() {
    // draws the square
  }
}
```

Violation of LSP Causing Violation of OCP

```
public abstract class Circle extends Shape {
  double radius;
  Point center;

  public Circle(ShapeType t) { super(t); }

  public void Draw() {
    // draws the circle
  }
}
```

The Liskov Substitution Principle

Violation of LSP

- Classes `Circle` and `Square` do not override `DrawShape` so objects of type `Shape` cannot be **substituted by** objects of class `Circle` or `Square`.

The Liskov Substitution Principle

Violation of LSP

- Classes `Circle` and `Square` do not override `DrawShape` so objects of type `Shape` cannot be **substituted by** objects of class `Circle` or `Square`.

Violation of OCP

- `DrawShape` violates **OCP** (**why?**)

LSP

Consequences

1. **weaker preconditions** in a sub-type
2. **stronger postconditions** in a sub-type
3. **invariants** of the supertype must be preserved in a sub-type
4. **Contravariance** of method arguments in the sub-type
5. **Covariance** of return types in the sub-type
6. **No new exceptions** are thrown by methods of the sub-type, except when those exception are sub-types of the exception thrown by the methods of the super-type.

LSP

Stronger Vs. Weaker

- `(x > 5 && y > 7)` is **stronger** than `(x > 5)`
- `(x > 5 || y > 7)` is **weaker** than `(x > 5)`
- `(x > 5)` is **stronger** than `true`
- `P` is **stronger** and **weaker** than `P`
- `false` is **stronger** than any predicate `P`

Weaker Precondition Violation

```
public class Rectangle {
   protected int x, y;

   //@ requires a >= 0;
   //@ ensures x == a;
   public void setX(int a) {
      x = a;
   }

   //@ requires a >= 0;
   //@ ensures y == a;
   public void setY(int a) {
      y = a;
   }
}
```

Weaker Precondition Violation

```
public class Square extends Rectangle {
  //@ inv: x == y;

  //@ requires a >= 0 && a == y;
  //@ ensures x == a;
  public void setX(int a) {
    x = a;
  }
}
```

Weaker Precondition Violation

```
(a >= 0)   ⇏   (a >= 0 && a == y)
```

A Weird Way to Solve this Problem

```
public class Square extends Rectangle {
  //@ inv: x == y;

  //@ requires a >= 0
  //@ ensures x == a && y == a;
  public void setX(int a) {
    x = a;
    y = a;
  }
}
```

A Weird Way to Solve this Problem

Weaker Precondition

`(a >= 0)` $\Rightarrow$ `(a >= 0)`

Stronger Postcondition

`(x == a && y == a)` $\Rightarrow$ `(x == a)`

Covariant and Contravariant

Covariant

if `Collection<T>` is **Covariant**, then `Collection<Cat>` is **sub-type** of `Collection<Animal>`

Contravariant

if `Collection<T>` is **Contravariant**, then `Collection<Animal>` is **sub-type** of `Collection<Cat>`

LSP requires Contravariance of method arguments
LSP requires Covariance of return types

O.K: Contravariance on Parameters

```
public class Shape {
    public Shape set(Rectangle a) {
      // something
    }
}
public class Rectangle extends Shape {
    public Shape set(Shape a) {
      // something
    }
}
```

O.K: Variance on Return Types

```
public class Shape {
    public Shape set(Shape a) {
      // something
    }
}
public class Rectangle {

    public Rectangle set(Shape a) {
      // something
    }
}
```

Outline

The Interface Segregation Principle

SIP

Independent Reading: **Chapter 12** of Book: **Agile Principles, Patterns and Practices in C♯**)

Outline

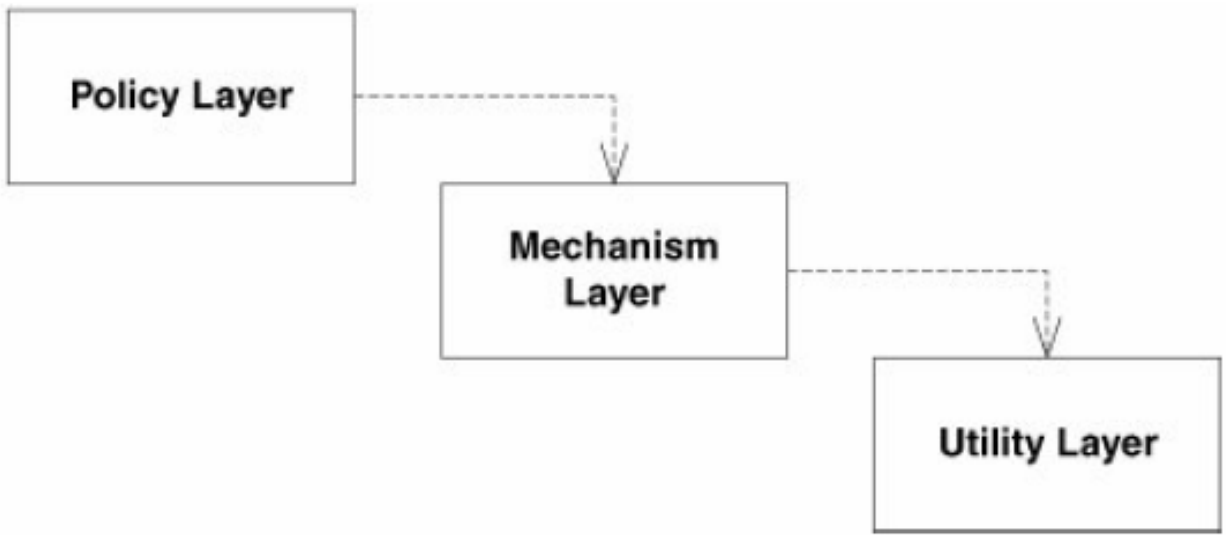
The Dependency Inversion Principle

DIP

- **high-level** modules should not depend on **low-level** modules; both should depend on **abstractions**.
- **abstractions** should not depend upon **details**; **details** should depend upon **abstractions**.

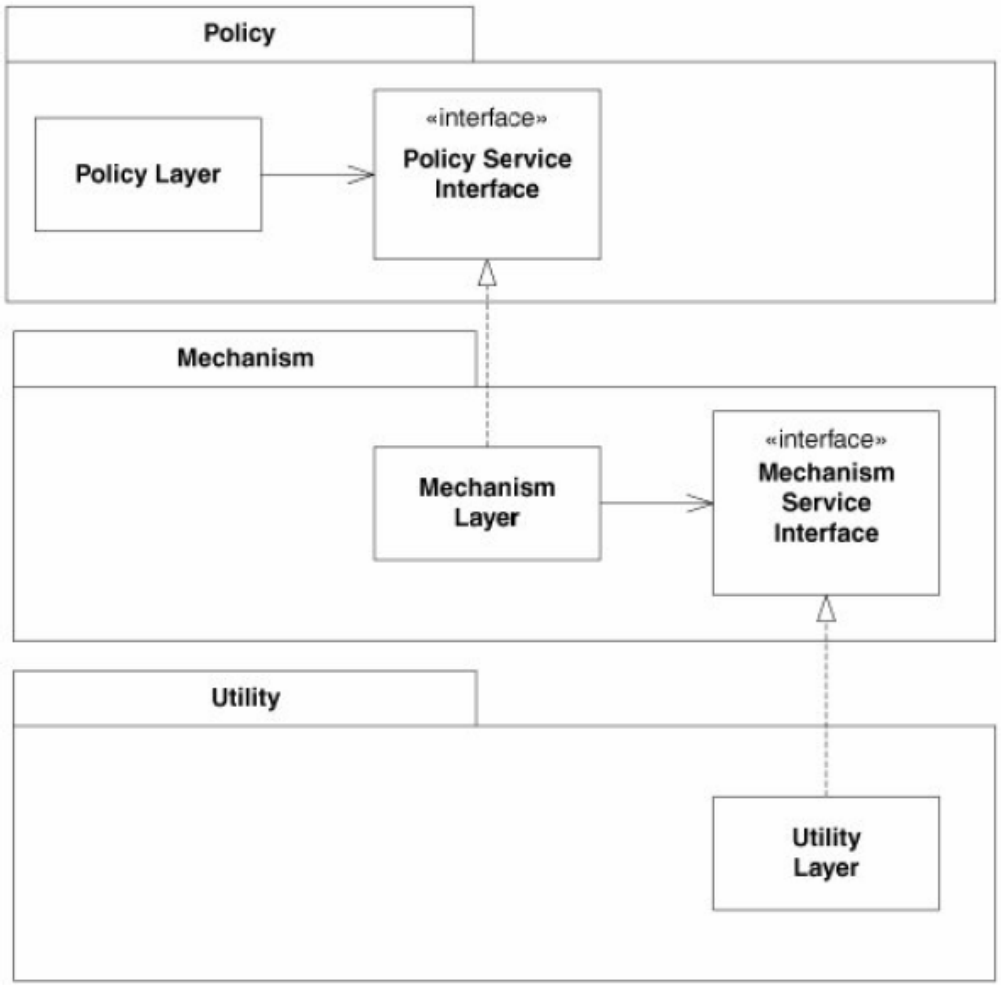
Naive Layering Scheme

Policy depends on Utility



Naive Layering Scheme

Policy depends on Utility



Further Reading

Literature

- Agile Principles, Patterns and Practices in C♯, by Robert C. Martin, Chapters 8 to 12
- Object Oriented Software Construction (Second Edition), by Bertrand Meyer, Section 3.3: **Five Principles**