Open Closed Principle (OCP)

Produced by:

Eamonn de Leastar (edeleastar@wit.ie)

Dr. Siobhán Drohan (sdrohan@wit.ie)

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

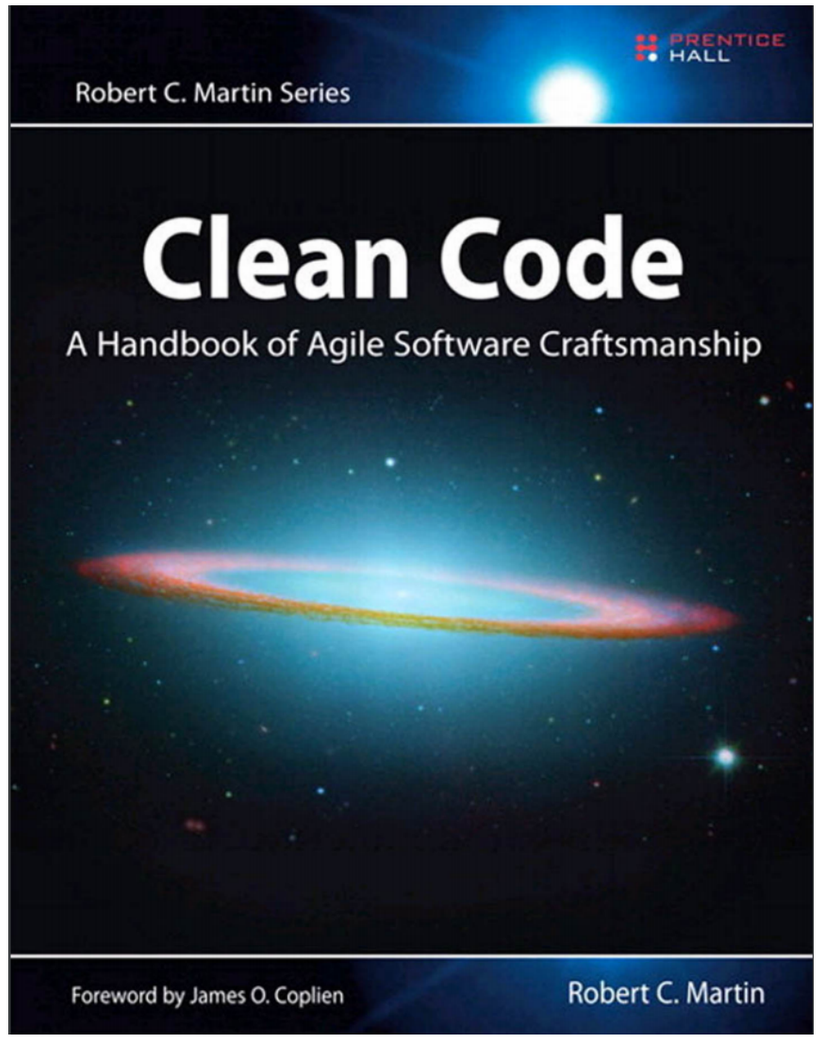
Department of Computing and Mathematics
http://www.wit.ie/

SOLID Class Design Principles

In this talk, we will refer to the SOLID principles examples in this book and also this website.

SOLID → five principles for object-oriented class design i.e.

best guidelines for building a object-oriented maintainable system.



SOLID Class Design Principles

S — Single Responsibility Principle (SRP). Classes should have one, and only one, reason to change. Keep your classes small and single-purposed.

O — Open-Closed Principle (OCP). Design classes to be open for extension but closed for modification; you should be able to extend a class without modifying it. Minimize the need to make changes to existing classes.

L — Liskov Substitution Principle (LSP). Subtypes should be substitutable for their base types. From a client's perspective, override methods shouldn't break functionality.

I — Interface Segregation Principle (ISP). Clients should not be forced to depend on methods they don't use. Split a larger interface into a number of smaller interfaces.

D — Dependency Inversion Principle (DIP). 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.

Open-Closed Principle

Open-chest surgery isn’t needed when putting on a coat.

Change happens!

- *"All systems change during their lifecycles. This must be borne in mind when developing systems expected to last longer than the first version."*

(Jacobson et al., 1992)

“O” in SOLID - Open-Closed Principle

The Open/Closed Principle

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

“O” in SOLID - Open-Closed Principle

The Open/Closed Principle

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

- It is often better to make changes by adding to or building on top of software entities rather than modifying them directly.

OCP - avoidance of rigid code

- When a single change to a program results in a cascade of changes to dependent modules:

  → design smells of rigidity.

- OCP advice:

  → refactor the system so further changes of that kind will not cause more modifications.

- OCP ideal:

  → further changes of that kind are achieved by adding new code, NOT by changing code that already exists.

Open and Closed? Contradiction?

Open and Closed? Contradiction?

- Open For Extension:
  - the behavior of the module can be extended.
  - the module can be made to behave in new and different ways as the requirements of the application change.
- Closed for Modification:
  - extending the behaviour of the module does not result in changes to the source (or binary) code of the module.

Open and Closed? Contradiction?

- The normal way to extend the behavior of a module **is** to make changes to the source code of that module.
- A module that cannot be changed is normally thought to have a fixed behavior.
- How can these two opposing attributes be resolved?

Abstraction

- Create abstractions that:
  - are fixed.
  - May be realised as interfaces or abstract base classes.
  - represent an unbounded group of possible behaviours (i.e. all the possible derived classes).

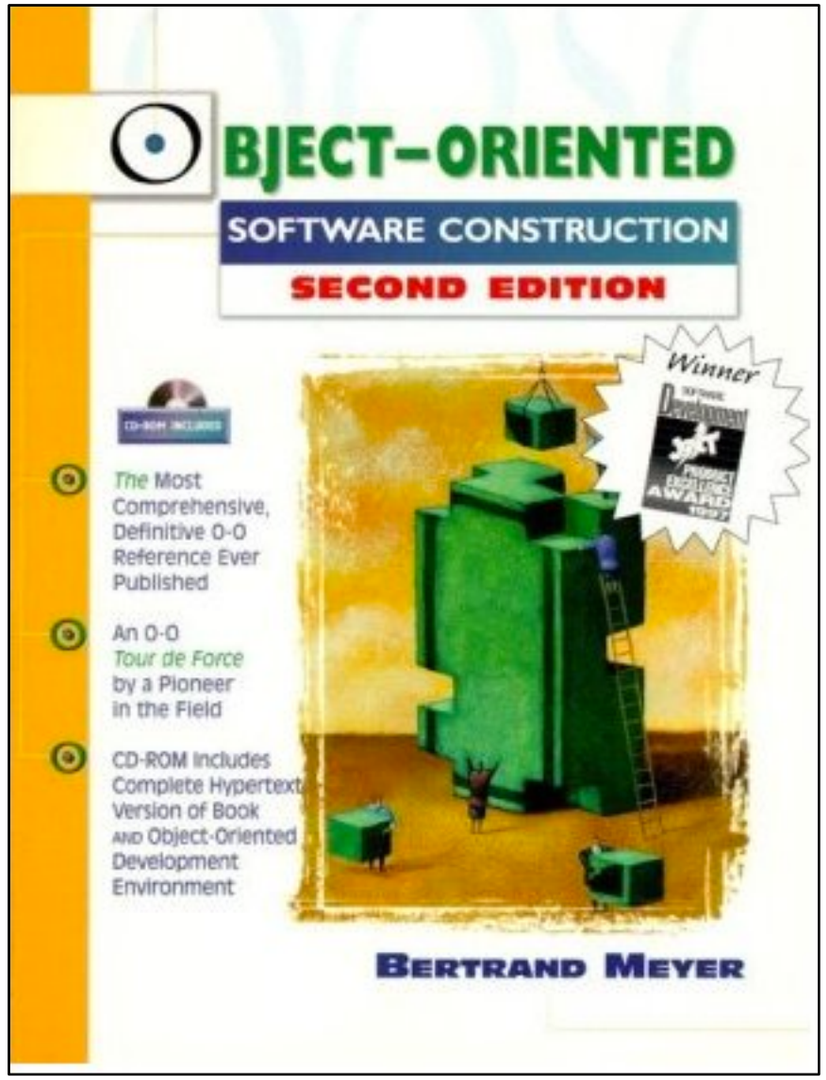
- It is possible for a module to manipulate an abstraction.

- The abstraction is:
  - Closed for modification – it is fixed.
  - Open for extension – can create new derivative classes or implementations of the abstraction.

Open-Closed Principle

- OCP states:
  - Design modules that *never change.*
  - When requirements change:
    - you extend the behavior of such modules by adding new code.
    - not by changing old code that already works.

Source Material - OCP first characterized in OOSC...

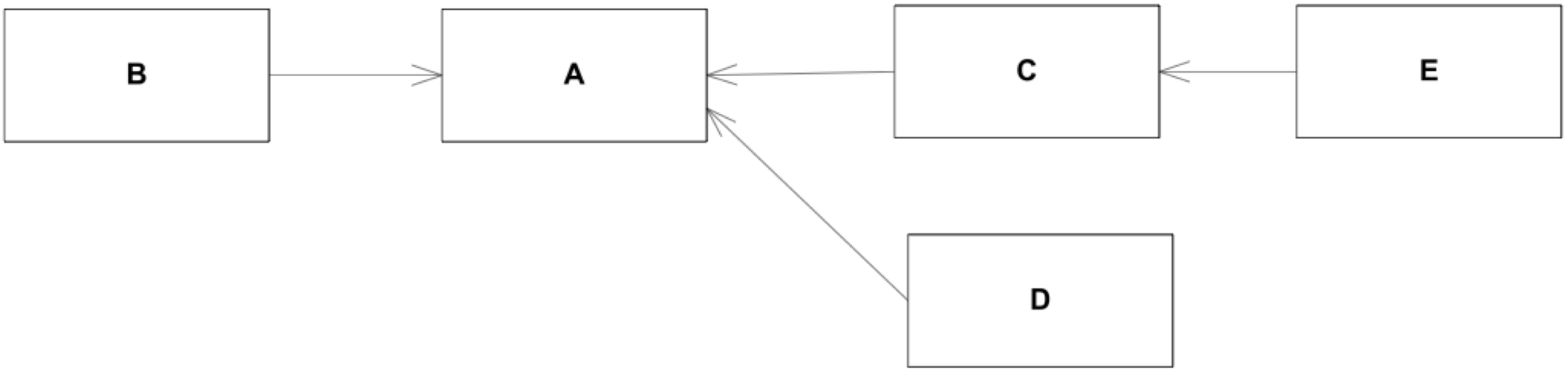


- *"The book, known among its fans as "OOSC", presents object technology as an answer to major issues of software engineering, with a special emphasis on addressing the software quality factors of correctness, robustness, extendibility and reusability. It starts with an examination of the issues of software quality, then introduces abstract data types as the theoretical basis for object technology and proceeds with the main object-oriented techniques: classes, objects, genericity, inheritance, Design by Contract, concurrency, and persistence. It includes extensive discussions of methodological issues."*

http://en.wikipedia.org/wiki/Object-Oriented_Software_Construction
http://en.wikipedia.org/wiki/Open/closed_principle

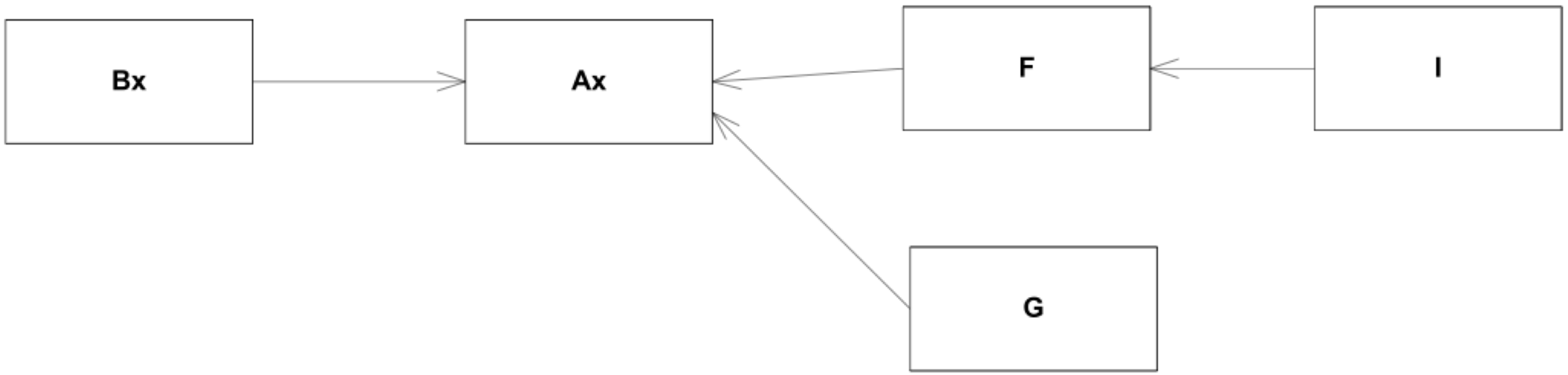
An Abstract Example of OCP

Example (1)



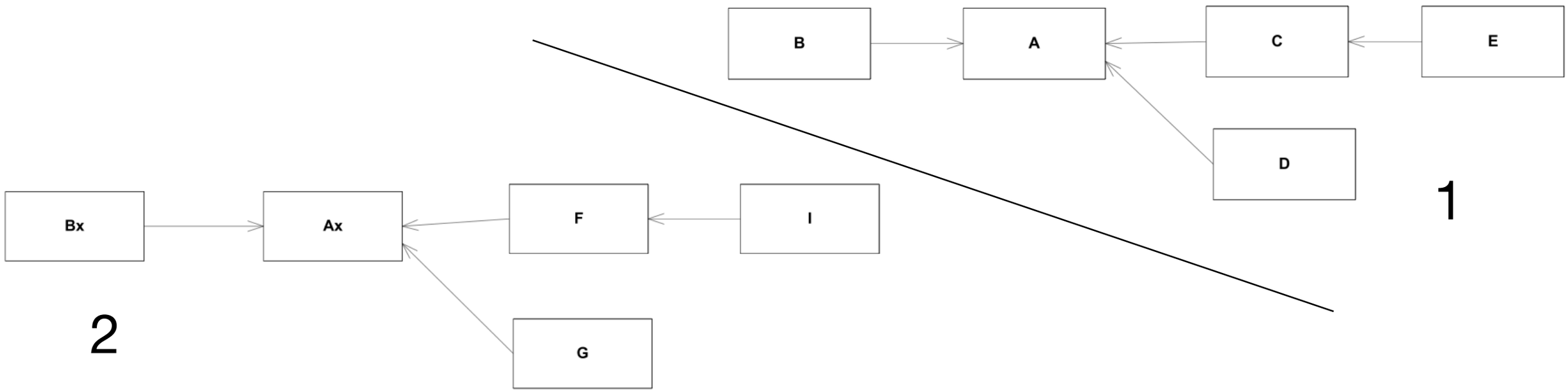
- Module *A* is used by client modules *B*, *C*, *D*, which may themselves have their own clients (*E*).

Example (2)



- A new client – Bx – requires an extended or adapted version of A – called Ax.
- Further clients of Ax are developed (F, G, I)

Example (3)



- Two possible solutions:

1. Adapt module *A* so that it will offer the extended or modified functionality required by the new clients.

2. Leave *A* as it is, make a copy, change the module's name to *Ax*, and perform all the necessary adaptations on the new module. With this technique *Ax* retains no further connection to *A*.

Solution-1 (modify *A's* functionality)

- *A* may have been around for a long time and have many clients such as *B*, *C* and *D*.
- The adaptations needed to satisfy the new clients' requirements may invalidate the assumptions on the basis of which the old ones used *A*.
  - Change to *A* may start a dramatic series of changes in clients, clients of clients and so on.
- Nightmare scenario:
  - Entire parts of the software that were supposed to have been finished and sealed are reopened, triggering a new cycle of development, testing, debugging and documentation.

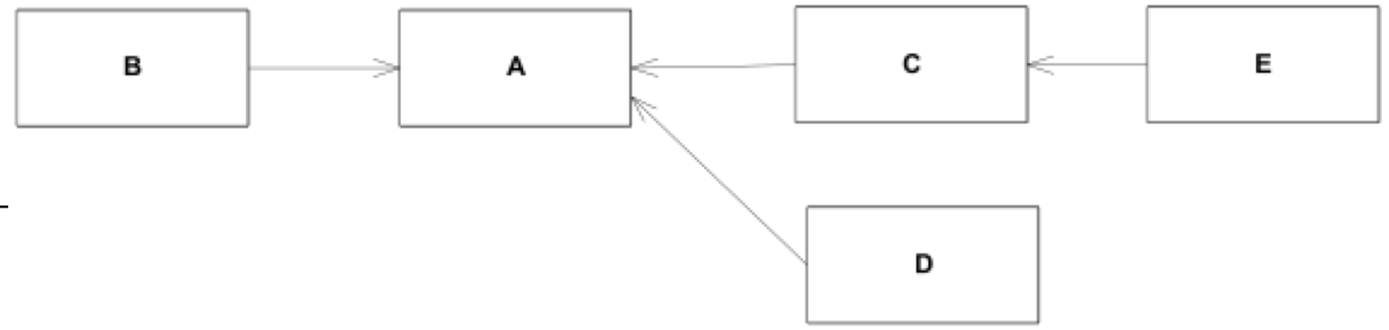
Solution-2 (copy *A*, paste as *Ax*)

- On the surface, solution 2 seems better as it does not require modifying any existing software.

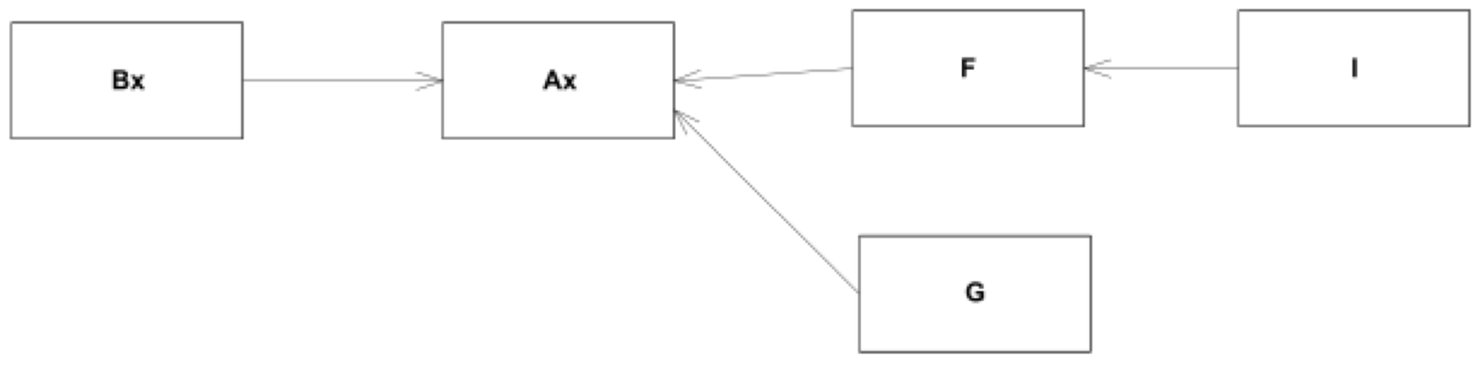
- But this solution may be even more catastrophic since it only postpones the day of reckoning.

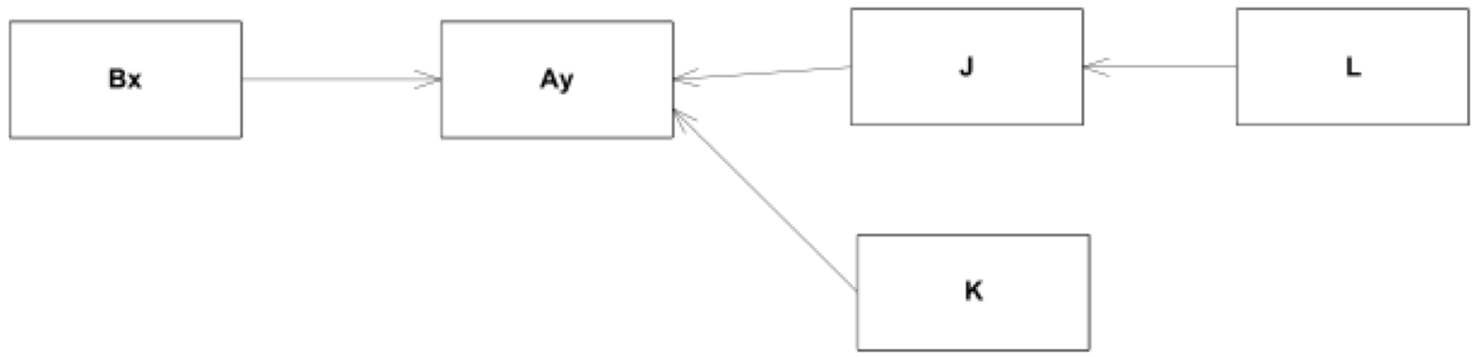
- Leads to an explosion of variants of the original modules, many of them very similar to each other although never quite identical.

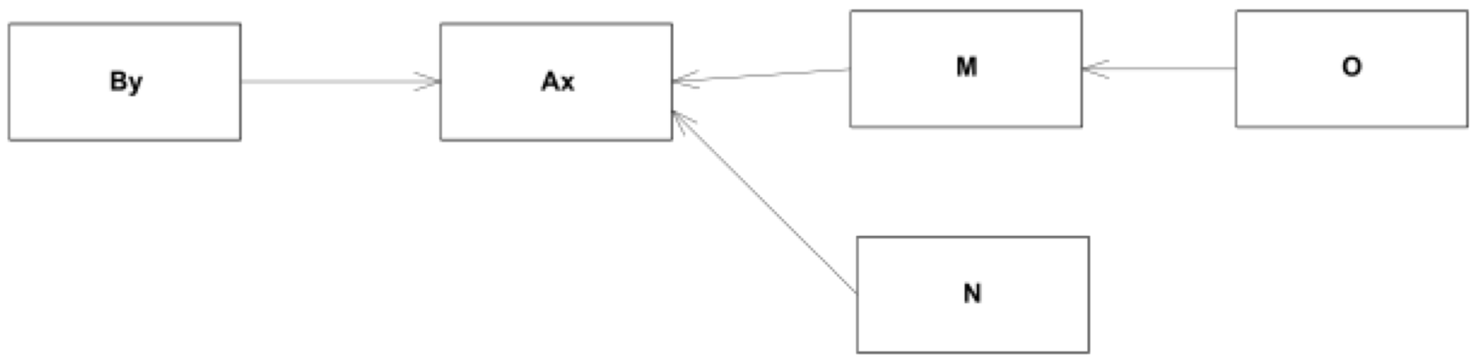
Solution-2



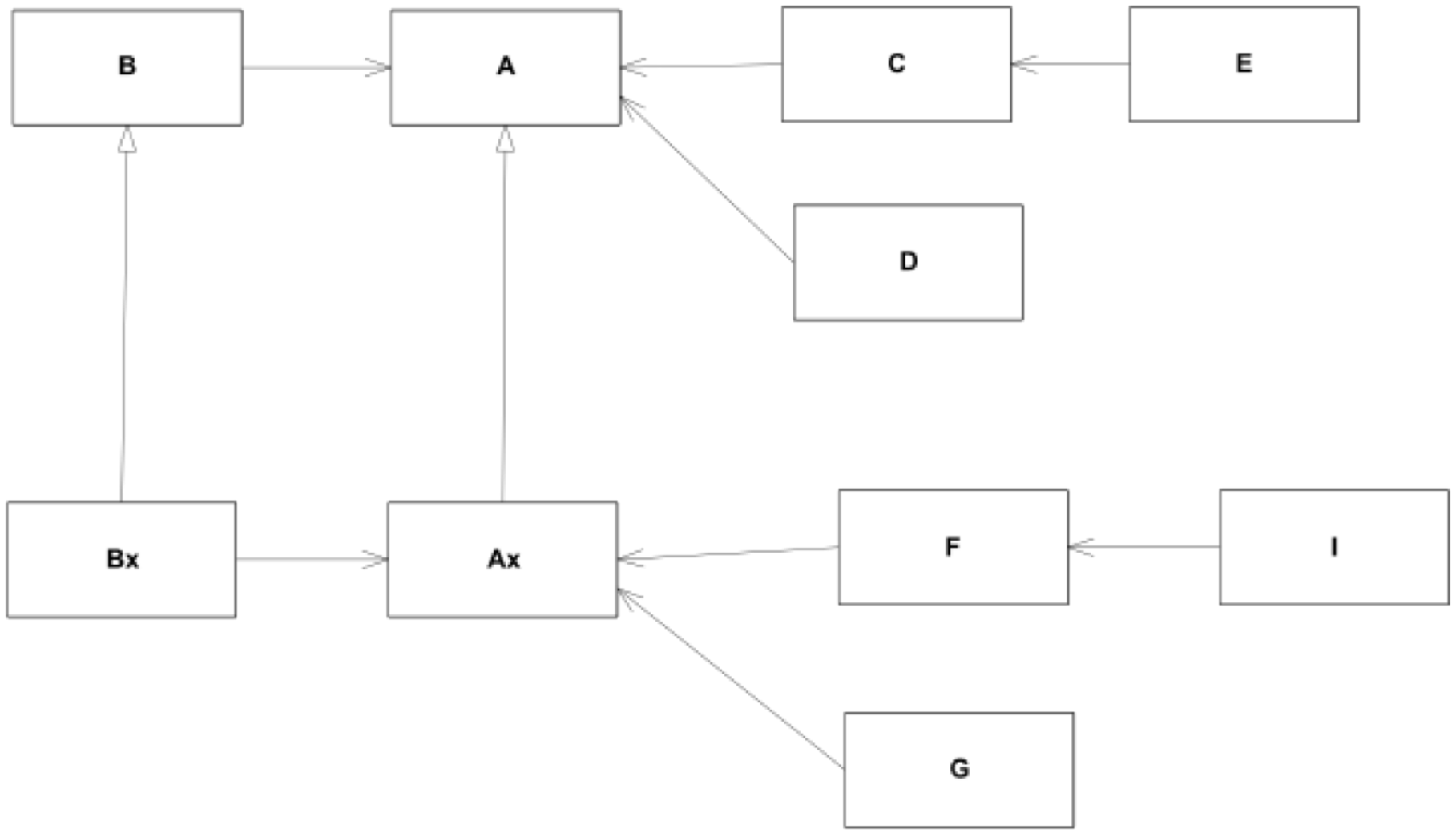
- Abundance of modules, not matched by abundance of available functionality (many of the apparent variants being in fact quasi-clones), creates a huge *configuration management* problem.







OO Solution – Abstraction, Interfaces & Inheritance



- Using interfaces and inheritance developers can adopt a much more incremental approach.

Returning to Abstraction

- In Java it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors:
  - Specified as interfaces and abstract classes.
  - The unbounded group of possible behaviors is represented as derived classes and classes that implement the interfaces.
- A module can implement these abstractions:
  - Can be closed for modification since it fulfills an abstraction that is fixed.
  - Yet the behavior of that module can be extended by creating new implementations of the abstraction.

Client and Server Example

Client is not open and closed

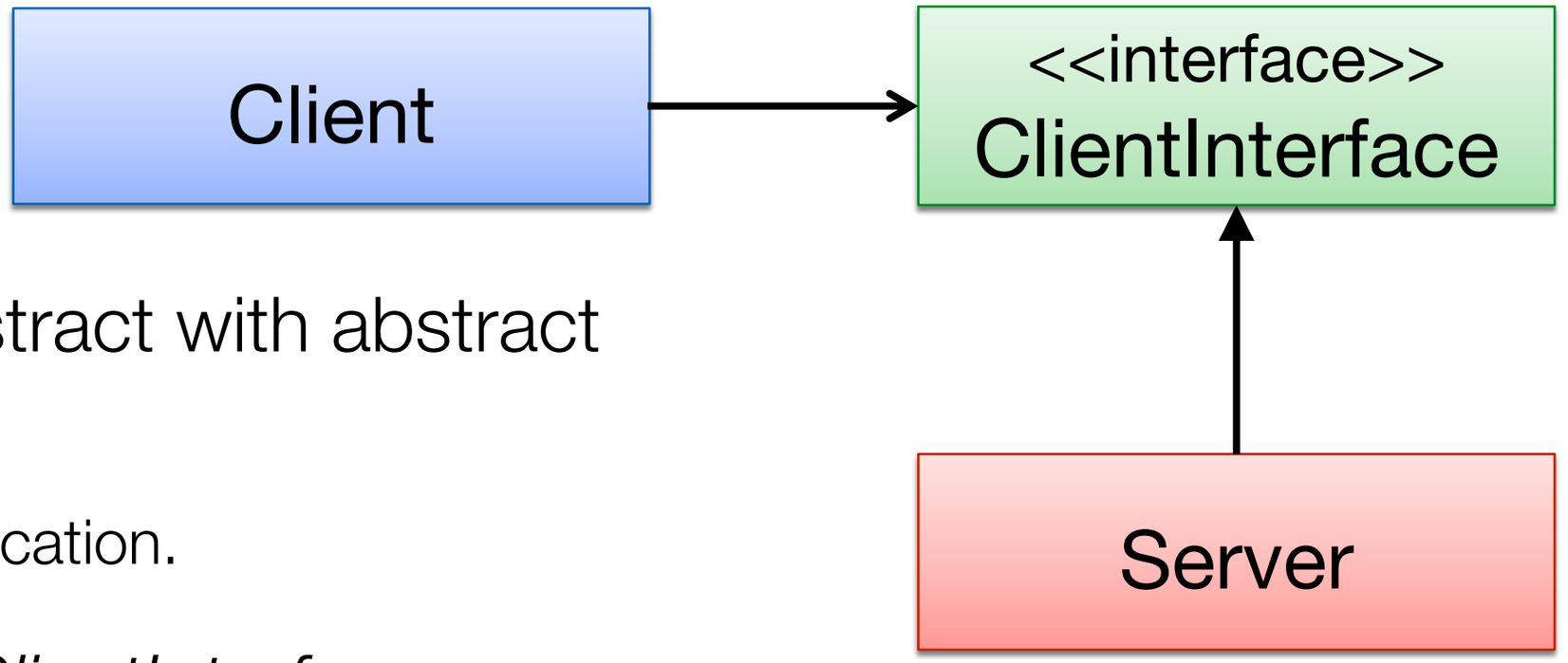
- Both *Client* and *Server* are concrete Java classes and *Client* uses *Server*.

- If we want the *Client* to use a different server, the *Client* source code must be modified to use the new server class:

  - *Client* is directly coupled to *Server*.

  - **Violation of OCP:** extending the behaviour of *Client* results in modifications to its source code.

Client is both open and closed



- *ClientInterface* is abstract with abstract member functions:
  - i.e. closed for modification.
- *Server* implements *ClientInterface*.
- *Client* uses *ClientInterface*:
  - However, objects of the *Client* class will be using objects of the derivative *Server* class.
  - If we want *Client* to use a different server class, a new derivative of the *ClientInterface* can be created. The *Client* class can remain unchanged.
  - *i.e. open for extension.*

Shape Example

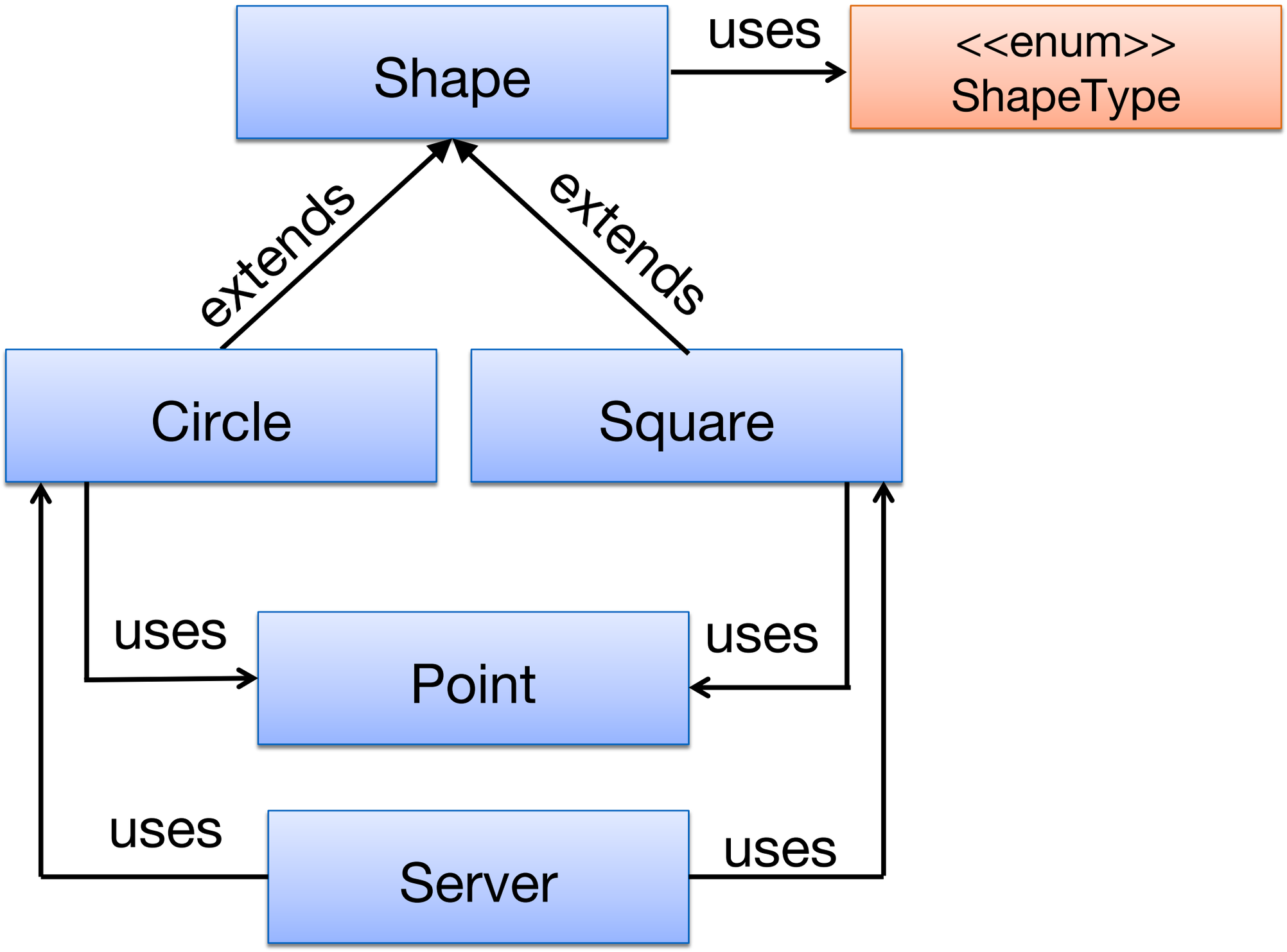
Shape Example

- Application must be able to draw circles and squares on a standard GUI.

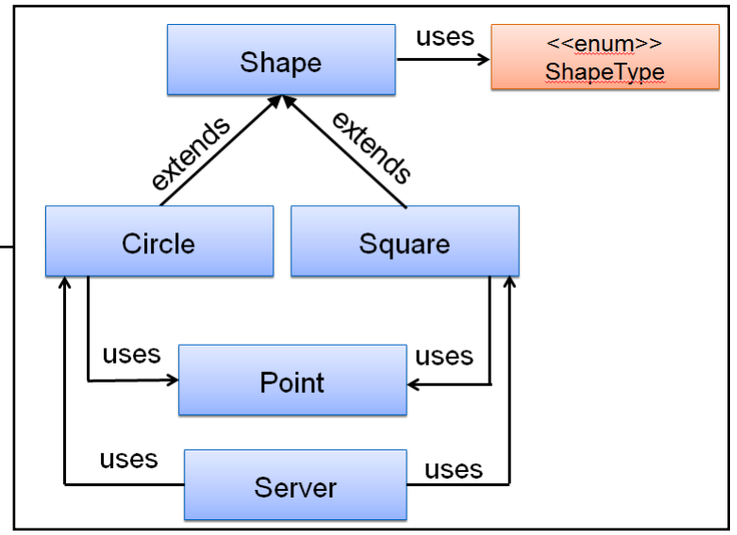
- The circles and squares must be drawn in a particular order.

- A list of the circles and squares will be created in the appropriate order and the program must walk the list in that order and draw each circle or square.

Shape Example 1 (violates OCP)



Shape Example 1 (violates OCP)

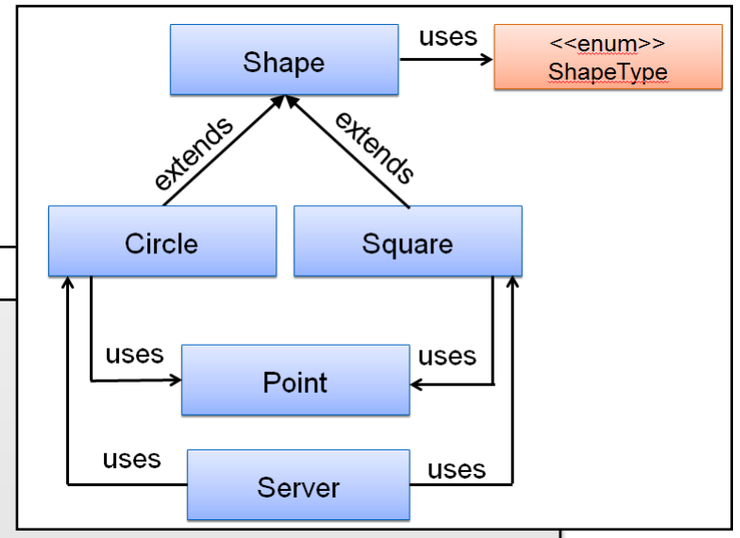



```
public class Point {

    private double xCoord;
    private double yCoord;

    public Point(){
        this.setxCoord(0.0);
        this.setyCoord(0.0);
    }

    public Point(double xCoord, double yCoord){
        this.setxCoord(xCoord);
        this.setyCoord(yCoord);
    }

    // getter and setter methods…
}
```

```
public class Shape
{
    private ShapeType type;

    // getter and setter…
}
```

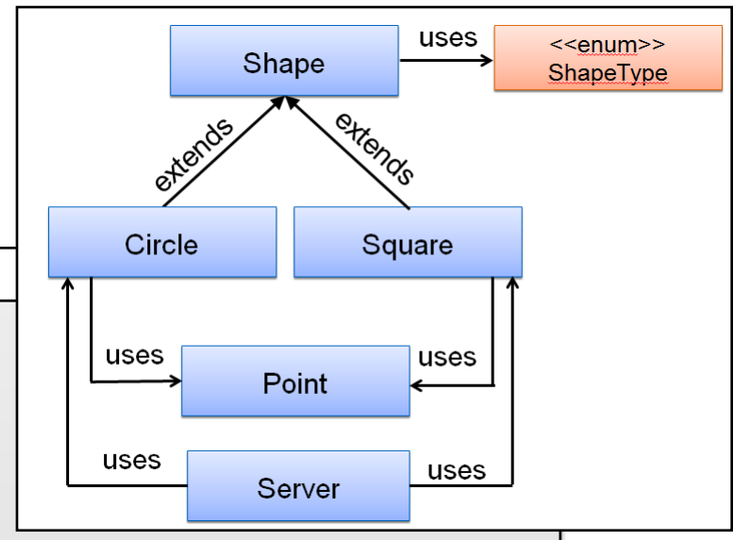
```
public enum ShapeType
{
    CIRCLE, SQUARE
}
```

Shape Example 1 (violates OCP)



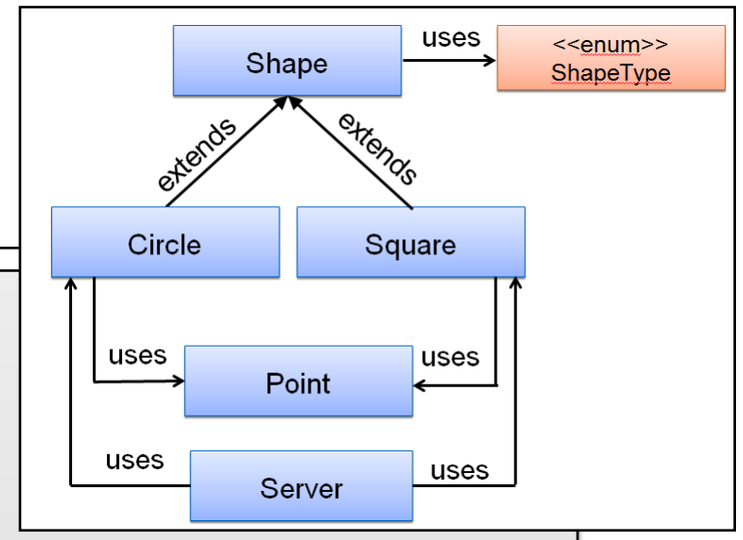

```
public class Circle extends Shape
{
    private double radius;
    private Point center;

    public Circle() {
        super.setType(ShapeType.CIRCLE);
        this.setRadius(10);                 //default size
        this.setCenter(new Point());  //default location
    }

     // getter methods…
     // setter methods…
}
```

Shape Example 1 (violates OCP)




```
public class Square extends Shape
{
    private double length;
    private Point leftCorner;

    public Square() {
        super.setType(ShapeType.SQUARE);
        this.setLength(10);                    //default size
        this.setLeftCorner(new Point());  //default location
    }

     // getter methods…
     // setter methods…
}
```

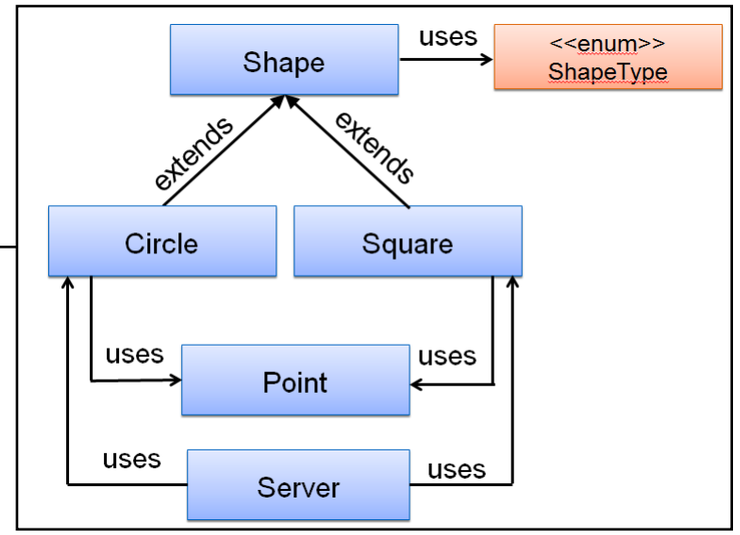
Shape Example 1 (violates OCP)




```
public class Server {

    void drawSquare(Square square) {
        System.out.println("I'm drawing a square");
    }

    void drawCircle(Circle circle) {
        System.out.println("I'm drawing a circle");
    }

    void drawAllShapes(List<Shape> shapes) {
        for (Shape shape : shapes) {
            switch (shape.getType()) {
            case SQUARE:
                drawSquare((Square) shape);
                break;
            case CIRCLE:
                drawCircle((Circle) shape);
                break;
            }
        }
    }
```

```
    public static void main(String args[]){
        List<Shape> shapes = new ArrayList<>();
        shapes.add(new Circle());
        shapes.add(new Square());
        shapes.add(new Circle());
        Server app = new Server();
        app.drawAllShapes(shapes);
    }
}
```

Proposed Extension : Triangle



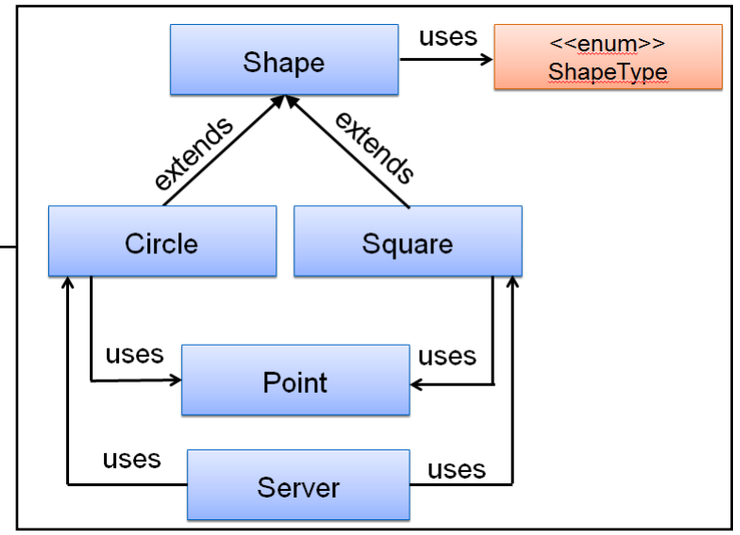

```
public class Triangle extends Shape
{
    private Point pointA;
    private Point pointB;
    private Point pointC;

    public Triangle()
    {
        super.setType(ShapeType.TRIANGLE);
        this.setPointA(new Point());
        this.setPointB(new Point());
        this.setPointC(new Point());
    }
     // getter methods…
     // setter methods…
}
```

```
public enum ShapeType
{
    CIRCLE,
    SQUARE,
    TRIANGLE
}
```

- Open for Extension?
  - Yes. New Triangle class.

Proposed Extension : Triangle

```
public class Server {

    void drawSquare(Square square) {
        System.out.println("I'm drawing a square");
    }

    void drawCircle(Circle circle) {
        System.out.println("I'm drawing a circle");
    }

  void drawAllShapes(List<Shape> shapes) {
      for (Shape shape : shapes) {
          switch (shape.getType()) {
          case SQUARE:
              drawSquare((Square) shape);
              break;
          case CIRCLE:
              drawCircle((Circle) shape);
              break;
          }
      }
  }
```



Open or Closed for Modification ?

Proposed Extension : Triangle

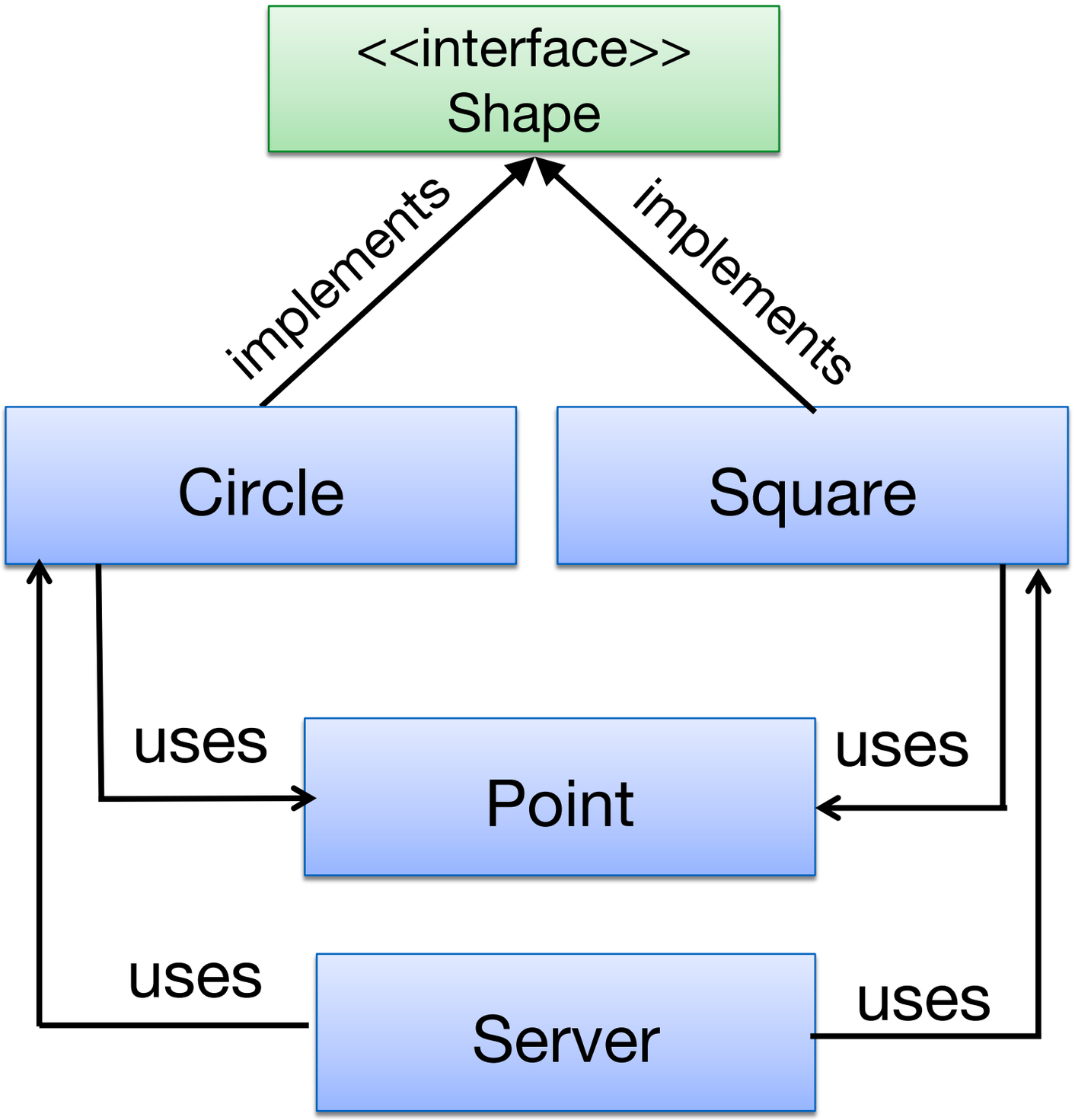
```
void drawSquare(Square square) {
    System.out.println("I'm drawing a square");
}

void drawCircle(Circle circle) {
    System.out.println("I'm drawing a circle");
}

void drawTriangle(Triangle triangle) {
    System.out.println("I'm drawing a triangle");
}
```

Class has to be modified to include new shape type – **Not** Closed for Modification!

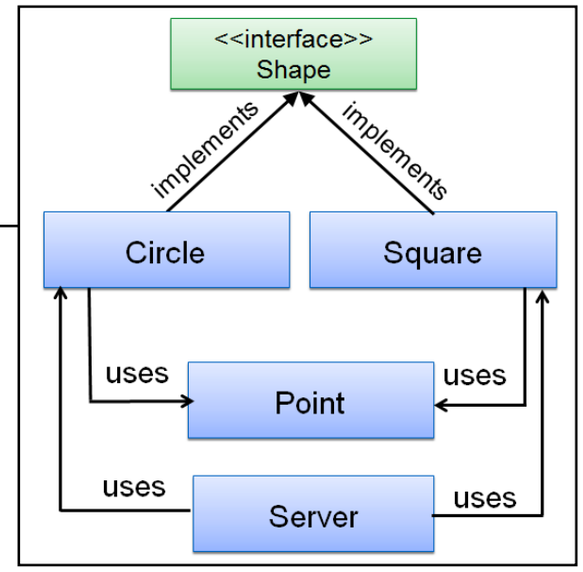
Server class

```
void drawAllShapes(List<Shape> shapes) {
    for (Shape shape : shapes) {
        switch (shape.getType()) {
        case SQUARE:
            drawSquare((Square) shape);
            break;
        case CIRCLE:
            drawCircle((Circle) shape);
            break;
        case TRIANGLE:
            drawTriangle((Triangle) shape);
            break;
        }
    }
}
```

Shape Example 2 (conforming to OCP)



Shape Example 2 (conforming to OCP)



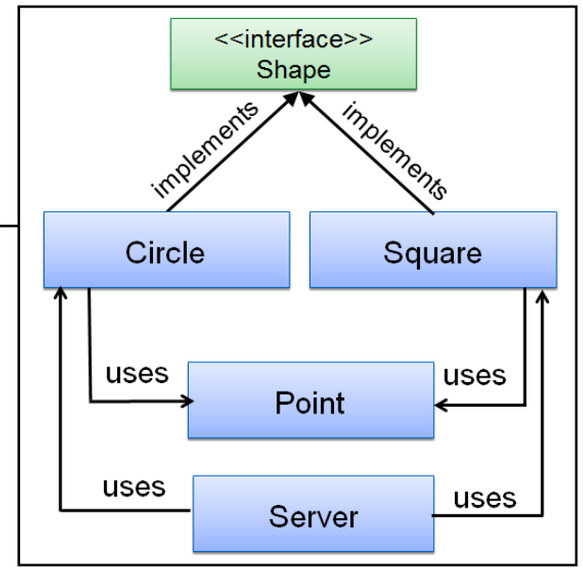

```
public class Circle implements Shape
{
    private double radius;
    private Point center;

    public Circle() {
        this.setRadius(10);                  //default size
        this.setCenter(new Point());  //default location
    }

    public void draw() {
        System.out.println("I'm drawing a circle");
    }
    //getters and setters
}
```

```
public interface Shape
{
    public void draw();
}
```

Interface defines abstraction Shape.

Circle implements this abstraction.

Shape Example 2 (conforming to OCP)



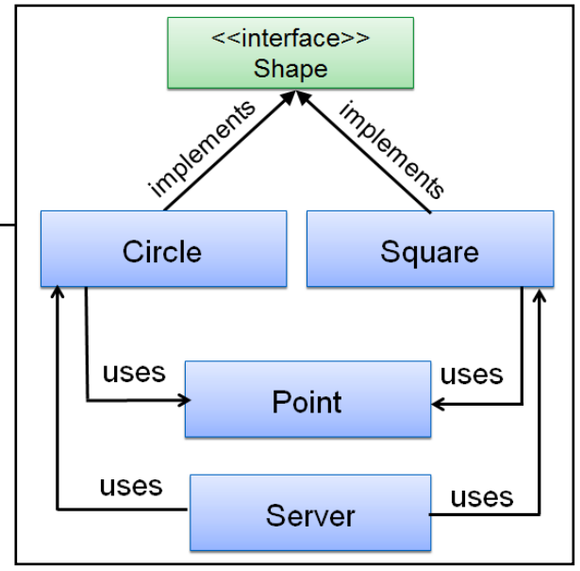

```
public class Square implements Shape
{
    private double length;
    private Point leftCorner;

    public Square() {
       this.setLength(10);                    //default size
       this.setLeftCorner(new Point());  //default location
    }

    public void draw() {
        System.out.println("I'm drawing a square");
     }
     //getters and setters
}
```

```
public interface Shape
{
    public void draw();
}
```

Interface defines abstraction Shape.

Square implements this abstraction.

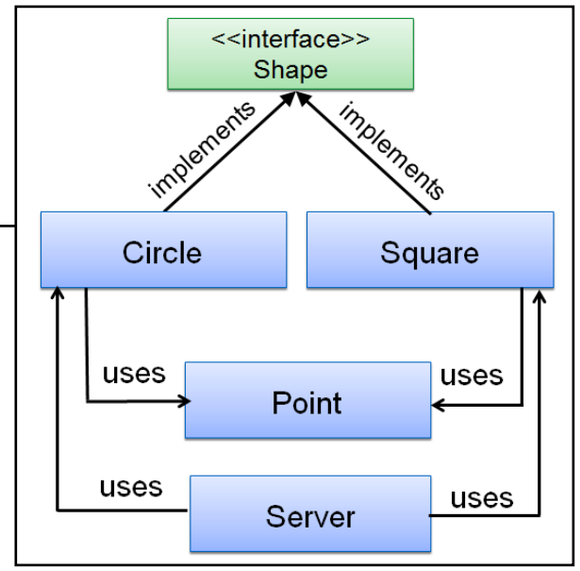
Shape Example 2 (conforming to OCP)




```
public class Server
{
    void drawAllShapes(List<Shape> shapes) {
        for (Shape shape : shapes)
        {
            shape.draw();
        }
    }

   //rest of code…
}
```

drawAllShapes method is changed to refer to Shape only.

Open and Closed



- Is Shape interface open to extension (e.g. Triangle)?
  Yes
- Is Server closed for this extensions?
  Yes

```
public class Triangle implements Shape
{
    private Point pointA;
    private Point pointB;
    private Point pointC;

    public void draw()
    {
       System.out.println("I'm drawing a triangle");
    }
}
```

```
public class Server
{
    void drawAllShapes(List<Shape> shapes)
    {
        for (Shape shape : shapes)
        {
            shape.draw();
        }
    }

   //rest of code…
}
```

- Server's behaviour can be extended without modification – it is closed.

Strategic Closure (1)

- It should be clear that no significant program can be 100% closed.

- Consider what would happen to the drawAllShapes methods (on the previous slide) if:
  - we decided that all Circles should be drawn before any Squares. The drawAllShapes method is **NOT** closed against a change like this.

- In general, no matter how "closed" a module is, there will always be some kind of change against which it is not closed.

Strategic Closure (2)

- Since closure cannot be complete, it must be strategic.
- That is, the designer must choose the kinds of changes against which to close his/her design.
- This takes a certain amount of foresight derived from experience. The experienced designer knows the users and the industry well enough to judge the probability of different kinds of changes. S/he then makes sure that the open-closed principle is invoked for the most probable changes.

OCP Summary

- Without OCP:
  - code tends to have large switch statements or if-then constructions, possibly enums.
  - adding a new implementation (type) means adding new code to the client of the type.
  - the client of a class must be aware of all subclasses.
- With OCP:
  - Use abstraction and dynamic binding to avoid dependency on a concrete class.
  - The abstraction is:
    - Closed for modification – it is fixed.
    - Open for extension – can create new derivative classes of the abstraction.