

# Agile Software Development

---

Produced  
by

Eamonn de Leastar (edelestar@wit.ie)  
Dr. Siobhán Drohan (sdrohan@wit.ie)

# Open Closed Principle

---

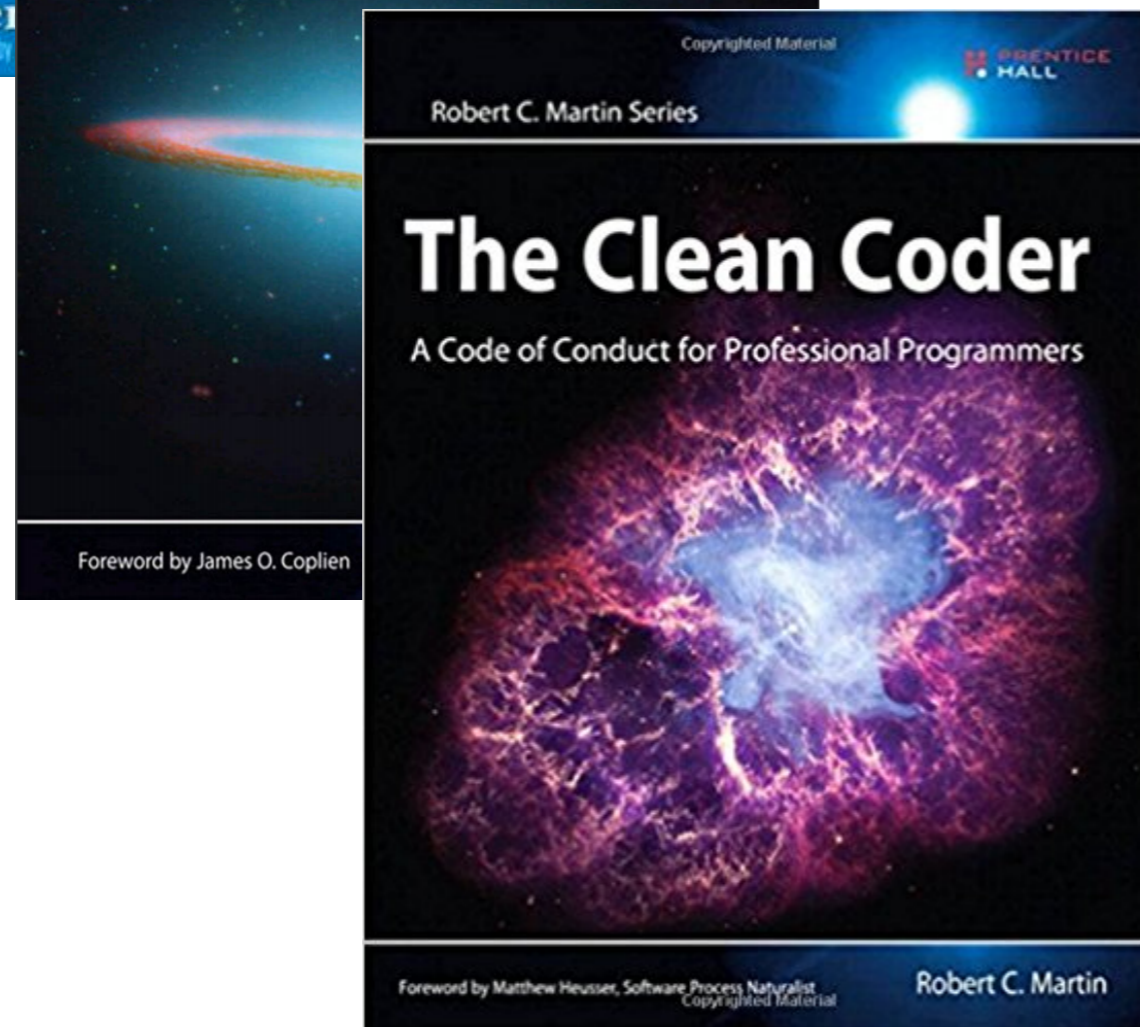
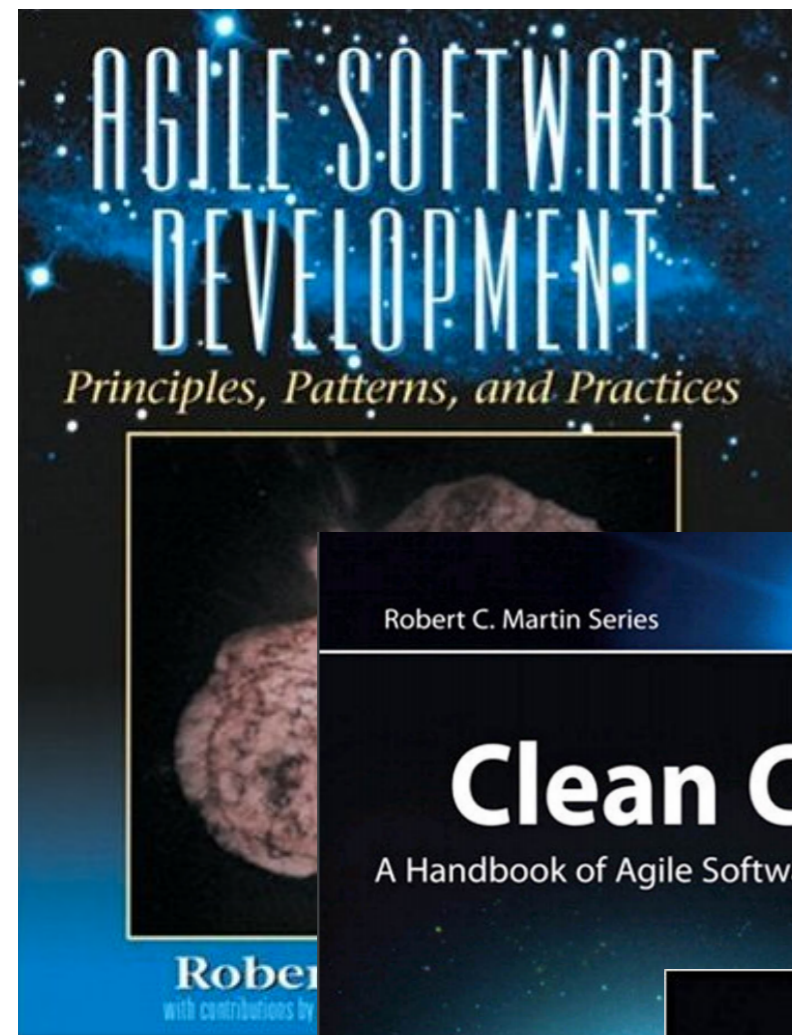
# SOLID Principles

---

- *The **S**ingle Responsibility Principle*
  - A class should have one, and only one, reason to change.
- *The **O**pen Closed Principle*
  - You should be able to extend a classes behavior, without modifying it.
- *The **L**iskov Substitution Principle*
  - Derived classes must be substitutable for their base classes.
- *The **I**nterface Segregation Principle*
  - Make fine grained interfaces that are client specific.
- *The **D**ependency Inversion Principle*
  - Depend on abstractions, not on concretions.

# References

---



- ⊕ Agile principles, and the fourteen practices of Extreme Programming
- ⊕ Spiking, splitting, velocity, and planning iterations and releases
- ⊕ Test-driven development, test-first design, and acceptance testing
- ⊕ Refactoring with unit testing
- ⊕ Pair programming
- ⊕ Agile design and design smells
- ⊕ The five types of UML diagrams and how to use them effectively
- ⊕ Object-oriented package design and design patterns
- ⊕ How to put all of it together for a real-world project

# Solid on Wikipedia

Initial	Stands for (acronym)	Concept
<b>S</b>	SRP	<b>Single responsibility principle</b> the notion that an <b>object</b> should have only a single responsibility.
<b>O</b>	OCP	<b>Open/closed principle</b> the notion that “software entities ... should be open for extension, but closed for modification”.
<b>L</b>	LSP	<b>Liskov substitution principle</b> the notion that “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program”. See also <b>design by contract</b> .
<b>I</b>	ISP	<b>Interface segregation principle</b> the notion that “many client specific interfaces are better than one general purpose interface.” <sup>[5]</sup>
<b>D</b>	DIP	<b>Dependency inversion principle</b> the notion that one should “Depend upon Abstractions. Do not depend upon concretions.” <sup>[5]</sup> <b>Dependency injection</b> is one method of following this principle.

[http://en.wikipedia.org/wiki/Solid\\_\(object-oriented\\_design\)](http://en.wikipedia.org/wiki/Solid_(object-oriented_design))

# 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)

# Change Strategy

---

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

-> design exhibits 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.

# OCP

---

## **SOFTWARE ENTITIES SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION**

- ⊕ If a single change to a program results in a cascade of changes to dependent modules:
  - ⊕ program exhibits the undesirable attributes
  - ⊕ program becomes fragile, rigid, unpredictable and unreusable
- ⊕ 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 (3) - 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/Object-Oriented_Software_Construction)  
[http://en.wikipedia.org/wiki/Open/closed\\_principle](http://en.wikipedia.org/wiki/Open/closed_principle)

# Open and Closed?

---

**SOFTWARE ENTITIES SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION**

⊕ *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.
- ⊕ the source code of such a module is inviolate

# Open and Closed?

---

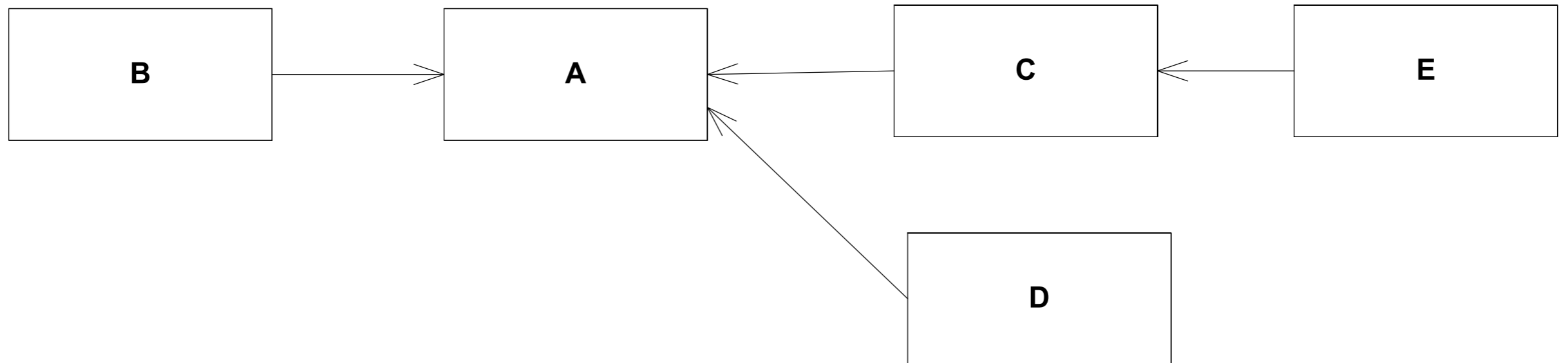
***SOFTWARE ENTITIES SHOULD BE OPEN FOR EXTENSION BUT  
CLOSED FOR MODIFICATION***

⊕ Contradiction?

- ⊕ the normal way to extend the behavior of a module is to make changes to that module
- ⊕ a module that cannot be changed is normally thought to have a fixed behavior

# Example (1)

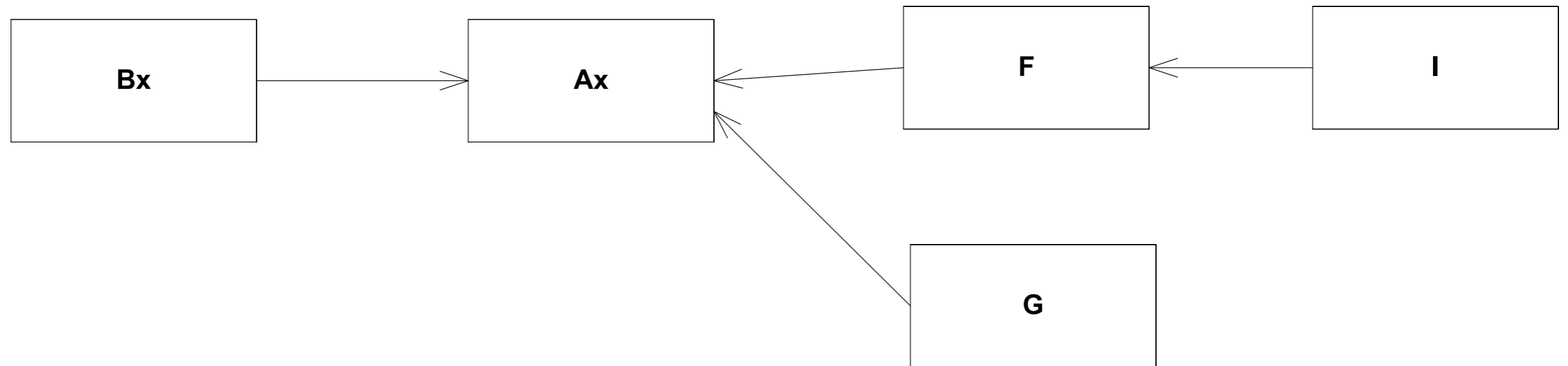
---



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

# Example (2)

---

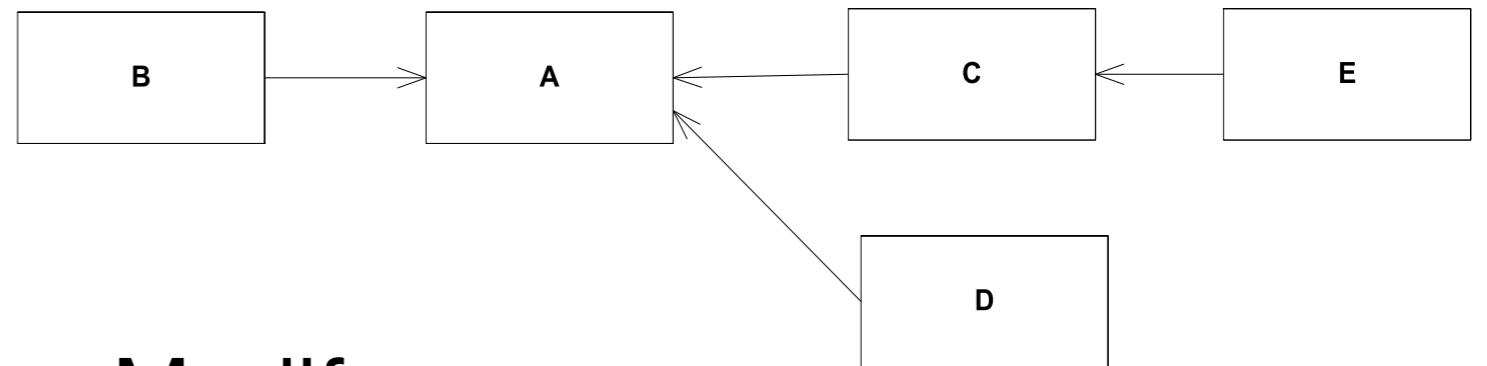


- ⊕ A new client – **Bx** – requires an extended or adapted version of a – called **Ax**.
- ⊕ Further clients of **Ax** are developed (**F**, **G**, **I**)

# 2 Possible Solutions

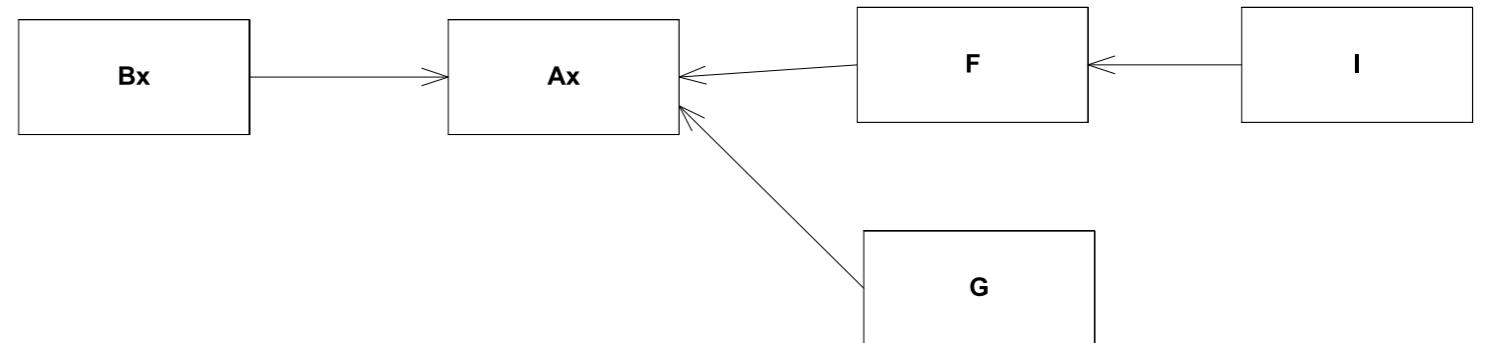
---

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



**Modify**

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*.

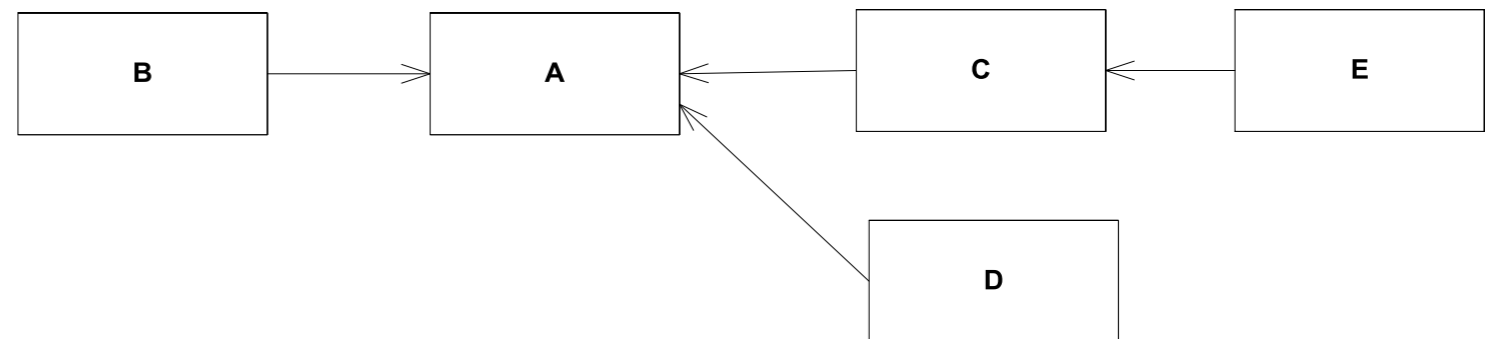


**Copy/Paste**

# Modify Solution

---

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



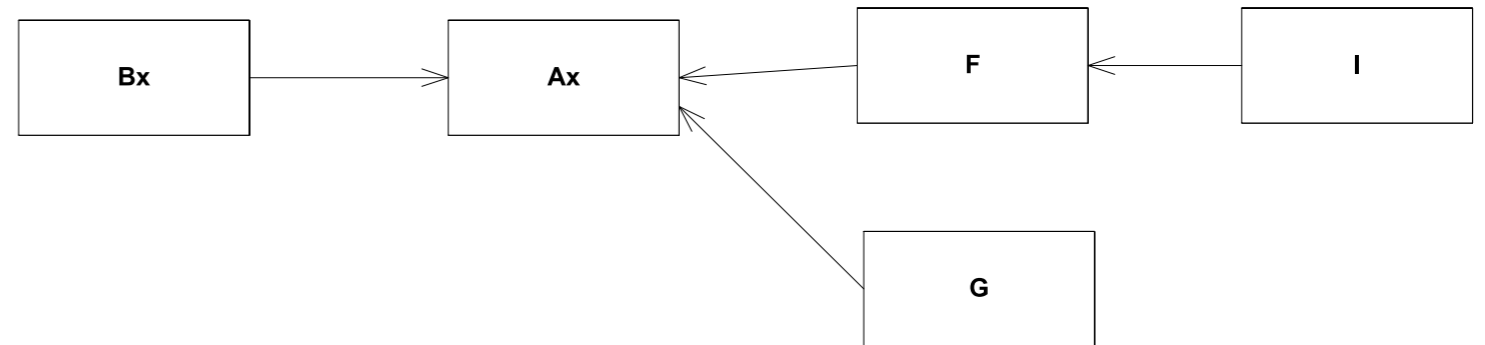
**Modify**

- ⊕ *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.*

# Copy/Paste Solution

---

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*.



**Copy/Paste**

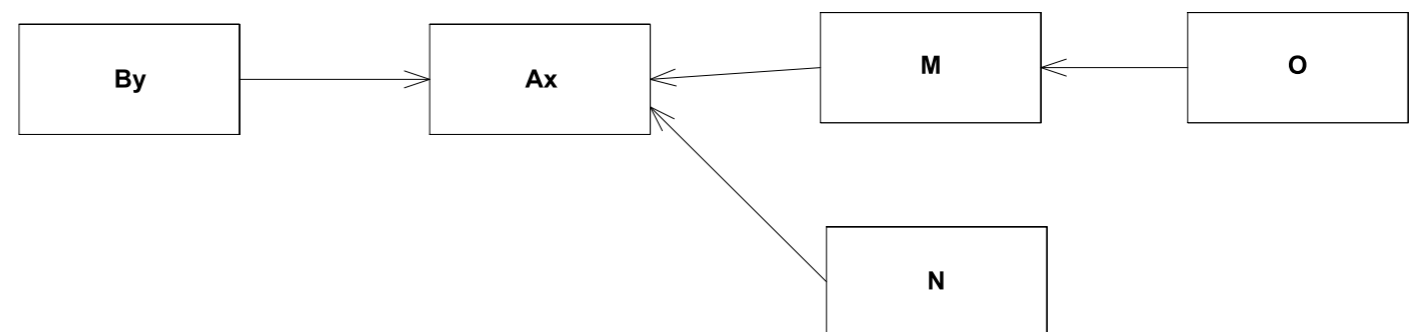
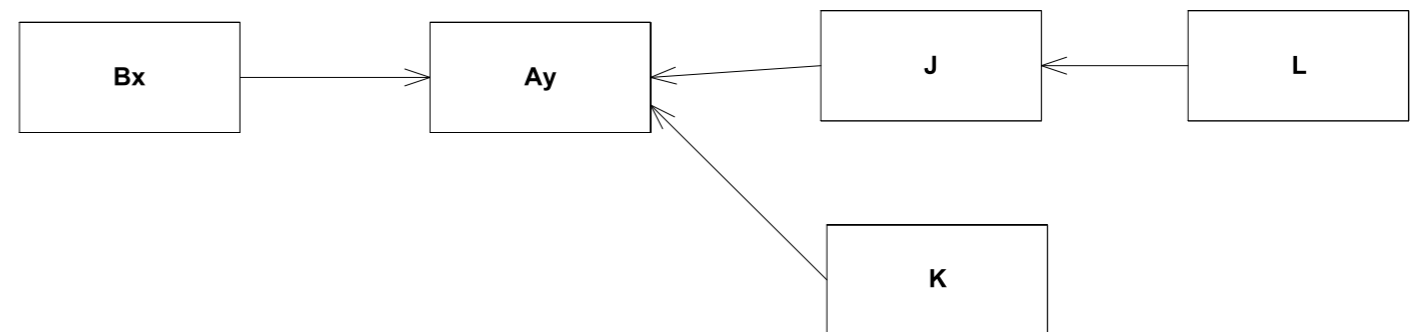
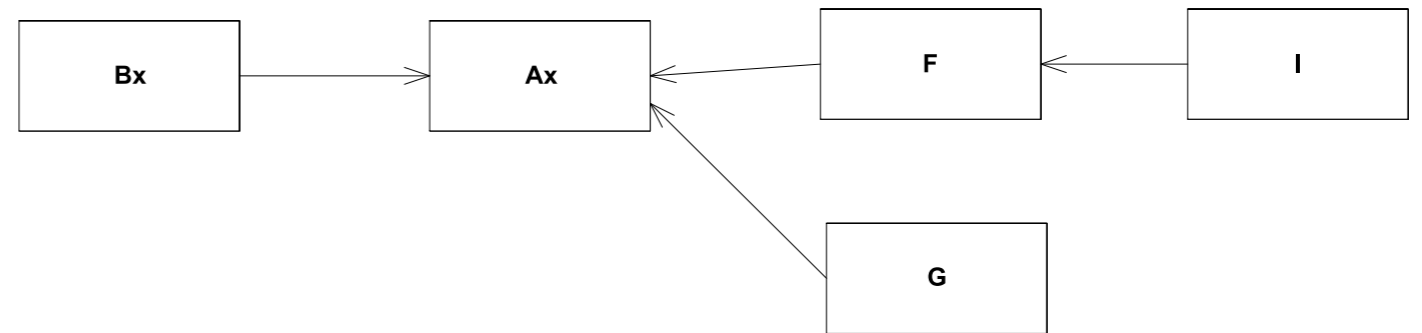
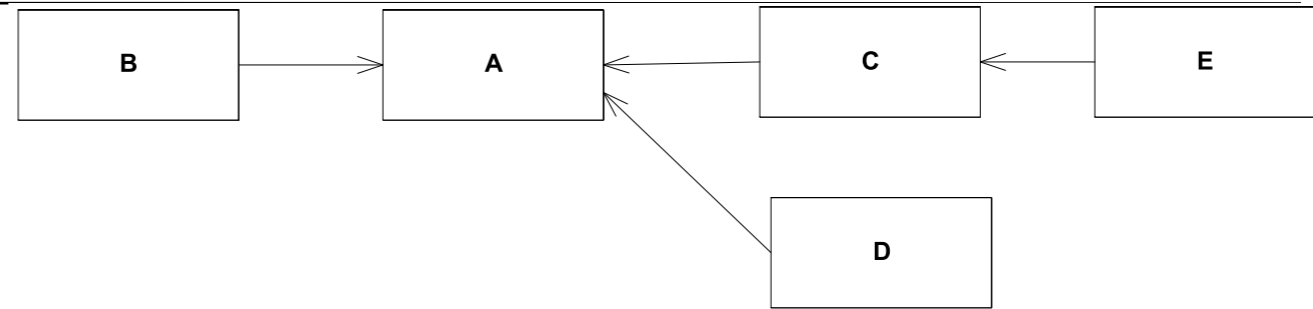
- ⊕ On the surface, solution 2 seems better as it does not require modifying any existing software.
- ⊕ But this solution may be even more catastrophic since it only postpones the day of reckoning.
- ⊕ Leads to an explosion of variants of the original modules, many of them very similar to each other although never quite identical.



# Copy/Paste Solution Implications

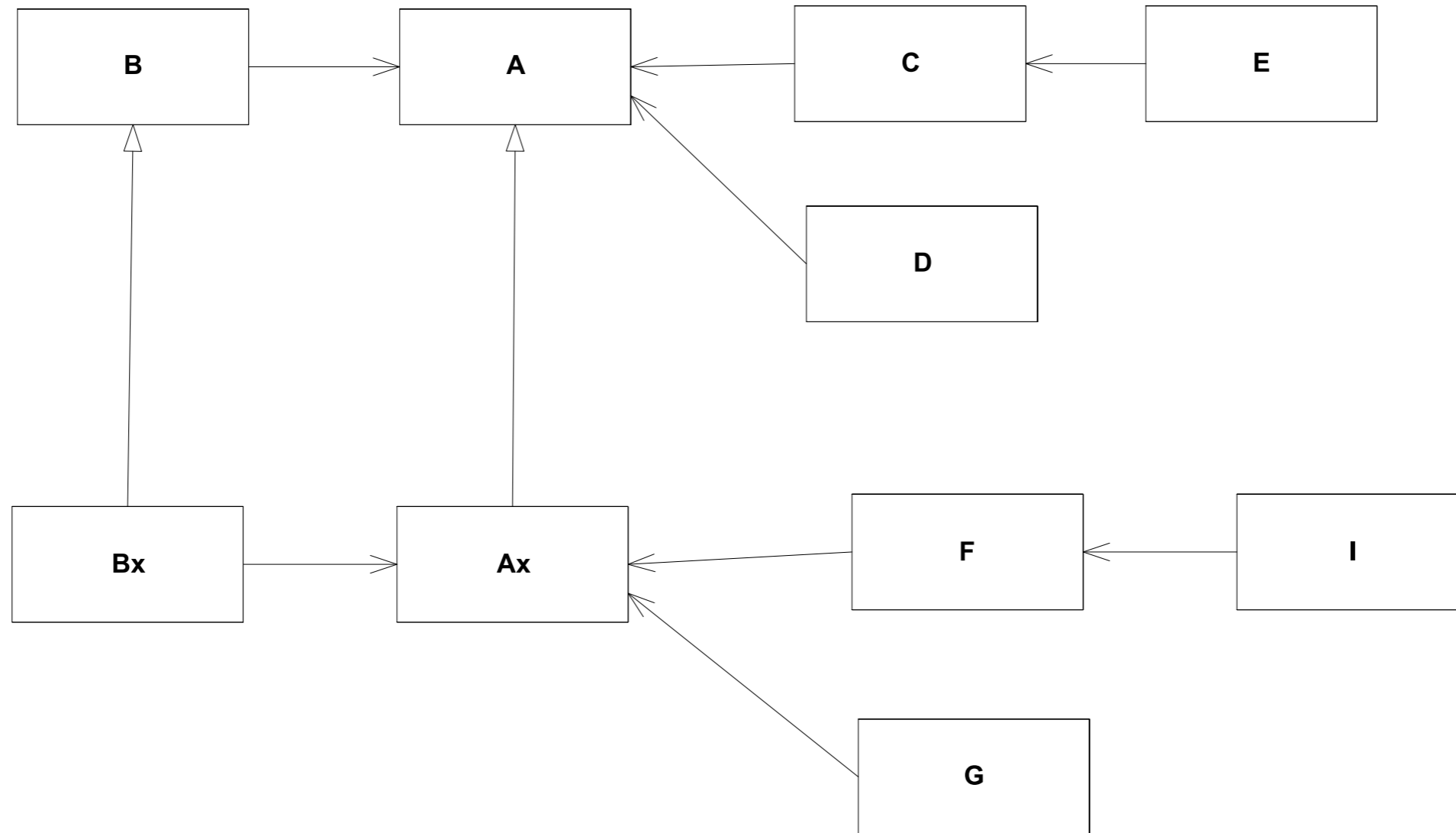
---

- ⊕ 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



# OCP Solution – Abstraction, Interfaces & Inheritance

---



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

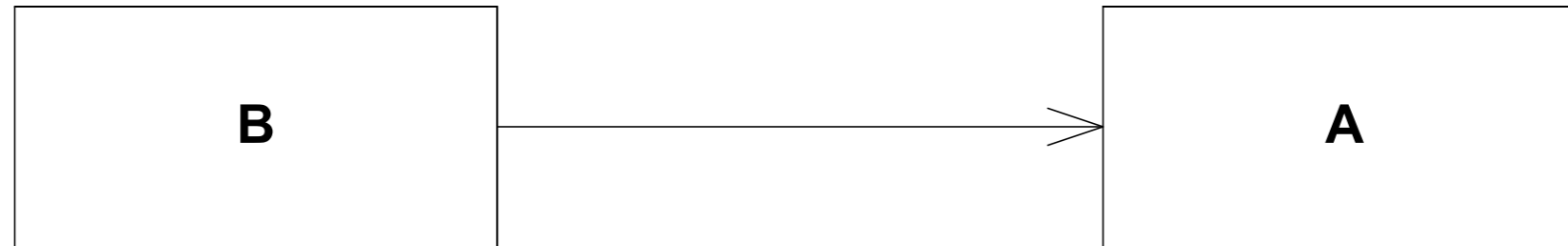
# Abstraction

---

- ⊕ 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
- ⊕ A module can implement these abstractions:
  - ⊕ Can be closed for modification since it fulfils an abstraction that is fixed.
  - ⊕ Yet the behavior of that module can be extended by creating new implementations of the abstraction

# Not Closed

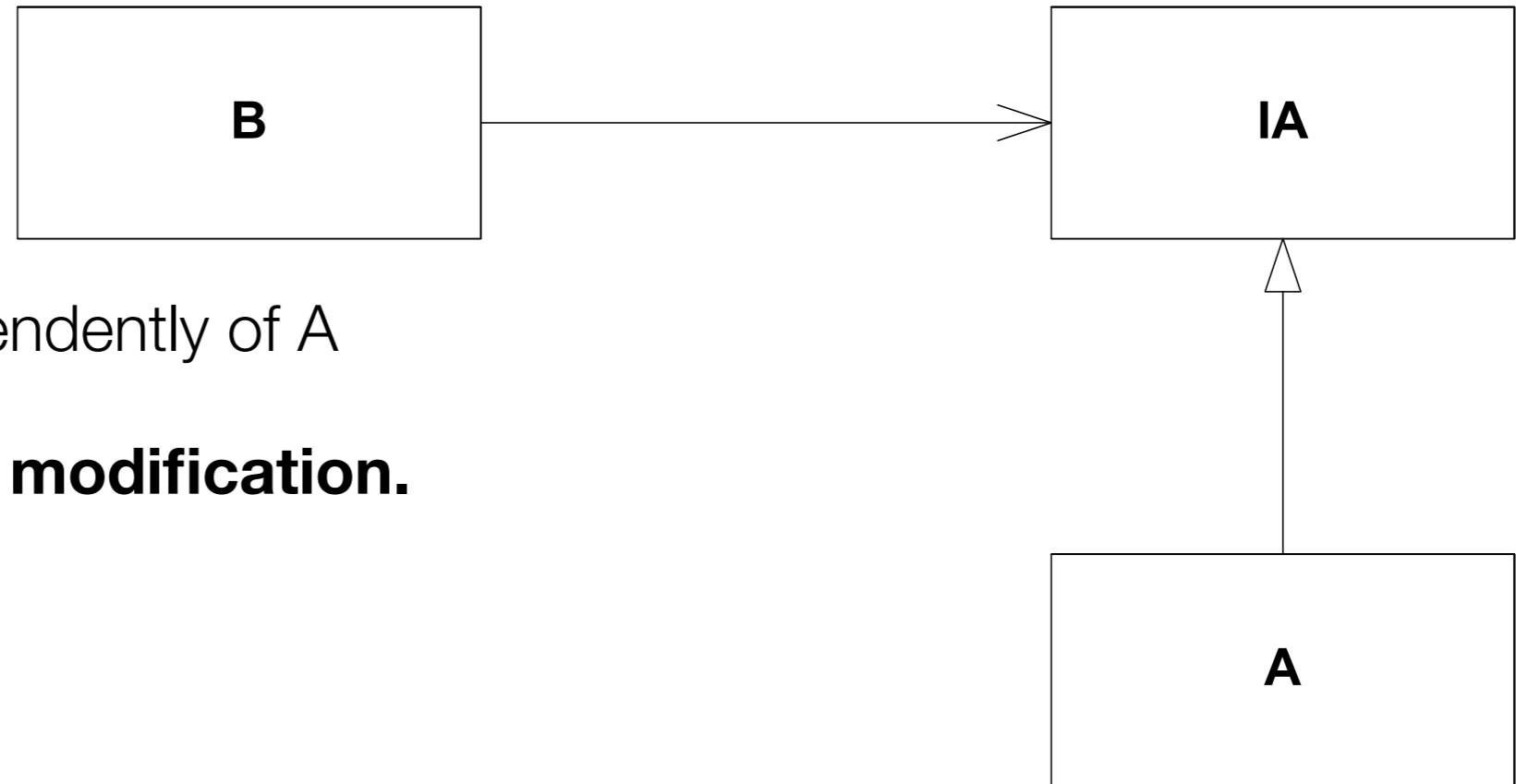
---



- ⊕ Both A and B are concrete Java classes, and B uses A.
- ⊕ To extend the behaviour of A, B must also be changed as it is directly coupled to A
- ⊕ A is not “Closed” to modifications - introducing changes to A ripples through to B

# Closing A

---



⊕ IA is defined independently of A

⊕ **It is closed for modification.**

⊕ A can be:

⊕ Used as is

⊕ Modified

⊕ Extended via inheritance

⊕ Completely replaced

⊕ I.e. => **It is open for extension**

# 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.

# Procedural Solution - C

---

```
enum ShapeType {circle, square};

struct Shape
{
    ShapeType itsType;
};

struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
};

struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};
```

⊕ The first element of each is a type code that identifies the data structure as either a circle or a square.

# Draw Functions

---

```
void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);
typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->itsType)
        {
            case square: DrawSquare((struct Square*)s);
                          break;
            case circle: DrawCircle((struct Circle*)s);
                          break;
        }
    }
}
```



# Proposed Extension : Triangle

---

```
enum ShapeType {circle, square, Triangle}

struct Triangle
{
    Point A;
    Point B;
    Point C;
}
void DrawTriangle(struct Triangle*);
```

⊕ Open for Extension

⊕ New enum entry and DrawTriangle() function.

# Open or Closed for Modification?

---

```
void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);
typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->itsType)
        {
            case square: DrawSquare((struct Square*)s);
                          break;
            case circle: DrawCircle((struct Circle*)s);
                          break;
        }
    }
}
```

# DrawAllShapes() violates OCP

---

```
struct Shape* s = list[i];
switch (s->itsType)
{
    case square:    DrawSquare((struct Square*)s);
                   break;
    case circle:   DrawCircle((struct Circle*)s);
                   break;
    case triangle : DrawTriangle(((struct Triangle*)s);
                               break
}
}
```

⊕ DrawAllShapes() would have to be modified to include new shape type – not closed for modification.

# Conforming to OCP

---

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

class Circle implements Shape
{
    private double itsRadius;
    private Point itsCenter;
    public void draw()
    { //... }
}

class Square implements Shape
{
    private double itsSide;
    private Point itsTopLeft;
    public void draw()
    { //... }
}
```

- ⊕ Interface defines abstraction Shape.
- ⊕ Circle and Square implement this abstraction.

# drawShapes()

---

```
class Canvas
{
    void drawShape (Shape s)
    {
        s.draw();
    }
    void drawShapes (Collection<Shape> shapes)
    {
        for (Shape s:shapes)
        {
            s.draw()
        }
    }
}
```

# Open and Closed

---

- ⊕ Is Shape interface open to extension?
- ⊕ Is Canvas closed for these extensions?

```
class Triangle implements Shape
{
    private Point A;
    private Point B;
    private point C

    public void draw()
    { //... }
}
```

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

# Strategic Closure

---

- ⊕ No significant program can be 100% closed.
- ⊕ 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
  - ⊕ makes sure that the open-closed principle is invoked for the most probable changes.

# OCP Succinctly

---

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