# Object-oriented programming

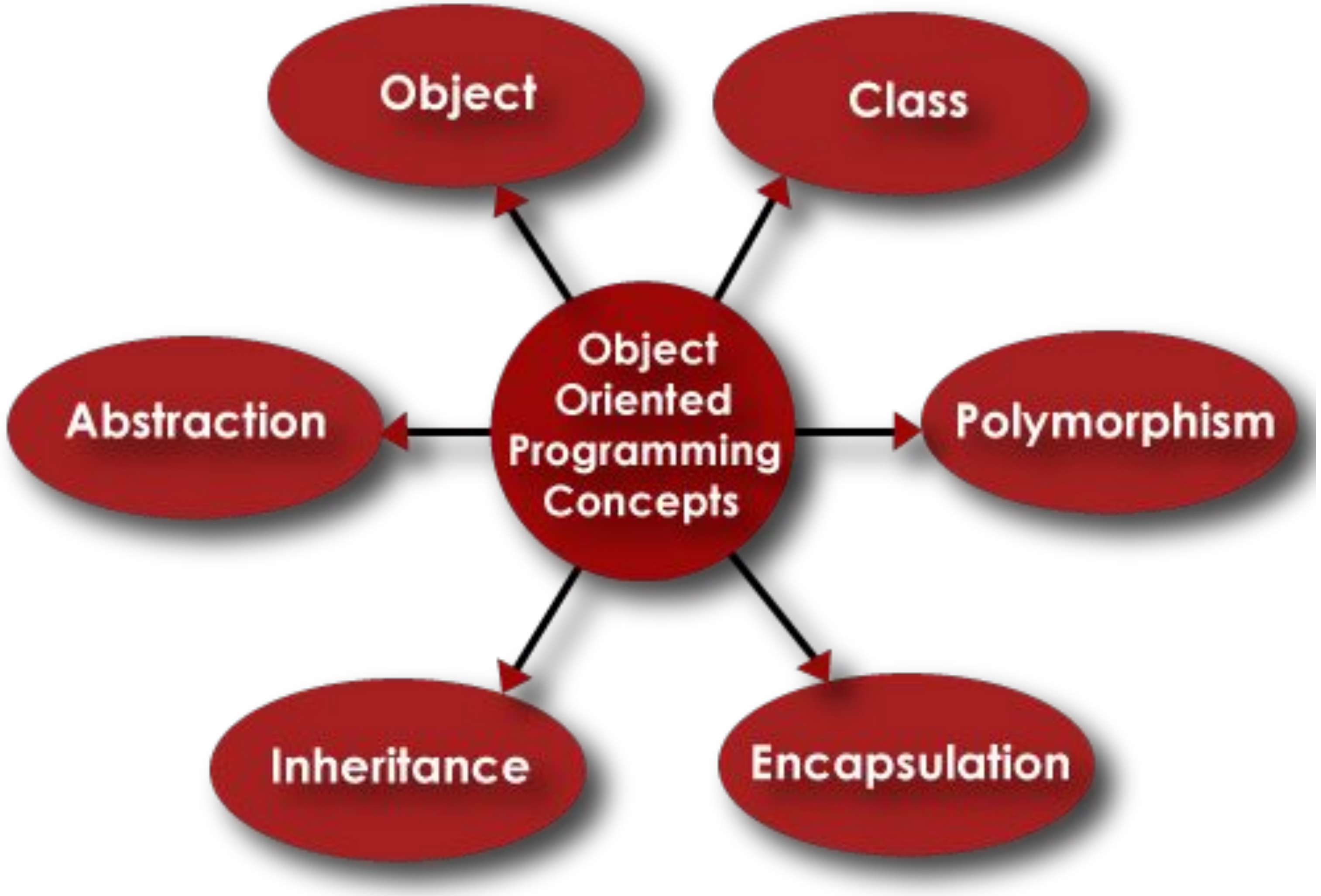
Second semester

Lecture №1

OOP Conception, SOLID principles of OOP, Relationships between classes

PhD, Alexander Vlasov

## Total Recall, key words



- Contract
- Modularity
- Typing
- Concurrency
- Persistence

# S.O.L.I.D

## Single Responsibility Principle

S

*"A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE."*

Gather together the things that change for the same reasons. Separate those things that change for different reasons.

## Open/Closed Principle

O

*"SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS etc) SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION"*

## Liskov Substitution Principle

L

*"SUBCLASSES SHOULD BEHAVE NICELY WHEN USED IN PLACE OF THEIR BASE CLASS"*

The sub-types must be replaceable for super-types without breaking the program execution.

## Interface Segregation Principle

I

*"A CLIENT SHOULD NEVER BE FORCED TO IMPLEMENT AN INTERFACE THAT IT DOESN't USE OR CLIENTS SHOULDN'T BE FORCED TO DEPEND ON METHODS THEY DON'T USE"*

Keep protocols small, don't force classes to implement methods they can't.

## Dependency Inversion Principle

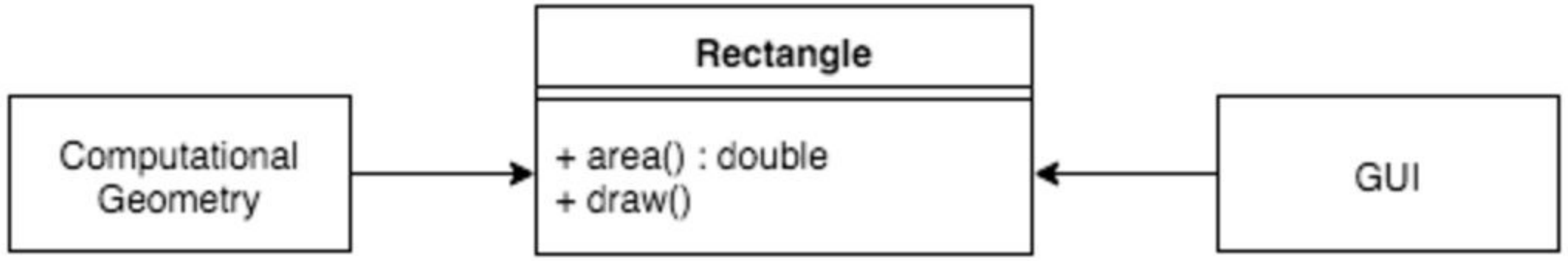
D

*"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"*

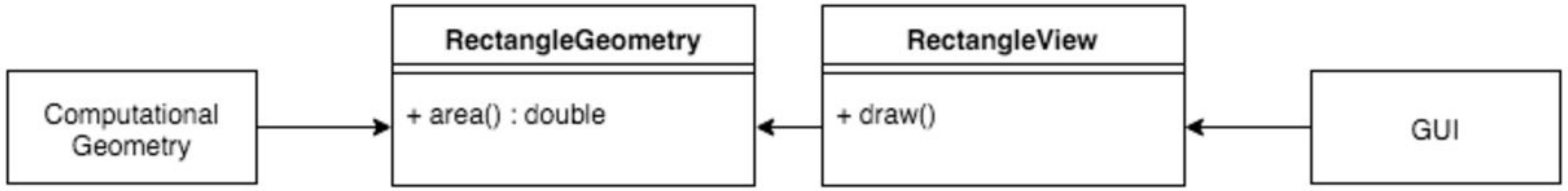
## Single Responsibility Principle (SRP)

- The Single Responsibility principle states that every module, class, or function should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or function.
- **A class should have only one reason to change**
- This means that every class should have a single responsibility or single job or single purpose.

# Single Responsibility Principle (SRP)



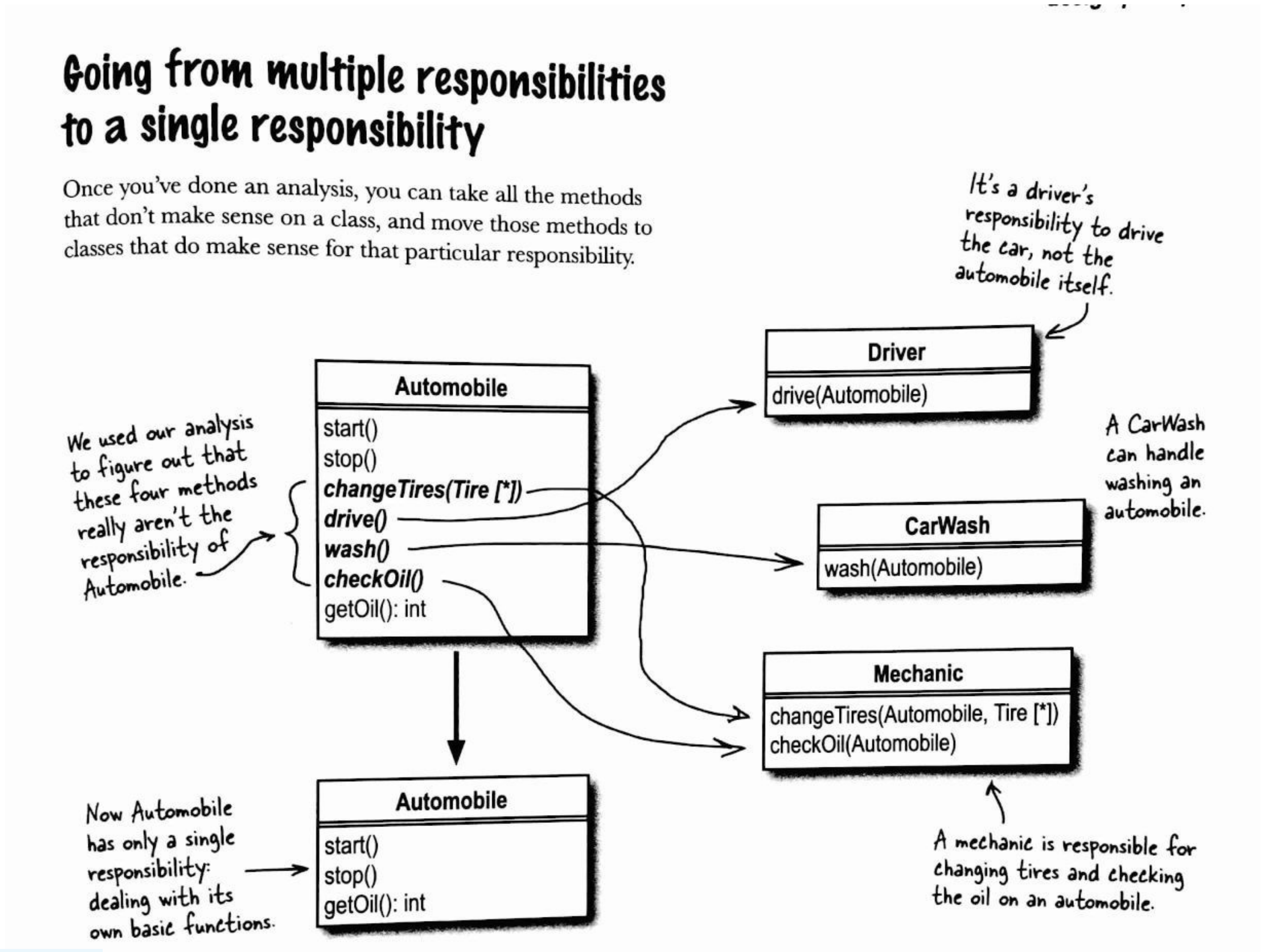
Wrong approach



SRP approach

# Single Responsibility Principle (SRP)

When SRP is followed, testing is easier. With a single responsibility, the class will have fewer test cases. Less functionality also means less dependencies to other modules or classes. It leads to better code organization since smaller and well-purposed classes are easier to search.



**Going from multiple responsibilities to a single responsibility**

Once you've done an analysis, you can take all the methods that don't make sense on a class, and move those methods to classes that do make sense for that particular responsibility.

We used our analysis to figure out that these four methods really aren't the responsibility of Automobile.

It's a driver's responsibility to drive the car, not the automobile itself.

A CarWash can handle washing an automobile.

A mechanic is responsible for changing tires and checking the oil on an automobile.

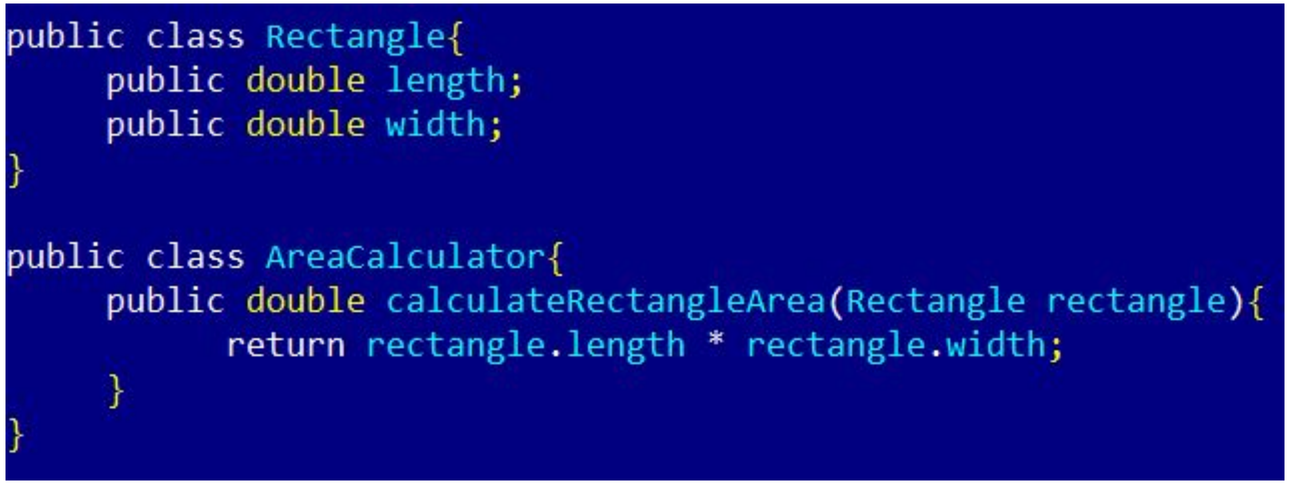
Now Automobile has only a single responsibility: dealing with its own basic functions.

Çapar, Ali Can. 2018. "SOLID -Single Responsibility Principle." December 19. Accessed 2019-05-23.

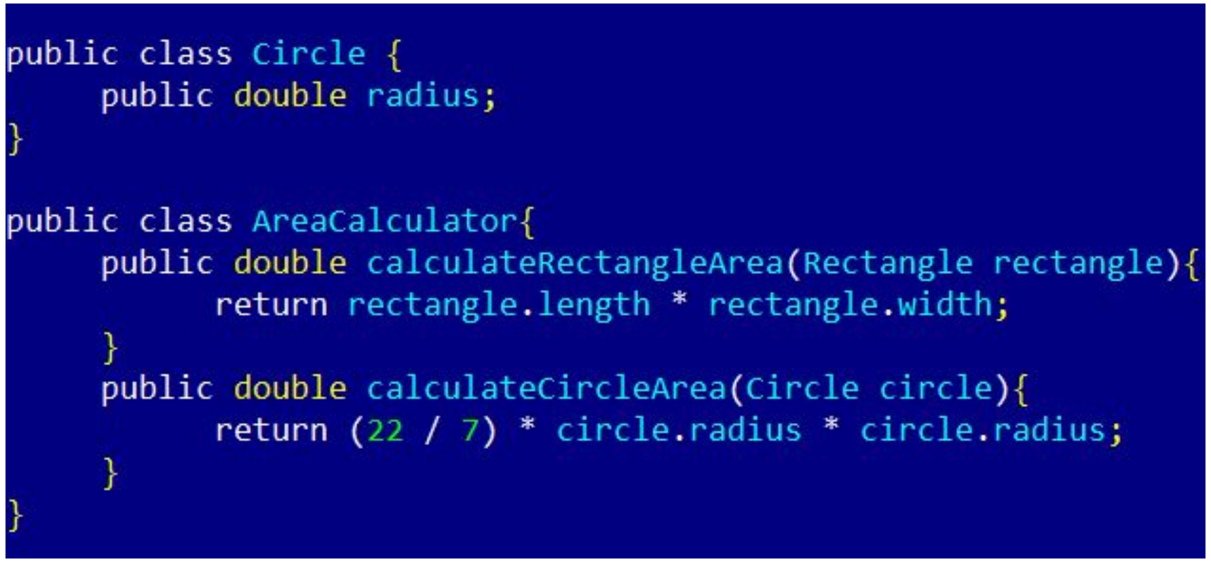
## Open/Closed Principle (OCP)

- The Open/Closed principle states that you should be able to extend a class behavior, without modifying it, that is we can extend the behavior of software entity without touching the source code of the entity.
- **Software entities … should be open for extension, but closed for modification**
- Open-Closed principle can be achieved using abstraction and inheritance.

# Open/Closed Principle (OCP)

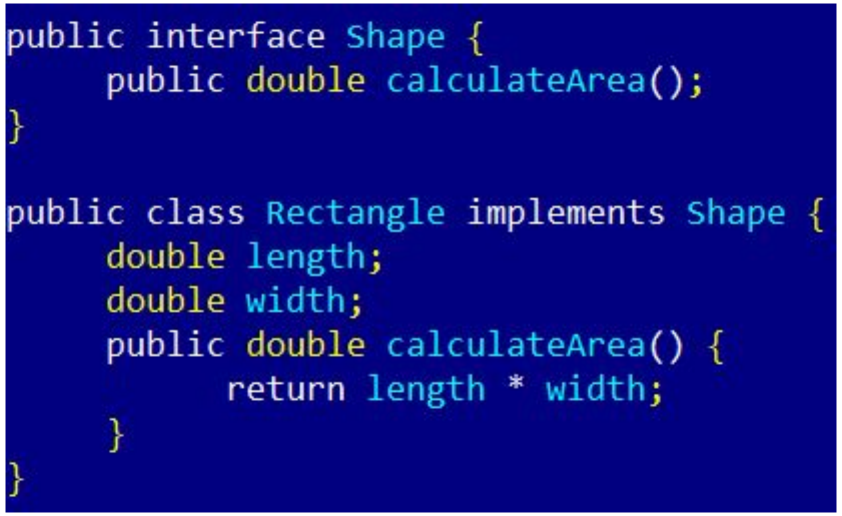


```java
public class Rectangle{
     public double length;
     public double width;
}

public class AreaCalculator{
     public double calculateRectangleArea(Rectangle rectangle){
           return rectangle.length * rectangle.width;
     }
}
```

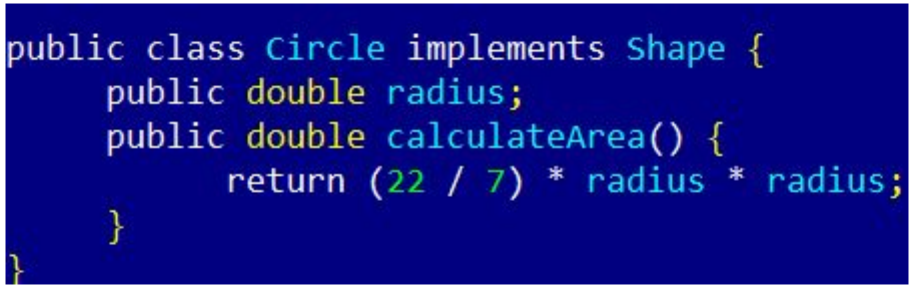


```java
public class Circle {
     public double radius;
}

public class AreaCalculator{
     public double calculateRectangleArea(Rectangle rectangle){
           return rectangle.length * rectangle.width;
     }
     public double calculateCircleArea(Circle circle){
           return (22 / 7) * circle.radius * circle.radius;
     }
}
```

# Open/Closed Principle (OCP)



```java
public interface Shape {
     public double calculateArea();
}

public class Rectangle implements Shape {
     double length;
     double width;
     public double calculateArea() {
           return length * width;
     }
}
```

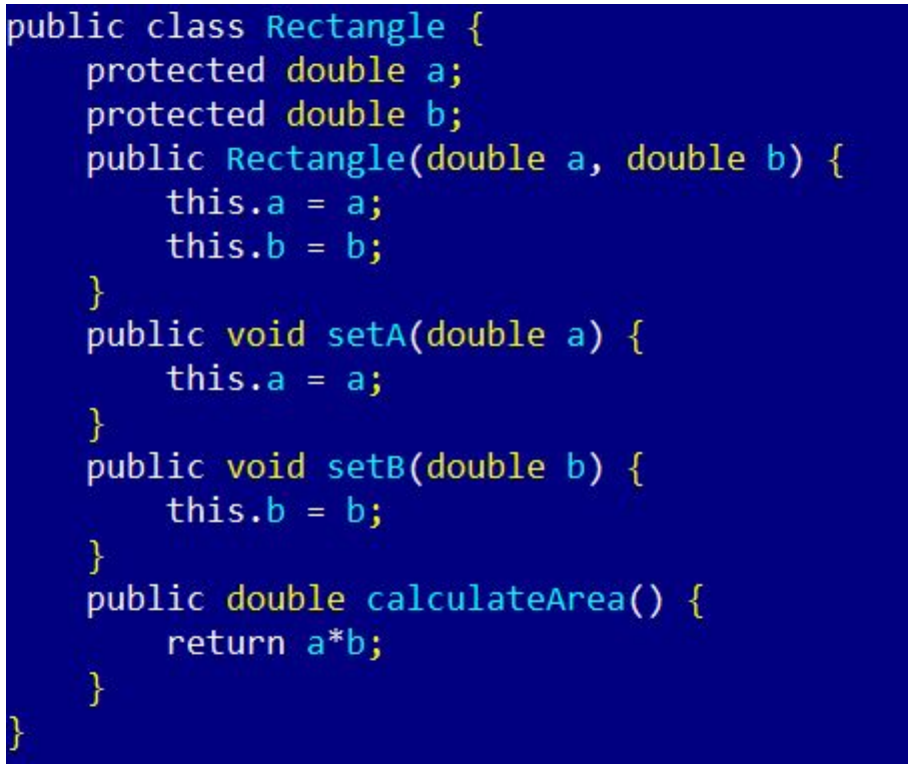


```java
public class Circle implements Shape {
     public double radius;
     public double calculateArea() {
           return (22 / 7) * radius * radius;
     }
}
```

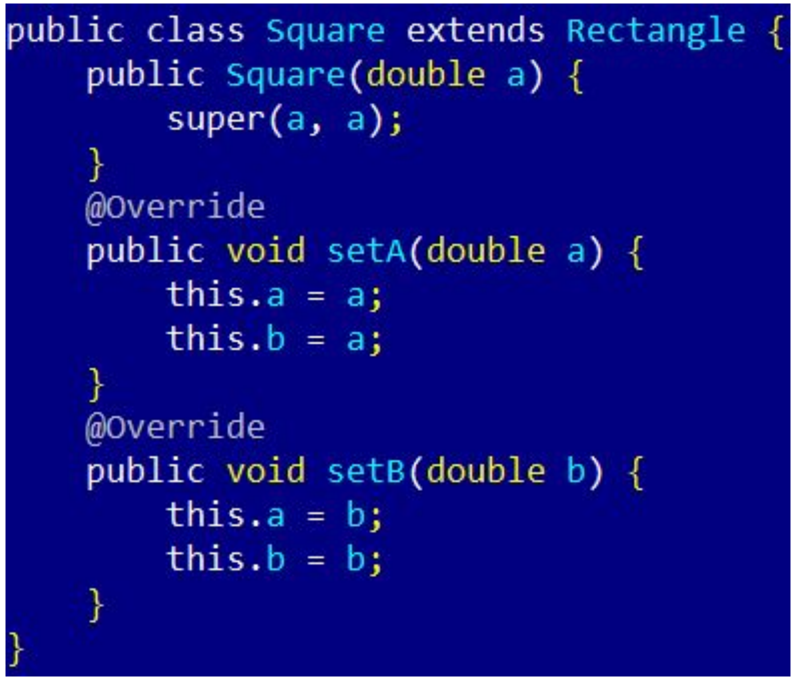
# Liskov's Substitution Principle (LSP)

- The Liskov's Substitution principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behaviour.
- **Derived or child classes must be substitutable for their base or parent classes**
- Liskov's notion of a behavioural subtype defines a notion of substitutability for objects; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

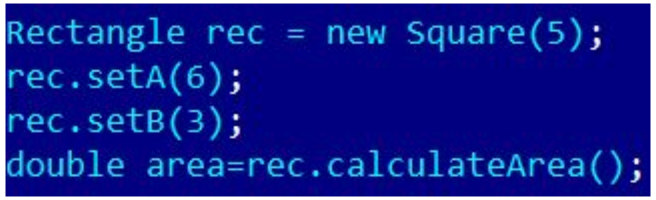
# LSP Square <- Rectangle



```
public class Rectangle {
    protected double a;
    protected double b;
    public Rectangle(double a, double b) {
        this.a = a;
        this.b = b;
    }
    public void setA(double a) {
        this.a = a;
    }
    public void setB(double b) {
        this.b = b;
    }
    public double calculateArea() {
        return a*b;
    }
}
```



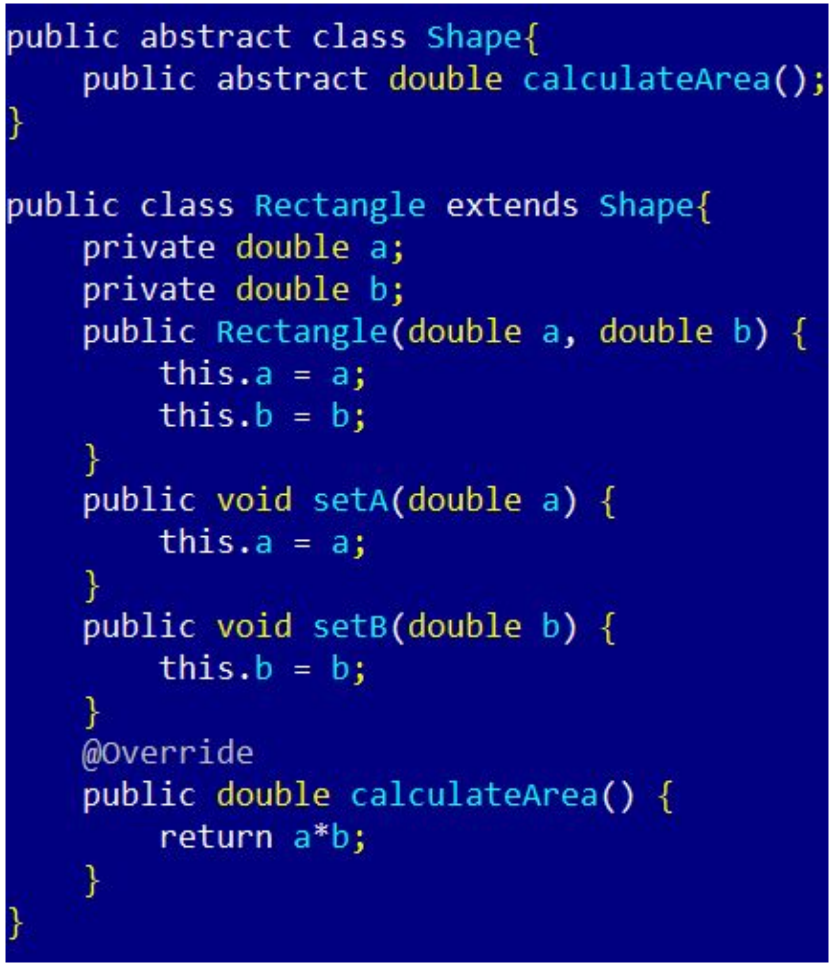
```
public class Square extends Rectangle {
    public Square(double a) {
        super(a, a);
    }
    @Override
    public void setA(double a) {
        this.a = a;
        this.b = a;
    }
    @Override
    public void setB(double b) {
        this.a = b;
        this.b = b;
    }
}
```



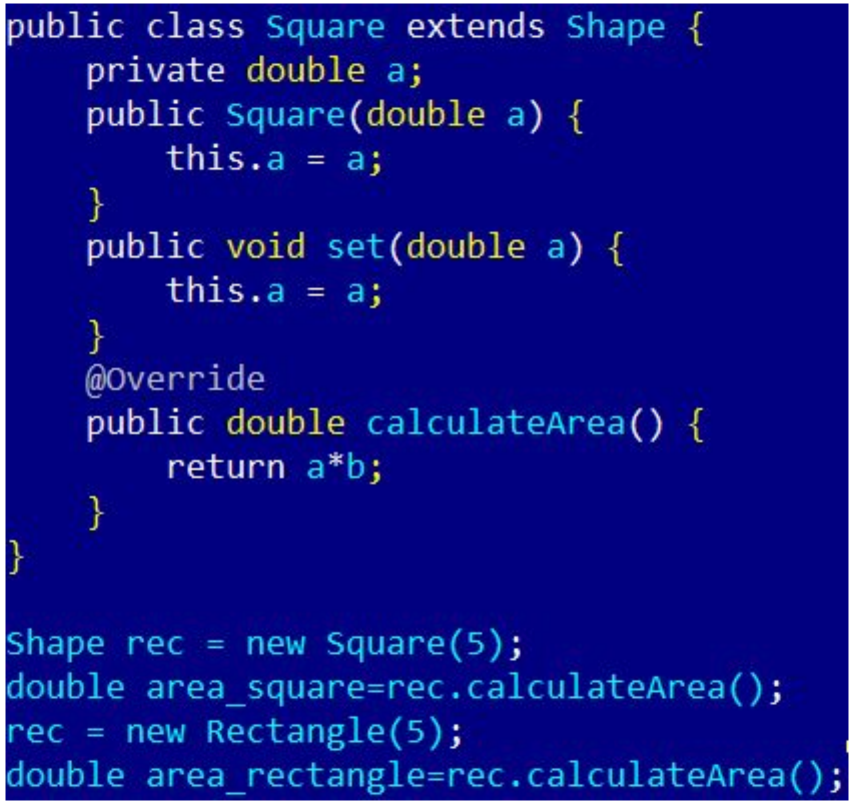
```
Rectangle rec = new Square(5);
rec.setA(6);
rec.setB(3);
double area=rec.calculateArea();
```

**May be inherit Rectangle from Square?**

# LSP Solution



```java
public abstract class Shape{
    public abstract double calculateArea();
}

public class Rectangle extends Shape{
    private double a;
    private double b;
    public Rectangle(double a, double b) {
        this.a = a;
        this.b = b;
    }
    public void setA(double a) {
        this.a = a;
    }
    public void setB(double b) {
        this.b = b;
    }
    @Override
    public double calculateArea() {
        return a*b;
    }
}
```

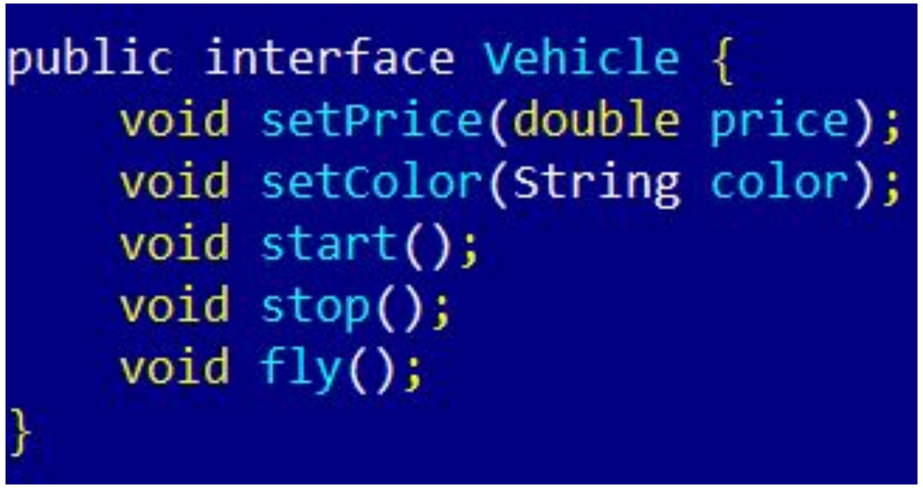


```java
public class Square extends Shape {
    private double a;
    public Square(double a) {
        this.a = a;
    }
    public void set(double a) {
        this.a = a;
    }
    @Override
    public double calculateArea() {
        return a*b;
    }
}

Shape rec = new Square(5);
double area_square=rec.calculateArea();
rec = new Rectangle(5);
double area_rectangle=rec.calculateArea();
```

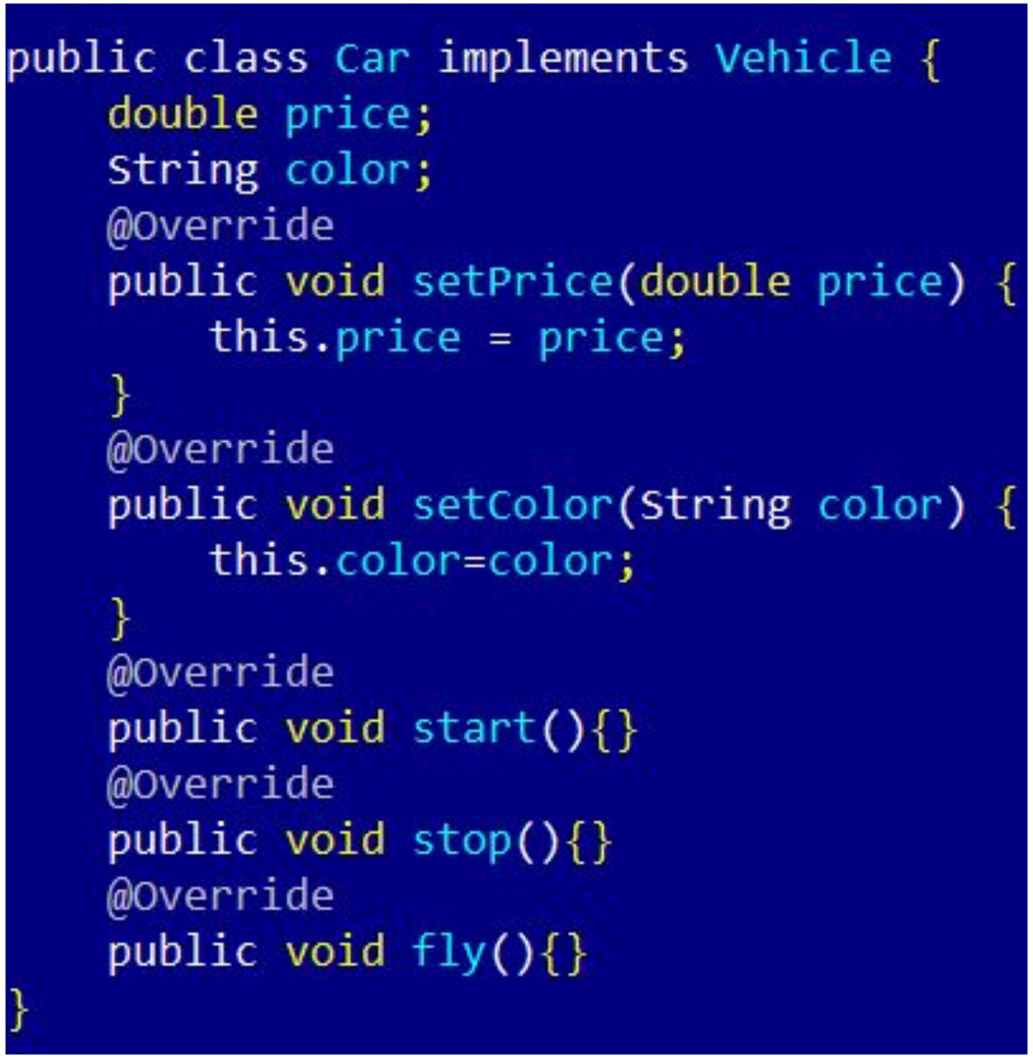
## Interface Segregation Principle (ISP)

- The Interface Segregation principle states that no client should be forced to depend on methods it does not use. Interfaces that are very large should be splitted into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
- **Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program**
- You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

# Interface Segregation Principle (ISP)

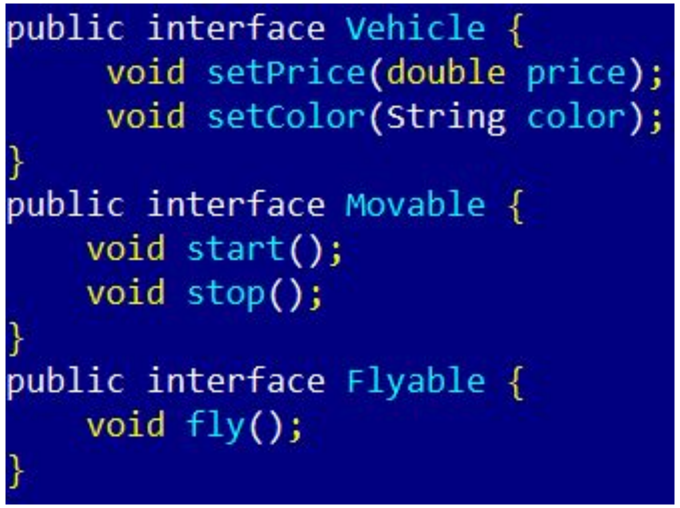


```java
public interface Vehicle {
    void setPrice(double price);
    void setColor(String color);
    void start();
    void stop();
    void fly();
}
```

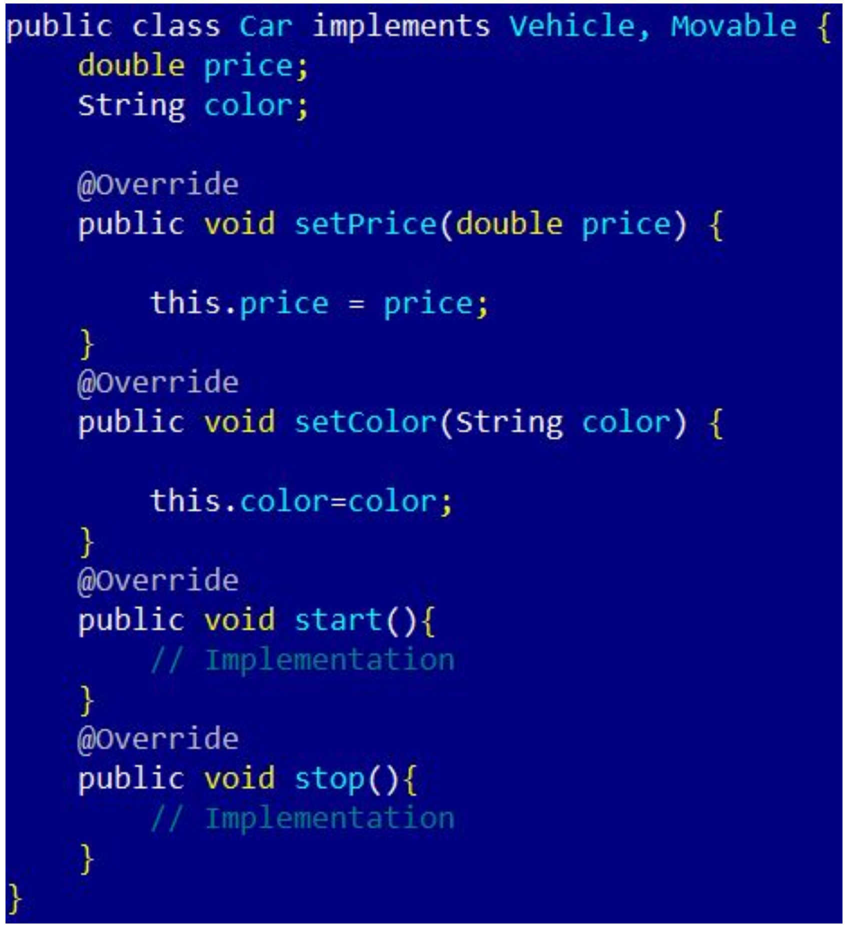


```java
public class Car implements Vehicle {
    double price;
    String color;
    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void start(){}
    @Override
    public void stop(){}
    @Override
    public void fly(){}
}
```

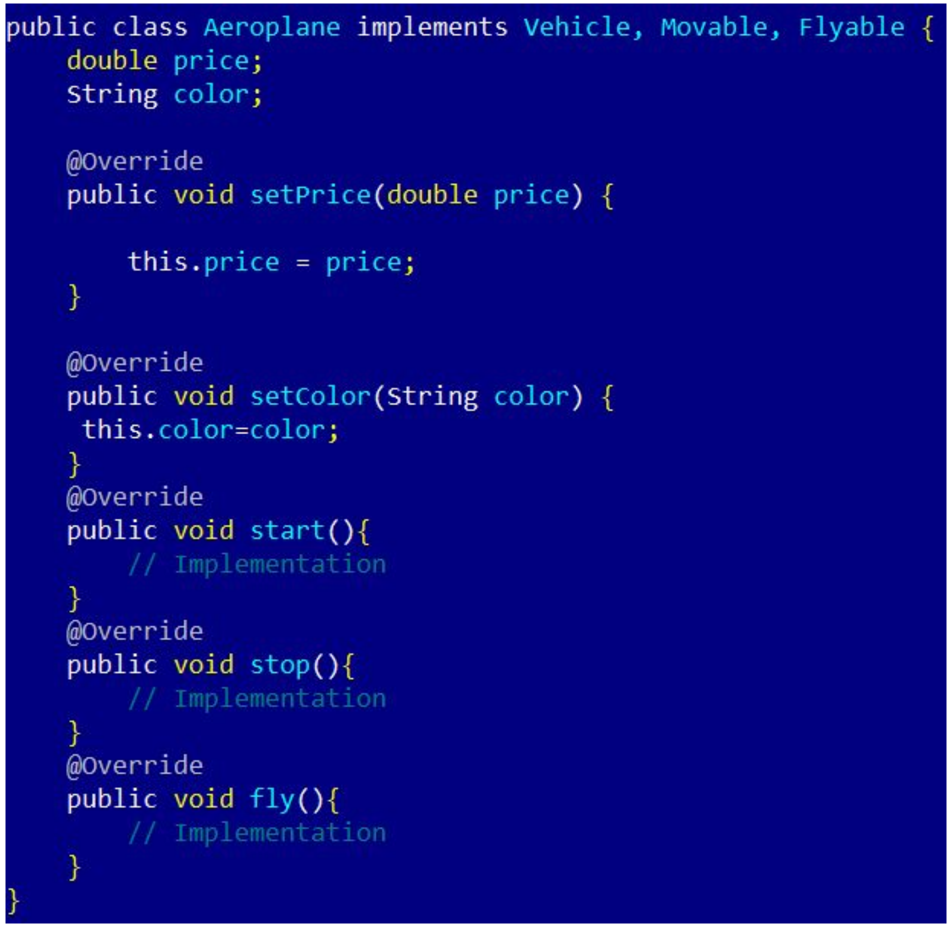
## Interface Segregation Principle (ISP)



```java
public interface Vehicle {
     void setPrice(double price);
     void setColor(String color);
}
public interface Movable {
    void start();
    void stop();
}
public interface Flyable {
    void fly();
}
```



```java
public class Car implements Vehicle, Movable {
    double price;
    String color;

    @Override
    public void setPrice(double price) {

        this.price = price;
    }
    @Override
    public void setColor(String color) {

        this.color=color;
    }
    @Override
    public void start(){
        // Implementation
    }
    @Override
    public void stop(){
        // Implementation
    }
}
```

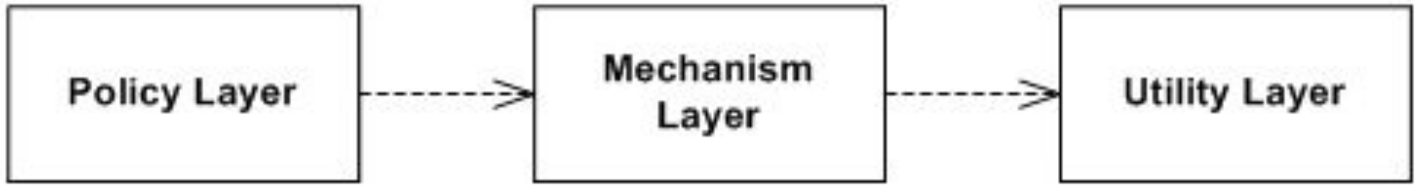


```java
public class Aeroplane implements Vehicle, Movable, Flyable {
    double price;
    String color;

    @Override
    public void setPrice(double price) {

        this.price = price;
    }

    @Override
    public void setColor(String color) {
     this.color=color;
    }
    @Override
    public void start(){
        // Implementation
    }
    @Override
    public void stop(){
        // Implementation
    }
    @Override
    public void fly(){
        // Implementation
    }
}
```

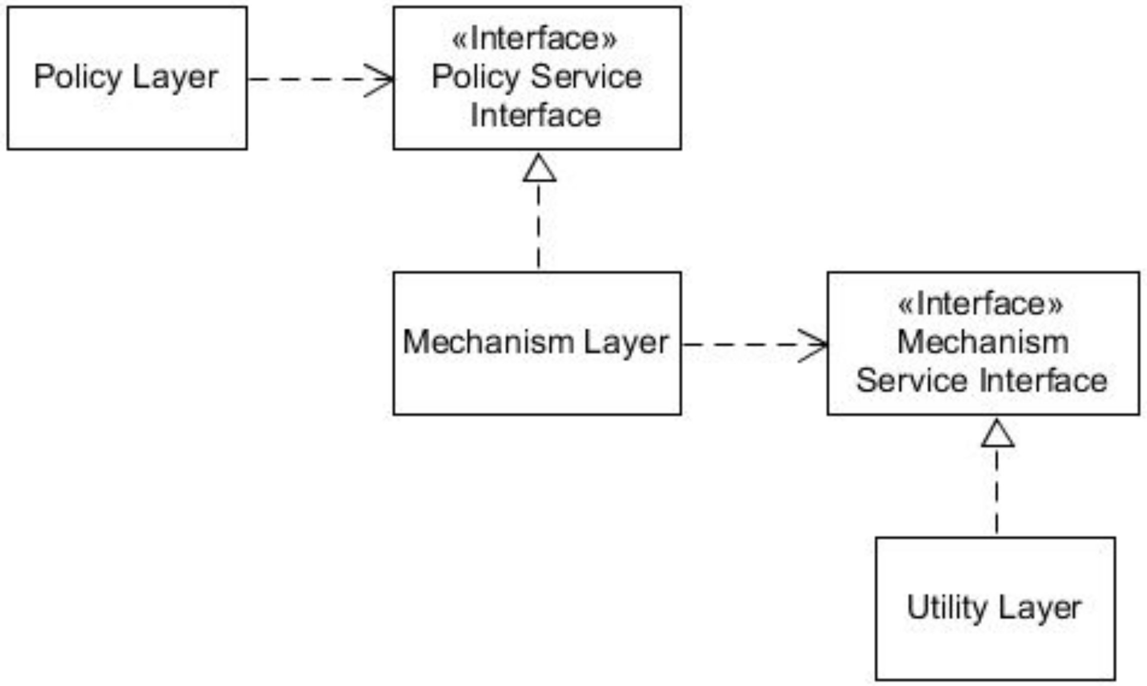
# Dependency Inversion Principle (DIP)

- The Dependency Inversion principle states that high-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features.
- **High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).**
- **Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.**
- The main motive of this principle is decoupling the dependencies so if class A changes the class B doesn’t need to care or know about the changes.

# Dependency Inversion Principle (DIP)



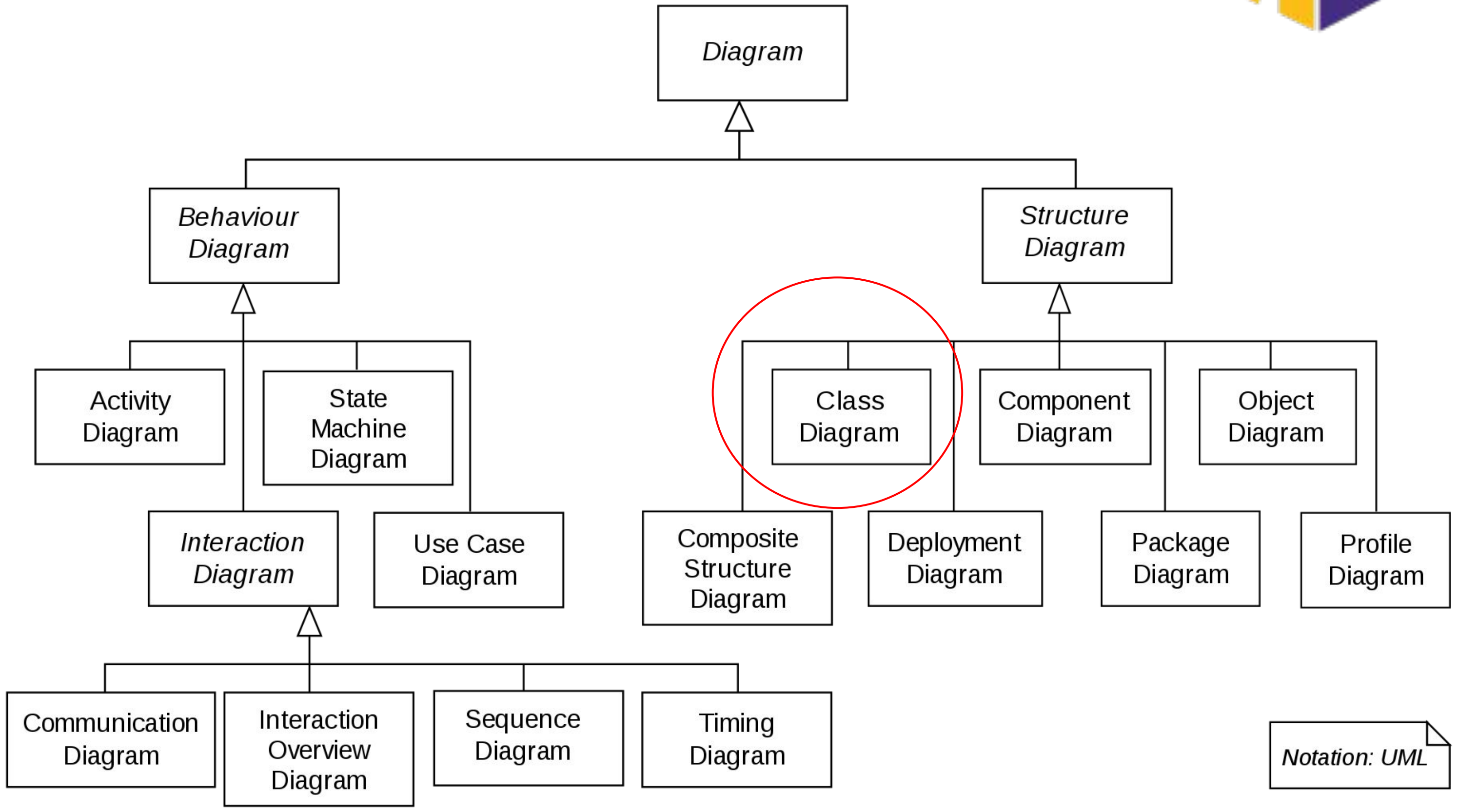
Wrong approach


Policy Layer
«Interface» Policy Service Interface
Mechanism Layer
«Interface» Mechanism Service Interface
Utility Layer

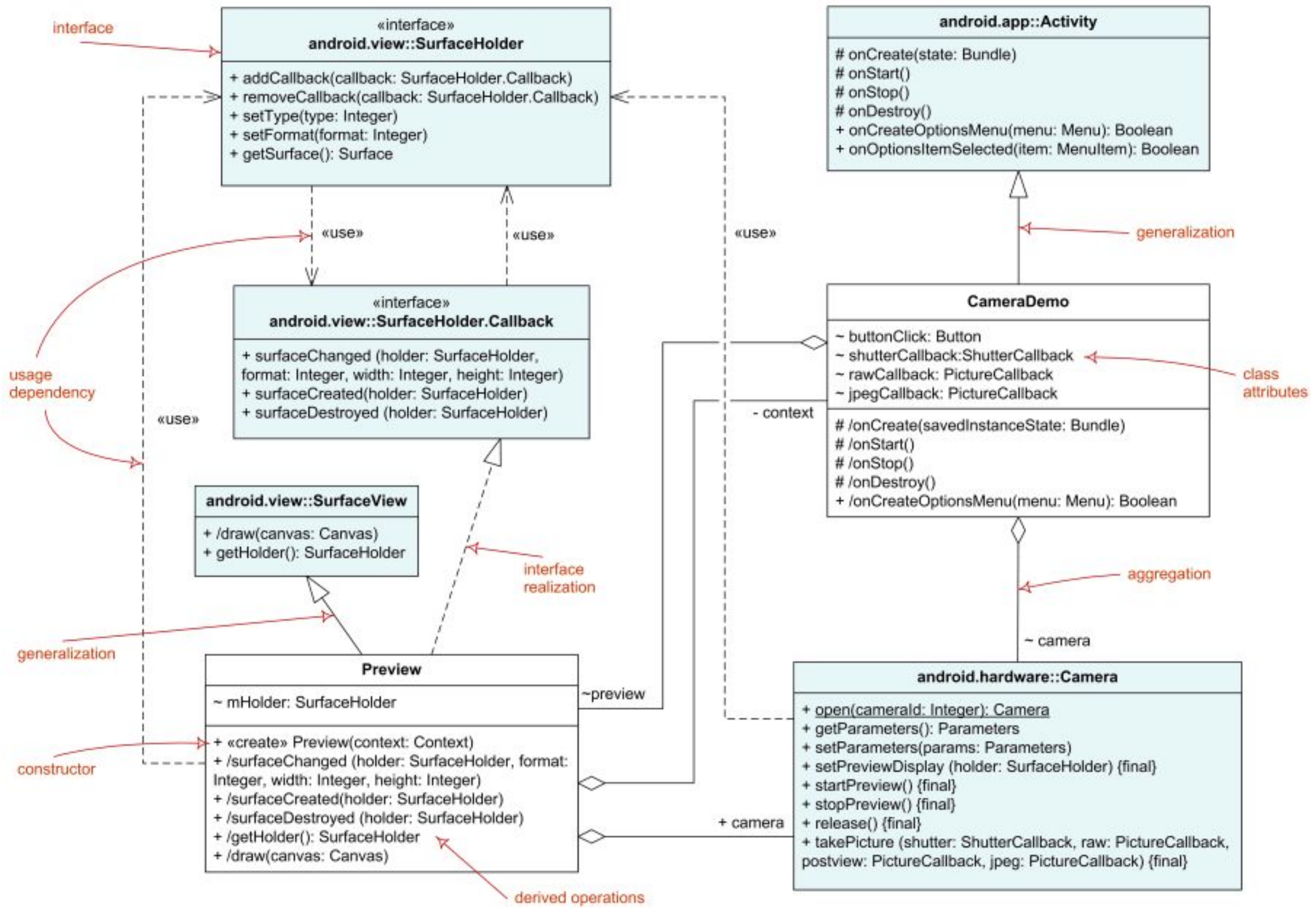
DIP approach

# UML

is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system

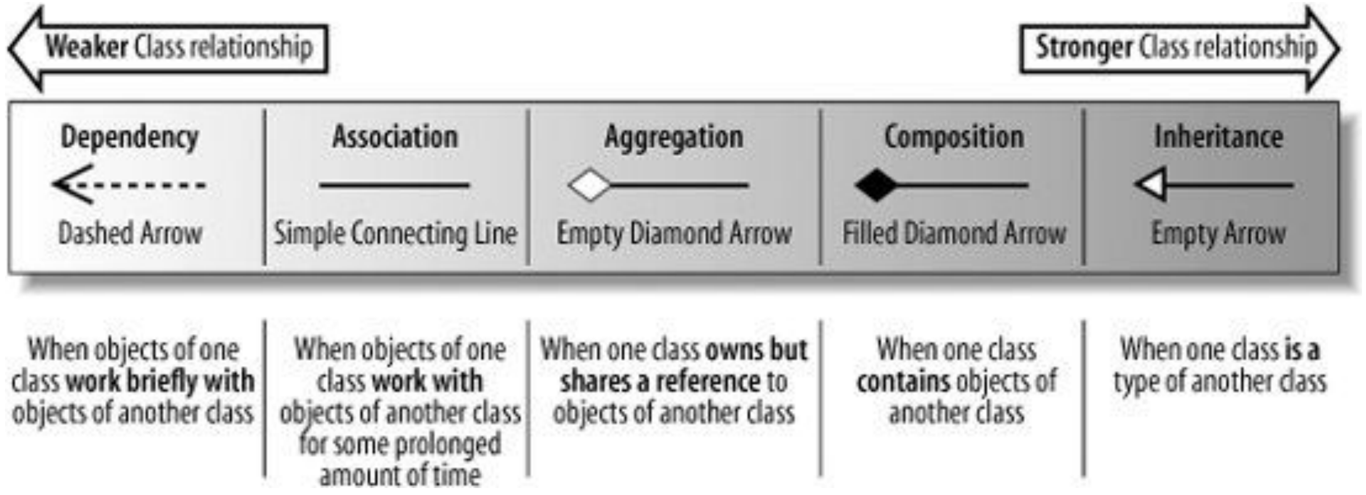


# UML Class Diagram

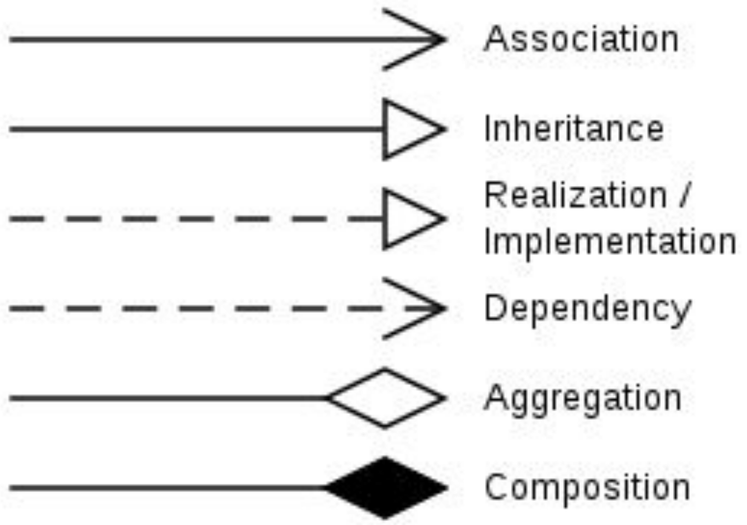


interface
«interface»
android.view::SurfaceHolder
+ addCallback(callback: SurfaceHolder.Callback)
+ removeCallback(callback: SurfaceHolder.Callback)
+ setType(type: Integer)
+ setFormat(format: Integer)
+ getSurface(): Surface
«use»
«use»
«use»
usage dependency
«use»
«interface»
android.view::SurfaceHolder.Callback
+ surfaceChanged (holder: SurfaceHolder, format: Integer, width: Integer, height: Integer)
+ surfaceCreated(holder: SurfaceHolder)
+ surfaceDestroyed (holder: SurfaceHolder)
android.view::SurfaceView
+ /draw(canvas: Canvas)
+ getHolder(): SurfaceHolder
interface realization
generalization
Preview
~ mHolder: SurfaceHolder
+ «create» Preview(context: Context)
+ /surfaceChanged (holder: SurfaceHolder, format: Integer, width: Integer, height: Integer)
+ /surfaceCreated(holder: SurfaceHolder)
+ /surfaceDestroyed (holder: SurfaceHolder)
+ /getHolder(): SurfaceHolder
+ /draw(canvas: Canvas)
constructor
derived operations
~preview
- context
+ camera
android.app::Activity
# onCreate(state: Bundle)
# onStart()
# onStop()
# onDestroy()
+ onCreateOptionsMenu(menu: Menu): Boolean
+ onOptionsItemSelected(item: MenuItem): Boolean
generalization
CameraDemo
~ buttonClick: Button
~ shutterCallback:ShutterCallback
~ rawCallback: PictureCallback
~ jpegCallback: PictureCallback
# /onCreate(savedInstanceState: Bundle)
# /onStart()
# /onStop()
# /onDestroy()
+ /onCreateOptionsMenu(menu: Menu): Boolean
class attributes
aggregation
~ camera
android.hardware::Camera
+ open(cameraId: Integer): Camera
+ getParameters(): Parameters
+ setParameters(params: Parameters)
+ setPreviewDisplay (holder: SurfaceHolder) {final}
+ startPreview() {final}
+ stopPreview() {final}
+ release() {final}
+ takePicture (shutter: ShutterCallback, raw: PictureCallback, postview: PictureCallback, jpeg: PictureCallback) {final}

# Relationships

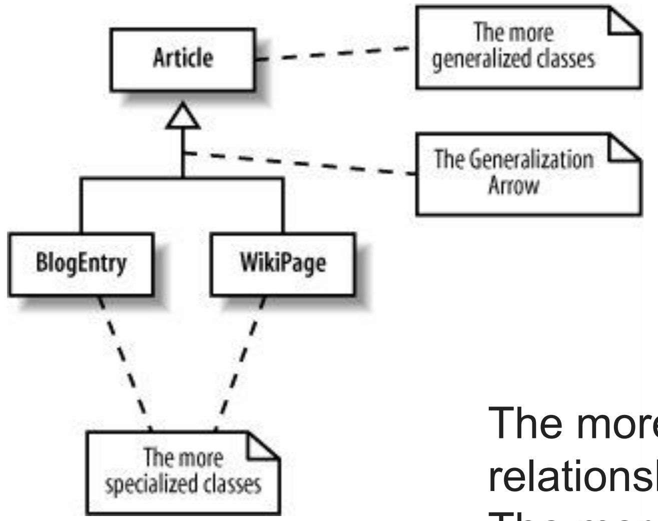


**Weaker** Class relationship
**Stronger** Class relationship
**Dependency**
Dashed Arrow
**Association**
Simple Connecting Line
**Aggregation**
Empty Diamond Arrow
**Composition**
Filled Diamond Arrow
**Inheritance**
Empty Arrow
When objects of one class **work briefly with** objects of another class
When objects of one class **work with** objects of another class for some prolonged amount of time
When one class **owns but shares a reference** to objects of another class
When one class **contains** objects of another class
When one class **is a** type of another class



# Generalization (Otherwise Known as Inheritance)

Generalization and inheritance are used to describe a class that is a type of another class.

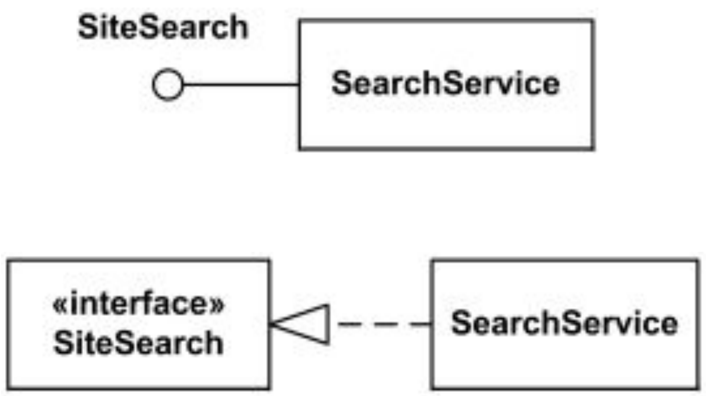


Showing that a BlogEntry and WikiPage are both types of Article

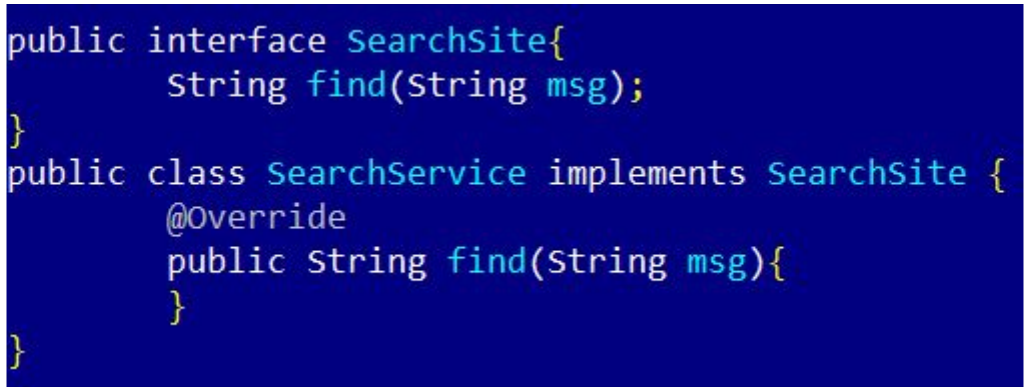
The more generalized class that is inherited fromat the arrow end of the generalization relationship, Article in this caseis often referred to as the parent, base, or superclass. The more specialized classes that do the inheritingBlogEntry and WikiPage in this caseare often referred to as the children or derived classes. The specialized class inherits all of the attributes and methods that are declared in the generalized class and may add operations and attributes that are only applicable in specialized cases.

# Realization

Realization is a relationship between two model elements, in which one model element (the client) realizes (implements or executes) the behavior that the other model element (the supplier) specifies.



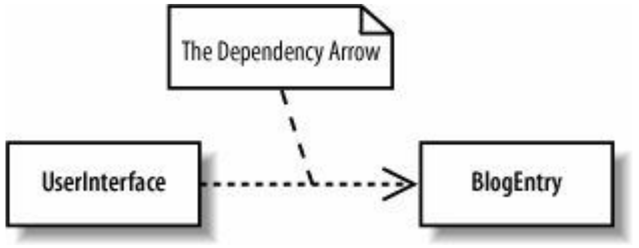
Interface SiteSearch is realized (implemented) by SearchService



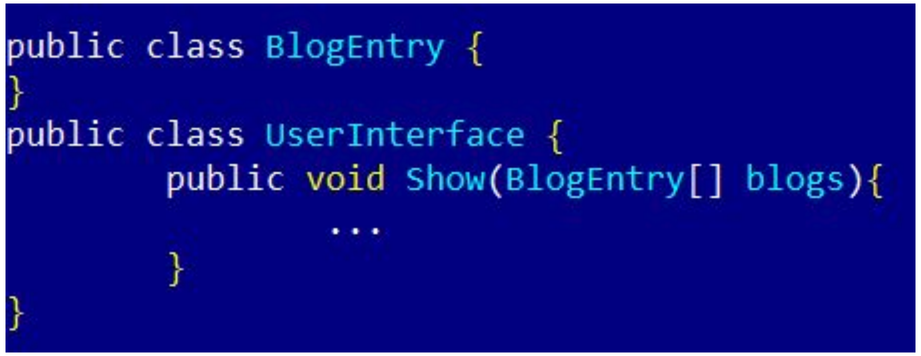
```
public interface SearchSite{
        String find(String msg);
}
public class SearchService implements SearchSite {
        @Override
        public String find(String msg){
        }
}
```

# Dependency

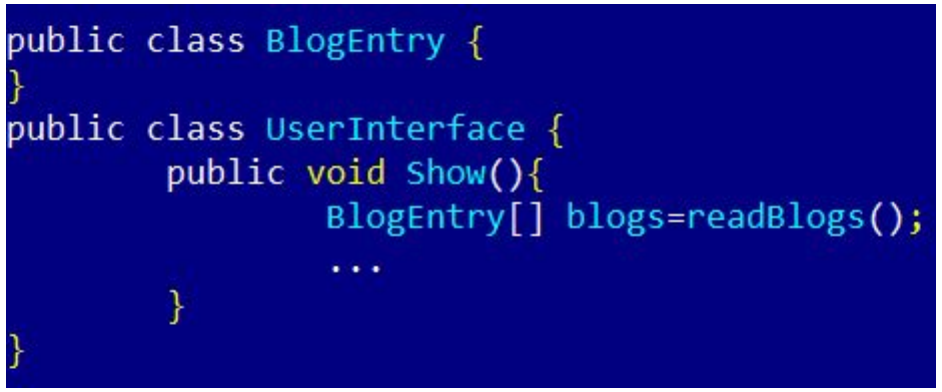
A dependency between two classes declares that a class needs to know about another class to use objects of that class.



The UserInterface is dependent on the BlogEntry class because it will need to read the contents of a blog's entries to display them to the user



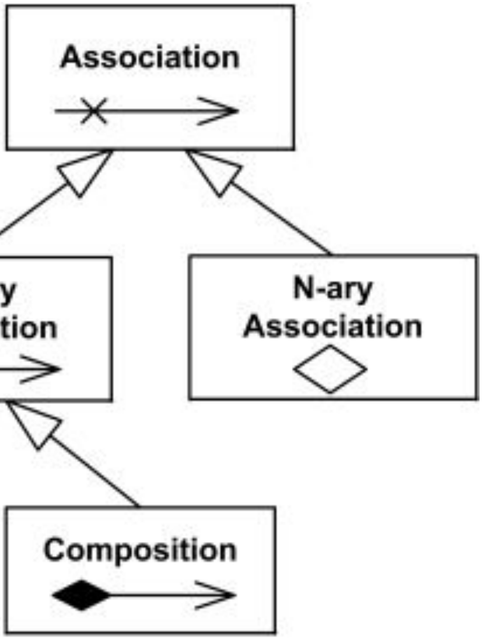
```
public class BlogEntry {
}
public class UserInterface {
        public void Show(BlogEntry[] blogs){
                ...
        }
}
```

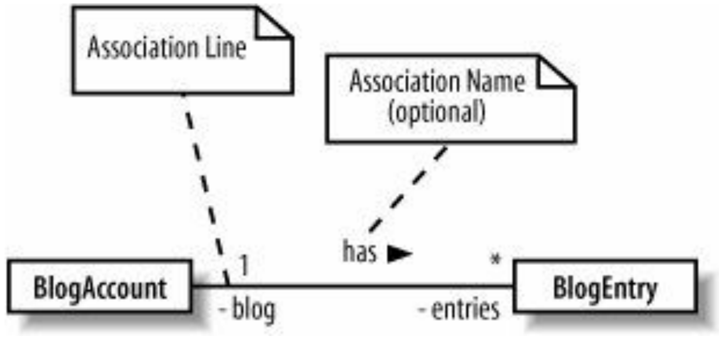


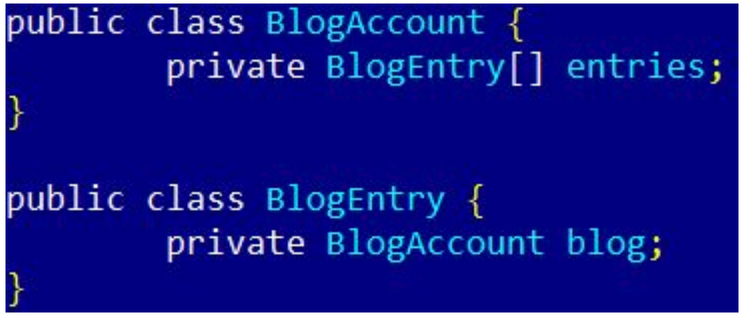
```
public class BlogEntry {
}
public class UserInterface {
        public void Show(){
                BlogEntry[] blogs=readBlogs();
                ...
        }
}
```

# Association

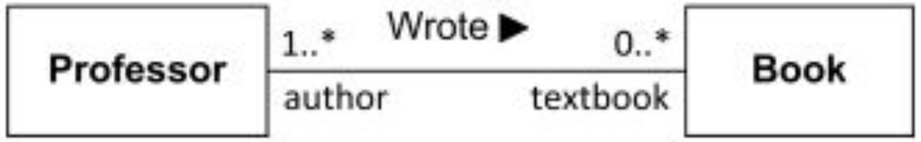
Although dependency simply allows one class to use objects of another class, association means that a class will actually contain a reference to an object, or objects, of the other class in the form of an attribute. If you find yourself saying that a class works with an object of another class, then the relationship between those classes is a great candidate for association rather than just a dependency

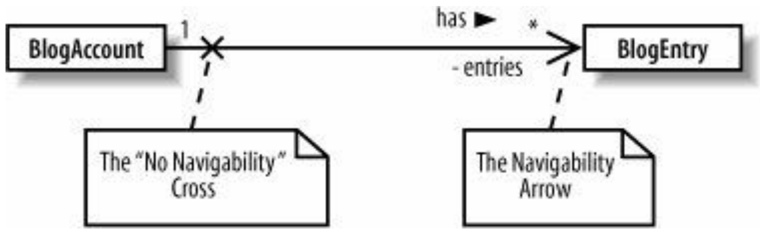


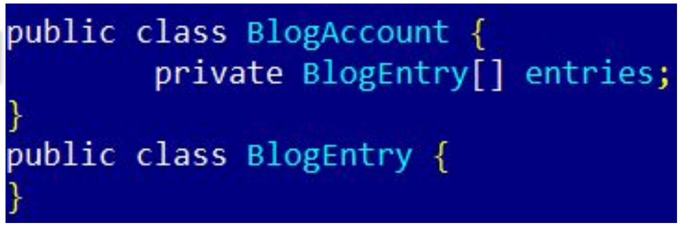




```
public class BlogAccount {
        private BlogEntry[] entries;
}

public class BlogEntry {
        private BlogAccount blog;
}
```



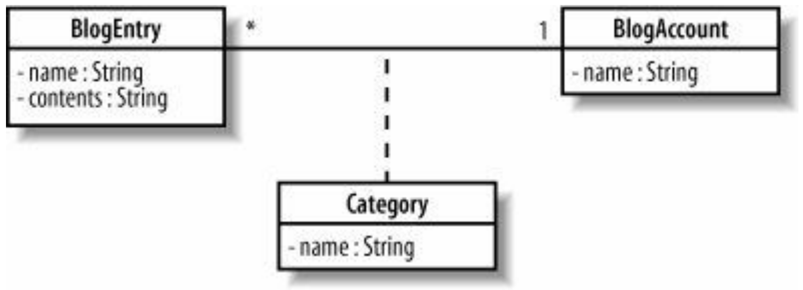




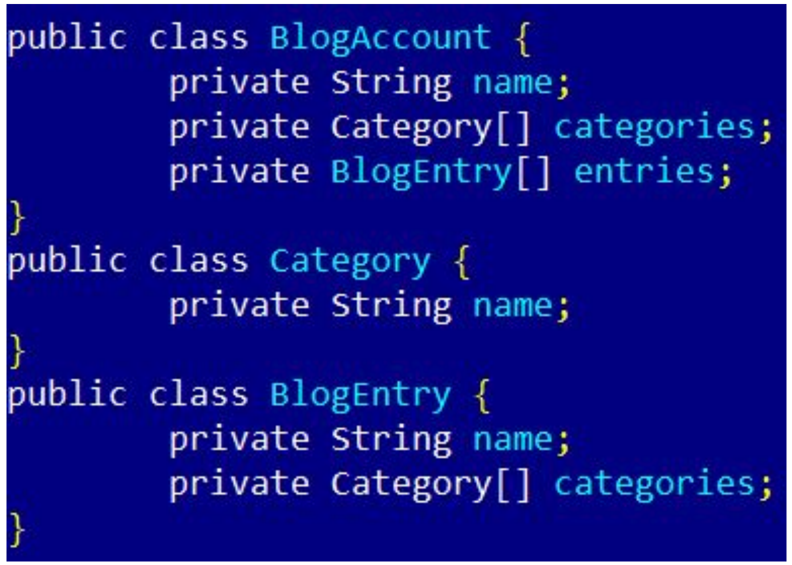
```
public class BlogAccount {
        private BlogEntry[] entries;
}
public class BlogEntry {
}
```

Association Wrote between Professor and Book with association ends author and textbook.

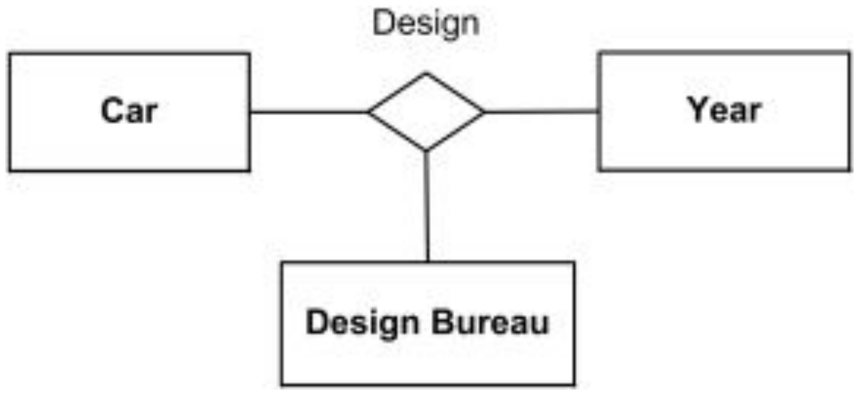
# N-ary Association



A BlogEntry is associated with an Author by virtue of the fact that it is associated with a particular BlogAccount

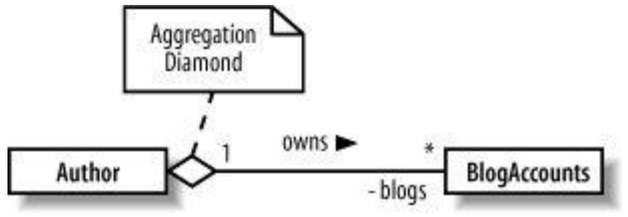


```
public class BlogAccount {
        private String name;
        private Category[] categories;
        private BlogEntry[] entries;
}
public class Category {
        private String name;
}
public class BlogEntry {
        private String name;
        private Category[] categories;
}
```



## Aggregation

Moving one step on from association, we encounter the aggregation relationship. Aggregation is really just a stronger version of association and is used to indicate that a class actually owns but may share objects of another class.



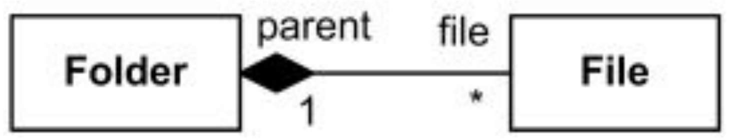
An aggregation relationship can show that an Author owns a collection of blogs

Where's the code? Actually, the Java code implementation for an aggregation relationship is exactly the same as the implementation for an association relationship; it results in the introduction of an attribute.

# Composition

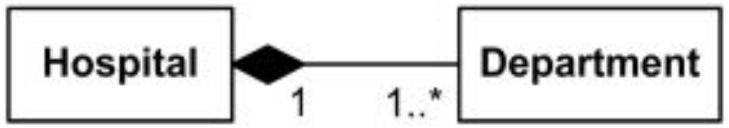
Composite aggregation (composition) is a "strong" form of aggregation with the following characteristics:

- it is binary association,
- it is a whole/part relationship,
- a part could be included in at most one composite (whole) at a time, and
- if a composite (whole) is deleted, all of its composite parts are "normally" deleted with it.



*Folder could contain many files, while each File has exactly one Folder parent.*

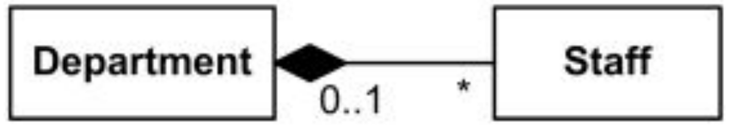
*If Folder is deleted, all contained Files are deleted as well.*



*Hospital has 1 or more Departments, and*

*each Department belongs to exactly one Hospital.*

*If Hospital is closed, so are all of its Departments.*



*Each Department has some Staff, and each Staff could be*

*a member of one Department (or none). If Department is closed,*

*its Staff is relieved (but excluding the "stand alone" Staff).*

Similar to aggregation, the Java code implementation for a composition relationship results only in the introduction of an attribute