CPT304 – Object-oriented Concepts

Software Engineering II

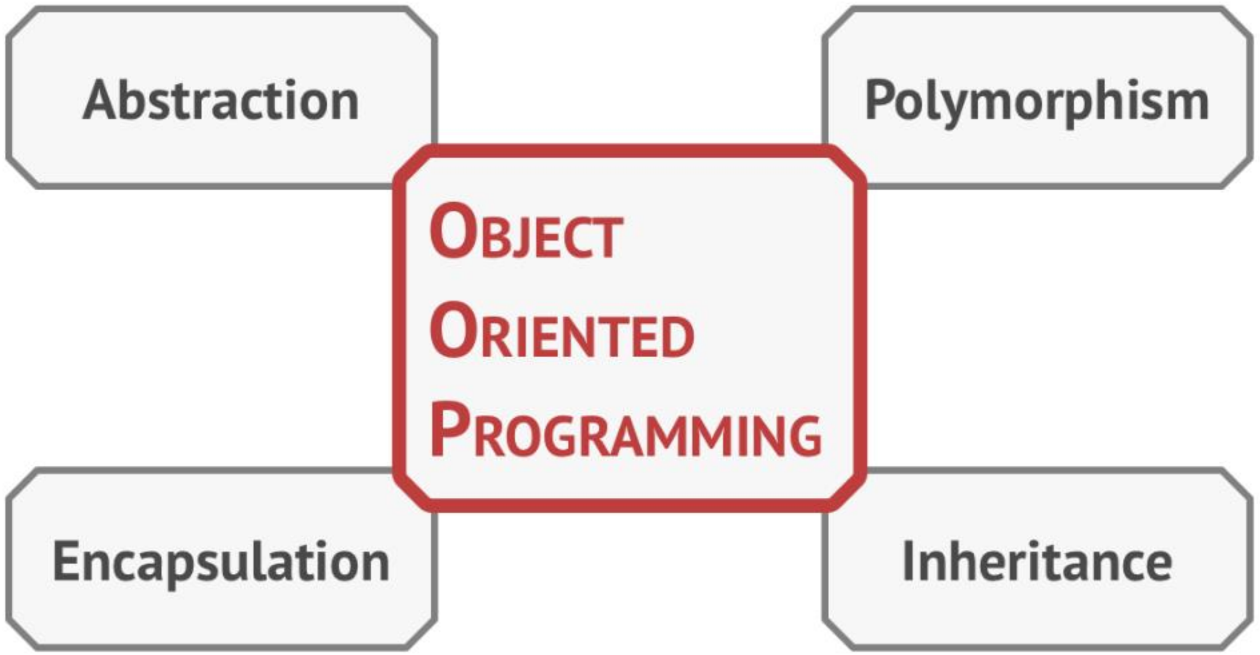
Soon Phei, Tin

Topic

- OOP Recap
- What is a good design
- SOLID Principles
- Loosen coupling using composition

A good design is the one that hides the inherent complexity and eliminates accidental complexity

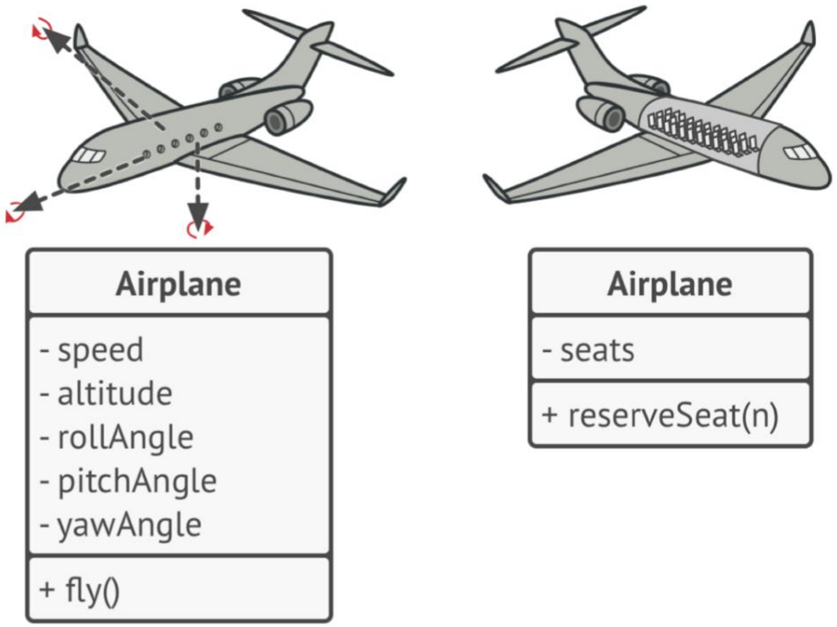
Four Pillar of OOP



OOP - Abstraction

- Objects only *model* attributes and behaviors of real objects in a specific context, ignoring the rest.

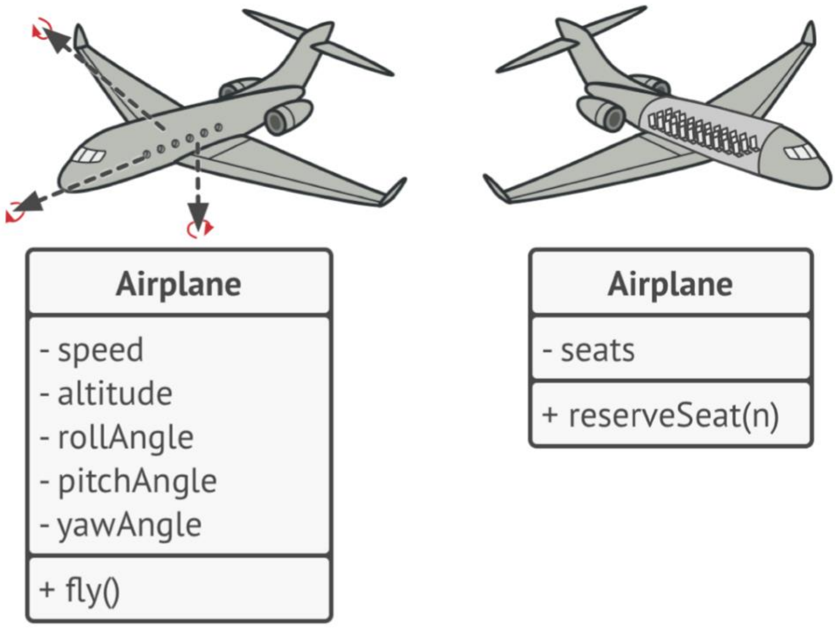
- *Abstraction* is a model of a real-world object or phenomenon, limited to a specific context, which represents all details relevant to this context with high accuracy and omits all the rest.



Different models of the same real-world object.

OOP - Abstraction

- Abstraction allows developers to interact with a system at a higher level without needing to understand the intricate details of its implementation



Different models of the same real-world object.

OOP - Encapsulation

- The ability of an object to hide parts of its state and behaviors from other objects, exposing only a limited interface to the rest of the program.
- The interface mechanism lets you define contracts of interaction between objects.

Interface with different meaning: -

1. Public part of an object
2. Keyword in some programming language
3. The mean of communication between human and computer

OOP - Encapsulation

- The class that bind to the contract must implement the methods defined in the contract.
- When class X is working with other classes that binds to a contract, class X has definite confident that the classes it is working with have implemented methods defined in the contract.

OOP – Encapsulation
Exercise

- How do you create a contract in Java?
- How do you create a Java class that bind to a contract?

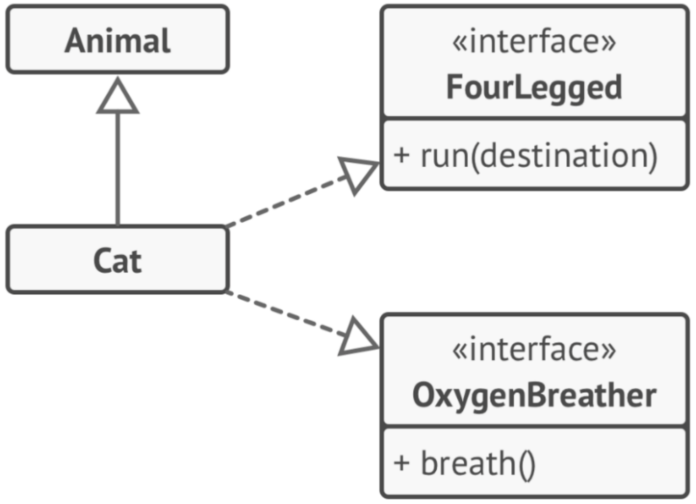
OOP - Inheritance

- The ability to build new classes on top of existing ones.
- The main benefit of inheritance is code reuse.
- Achieve code reuse by extending the existing class and put the extra functionality into a resulting subclass, which inherits fields and methods of the superclass.

OOP - Inheritance

- In most programming languages a subclass can extend only one superclass.
- On the other hand, any class can implement several interfaces at the same time.

- if a superclass implements an interface, subclasses inherits the implementation of the interface from its superclass.



UML diagram of extending a single class versus implementing multiple interfaces at the same time.

OOP - Polymorphism

- Allow a class to can takes different forms.
- The concept that you can access objects of different types through the same interface.
- Parent class reference is used to refer to a child class object.

Good Design

What is a good design?

- A software has a good design if the cost of changing it is minimum
- How to achieve?
  - Cohesion
    - The code is narrow and focus; It does only one thing and does it well.
  - Coupling
    - The degree of dependency in the code

What is a good design?

- The one that hides inherent complexity and eliminates the accidental complexity
  - Hides inherent complexity
    - Abstraction, encapsulation, inheritance
  - Eliminate accidental complexity
    - Simplification, reusability, design patterns

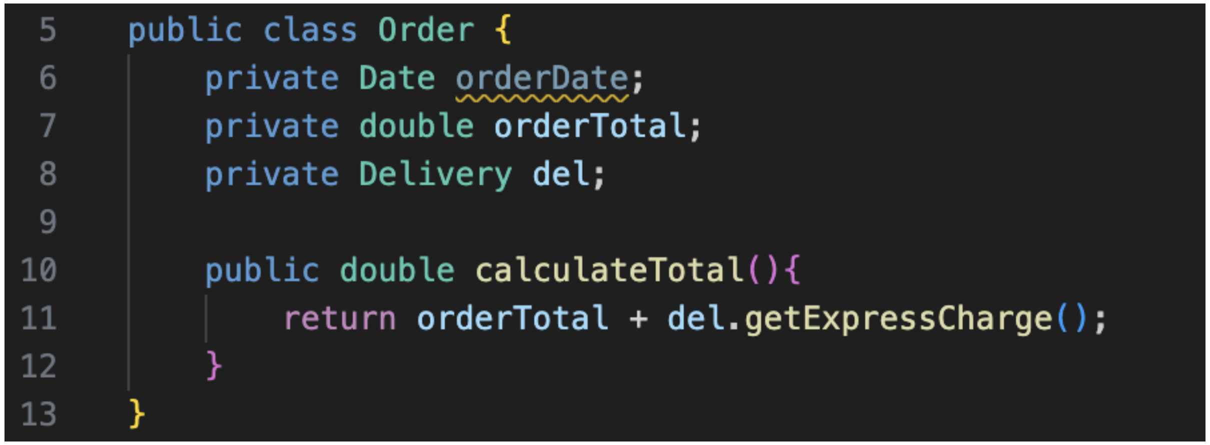
Highly cohesive code

- Code with similar/related (narrow and focus) functions stay together.
- For the software that evolve constantly, high cohesive code got to <u>change less frequently</u>

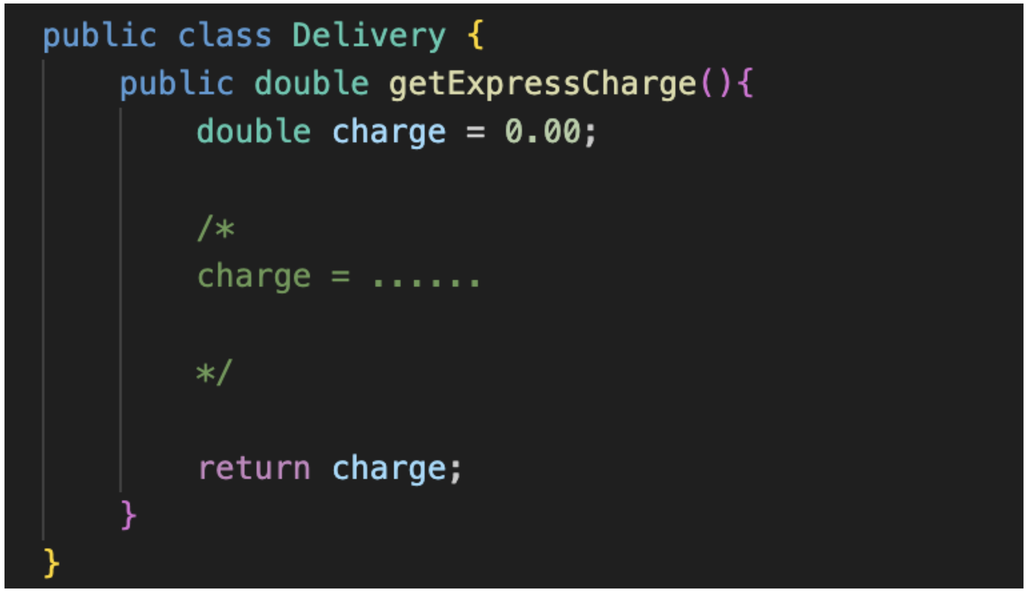
Loose coupling code

- Loose(low) coupling, instead of No coupling. Impossible to have no dependency.
- Make the coupling loose.
- Depending on a class is tight(high) coupling.
- Depending on an interface is loose coupling.

Tight coupling code

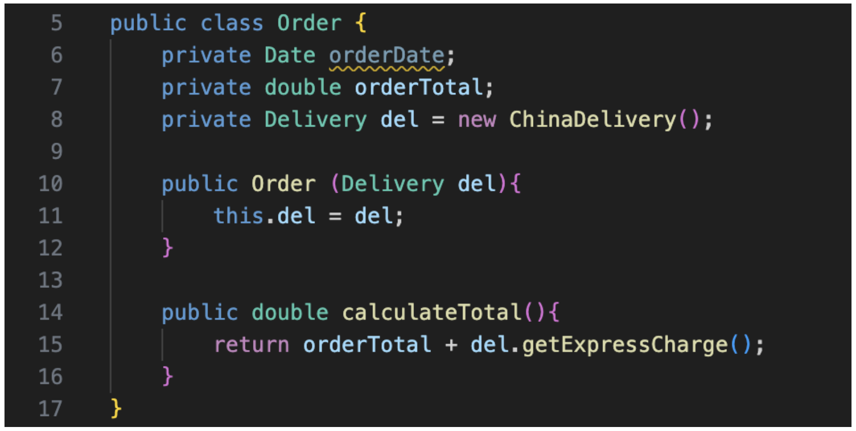


```java
public class Order {
    private Date orderDate;
    private double orderTotal;
    private Delivery del;

    public double calculateTotal(){
        return orderTotal + del.getExpressCharge();
    }
}
```

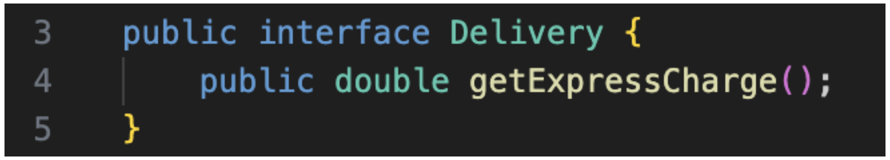


```java
public class Delivery {
    public double getExpressCharge(){
        double charge = 0.00;

        /*
        charge = ......

        */

        return charge;
    }
}
```

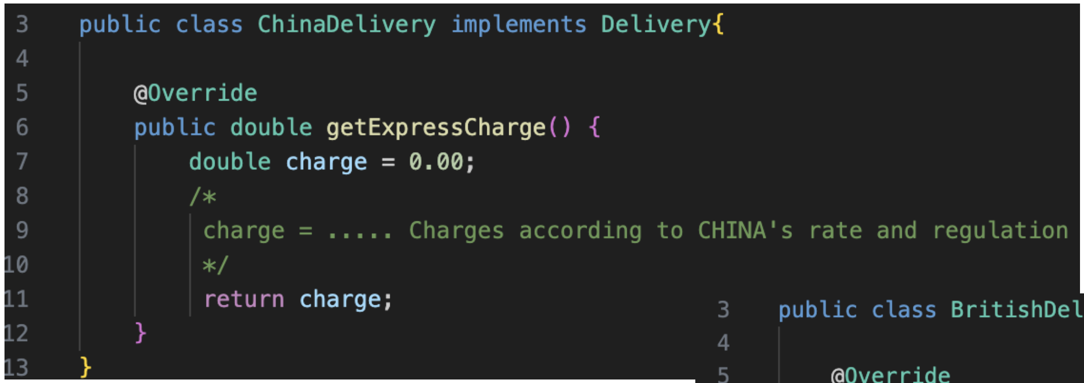
Loose coupling code



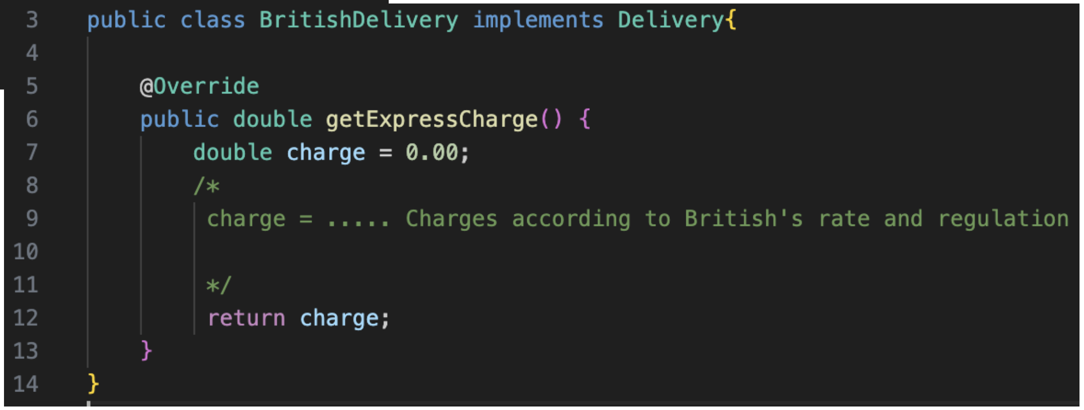
```java
public class Order {
    private Date orderDate;
    private double orderTotal;
    private Delivery del = new ChinaDelivery();

    public Order (Delivery del){
        this.del = del;
    }

    public double calculateTotal(){
        return orderTotal + del.getExpressCharge();
    }
}
```



```java
public interface Delivery {
    public double getExpressCharge();
}
```



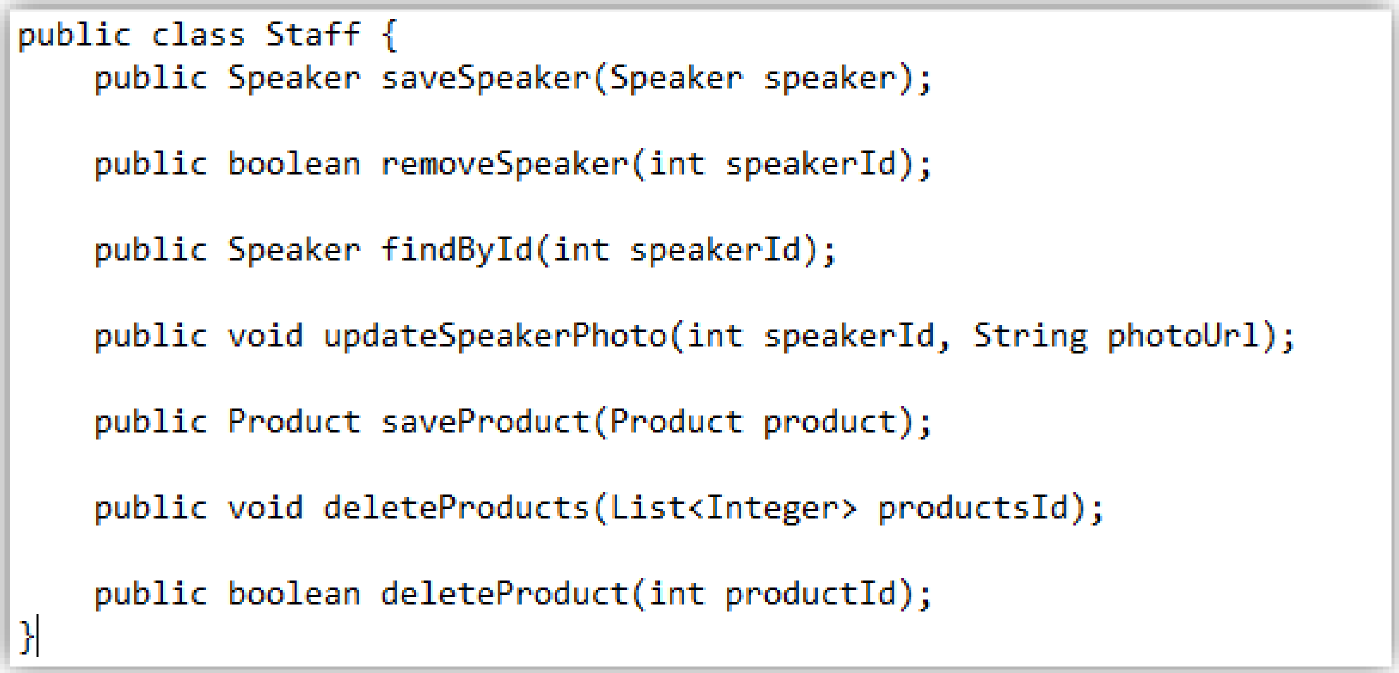
```java
public class ChinaDelivery implements Delivery{

    @Override
    public double getExpressCharge() {
        double charge = 0.00;
        /*
         charge = ..... Charges according to CHINA's rate and regulation
         */
         return charge;
    }
}
```



```java
public class BritishDelivery implements Delivery{

    @Override
    public double getExpressCharge() {
        double charge = 0.00;
        /*
         charge = ..... Charges according to British's rate and regulation

         */
         return charge;
    }
}
```

Exercise

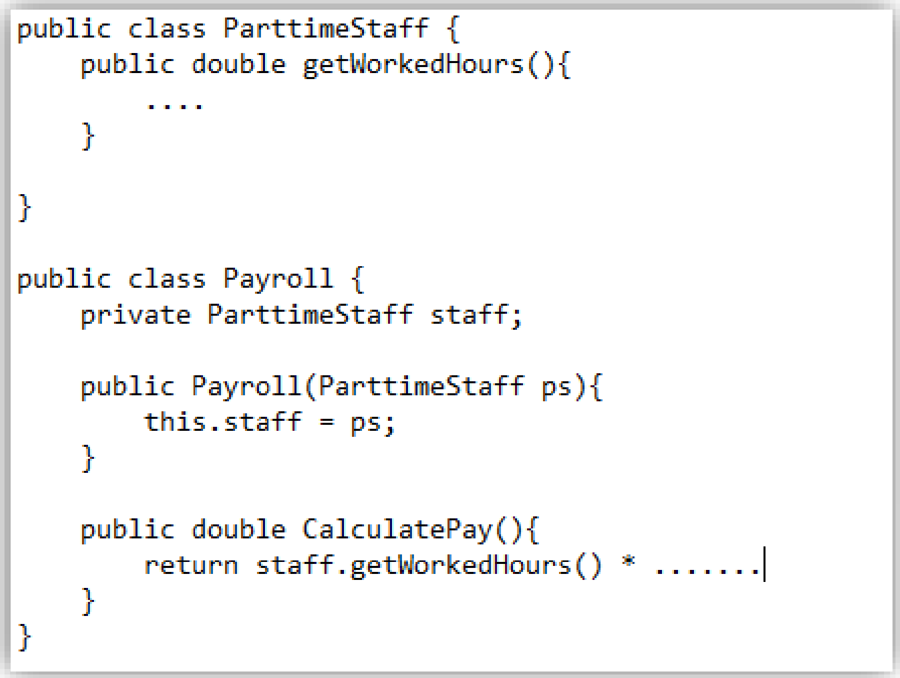
- Modify the following code to improve its cohesion



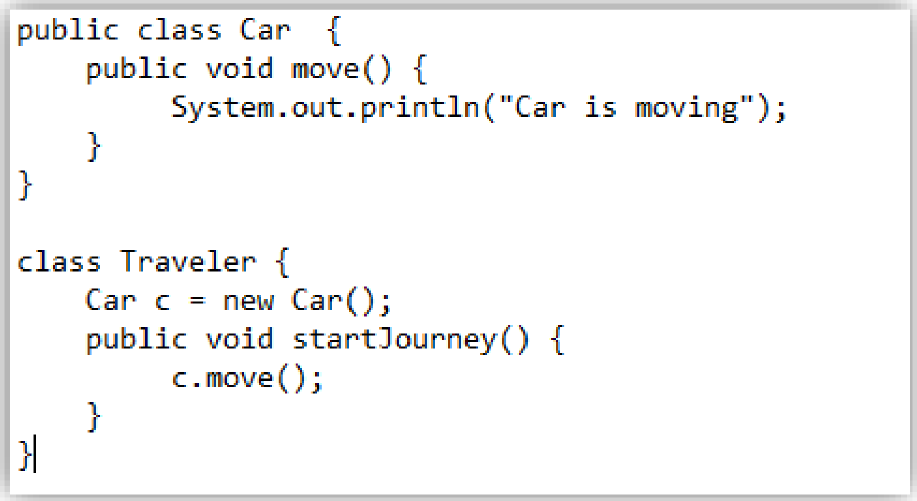
```java
public class Staff {
    public Speaker saveSpeaker(Speaker speaker);

    public boolean removeSpeaker(int speakerId);

    public Speaker findById(int speakerId);

    public void updateSpeakerPhoto(int speakerId, String photoUrl);

    public Product saveProduct(Product product);

    public void deleteProducts(List<Integer> productsId);

    public boolean deleteProduct(int productId);
}
```

Exercise

- Modify the following code to lower its coupling



```
public class ParttimeStaff {
    public double getWorkedHours(){
        ....
    }

}

public class Payroll {
    private ParttimeStaff staff;

    public Payroll(ParttimeStaff ps){
        this.staff = ps;
    }

    public double CalculatePay(){
        return staff.getWorkedHours() * .......
    }
}
```



```
public class Car  {
    public void move() {
        System.out.println("Car is moving");
    }
}

class Traveler {
    Car c = new Car();
    public void startJourney() {
        c.move();
    }
}
```

SOLID Principles

SOLID is a mnemonic for five design principles intended to make software designs more understandable, flexible and maintainable.

The cost of applying these principles into a program's architecture might be making it more complicated than it should be.

Striving for these principles is good, but always try to be pragmatic.

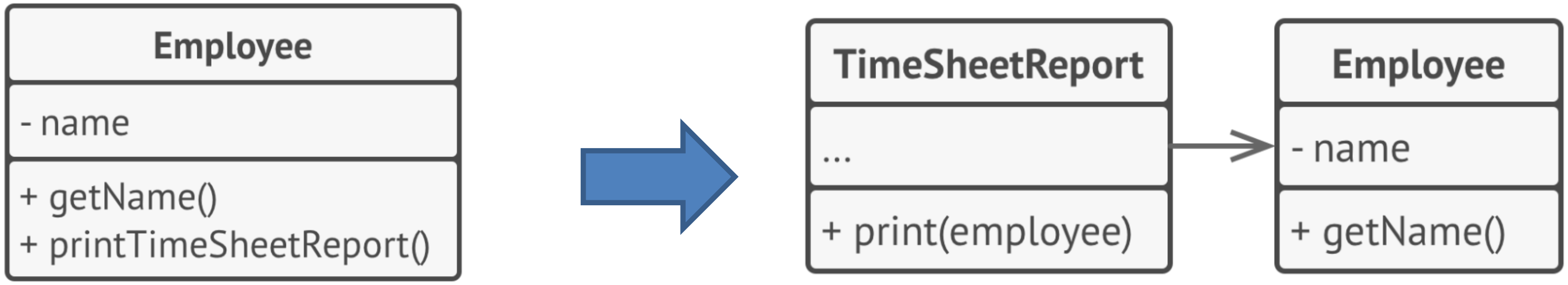
SOLID Principles

S – Single Responsibility Principle
O – Open/closed Principle
L – Liskov Substitute Principle
I – Interface Segregation Principle
D – Dependency inversion Principle

S – Single Responsibility Principle

- *A class should have one, and only one, reason to change.*
- *Make every class to have single responsibility, and make that responsibility entirely encapsulated by the class.*
- *The main goal of this principle is reducing complexity.*
- *A class with many responsibilities: -*
  - is more difficult to understand, making change to the code becomes risky
  - often clutter with codes
  - difficult to navigate and hard to find a specific code

Example 6



BEFORE: the class contains several different behaviors.

AFTER: the extra behavior is in its own class.

O – Open/Closed Principle

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

- *The main idea of this principle is to keep existing code from breaking when you implement new features.*

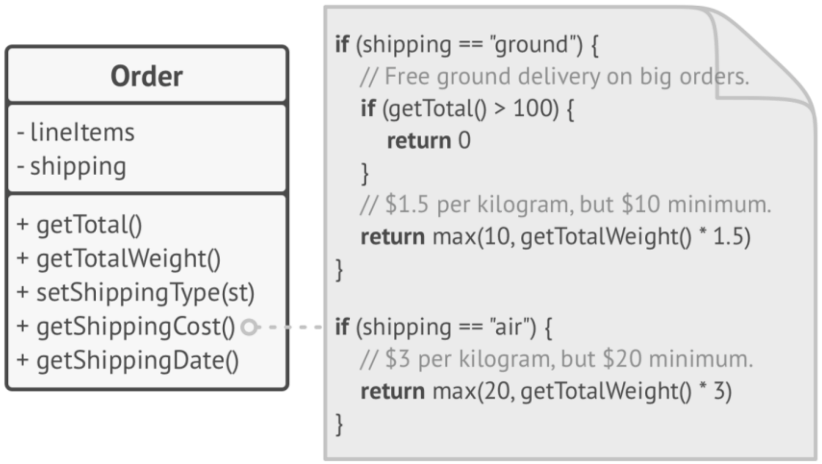
O – Open/Closed Principle

- *We can achieve Open/Closed Principle using inheritance. However, inheritance introduces tight coupling if the subclasses depend on implementation details of their parent class.*

- *Use interfaces instead of superclasses to allow different implementations without changing the code that uses them.*

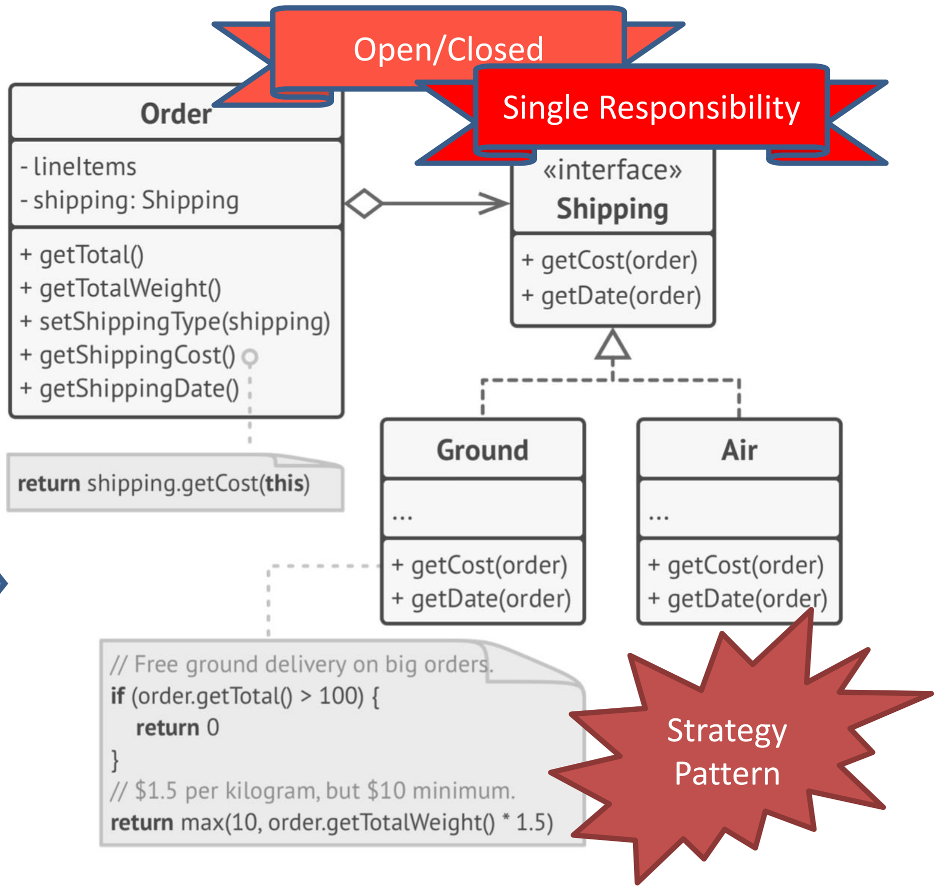
- *The interfaces are closed for modifications, and you can provide new implementations to extend the functionality of your software.*

Example 7



```
if (shipping == "ground") {
    // Free ground delivery on big orders.
    if (getTotal() > 100) {
        return 0
    }
    // $1.5 per kilogram, but $10 minimum.
    return max(10, getTotalWeight() * 1.5)
}

if (shipping == "air") {
    // $3 per kilogram, but $20 minimum.
    return max(20, getTotalWeight() * 3)
}
```

BEFORE: you have to change the `Order` class whenever you add a new shipping method to the app.



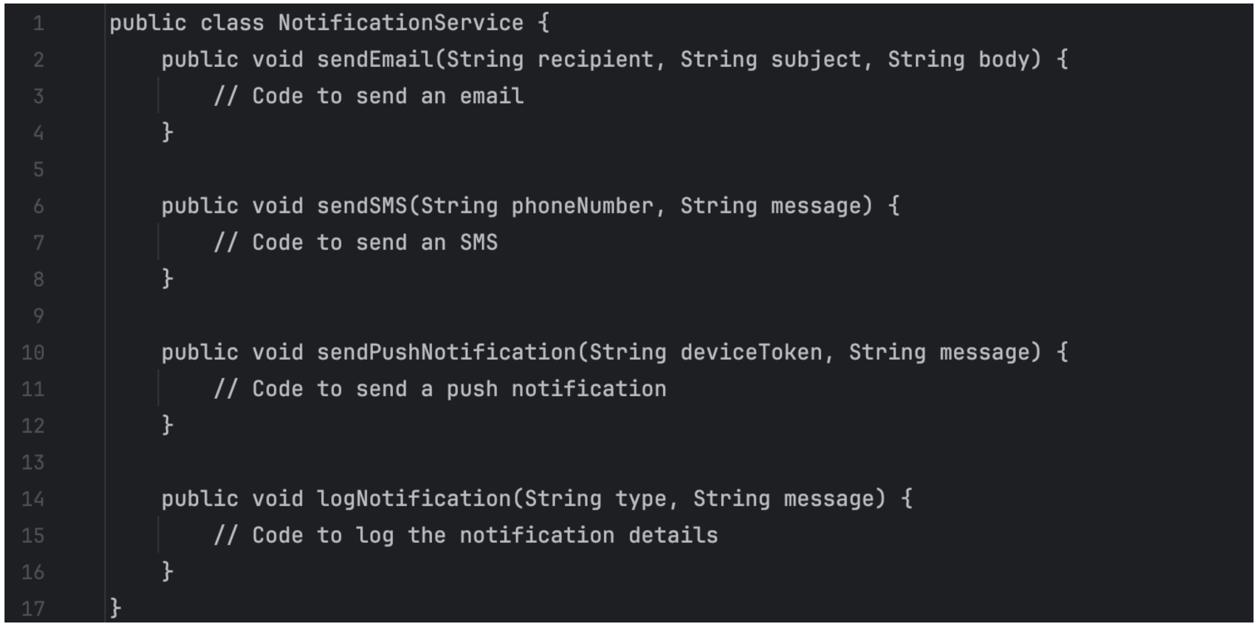
```
return shipping.getCost(this)
```

```
// Free ground delivery on big orders.
if (order.getTotal() > 100) {
    return 0
}
// $1.5 per kilogram, but $10 minimum.
return max(10, order.getTotalWeight() * 1.5)
```

AFTER: adding a new shipping method doesn't require changing existing classes.

Exercise

You are provided with the following class design for a simple notification system. The class below does not adhere to the SOLID principles, particularly the Single Responsibility Principle (SRP) and the Open/Closed Principle (OCP).



```java
public class NotificationService {
    public void sendEmail(String recipient, String subject, String body) {
        // Code to send an email
    }

    public void sendSMS(String phoneNumber, String message) {
        // Code to send an SMS
    }

    public void sendPushNotification(String deviceToken, String message) {
        // Code to send a push notification
    }

    public void logNotification(String type, String message) {
        // Code to log the notification details
    }
}
```

Exercise

Tasks:

- *Redesign the class structure to ensure that each class has only one reason to change, adhering to the Single Responsibility Principle.*
- *Modify your design to comply with the Open/Closed Principle, allowing the system to support new notification methods without modifying existing code.*

L – Liskov Substitution Principle

Inheritance should be used only for substitutability

If an object of B should be used anywhere an object of A is used, then use inheritance.

If an object of B should use an object of A, then use composition.

Inheritance demands more from a developer than composition does

Services of the derived class should require no more and promise no less than the corresponding services of the base class.

Why?

The user of a base class should be able to use an instance of a derived class without knowing the difference.

L – Liskov Substitution Principle

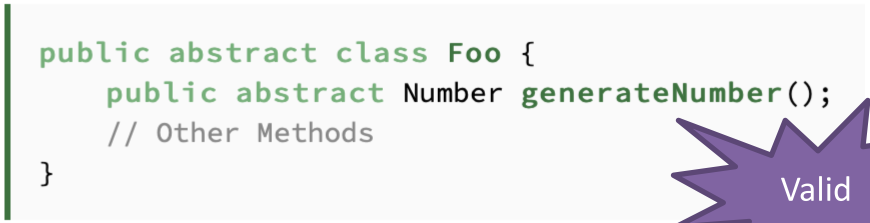
- *When extending a class, remember that you should be able to pass objects of the subclass in place of objects of the parent class without breaking the client code.*

- *This means that the subclass should remain compatible with the behavior of the superclass.*

L – Liskov Substitution Principle

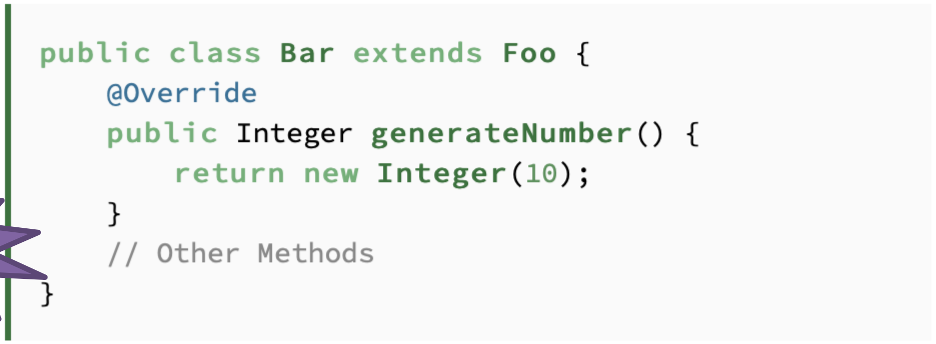
Rule 1 - ***Parameter types in a method of a subclass should match or be more abstract (more general) than parameter types in the method of the super- class.***

Rule 2 - ***The return type in a method of a subclass should match or be a subtype (more specific) of the return type in the method of the superclass.***



```java
public abstract class Foo {
    public abstract Number generateNumber();
    // Other Methods
}
```

Valid



```java
public class Bar extends Foo {
    @Override
    public Integer generateNumber() {
        return new Integer(10);
    }
    // Other Methods
}
```

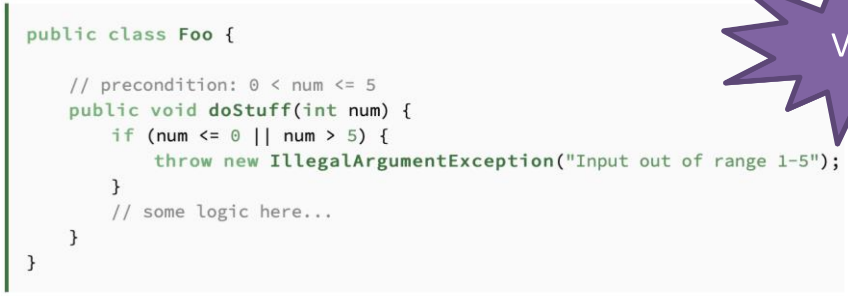
L – Liskov Substitution Principle

Rule 3 - ***A method in a subclass shouldn't throw types of exceptions which the base method isn't expected to throw.***

types of exceptions should *match* or be *subtypes* of the ones that the base method is already able to throw.

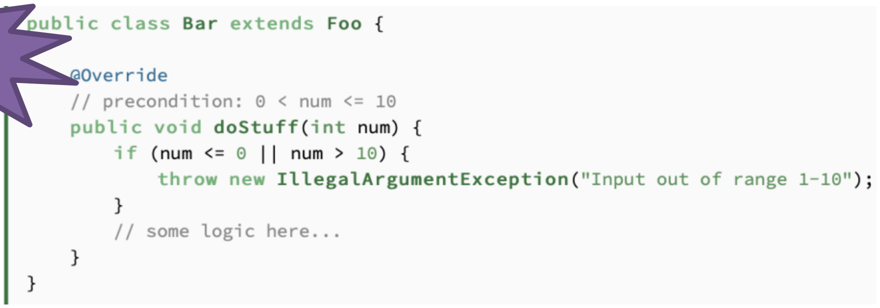
Rule 4 - ***A subclass shouldn't strengthen pre-conditions.***

A subclass shouldn't override some methods and strengthen/increase the pre-condition.



```java
public class Foo {

    // precondition: 0 < num <= 5
    public void doStuff(int num) {
        if (num <= 0 || num > 5) {
            throw new IllegalArgumentException("Input out of range 1-5");
        }
        // some logic here...
    }
}
```

Valid

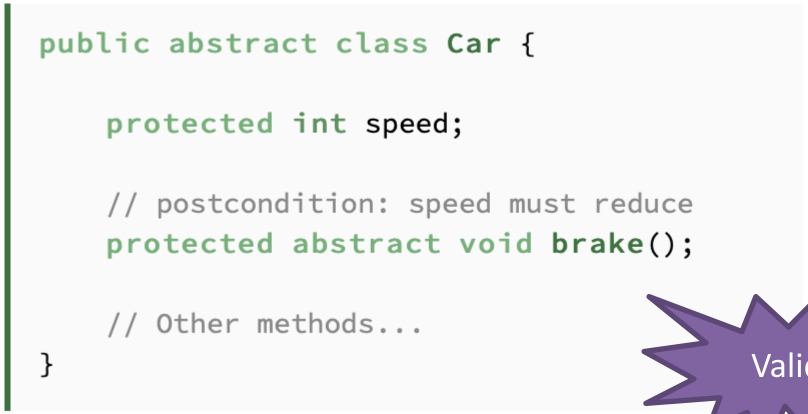


```java
public class Bar extends Foo {

    @Override
    // precondition: 0 < num <= 10
    public void doStuff(int num) {
        if (num <= 0 || num > 10) {
            throw new IllegalArgumentException("Input out of range 1-10");
        }
        // some logic here...
    }
}
```

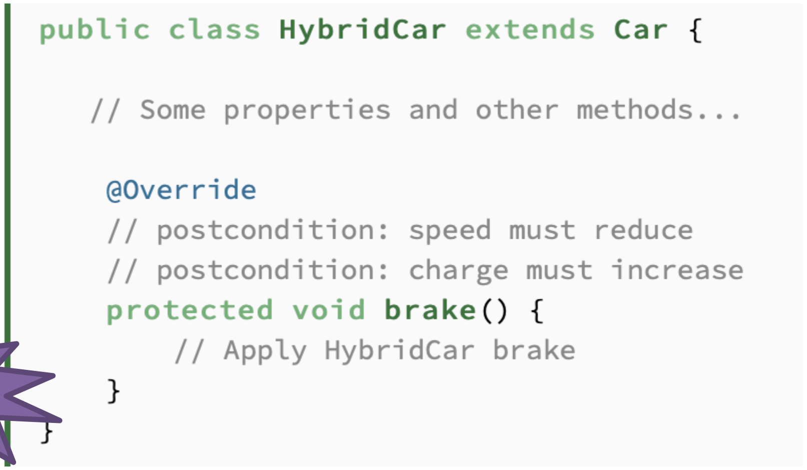
L – Liskov Substitution Principle

*Rule 5 - **A subclass shouldn't weaken post-conditions.***

A subclass shouldn't override some methods and weaken/reduce the post-condition.



```java
public abstract class Car {

    protected int speed;

    // postcondition: speed must reduce
    protected abstract void brake();

    // Other methods...
}
```



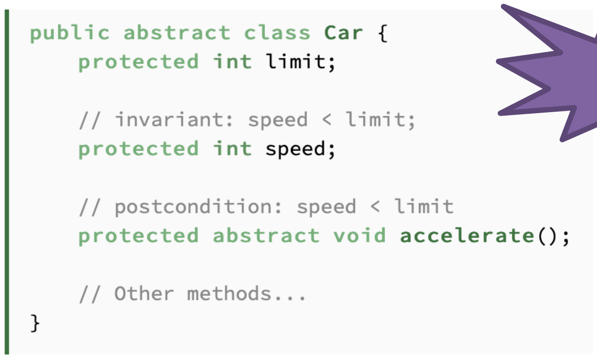
```java
public class HybridCar extends Car {

    // Some properties and other methods...

     @Override
     // postcondition: speed must reduce
     // postcondition: charge must increase
     protected void brake() {
         // Apply HybridCar brake
     }
}
```

Valid

L – Liskov Substitution Principle

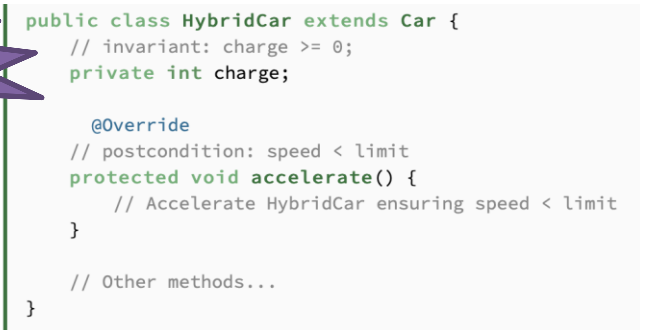
Rule 6 - ***Invariants of a superclass must be preserved.***

- This is probably the least formal rule of all.
- A class invariant is an assertion concerning object properties that must be true for all valid states of the object.
- The rule on invariants is the easiest to violate because you might misunderstand or not realize all of the invariants of a complex class.
- The safest way to extend a class is to introduce new fields and methods, and not to modify with any existing members of the superclass.



```
public abstract class Car {
    protected int limit;

    // invariant: speed < limit;
    protected int speed;

    // postcondition: speed < limit
    protected abstract void accelerate();

    // Other methods...
}
```

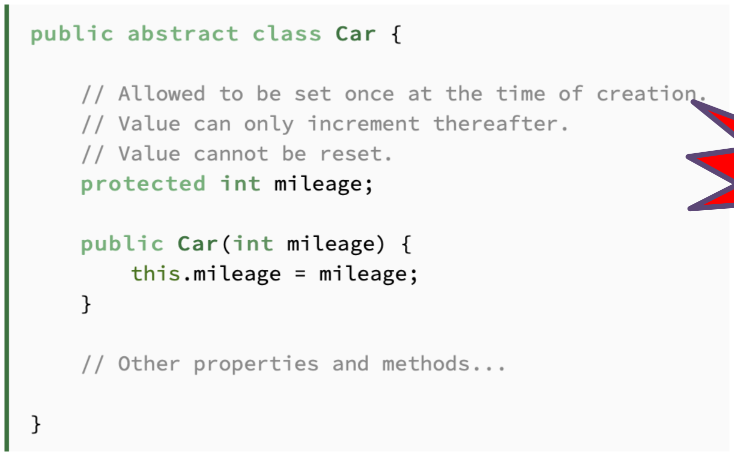
Valid



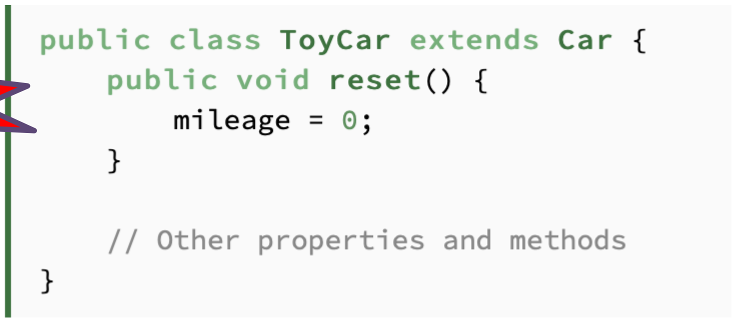
```
public class HybridCar extends Car {
    // invariant: charge >= 0;
    private int charge;

      @Override
    // postcondition: speed < limit
    protected void accelerate() {
        // Accelerate HybridCar ensuring speed < limit
    }

    // Other methods...
}
```

L – Liskov Substitution Principle

Rule 7 - A subclass shouldn't change values of private/protected states of the superclass



```java
public abstract class Car {

    // Allowed to be set once at the time of creation.
    // Value can only increment thereafter.
    // Value cannot be reset.
    protected int mileage;

    public Car(int mileage) {
        this.mileage = mileage;
    }

    // Other properties and methods...

}
```



```java
public class ToyCar extends Car {
    public void reset() {
        mileage = 0;
    }

    // Other properties and methods
}
```

L – Liskov Substitution Principle

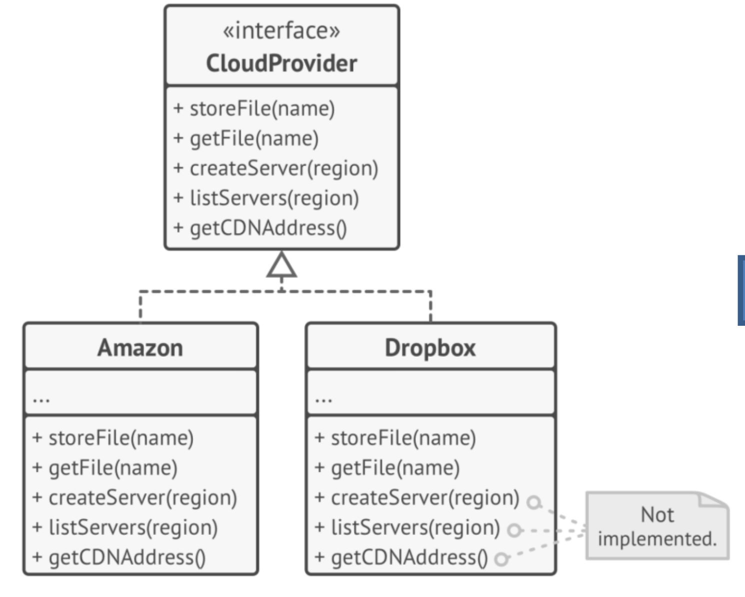
Why LSP is important?

- Code that adhere to LSP is flexible and promote high degree of reusability.
- Code that violate LSP is tightly coupled and creates unnecessary entanglements.
- If client code cannot substitute a superclass reference with a subclass object freely, it would be forced to do ***instanceof*** checks and specially handle some subclasses.
- If this kind of conditional code is spread across the codebase, it will be difficult to maintain.
- Code that adheres to the LSP is code that makes the right abstractions.

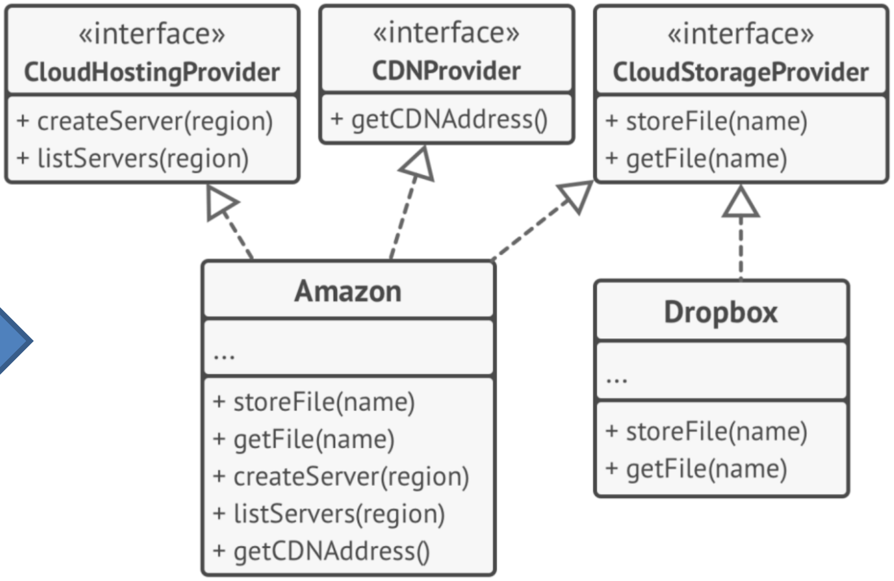
I – Interface Segregation Principle

- *Clients should not be forced to depend upon interfaces that they do not use.*
- *Make your interface narrow enough that client don’t need to implements the behaviors they don’t need.*
- *This principle is easy to violate, especially if your software evolves and you have to add more and more features over time.*
- *Like the Single Responsibility Principle, the goal of this principle is to reduce complexity and hence the side effects and frequency of required changes by splitting the software into multiple, independent parts.*

Example 8



BEFORE: not all clients can satisfy the requirements of the bloated interface.



AFTER: one bloated interface is broken down into a set of more granular interfaces.

D – Dependency Inversion Principle

- *High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features.*

- *To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.*

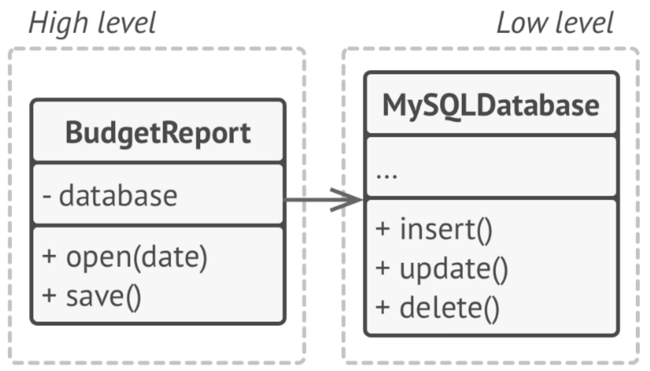
- *Definition of the Dependency Inversion Principle consists of two parts:*
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions.

D – Dependency Inversion Principle

- ***Low-level classes*** *implement basic operations such as working with a disk, transferring data over a network, connecting to a database, etc.*
- ***High-level classes*** *contain complex business logic that directs low-level classes to do something.*
- *The dependency inversion principle suggests to work on the high-level classes first.*
- *Complete the high-level classes by dependent on abstraction of the low-level classes (the interface).*
- *Then complete the low-level classes by writing the concrete classes of the abstraction (implementing the interface).*

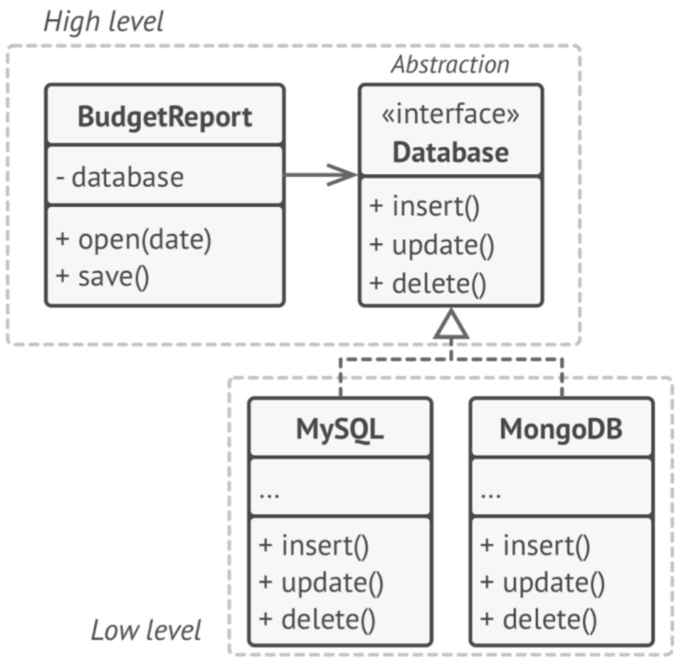
Example 9

Complete low-level class first



BEFORE: a high-level class depends on a low-level class.

Complete high-level class first



AFTER: low-level classes depend on a high-level abstraction.

Exercise

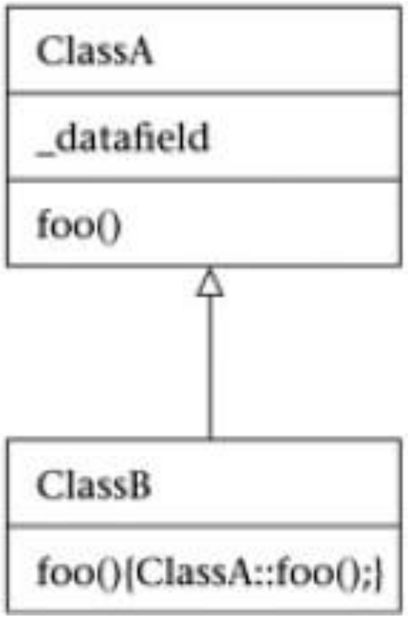
1. *According to SOLID principles, which is the most suitable situation to use inheritance?*
2. *According to SOLID principles, which is the most suitable situation to use composition?*
3. *Explain in detail the problem when the return type in an override method of a subclass is a long data type while the return type in the method of the superclass is an int data type?*
4. *Discuss the Single Responsibility Principle (SRP) and provide an example of how it can be applied in a software design.*

Exercise

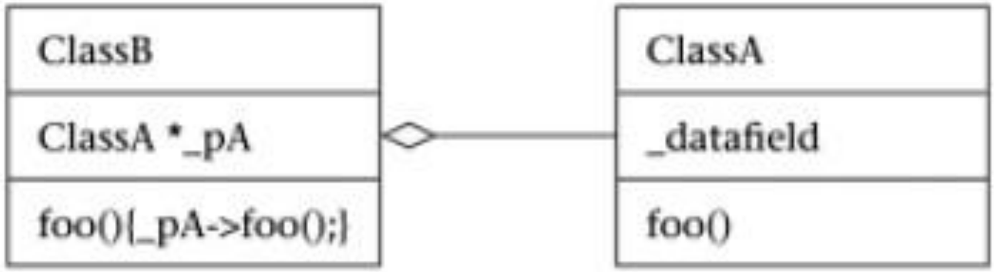
5. *How does the Open/Closed Principle (OCP) contribute to software maintainability and extensibility? Provide a real-world example to illustrate this principle.*
6. *Explain the Interface Segregation Principle (ISP) and discuss its role in software design.*
7. *Describe how the Dependency Inversion Principle (DIP) helps to decouple software modules and enhance flexibility in a software architecture.*

Loosen Coupling using Composition

Composition (has a) and Inheritance (is a)



Class B is a Class A

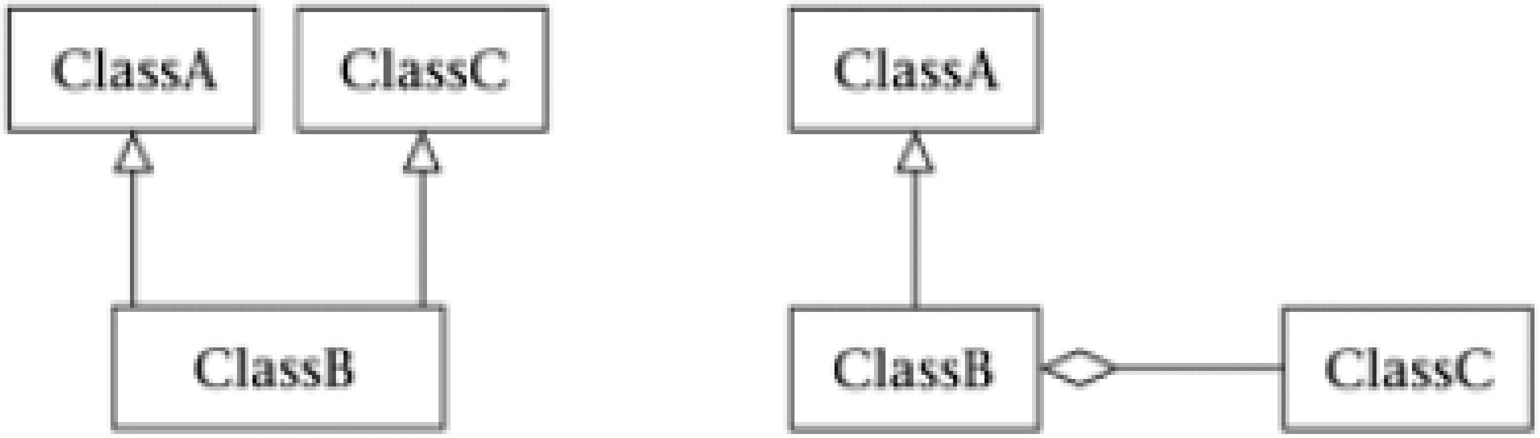


Class B has a Class A

You can always replace an inheritance relationship by a composition relationship

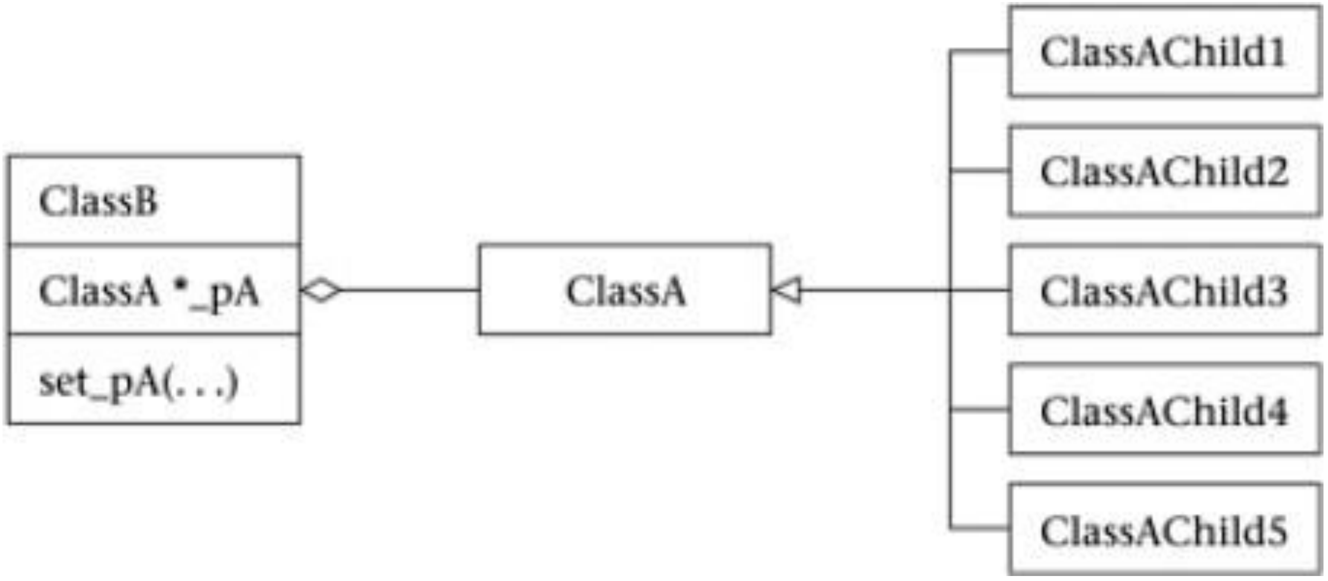
Usage of Composition

- Use composition as an alternative to *multiple inheritance*

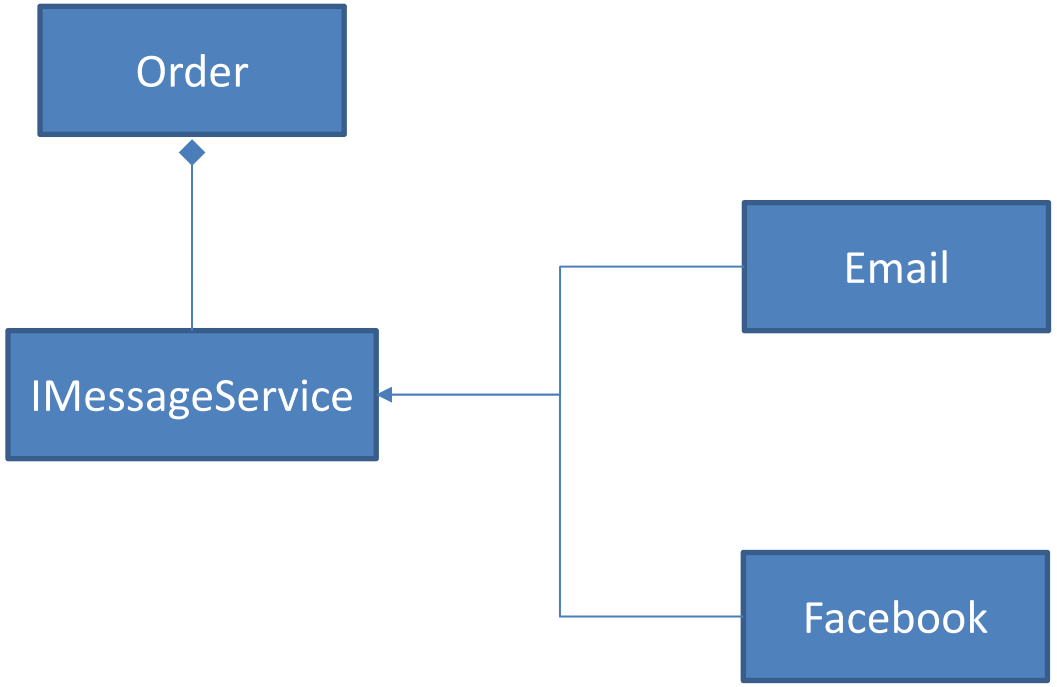


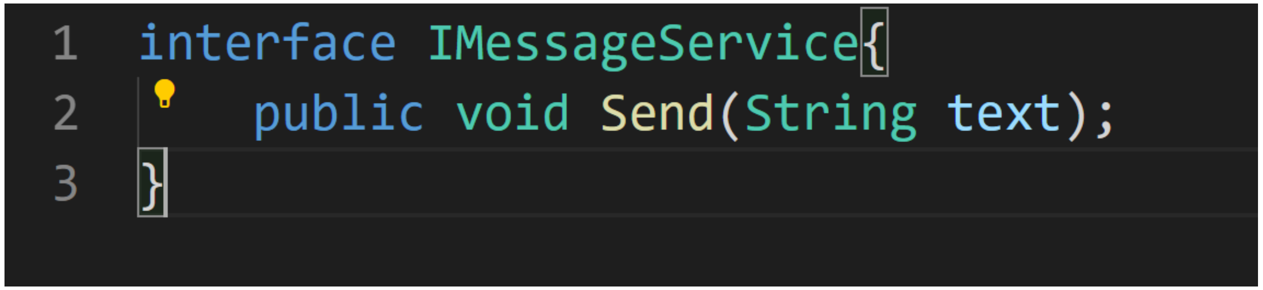
Usage of Composition

- Composition *makes dynamic change possible*
- Think about polymorphism

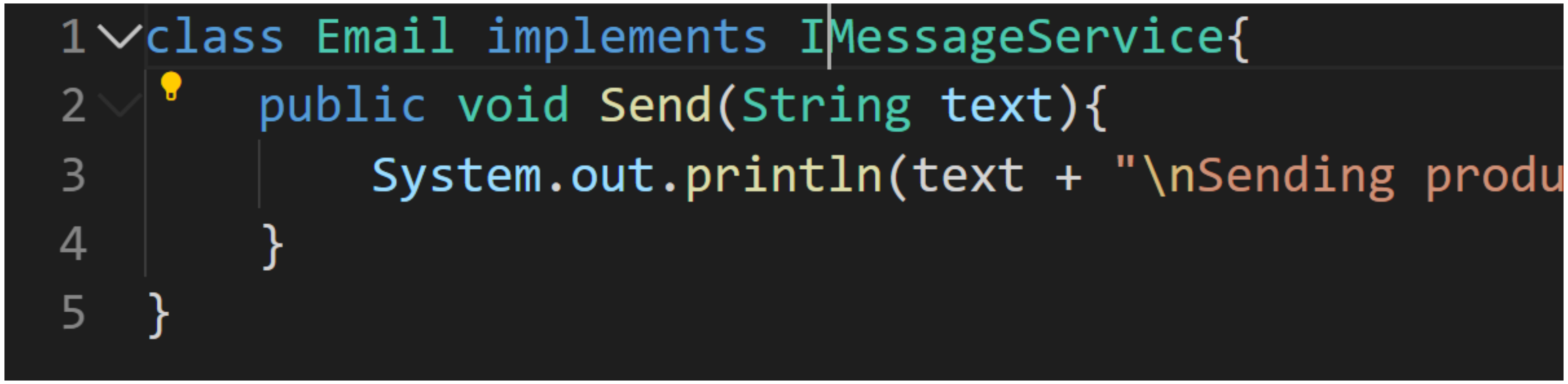


Example

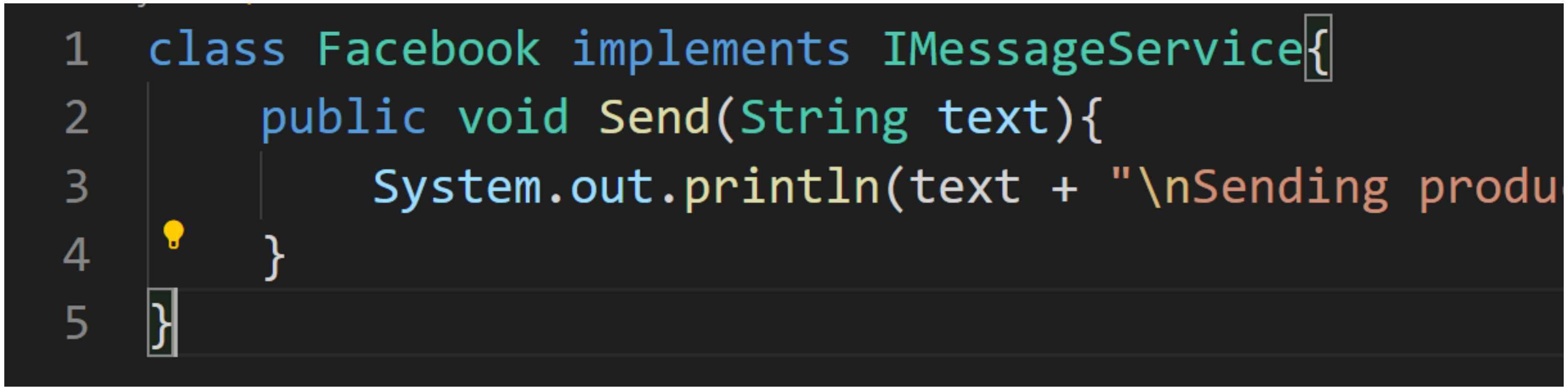




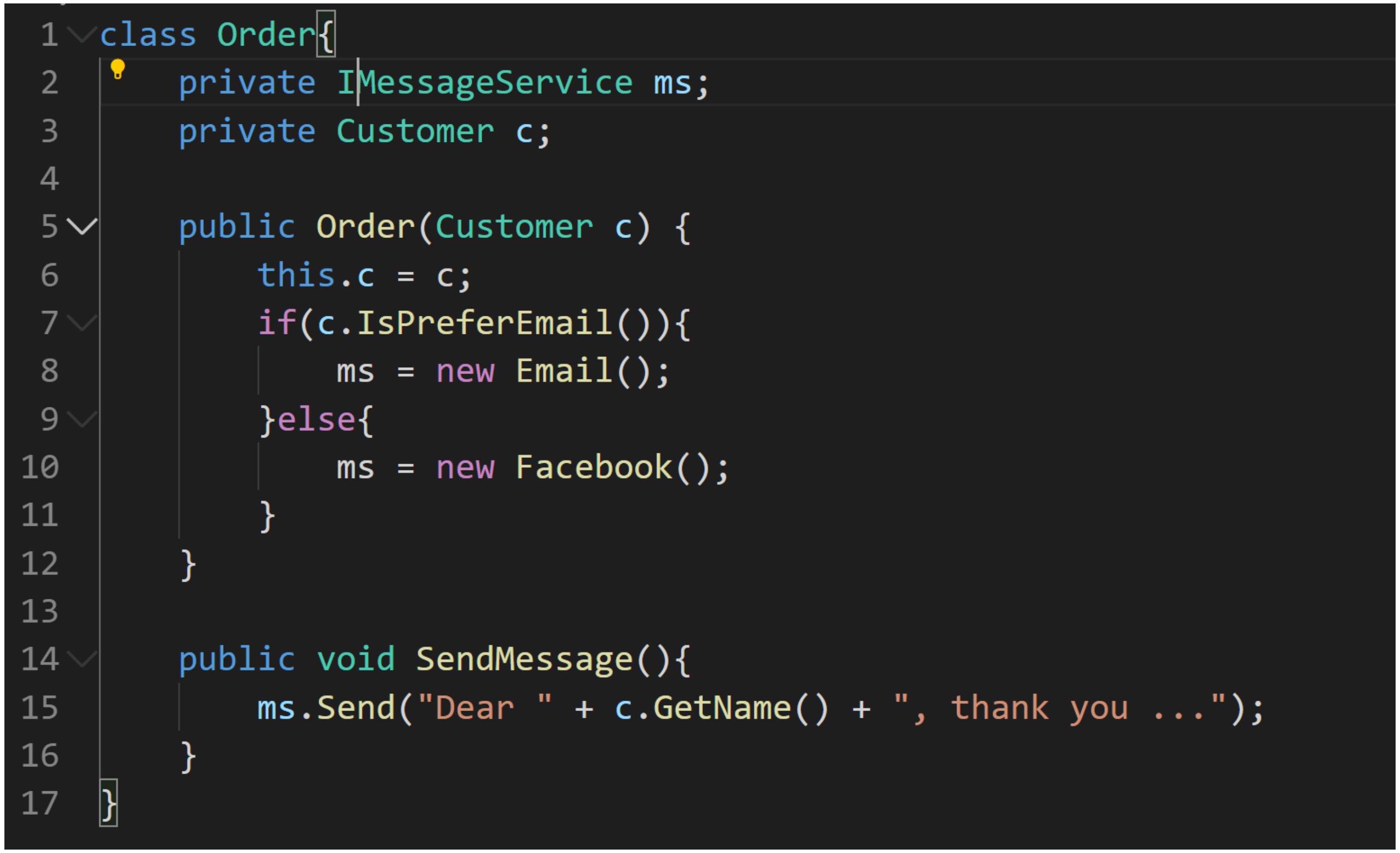
```
interface IMessageService{
    public void Send(String text);
}
```



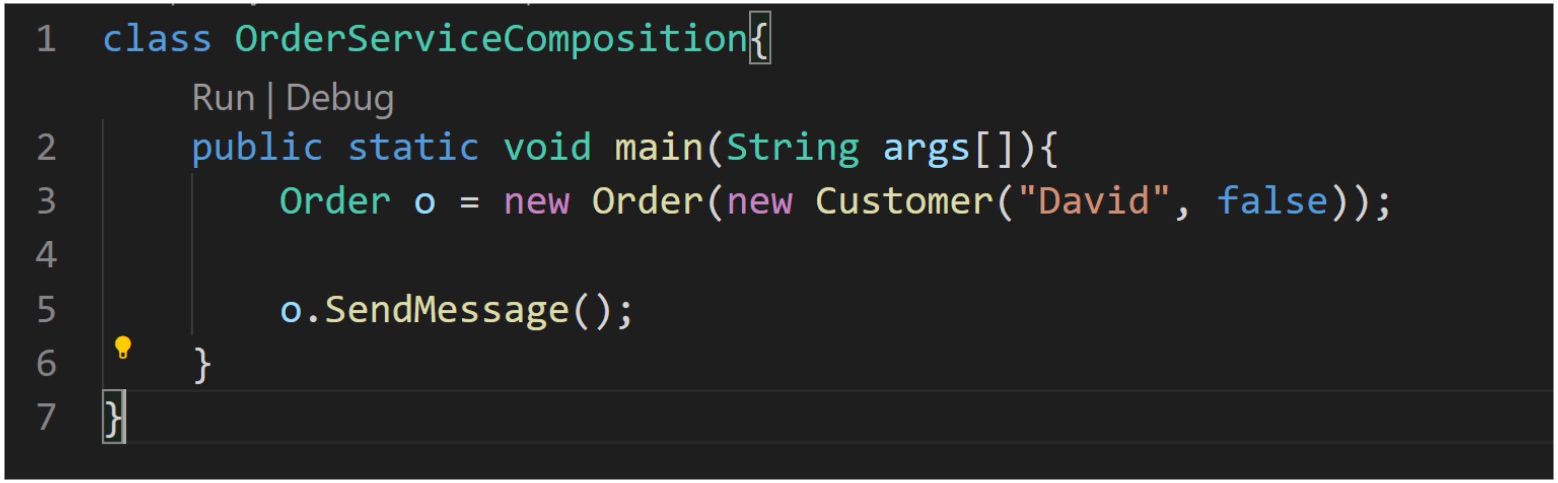
```
class Email implements IMessageService{
    public void Send(String text){
        System.out.println(text + "\nSending produ
    }
}
```



```
class Facebook implements IMessageService{
    public void Send(String text){
        System.out.println(text + "\nSending produ
    }
}
```



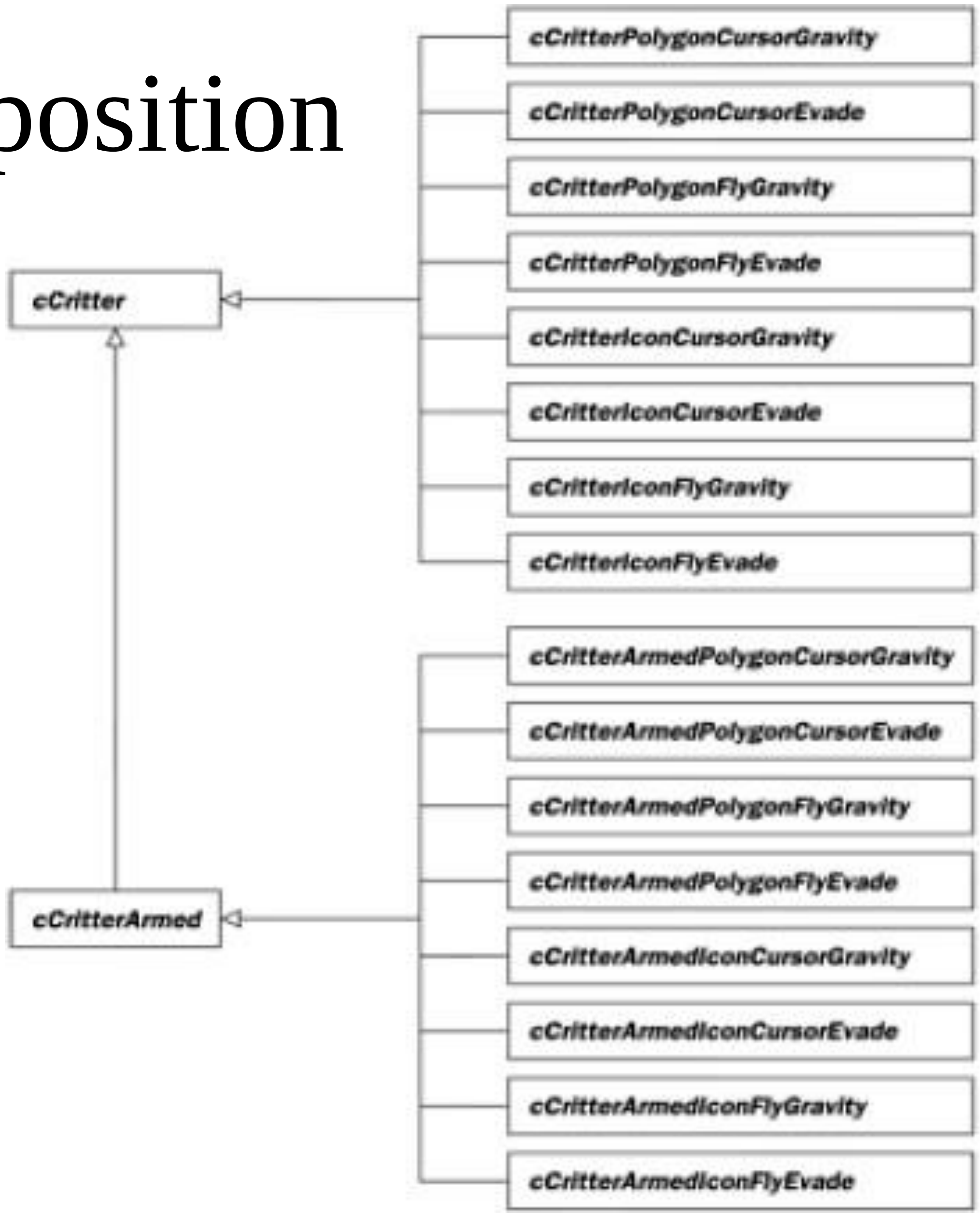
```java
class Order{
    private IMessageService ms;
    private Customer c;

    public Order(Customer c) {
        this.c = c;
        if(c.IsPreferEmail()){
            ms = new Email();
        }else{
            ms = new Facebook();
        }
    }

    public void SendMessage(){
        ms.Send("Dear " + c.GetName() + ", thank you ...");
    }
}
```



```java
class OrderServiceComposition{
    Run | Debug
    public static void main(String args[]){
        Order o = new Order(new Customer("David", false));

        o.SendMessage();
    }
}
```

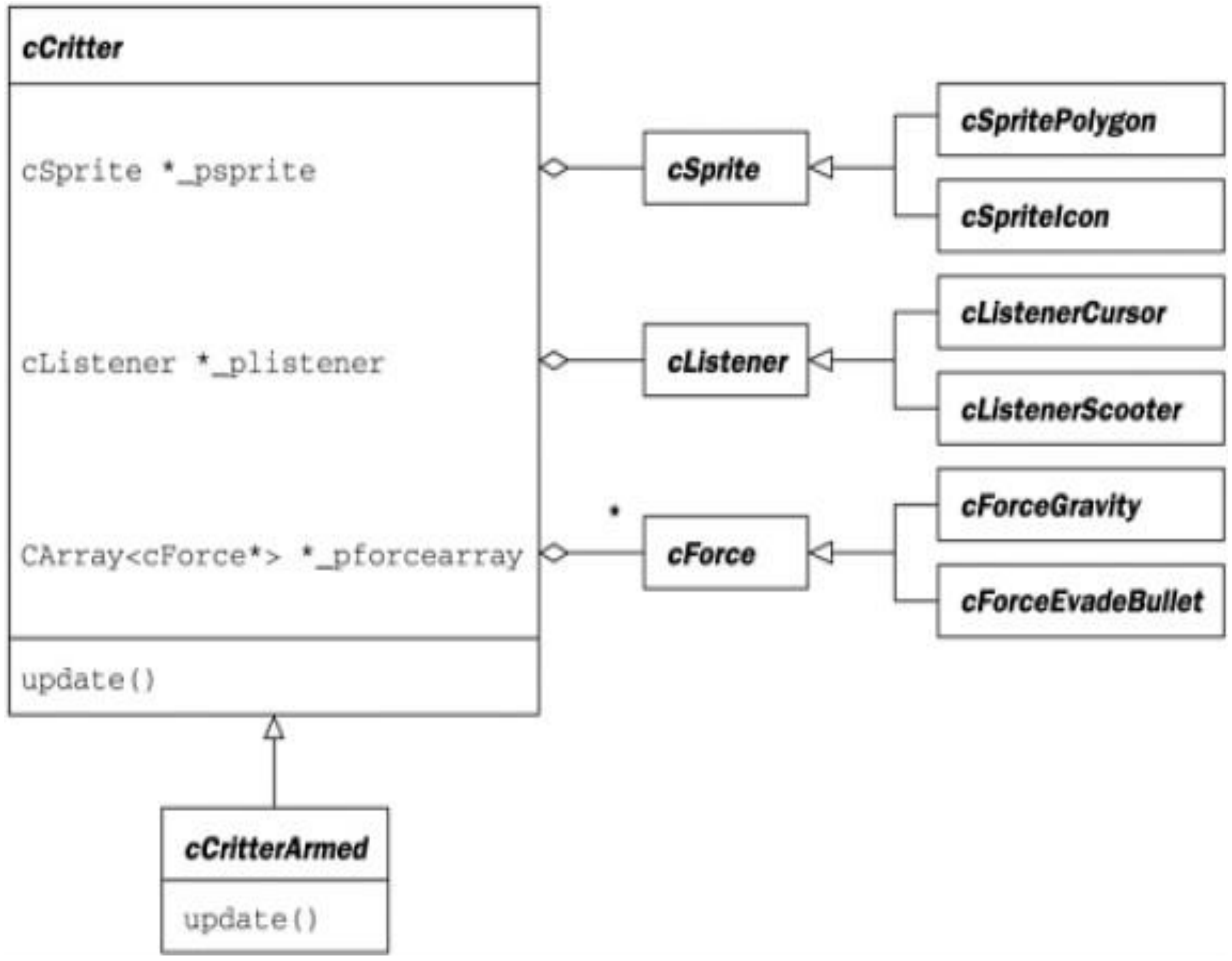
Usage of Composition

- Use composition to avoid the 'combinatorial explosion‘


cCritter
cCritterArmed
cCritterPolygonCursorGravity
cCritterPolygonCursorEvade
cCritterPolygonFlyGravity
cCritterPolygonFlyEvade
cCritterIconCursorGravity
cCritterIconCursorEvade
cCritterIconFlyGravity
cCritterIconFlyEvade
cCritterArmedPolygonCursorGravity
cCritterArmedPolygonCursorEvade
cCritterArmedPolygonFlyGravity
cCritterArmedPolygonFlyEvade
cCritterArmedIconCursorGravity
cCritterArmedIconCursorEvade
cCritterArmedIconFlyGravity
cCritterArmedIconFlyEvade

Usage of Composition

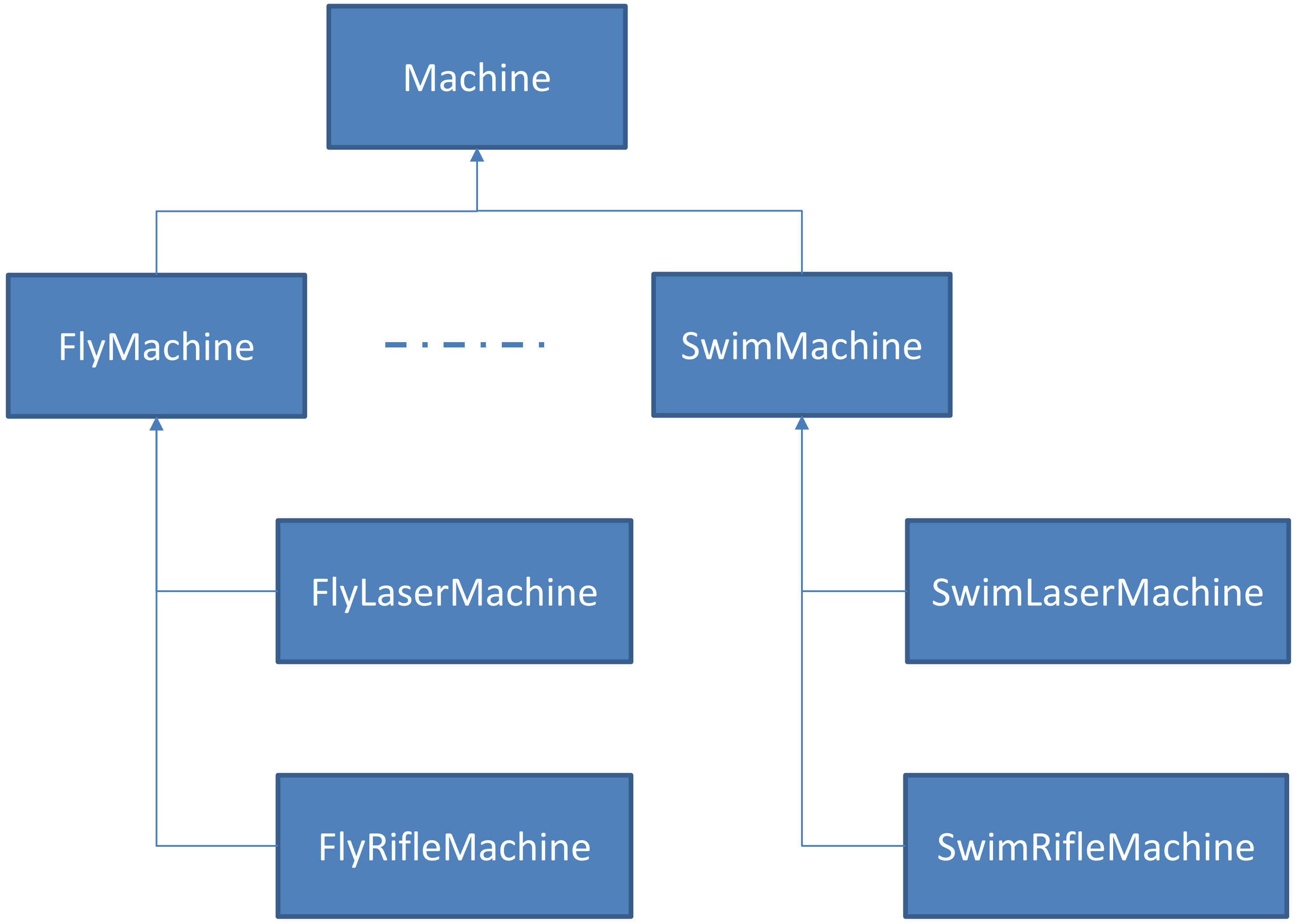
- Use composition to *avoid the 'combinatorial explosion'*

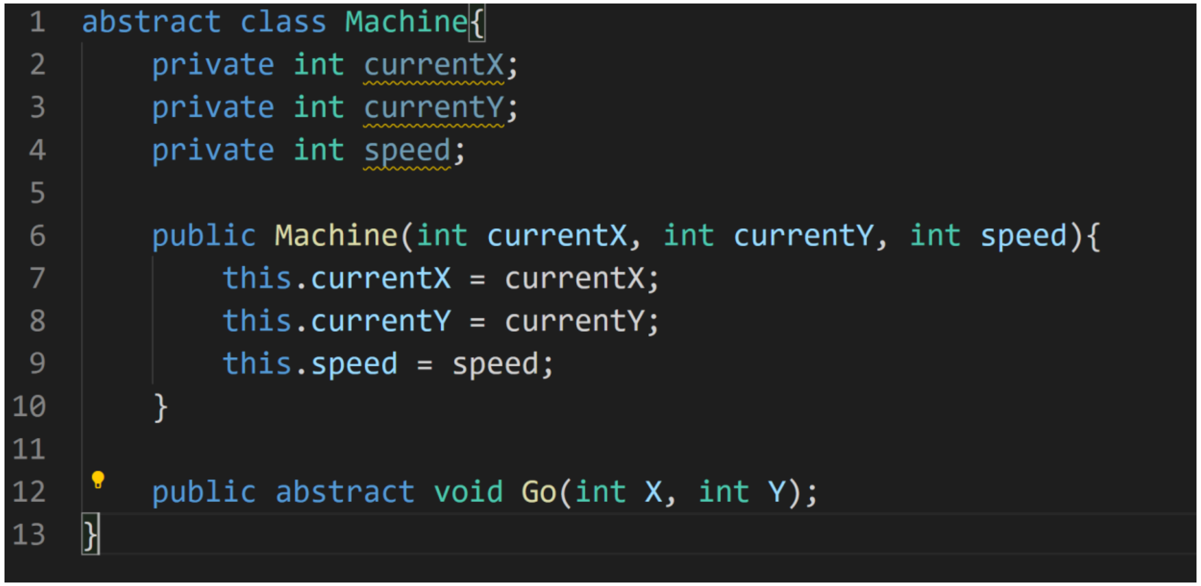


Machines

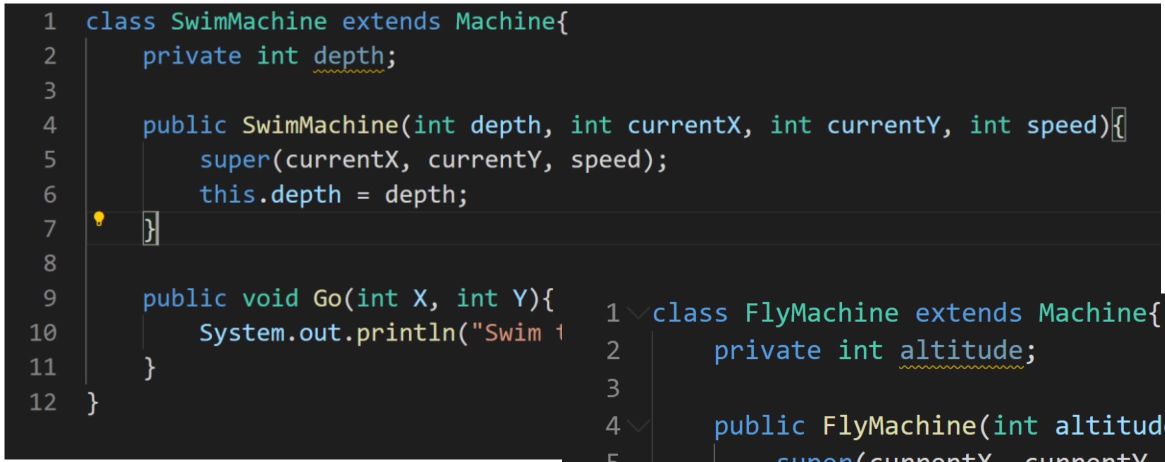
There are several types of machines. Some can fly and some can swim. Regardless of how the machine move, they can attack enemy by using either laser or rifle. Some machines does not have the attack capability.

Inheritance Hierarchy

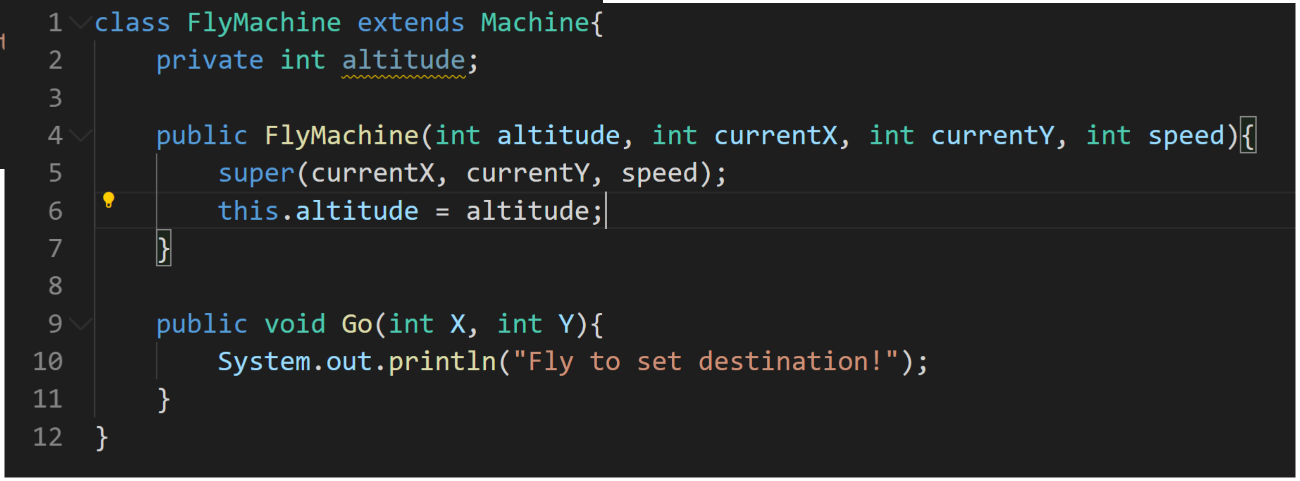




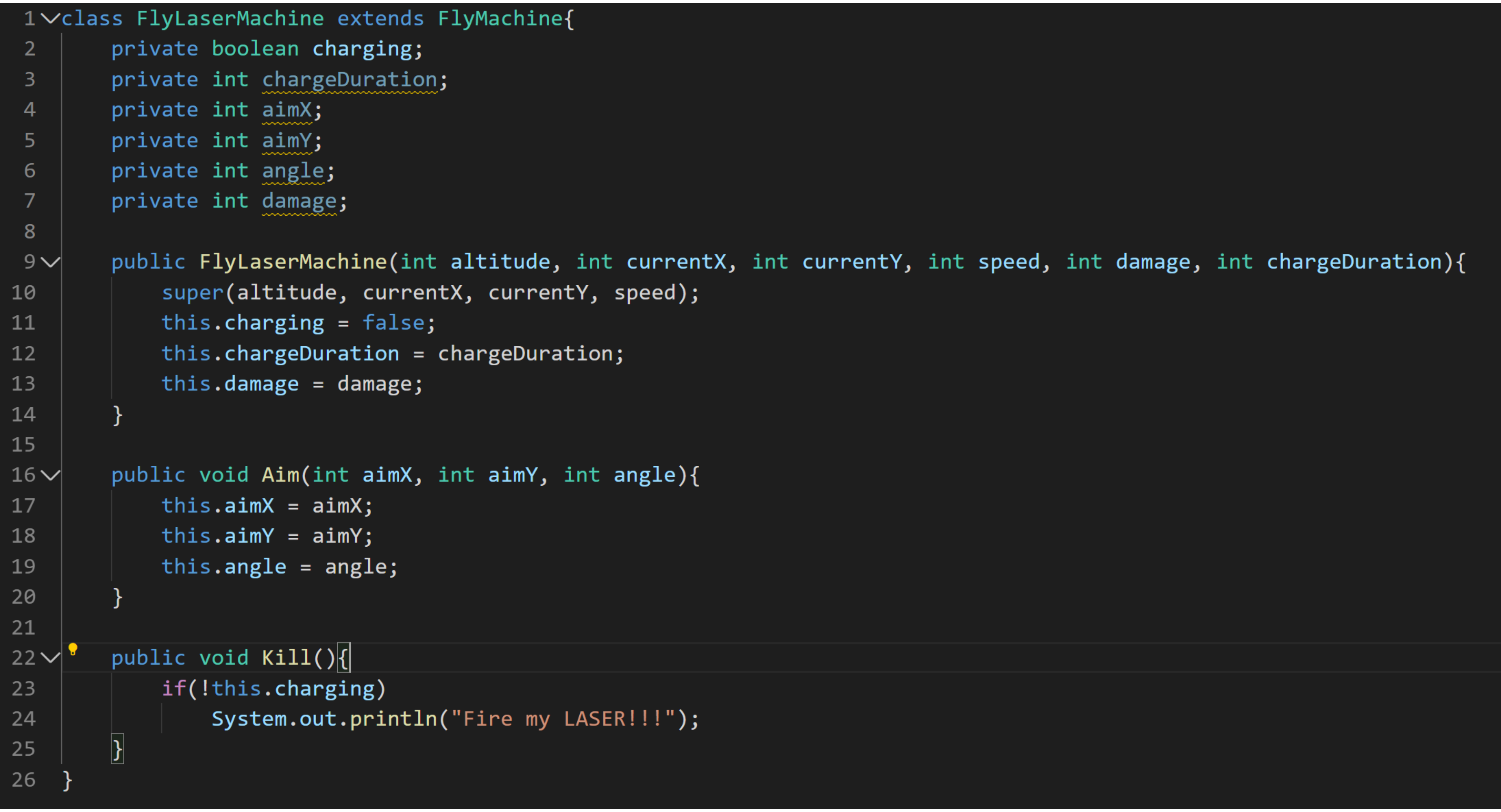
```java
abstract class Machine{
    private int currentX;
    private int currentY;
    private int speed;

    public Machine(int currentX, int currentY, int speed){
        this.currentX = currentX;
        this.currentY = currentY;
        this.speed = speed;
    }

    public abstract void Go(int X, int Y);
}
```



```java
class SwimMachine extends Machine{
    private int depth;

    public SwimMachine(int depth, int currentX, int currentY, int speed){
        super(currentX, currentY, speed);
        this.depth = depth;
    }

    public void Go(int X, int Y){
        System.out.println("Swim t
    }
}
```

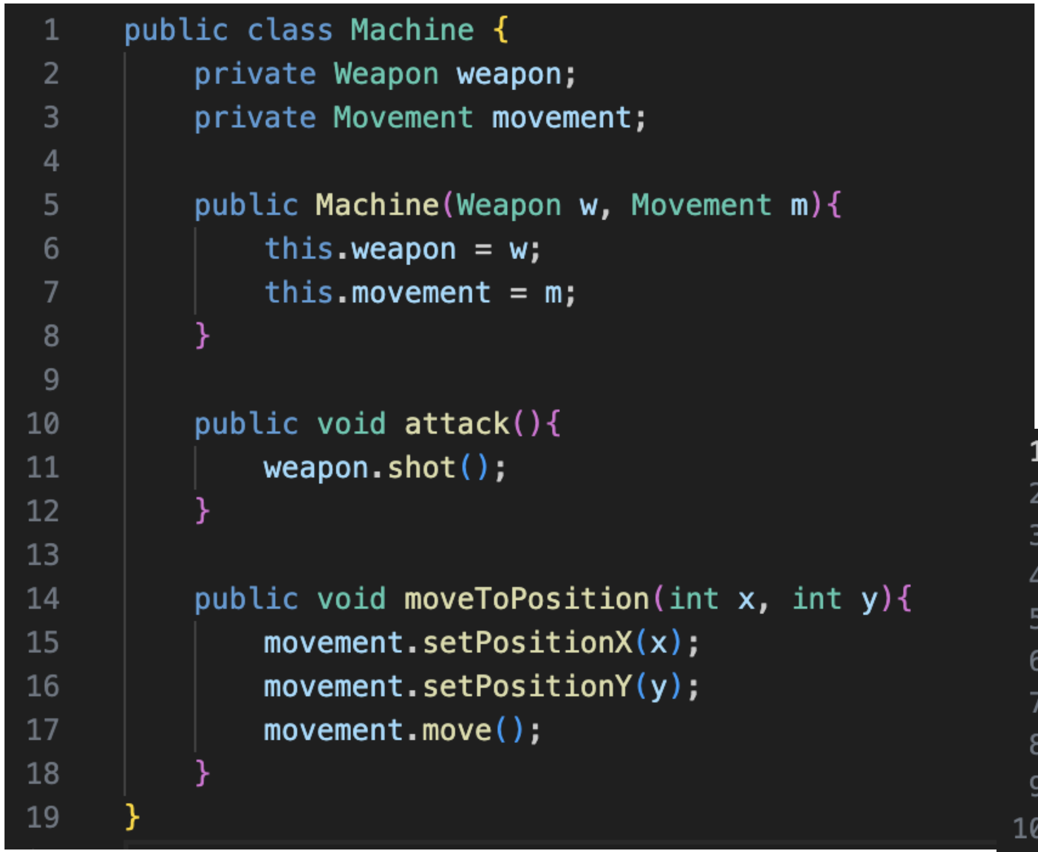


```java
class FlyMachine extends Machine{
    private int altitude;

    public FlyMachine(int altitude, int currentX, int currentY, int speed){
        super(currentX, currentY, speed);
        this.altitude = altitude;
    }

    public void Go(int X, int Y){
        System.out.println("Fly to set destination!");
    }
}
```

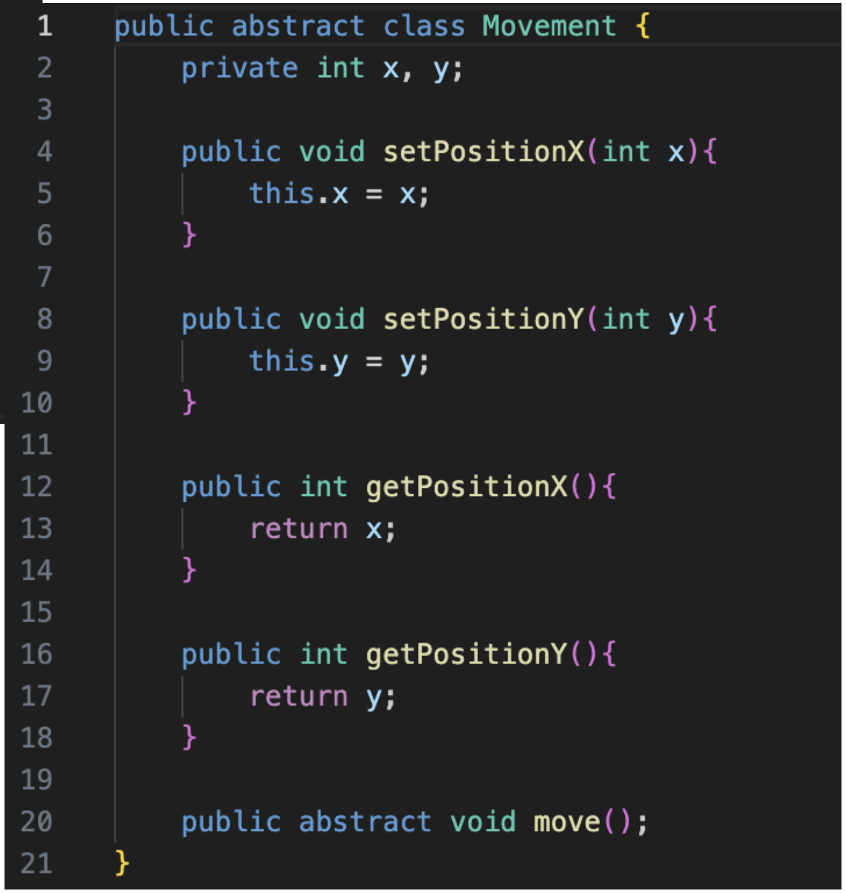

```java
class FlyLaserMachine extends FlyMachine{
    private boolean charging;
    private int chargeDuration;
    private int aimX;
    private int aimY;
    private int angle;
    private int damage;

    public FlyLaserMachine(int altitude, int currentX, int currentY, int speed, int damage, int chargeDuration){
        super(altitude, currentX, currentY, speed);
        this.charging = false;
        this.chargeDuration = chargeDuration;
        this.damage = damage;
    }

    public void Aim(int aimX, int aimY, int angle){
        this.aimX = aimX;
        this.aimY = aimY;
        this.angle = angle;
    }

    public void Kill(){
        if(!this.charging)
            System.out.println("Fire my LASER!!!");
    }
}
```

Machine Classes with Composition

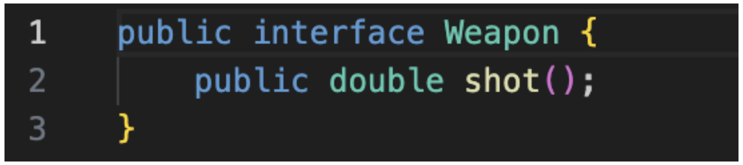
How can we achieve the same features using Composition instead of Inheritance?



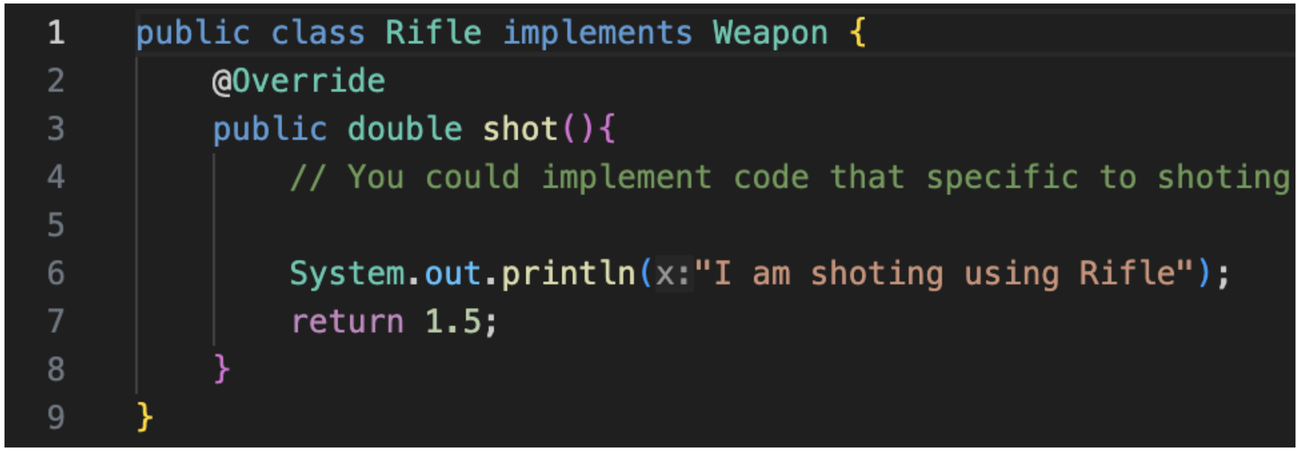
```java
public class Machine {
    private Weapon weapon;
    private Movement movement;

    public Machine(Weapon w, Movement m){
        this.weapon = w;
        this.movement = m;
    }

    public void attack(){
        weapon.shot();
    }

    public void moveToPosition(int x, int y){
        movement.setPositionX(x);
        movement.setPositionY(y);
        movement.move();
    }
}
```



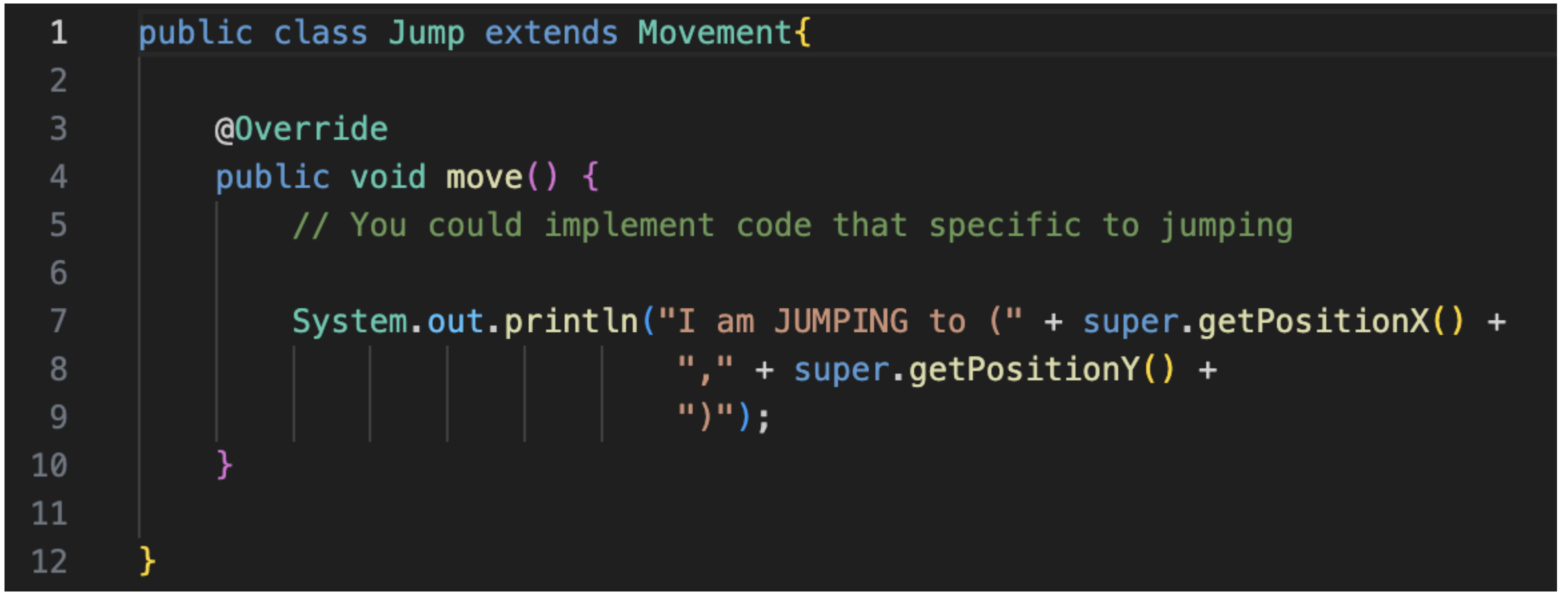
```java
public abstract class Movement {
    private int x, y;

    public void setPositionX(int x){
        this.x = x;
    }

    public void setPositionY(int y){
        this.y = y;
    }

    public int getPositionX(){
        return x;
    }

    public int getPositionY(){
        return y;
    }

    public abstract void move();
}
```



```java
public interface Weapon {
    public double shot();
}
```



```java
public class Rifle implements Weapon {
    @Override
    public double shot(){
        // You could implement code that specific to shoting

        System.out.println(x:"I am shoting using Rifle");
        return 1.5;
    }
}
```



```java
public class Jump extends Movement{

    @Override
    public void move() {
        // You could implement code that specific to jumping

        System.out.println("I am JUMPING to (" + super.getPositionX() +
                                    "," + super.getPositionY() +
                                    ")");
    }

}
```

End