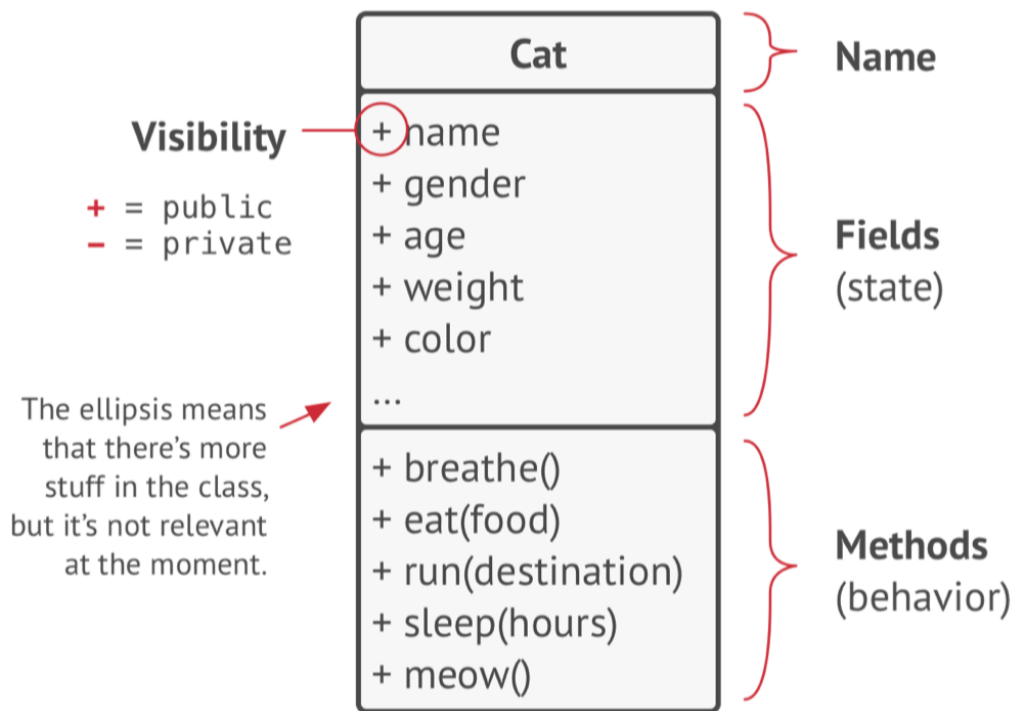


Clean Code in OOP

cuong@techmaster.vn

Object Oriented Programming

Class

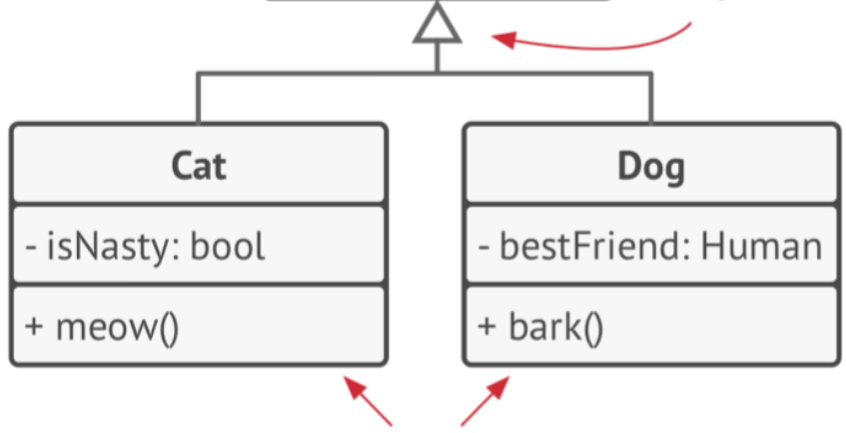


Objects

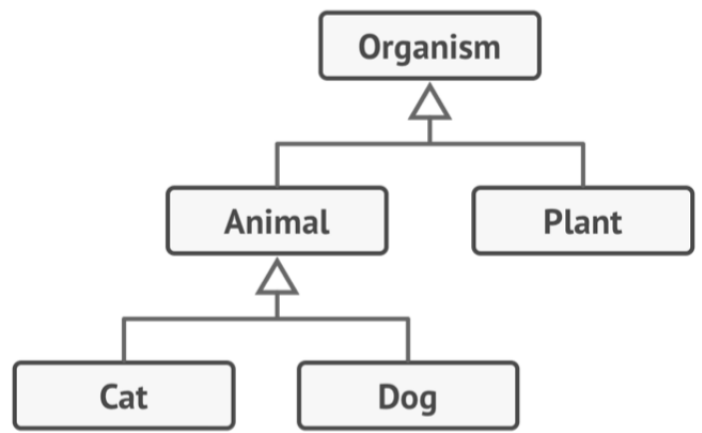
name	gender	age	weight	color
Belgan	male	8 months	1.2kg	Gray - white
Bobcat	female	2 years	3kg	Gray yellow
Persian	male	3.5 years	2.8kg	Yellow red



Arrows with empty triangle heads indicate inheritance and always go from a subclass to a superclass. Arrows from several subclasses can overlap (as in this diagram) or be drawn separately. This doesn't change their meaning.



Subclasses



Access Modifier at class and inside class

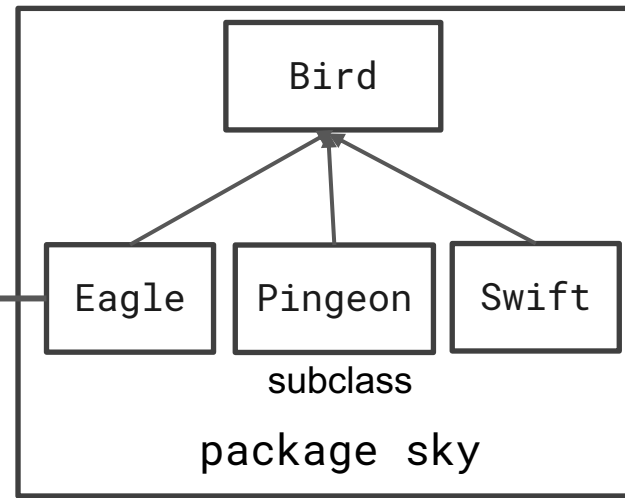
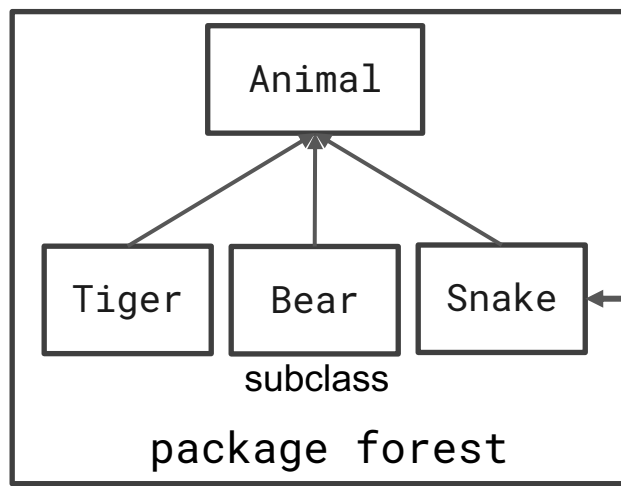
	public	private	protected	default
class	allowed	not allowed	not allowed	allowed
constructor	allowed	allowed	allowed	allowed
variable	allowed	allowed	allowed	allowed
method	allowed	allowed	allowed	allowed

`public class Animal`
`class Animal`



`private class Animal`
`protected Animal`





Access from
out side

	class	subclass	package	outside
private	allowed	not allowed	not allowed	not allowed
protected	allowed	allowed	allowed	not allowed
public	allowed	allowed	allowed	allowed
default	allowed	not allowed	allowed	not allowed

```
public class Animal {
    String name = "Bengal";
    private final float secret_magic = 1.61803398875;
    protected String favourite_food = "leaves";

    private void say() {
        System.out.println("I am animal");
    }
}

public class Jungle {
    public void raiseSomeAnimals() {
        Animal animal = new Animal();
        animal.say(); //Compile error access to private method
        System.out.println(animal.secret_magic); //Compile error access to private
        System.out.println(animal.favourite_food); //Ok
        System.out.println(animal.name); //Ok
    }
}
```

OOP

A diagram illustrating the four pillars of Object-Oriented Programming (OOP). At the top, the letters 'OOP' are written in a large, bold, black font, centered within a black triangular frame that resembles a pediment. Below this frame, four identical, classical-style columns are arranged in a row. Each column is white with vertical fluting and a decorative capital. The columns are positioned such that they appear to support the triangular frame above them. Below each column, its name is written in a bold, black, sans-serif font.

Abstraction

Encapsulation

Inheritance

Polymorphism

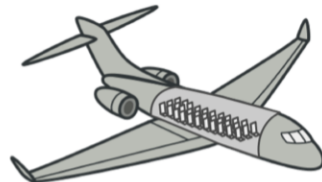
Abstract - Trừu tượng hoá



Đối tượng cụ thể



Airplane
- speed - altitude - rollAngle - pitchAngle - yawAngle
+ fly()



Airplane
- seats
+ reserveSeat(n)

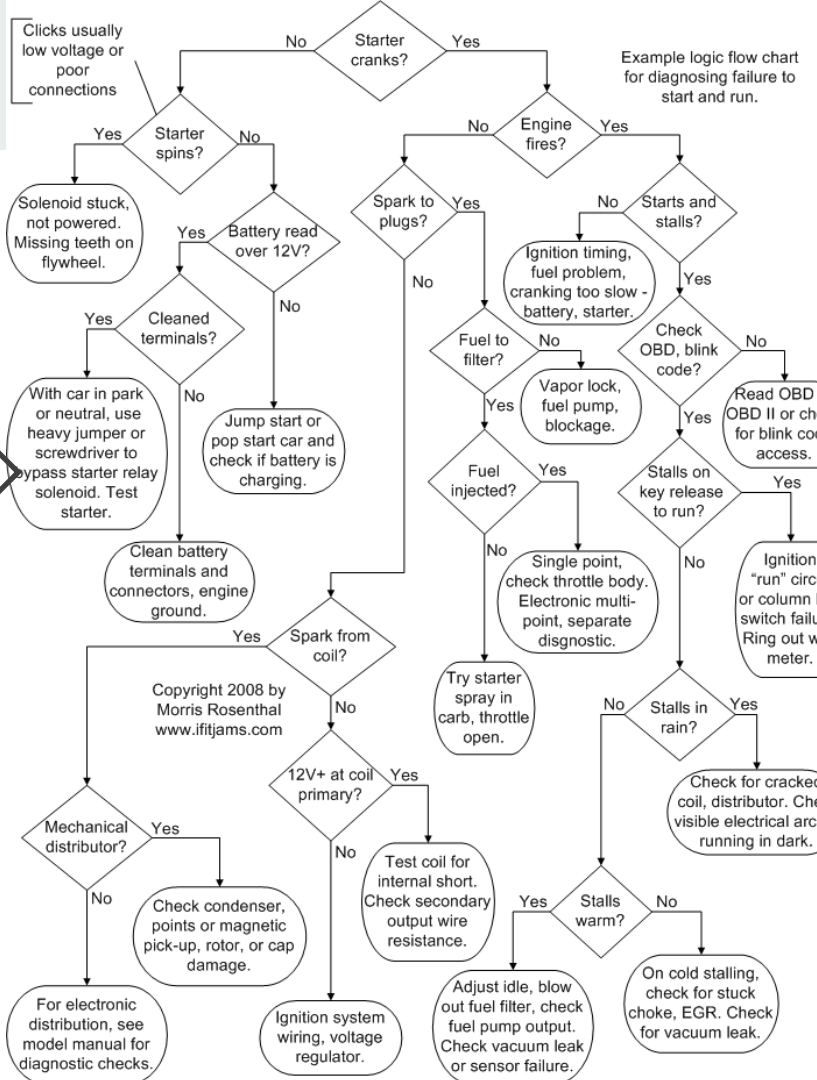
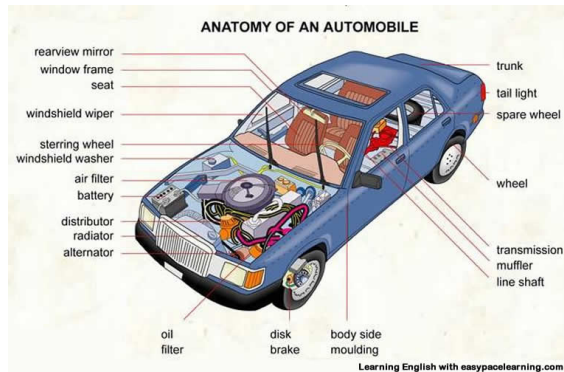
Class trừu tượng hoá

Encapsulation – Đóng gói

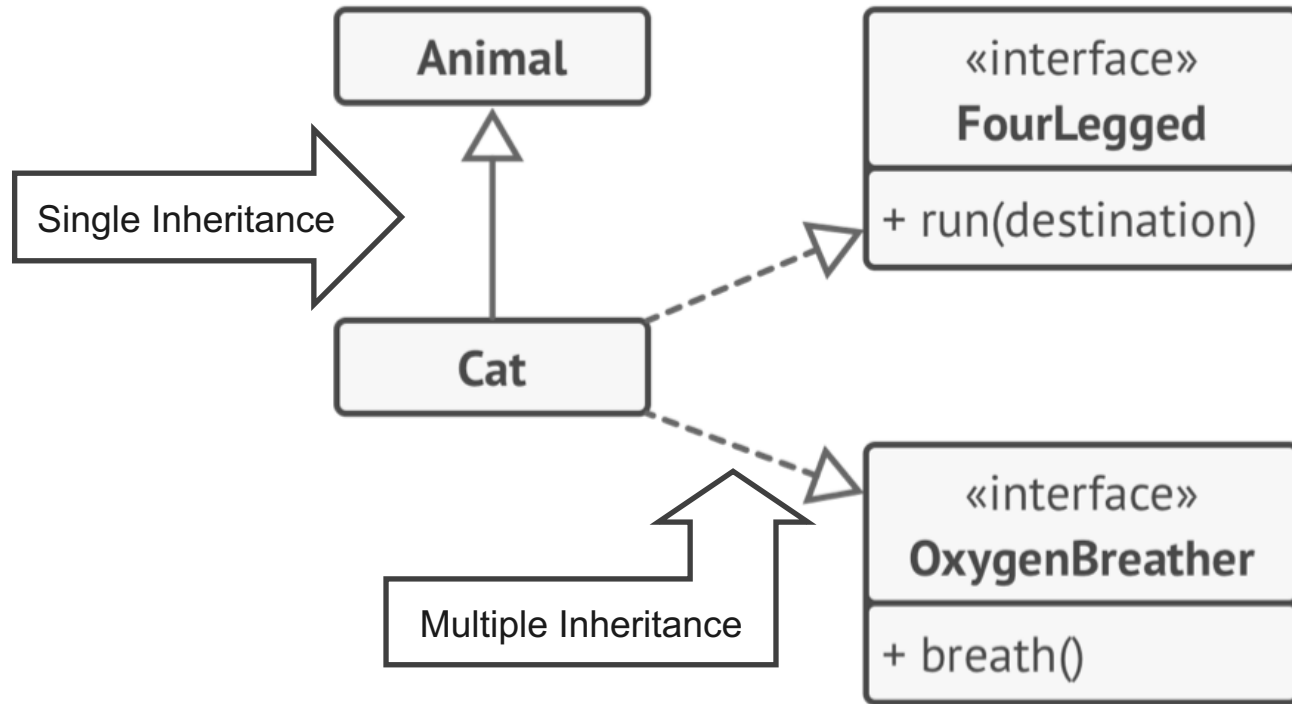


Đối với người dùng,
việc khởi động ô tô
chỉ cần bấm một nút

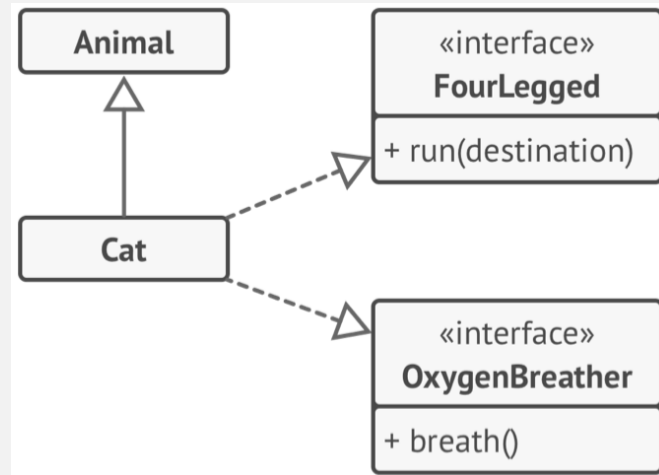
Thực tế vận hành bên trong
là một quá trình phức tạp



Inheritance – Kế thừa

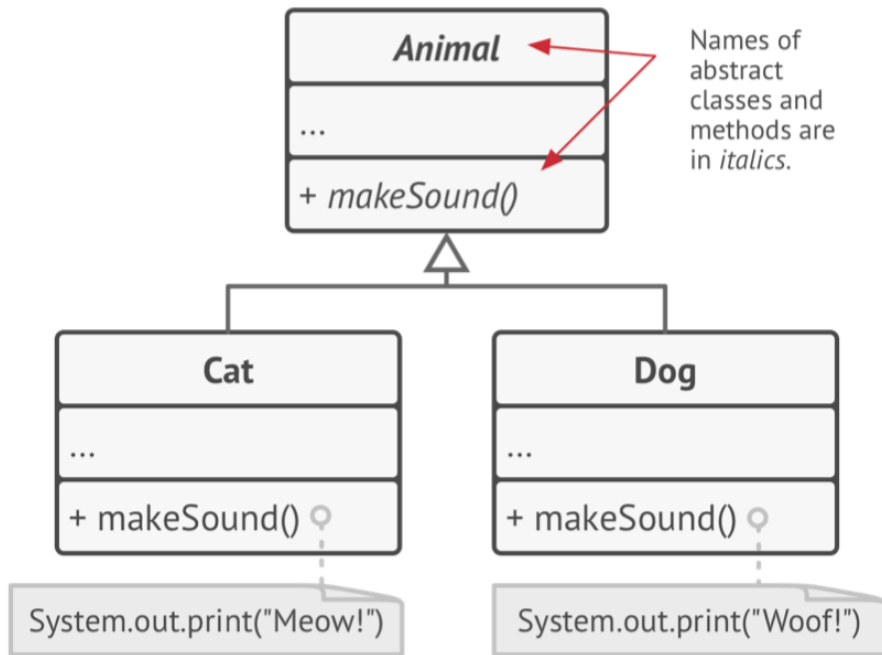


```
public class Animal {  
}  
  
public interface FourLegged {  
    public void run(Destination destination);  
}  
  
public interface OxygenBreather {  
    public void breath();  
}
```



```
public class Cat extends Animal implements FourLegged, OxygenBreather {  
    @Override  
    public void run(Destination destination) {}  
  
    @Override  
    public void breath() { }  
}
```

Polymorphism – Đa hình



```
abstract class Animal {
    public void makeSound() {
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}
```

```
public class PetHouse {  
    public void allMakeSound() {  
        Animal[] animals = {new Cat(), new Dog()};  
        for (Animal animal: animals) {  
            animal.makeSound();  
        }  
    }  
}
```



Dùng biến có kiểu base class nhưng khi thực hiện lại gọi phương thức của sub class

Terminal output

Meow!

Woof!

Poli + morphism

Nhiều
Đa

cách biến hoá
hình

Nếu không có đa hình, đời dev khổ

```
Cat cat = new Cat();
```

```
Dog dog = new Dog();
```

```
Tiger tiger = new Tiger();
```

// Games có 100 con vật thì làm thế nào???

```
cat.makeSound();
```

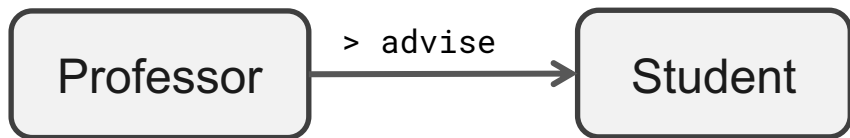
```
dog.makeSound();
```

```
tiger.makeSound();
```



Ôn tập lại UML

One way association

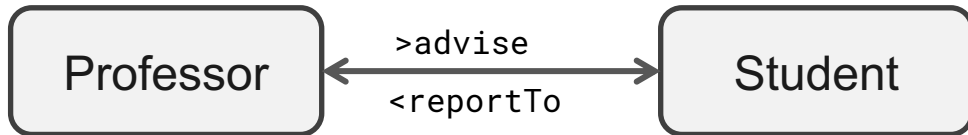


UML association: Professor advises student

```
class Professor {
    public void advise(Student student) {
        ...
    }
}

class Student {
    ...
}
```

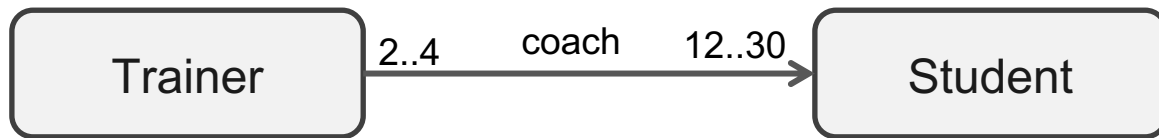
Two ways association



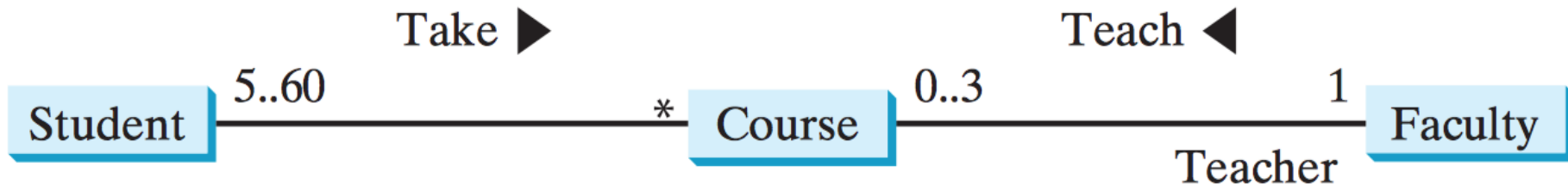
```
class Professor extends Person {
    public void advise(Student student) {
        System.out.println("Advise " + student.name);
    }
}

class Student extends Person {
    public void reportTo(Professor professor) {
        System.out.println("Report to " + professor.name);
    }
}
```

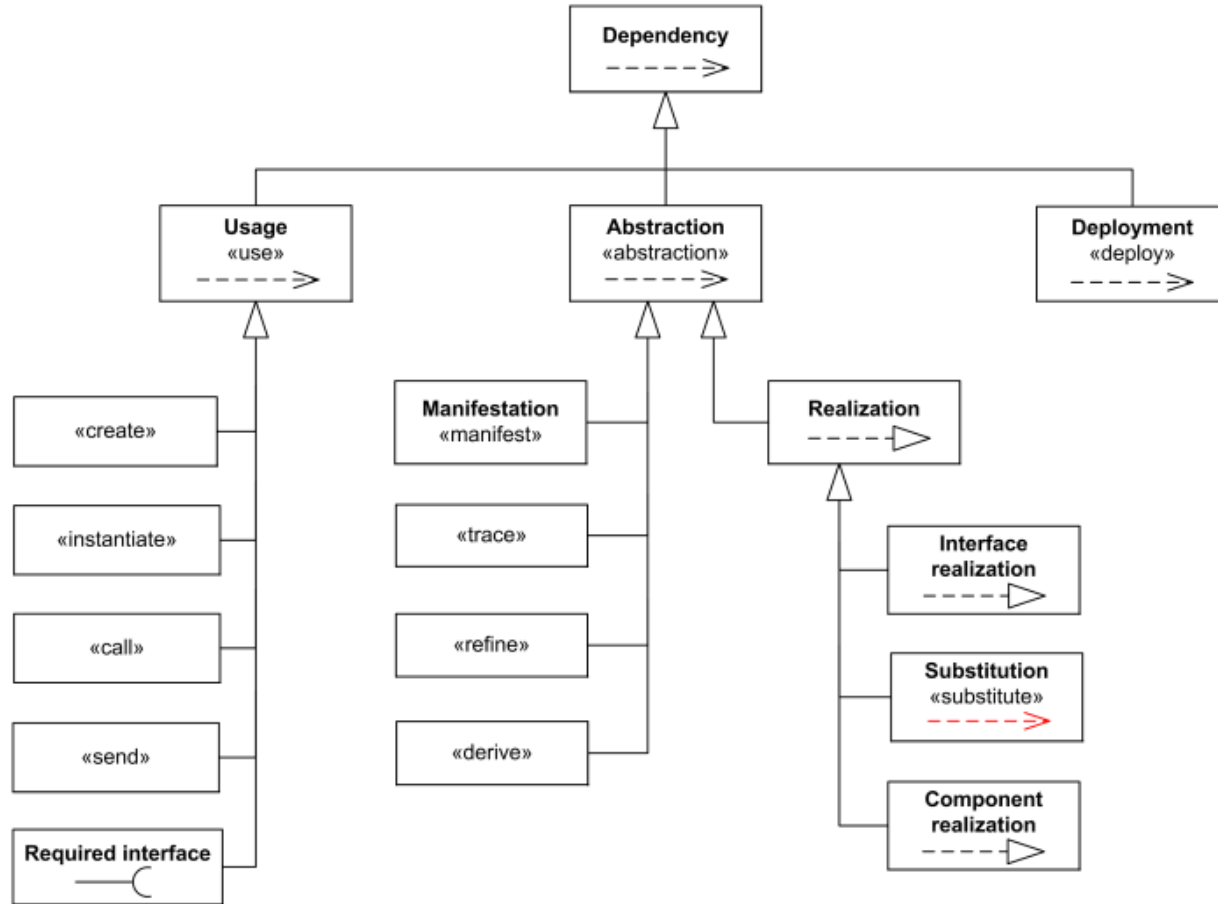
Cardinality – số phần tử tham gia quan hệ



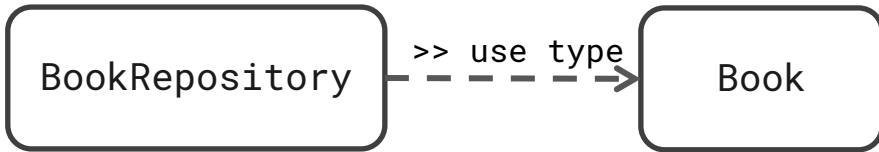
Ví dụ thực tế lớp CleanCode, có 2 đến 4 giảng viên hướng dẫn 12 đến 30 lập trình viên



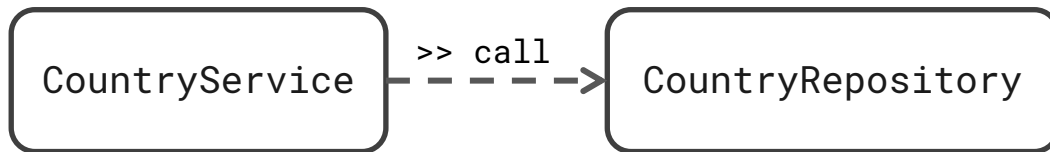
Dependency – quan hệ phụ thuộc nhiều biến thể



Depedency



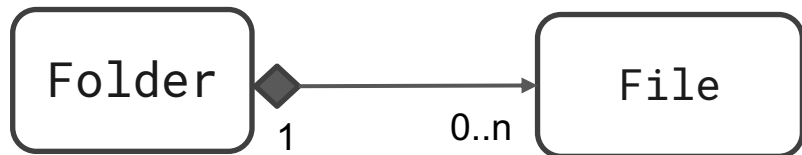
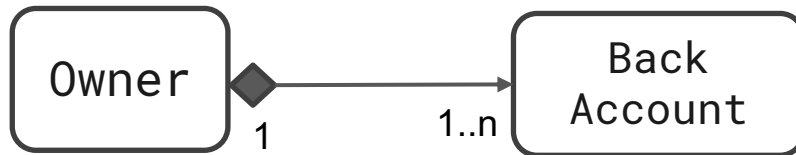
```
@Repository  
public interface BookRepository extends CrudRepository<Book, Long> {}
```



```
@Service
public class CountryService implements ICountryService {
    @Autowired
    private CountryRepository repository;

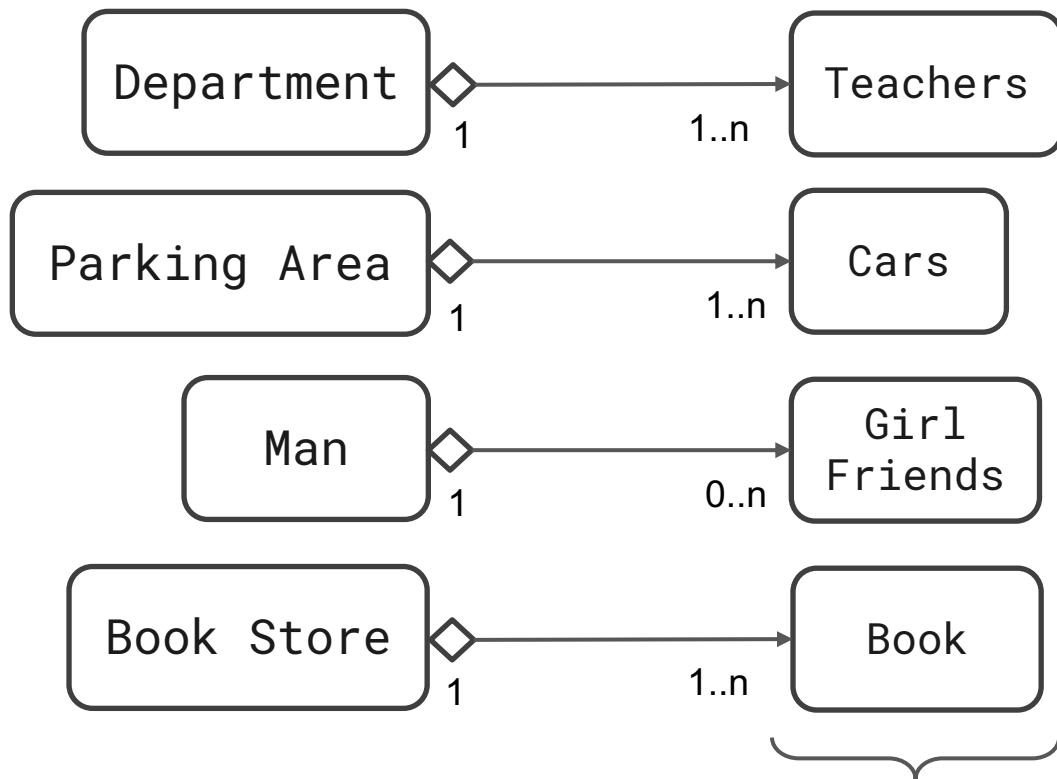
    @Override
    public List<Country> findAll() {
        return (List<Country>) repository.findAll();
    }
}
```

Composition



Không thể tồn tại, đứng riêng
độc lập thiếu vật chủ

Aggregation



Vẫn tồn tại, đứng riêng
độc lập không cần đối tượng chủ

Good Design

1. Code Reuse
2. Extensibility
3. Maintainability
4. Testability

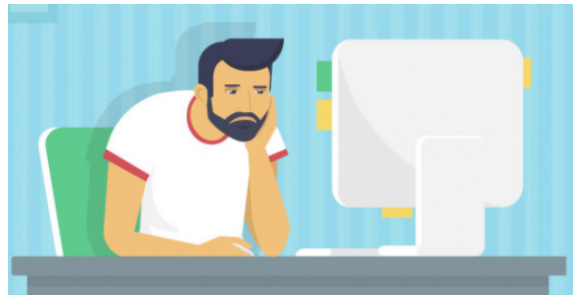
Interface vs Abstract Class

trong Java thôi nhé

Interface khác Class như thế nào?

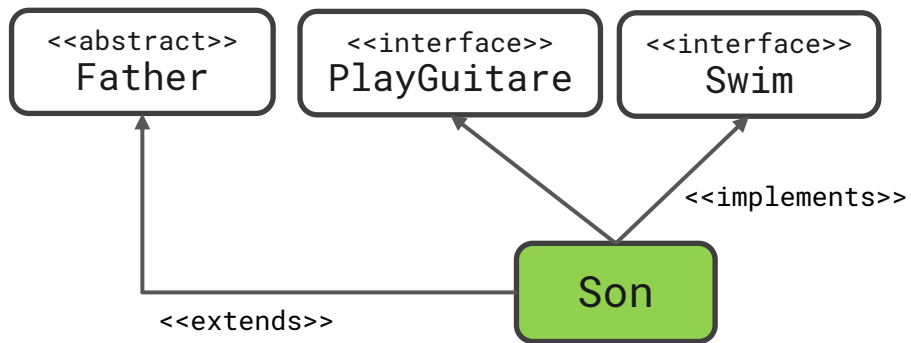
Một người chỉ có **duy nhất một ông bố** sinh học. Tuy nhiên anh ta có thể **học từ nhiều** giảng viên chơi đàn guitar, bơi lội, chơi piano và lập trình.

Bản chất của Interface **không phải là kế thừa** mà là **tuân thủ**.



```
abstract class Father {  
    abstract public void hasSomeTraits();  
}  
  
interface PlayGuitare {  
    public void play();  
}  
  
interface Swim {  
    public final double SWIM_PERFORM =  
10.0;  
    public void freeStyleSwim();  
    public void breastStrokeSwim();  
}
```

```
class Son extends Father implements  
PlayGuitare, Swim {  
    @Override  
    public void hasSomeTraits() {...}  
  
    @Override  
    public void play() {...}  
  
    @Override  
    public void breastStrokeSwim() {...}  
  
    @Override  
    public void freeStyleSwim() {...}  
}
```



Interface

abstract, default, static method

Chỉ có biến static hoặc final

Interface không thể kế thừa hay
tuân thủ Abstract Class

Interface mở rộng (**extends**) 01
Interface khác

Thành phần trong interface
luôn là public

Abstract Class

Có cả abstract và concrete method

Chứa mọi loại biến static vs non static
, final vs non final

Abstract Class có thể tuân thủ
Interface

Class mở rộng (**extends**) 01 abstract
class và tuân thủ (**implements**) 1-n
interface khác

Thành phần trong abstract class có
thể là private, public, protected

```
abstract class Shape {  
    protected String name = " ";  
    Shape(String name) {  
        this.name = name;  
    }  
  
    public void moveTo(double x, double y) {  
        System.out.println(this.name + " " + "has been moved to" +  
            " x = " + x + " and y = " + y);  
    }  
  
    public abstract double area();  
    public abstract void draw();  
}
```

```
interface Shape {  
    public void moveTo(double x, double y);  
    public double area();  
    public void draw();  
    public static final double GOLDEN_RATIO = 1.618;  
    public static double SQUARE_RATIO = 1.0;  
}
```

```
interface Fly {  
    public void fly();  
}  
  
interface BetterFly extends Fly {  
    public void glide();  
}  
  
class Eagle implements BetterFly {  
  
    @Override  
    public void glide() {  
    }  
  
    @Override  
    public void fly() {  
    }  
}
```

Interface mở rộng (extends) Interface

Class tuân thủ (implements) Interface

Nested Interface

Nằm trong 1 Class

```
class Utility {  
    public interface Zip {  
        public void compress();  
    }  
}  
  
interface Sevenzip extends Utility.Zip {  
    public void strongCompress();  
}
```

Nằm trong 1 Interface

```
interface Utility {  
    public interface Zip {  
        public void compress();  
    }  
}  
  
interface Sevenzip extends Utility.Zip {  
    public void strongCompress();  
}
```

Interface có thể nằm trong một class hay interface khác?

Dùng abstract class hay interface khi nào?

- Dùng abstract class khi vừa muốn tái sử dụng thuộc tính, phương thức vừa cho phép đè phương thức (**override method**)
- Có thể viết đè tất cả phương thức có access modifier **protected** và **public**
- Dùng interface, thì muốn 1 interface hay 1 class tuân thủ (**implements**) nhiều interface.
- Interface chỉ được chứa **public, static, final variables**.

Encapsulate What Often Changes !



Đóng gói những logic thay đổi với phần còn lại

Tìm ra những code block có logic thay đổi đóng gói lại bằng cách tách thành phương thức riêng:

- $\text{Lương net} = F (\text{lương gross} - \text{bảo hiểm xã hội} - \text{bảo hiểm y tế} + \text{thưởng} + \text{phụ cấp} + \text{over time} - \text{phạt} - \text{thuế})$
- $\text{Giá trị đơn hàng} = \text{Tổng} (\text{giá từng mặt hàng} * \text{số lượng}) - \text{khuyến mại} + \text{phí vận chuyển}$

```
public class Order {  
    String shipToCountry;  
    List<OrderItem> orderItems;  
  
    public float getOrderTotal() {  
        float total = 0;  
        for (OrderItem item: orderItems) {  
            total += item.quantity * item.unitPrice;  
        }  
  
        if (shipToCountry.equals("US")) {  
            total += total * 0.07;  
        } else if (shipToCountry.equals("EU")) {  
            total += total * 0.2;  
        }  
        return total;  
    }  
}
```

Hàm tính thuế mỗi lúc một phức tạp liên tục thay đổi sẽ ảnh hưởng đến phần còn lại.

Việc viết unit test cũng khó khăn vì phải bao nhiêu tham số, trường hợp

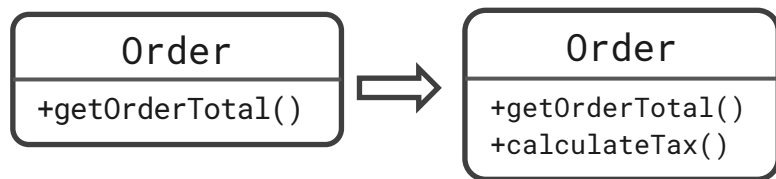
Cách xử lý

1. Chuyển phần tính thuế vào một phương thức nằm trong class Order
2. Chuyển phần tính thuế vào một class riêng TaxCalculator

Khi nào refactor ra phương thức mới, khi nào refactor ra class mới?
Xem trang tiếp theo nhé !

Refactor to new method

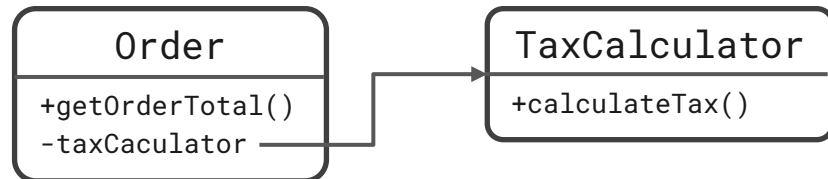
- Phương thức mới vẫn thuộc domain của class hiện tại: sử dụng nhiều thuộc tính, phương thức của class hiện tại
- Vẫn đảm bảo tính đóng gói, trừu tượng, hợp lý trong cùng 1 chủ đề



Khi calculateTax() vẫn trong domain của Order

Refactor to new class

- Phương thức mới phù hợp nằm trong một domain mới. Ít ràng buộc với class hiện tại sẽ tốt hơn
- Cần thêm nhiều tham số, thuộc tính không liên quan đến class hiện tại
- Sau khi tách ra giúp class hiện tại trở nên gọn gàng hơn, không ô nhiễm quá nhiều việc không liên quan

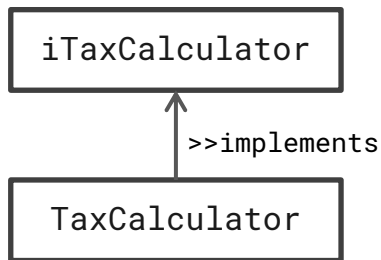


Khi calculateTax() nên nằm ở domain mới thì dễ bảo trì hơn
Biến động của TaxCalculator không ảnh hưởng đến Order !

```
public class Order {  
    String shipToCountry;  
    List<OrderItem> orderItems;  
  
    ITaxCalculator taxCalculator = new TaxCalculator();  
  
    public Order (List<OrderItem> orderItems, String shipToCountry) {  
        this.shipToCountry = shipToCountry;  
        this.orderItems = orderItems;  
    }  
  
    public float getOrderTotal() {  
        float total = 0;  
        for (OrderItem item: orderItems) {  
            total += item.quantity * item.unitPrice;  
        }  
        total = taxCalculator.calculateTax(total, shipToCountry);  
        return total;  
    }  
}
```

Chuyển thành một class mới, kỹ thuật DI nói sau nhé





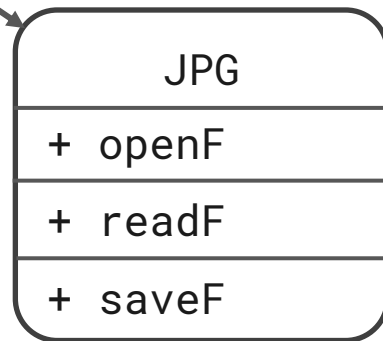
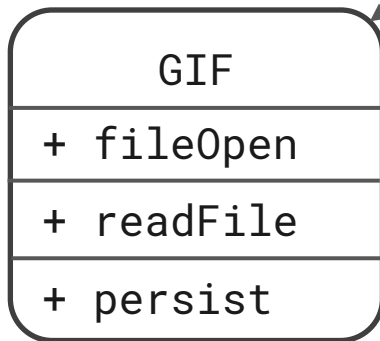
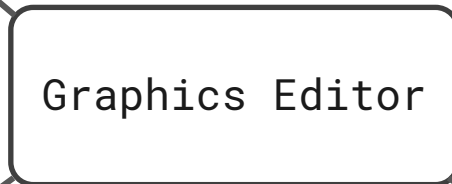
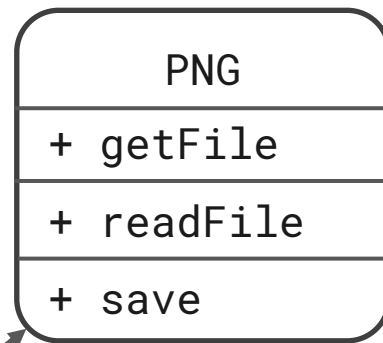
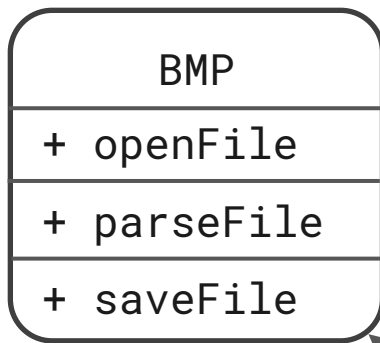
```
public interface ITaxCalculator {  
    public float caculateTax(float total, String shipToCountry);  
}
```

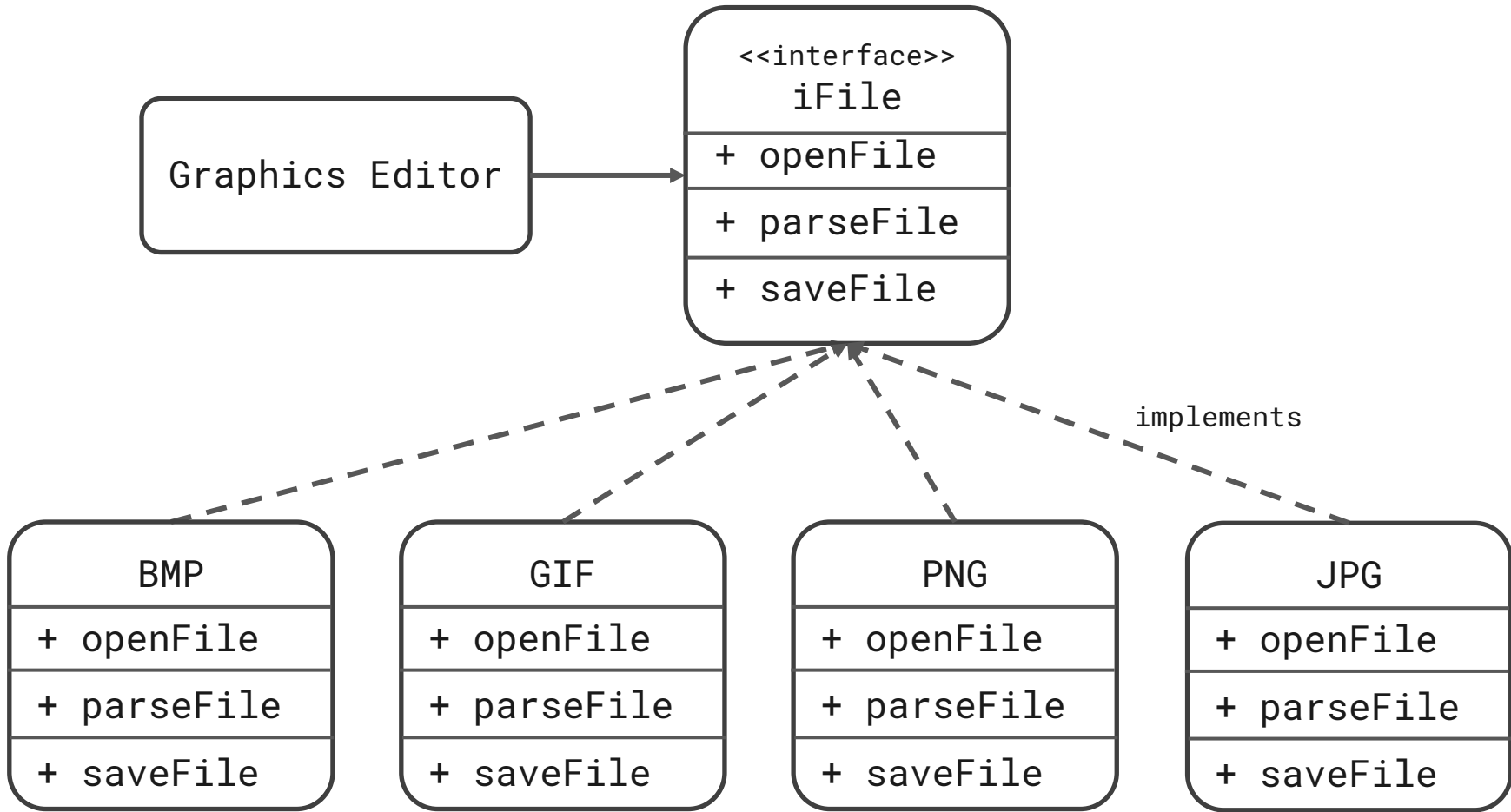
```
package orderafter;
```

```
public class TaxCalculator implements ITaxCalculator{  
    public float caculateTax(float total, String shipToCountry) {  
        if (shipToCountry.equals("US")) {  
            total += total * 0.07; //US sales tax  
        } else if (shipToCountry.equals("EU")) {  
            total += total * 0.2; //EU VAT  
        }  
        return total;  
    }  
}
```

Program to an Interface, not to an Implementation

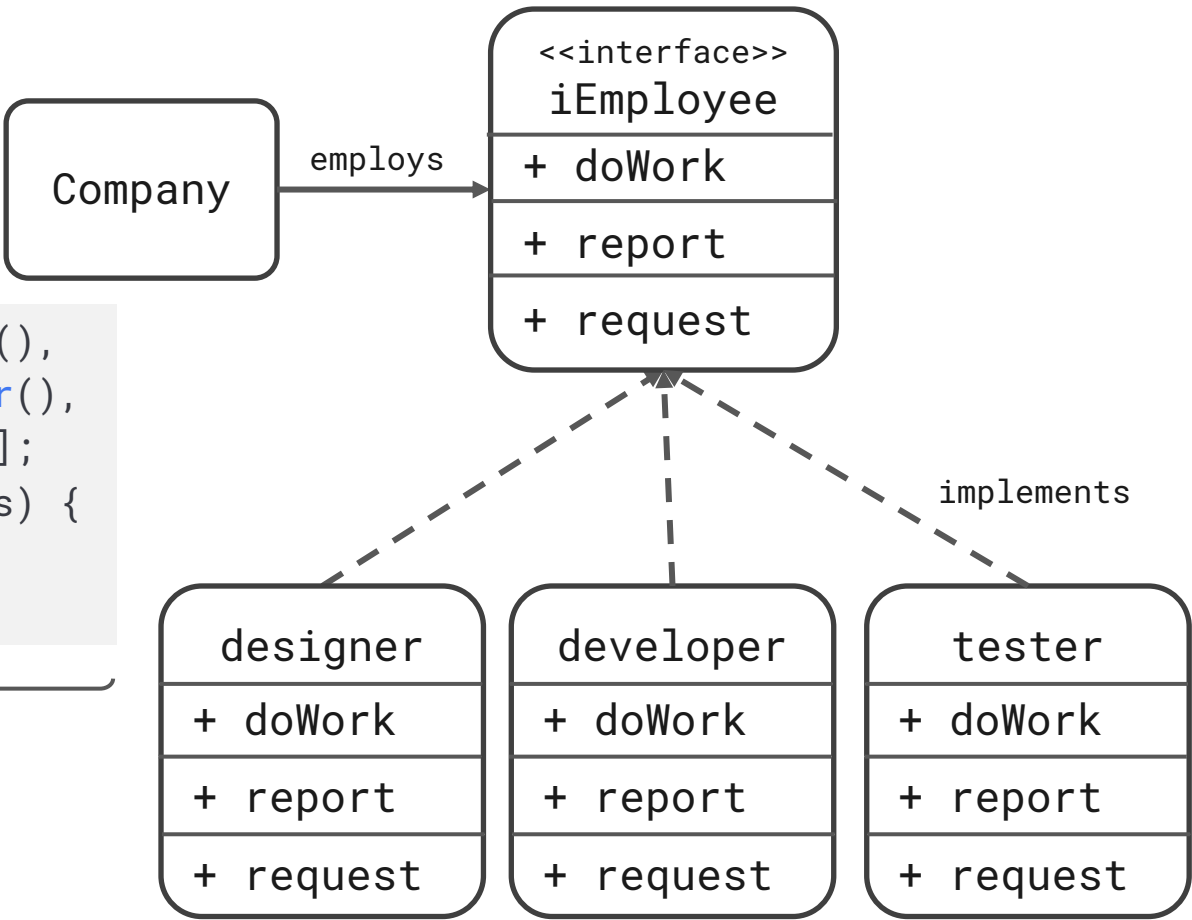


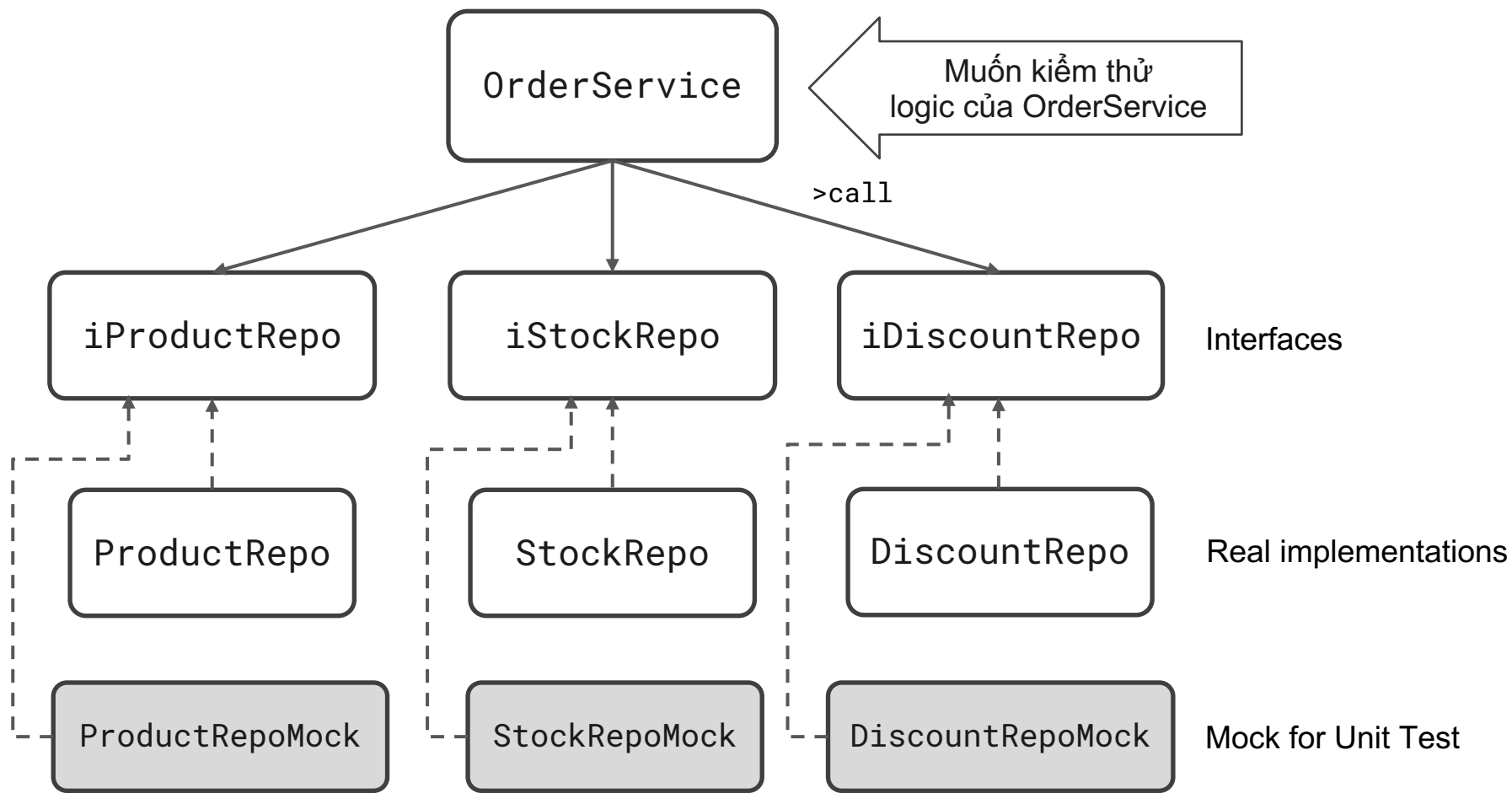




```
var employees = [new Designer(),  
                 new Developer(),  
                 new Tester()];  
for (iEmployee emp : employees) {  
    emp.doWork();  
}
```

Polymorphism – Đa hình





Mock object giả các object xịn do thể hiện interface giống nhau. Khi kiểm thử lắp Mock object vào.

Ví dụ Repository kết nối đến nhiều CSDL khác nhau

Trong ứng dụng quản lý nhân sự viết bằng Web SpringBoot, công ty làm ra 2 phiên bản: Free Edition và Premium Edition.

- **Free Edition** dùng SQLite chỉ hỗ trợ tối đa 3 tài khoản đồng thời
- **Premium Edition** dùng Postgresql hỗ trợ tối đa 500 tài khoản đồng thời.

Nếu phải viết lại toàn bộ logic truy cập CSDL cho SQLite và Premium thì quá tốn kém thời gian 🥲🥲🥲

JPA đề xuất Interface Repository thống nhất các thao tác truy cập vào các loại CSDL khác nhau.

Chức năng của Repository

- Tập trung giao tiếp với CSDL
- Thực hiện tác vụ: Thêm – Sửa – Tìm – Liệt kê – Xóa – Xử lý batch, Phân trang, Sắp xếp
- Cố gắng giảm tối đa sự phụ thuộc vào loại CSDL cụ thể (MySQL, Postgresql, H2, Oracle, MS-SQL)
- Vẫn cho phép viết custom query

```
@Query("SELECT u FROM User u WHERE u.status = 1")  
Collection<User> findAllActiveUsers();
```


Repository bản chất là Data Access Object

- Repository thay thế nhiều hàm phải viết thủ công ở DAO
- Xem repository/Dao.java và repository/UserDao.java để hiểu cách viết thủ công.

```
@Repository
public interface UserRepository extends CrudRepository<User, Integer> {
    List<User> findByNameContaining(String term);

    List<User> f
    }
    findByEmail(String email);
    findById(Integer id);
    findByName(String name);
    findByPhone(String phone);
```

Class of Entity

Type of ID

`CrudRepository<User, Integer>`

The diagram illustrates the mapping of generic types in the `CrudRepository` interface to their corresponding annotations. A light gray rectangular box at the bottom contains the text `CrudRepository<User, Integer>` in orange. Two dark gray rectangular boxes are positioned above this box. The left box, labeled 'Class of Entity', has a line pointing to the `User` type argument. The right box, labeled 'Type of ID', has a line pointing to the `Integer` type argument.

Interface with generic parameter

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity); //Lưu (create, update)

    T findOne(ID primaryKey); // Tìm theo ID

    Iterable<T> findAll(); // Liệt kê tất cả

    Long count(); // Đếm phần tử

    void delete(T entity); //Xoá

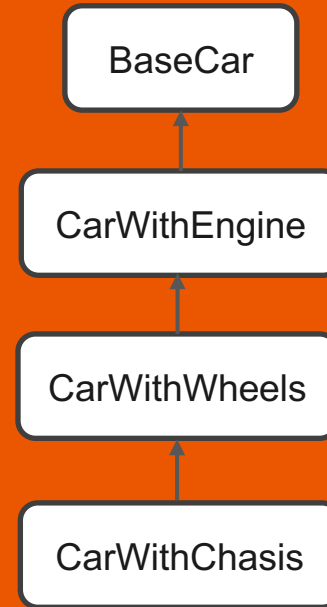
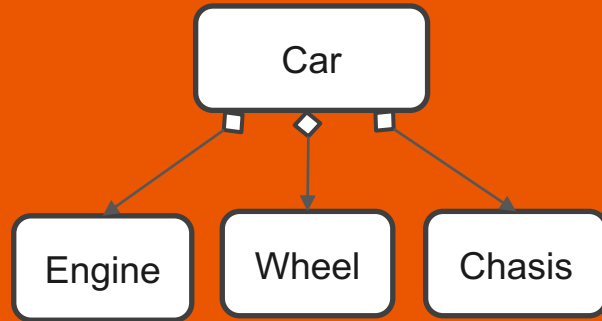
    boolean exists(ID primaryKey); // Kiểm tra tồn tại
}
```

Tóm lại

Hãy lập trình lập trình tuân thủ theo Interface chứ không lập trình thể hiện tùy ý.

Interface là quy ước giúp các đối tượng phải tuân thủ để có thể hoán đổi cho nhau hoặc gọi nhau dễ dàng.

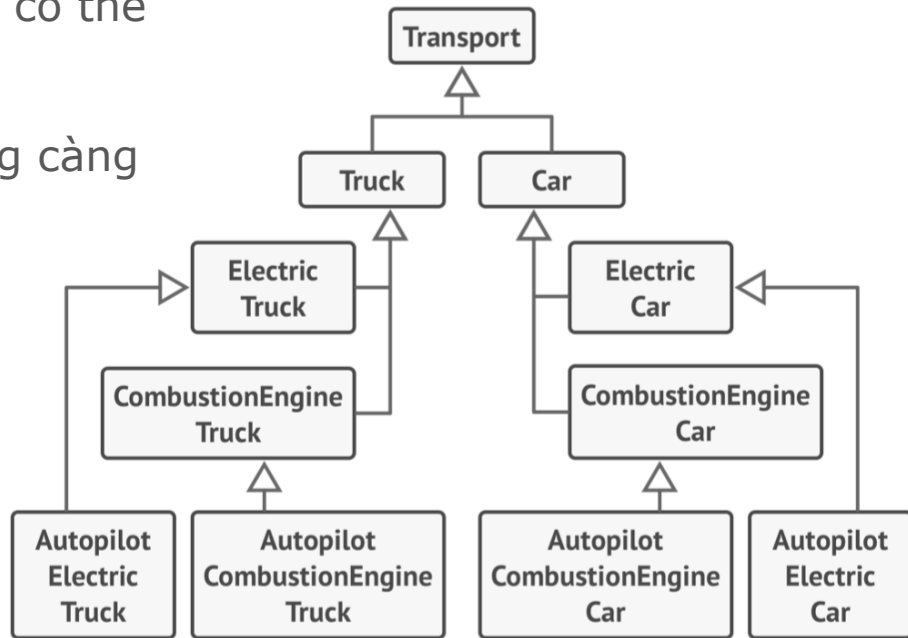
Favor Composition Over Inheritance



Vấn đề kế thừa

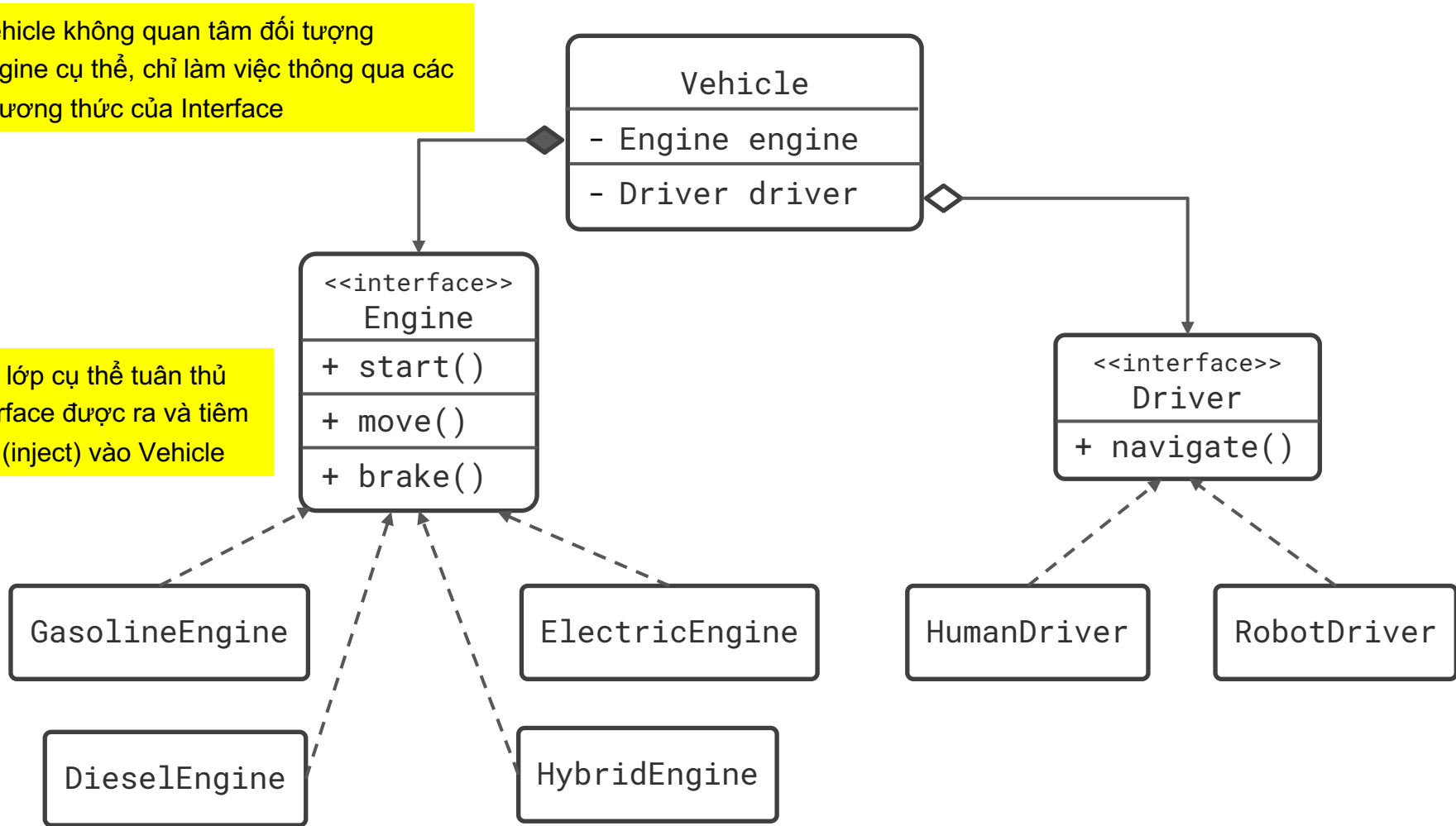
- Lớp con phải thể hiện đủ tất cả các phương thức trừu tượng của lớp cha
- Lớp con sẽ nhìn thấy hết tất cả thuộc tính không private của lớp cha
- Java không hỗ trợ đa kế thừa. Muốn mở rộng phải kế thừa chiều sâu: con > cháu > chắt > chít. Ngược lại một class có thể tuân thủ nhiều interface !
- Độ sâu kế thừa càng lớn, thì lớp dưới cùng càng lệ thuộc (nhận thuộc tính, phương thức của lớp tổ tiên)

Tái sử dụng code bằng
kế thừa hoá ra rắc rối !
Số lượng class nhiều lên,
Độ sâu kế thừa tăng lên



Vehicle không quan tâm đối tượng engine cụ thể, chỉ làm việc thông qua các phương thức của Interface

Các lớp cụ thể tuân thủ Interface được ra và tiêm vào (inject) vào Vehicle



4 lợi điểm của Composition so với Inheritance

1. Composition cho phép chứa các thuộc tính tuân thủ 1-nhiều interface khác nhau. Mức độ đa dạng hoá tăng, mà không tăng chiều sâu kế thừa
2. Do các thuộc tính trong Composition tuân thủ Interface nên có thể hoán đổi chúng. Thuận lợi cho kiểm thử từng thành phần (unit test dùng mock) hoặc cấu hình thành phần (dependency injection)
3. Cả Inheritance và Composition đều hỗ trợ tái sử dụng code. Những Inheritance phá vỡ tính đóng gói (encapsulation), nhưng composition bảo toàn tính đóng gói
4. Nhiều Design Patterns sử dụng Composition + Interface + Generic thay vì chỉ sử dụng Inheritance

Generic và Annotation

Generic và Annotation

- Generic và Annotation là 2 tính năng ngôn ngữ lập trình không trong lý thuyết lập trình hướng đối tượng căn bản nhưng rất mạnh Java, C#, C++, Dart
- **Generic** để động hoá kiểu biến thành phần, tham số phương thức của class hay interface, hay còn gọi là Type parameter. Generic cho phép tái sử dụng code với nhiều kiểu khác nhau.
- **Annotation** để bổ xung thông tin metadata cho class, method, variable. Metadata này được sử dụng trong quá trình biên dịch (compile time) và cả lúc chạy (run time)

Generic hay thể này, mà ít anh em dùng qua !

1. Kiểm tra kiểu chặt hơn lúc biên dịch. Nhận được báo lỗi khi biên dịch để sửa rõ ràng tốt hơn là phát sinh lỗi khi chạy.
2. Loại bỏ việc ép kiểu.
3. Cho phép lập trình viên dùng cho một thuật toán, logic cho nhiều kiểu khác nhau.

Generic khai báo kiểu rõ ràng, kiểm tra kiểu chặt chẽ

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0); //Cast !
```

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0); // no cast
```

```
public class GenericMethodTest {  
    public static <E> void printArray( E[] inputArray ) {  
        for(E element : inputArray) {  
            System.out.printf("%s ", element);  
        }  
        System.out.println();  
    }  
}
```

Hàm in mảng của bất kỳ kiểu gì

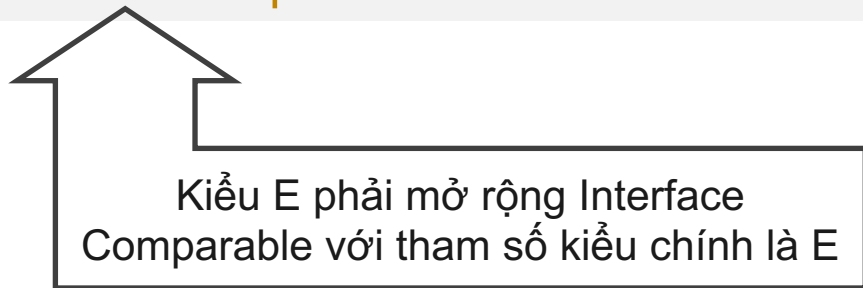
```
public static void main(String args[]) {  
    // Create arrays of Integer, Double and Character  
    Integer[] intArray = { 1, 2, 3, 4, 5 };  
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };  
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };  
  
    System.out.println("Array integerArray contains:");  
    printArray(intArray);    // pass an Integer array  
  
    System.out.println("\nArray doubleArray contains:");  
    printArray(doubleArray);    // pass a Double array  
  
    System.out.println("\nArray characterArray contains:");  
    printArray(charArray);    // pass a Character array  
}
```

```
private <E> void swap(E[] a, int i, int j) {  
    if (i != j) {  
        E temp = a[i];  
        a[i] = a[j];  
        a[j] = temp;  
    }  
}
```

Ví dụ hàm Selection Sort
Cùng một thuật toán nhưng áp
dụng cho nhiều kiểu khác nhau

```
public <E extends Comparable<E>> void selectionSort(E[] a) {  
    for (int i = 0; i < a.length - 1; i++) {  
        int smallest = i; // find index of smallest element  
        for (int j = i + 1; j < a.length; j++) {  
            if (a[j].compareTo(a[smallest]) <= 0) {  
                smallest = j;  
            }  
        }  
        swap(a, i, smallest); // swap smallest to front  
    }  
}
```

```
public <E extends Comparable<E>> void selectionSort(E[] a)
```



Kiểu E phải mở rộng Interface
Comparable với tham số kiểu chính là E

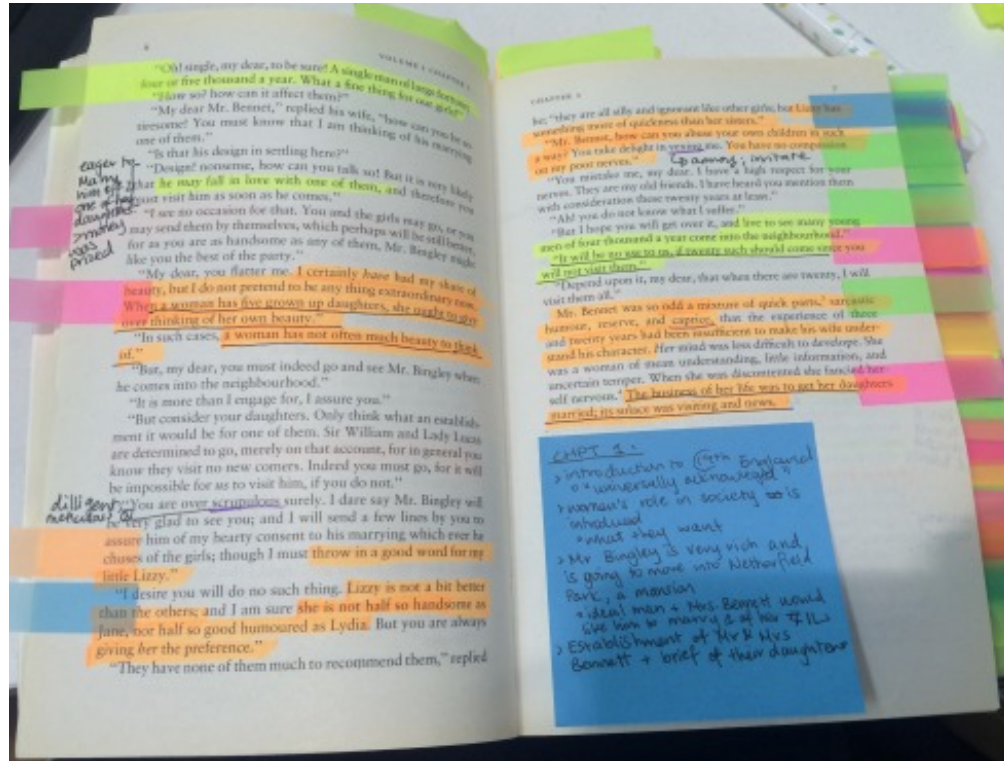
```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

```
public class Trainer<T> {  
    private List<T> trainees;  
    public Trainer(List<T> trainees) {  
        this.trainees = trainees;  
    }  
  
    @Override  
    public String toString() {  
        var sb = new StringBuilder("I coach following students:\n");  
        for (T trainee : this.trainees) {  
            sb.append(trainee.toString());  
            sb.append("\n");  
        }  
        return sb.toString();  
    }  
}
```

Generic Class

chú ý có thể truyền vào nhiều tham số kiểu <T, K, V>

Annotation để bổ xung thông tin metadata cho class, method, variable. Metadata này được sử dụng trong quá trình biên dịch (compile time) và cả lúc chạy (run time)



Annotation – Retention Policy

RetentionPolicy cho biết annotation sẽ được sử dụng đọc ra ở lúc nào

- SOURCE: sẽ bị compiler bỏ qua
- CLASS: được ghi chú vào class file lúc biên dịch (compile), nhưng bỏ qua khi chạy (run time). Giá trị mặc định
- RUNTIME: được ghi chú vào class file lúc biên dịch, và đọc ra khi chạy C9

TARGET	DESCRIPTION
Annotation Type	Annotates another annotation
Constructor	Annotates a constructor
Field	Annotates a field, such as an instance variable of a class or an <u>enum constant</u>
Local variable	Annotates a local variable
Method	Annotates a method of a class
Module	Annotates a module (new in Java 9)
Package	Annotates a package
Parameter	Annotates a parameter to a method or constructor
Type	Annotates a type, such as a class, interfaces, annotation types, or enum declarations
Type Parameter	Annotates a type parameter, such as those used as formal generic parameters

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE) // on class level
public @interface Entity {
    String name();
}
```

Annotation Entity giống với JPA có một tham số (thực ra là phương thức) name

1

```
@Entity(name = "person")
class Person {
}
```

2

Định nghĩa class Person ánh xạ vào bảng person trong CSDL

```
try {
    Entity anno = Person.class.getAnnotation(Entity.class);
    System.out.println("Table name attribute: " + anno.name());
} catch (Exception e) {
    System.out.println(e);
}
```

Lấy giá trị tham số name ra lúc run time

3

Ví dụ phức tạp hơn thôi có nhiều tham số

```
import java.lang.annotation.*;

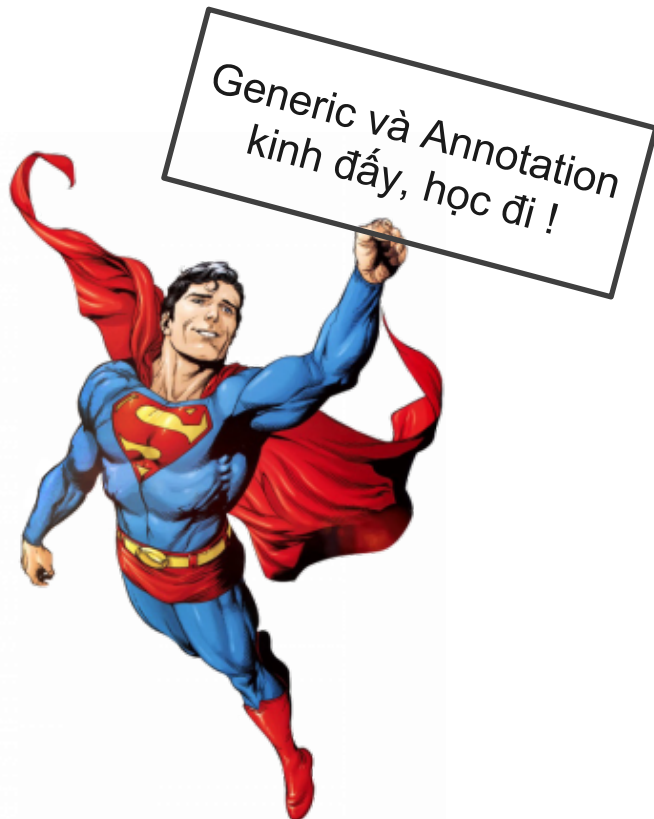
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE) // on class level
public @interface TesterInfo {
    public enum Priority {
        LOW, MEDIUM, HIGH
    }
    Priority priority() default Priority.MEDIUM;
    String[] tags() default "LogicTest";
    String createdBy() default "Techmaster";
    String lastModified() default "2020/12/03";
}
```

Liệt kê Annotation param bằng reflection

```
public static void listAnnotationParams()  
    throws IllegalAccessException, InvocationTargetException {  
    for (Annotation annotation : ZooCleaner.class.getAnnotations()) {  
        Class<? extends Annotation> type = annotation.annotationType();  
        System.out.println("Annotation name: " + type.getName());  
  
        for (Method method : type.getDeclaredMethods()) {  
            Object value = method.invoke(annotation, (Object[]) null);  
            System.out.println(" " + method.getName() + ": " + value);  
        }  
    }  
}
```

Tại sao phải tìm hiểu kỹ Annotation?

- Bạn có thấy SpringBoot sử dụng rất nhiều Annotation để đánh dấu các class, method, variable không?
- Annotation dùng để cấu hình, bổ xung thông tin phụ trợ cho class, interface, method, variable thay thế cho file cấu hình XML ngoài



Bài tập thực hành

Thiết kế class, interface ứng dụng Graphics Editor

Graphics Editor (GE) gồm các chức năng:

1. Có Layer Manager để quản lý hiển thị, ẩn, thay đổi opacity các layer
2. Vẽ đường thẳng, hình chữ nhật, vuông, ellipse, tròn, tam giác, đa giác.
3. Vẽ nét đậm nhạt, chọn màu cho nét, chọn màu để tô
4. Có thể transform: scale, rotate, skew
5. Có Extension Manager cho phép lập trình viên ngoài bổ xung các hình vẽ mới.
6. Có thể viết chữ với font style, font size, bold-italic
7. Có Font Manager để quét tất cả font chữ có trong hệ điều hành
8. Có File Manager để đọc, ghi file jpeg, gif, png

**Nếu bạn thiết kế tốt, bạn đủ điều kiện thi
vào Adobe Corp USA rồi đó**



Thiết kế web framework

Một web framework hiện đại cần hỗ trợ những chức năng sau đây:

1. Request / response. Với response trả về HTML, JSON, XML...
2. Routing: group routing, wildcard routing
3. Cookie, session
4. Security: plain authentication, JWT, OAuth
5. Middleware manager: các middle ware tuần tự xử lý request
6. HTML view rendering sử dụng các template engine khác nhau
7. Database: hỗ trợ nhiều kết nối CSDL khác nhau

Thiết kế phần mềm trình chiếu CoolSlide

- Để thiết kế hãy tham khảo chức năng Google Slide hoặc PowerPoint hoặc <https://slides.com/>
- Slide Editor
- Slide Previewer
- Slide that contains many pages