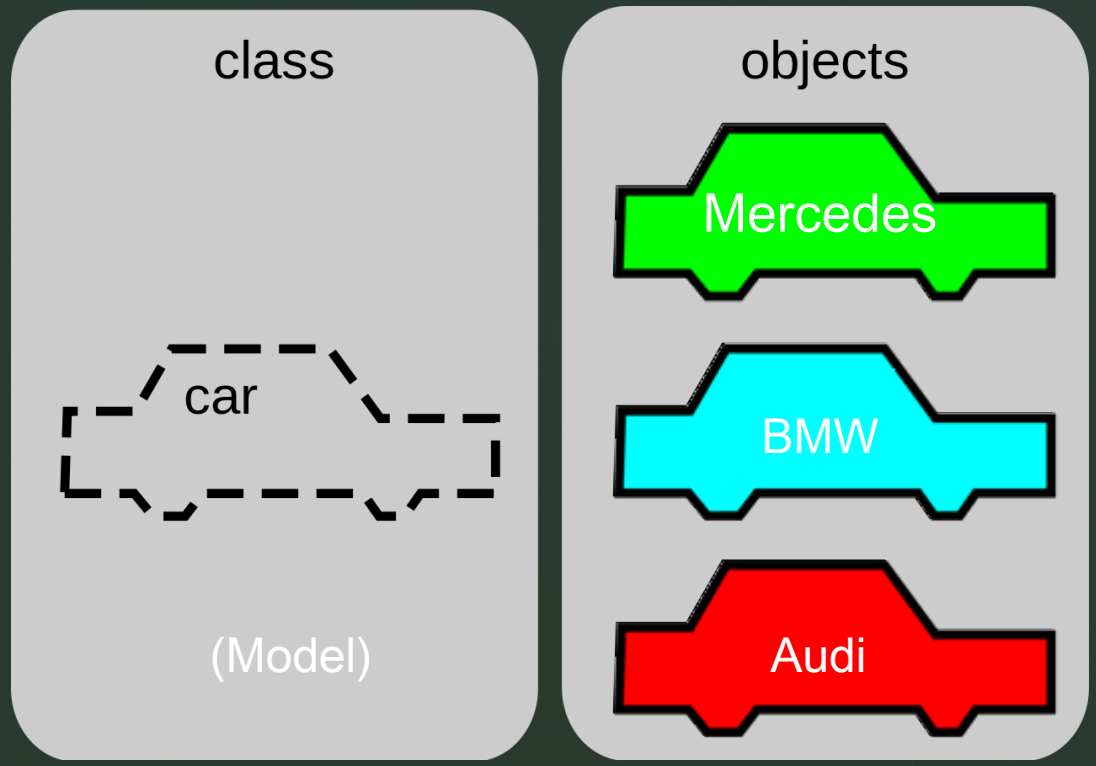

Lập trình hướng đối tượng

# Lập trình hướng đối tượng

# OOP – Tính chất

- Tính chất ?
  - Trừu tượng
  - Kế thừa
  - Đa hình
  - Đóng gói



# OOP & Class

- Class có thể kế thừa từ 1 hoặc nhiều interface
- Class chỉ được phép kế thừa 1 class khác
- Ex: Employee extends Person implements ICanWork, ICanMakeMoney, ICanBuyThing
- Ex: Bird extends Animal implements ICanFly, ICanEat, ICanSing

# Nâng cao: Nguyên lý SOLID

- SOLID LÀ GÌ?
- ÁP DỤNG CÁC NGUYÊN LÝ SOLID ĐỂ TRỞ THÀNH LẬP TRÌNH VIÊN CODE “CỨNG”

# Nâng cao: Nguyên lý SOLID

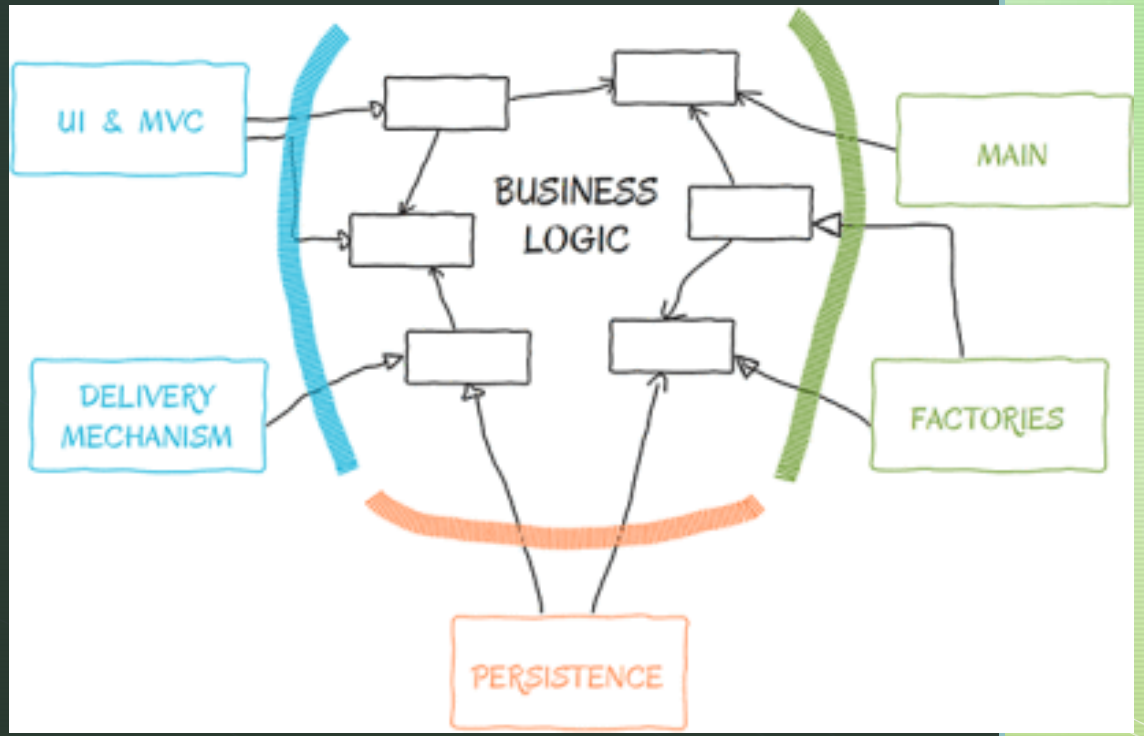
- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

# Single responsibility principle

- Một class chỉ nên giữ một trách nhiệm duy nhất
- Ví dụ 1 một con dao đa chức năng

# Single responsibility principle

- Áp dụng SRP ta có thể tách nó ra làm kéo, dao, mở nút chai,… riêng biệt là xong, cái gì hư chỉ cần sửa cái đấy.
- Với code cũng vậy, ta chỉ cần thiết kế các module sao cho đơn giản, một module chỉ có 1 chức năng duy nhất!



# Open/closed principle

- **Dễ mở rộng**: Có thể dễ dàng nâng cấp, mở rộng, thêm tính năng mới cho một module khi có yêu cầu.
- **Khó sửa đổi**: Hạn chế hoặc cấm việc sửa đổi source code của module sẵn có.

- Có thể thoải mái mở rộng 1 module, nhưng hạn chế sửa đổi bên trong module đó

# Liskov substitution principle

- Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình

```
1  public class Bird {
2    public virtual void Fly() { Console.Write("Fly"); }
3  }
4  public class Eagle : Bird {
5    public override void Fly() { Console.Write("Eagle Fly"); }
6  }
7  public class Duck : Bird {
8    public override void Fly() { Console.Write("Duck Fly"); }
9  }
10 public class Penguin : Bird {
11   public override void Fly() { throw new NoFlyException(); }
12 }
13
14 var birds = new List { new Bird(), new Eagle(), new Duck(), new Penguin() };
15 foreach(var bird in birds) bird.Fly();
16 // Tới pengiun thì lỗi vì cánh cụt quăng Exception
```

# Interface Segregation Principle

- Nguyên tắc phân chia giao diện
- Áp dụng ISP, ta sẽ chia interface này ra thành nhiều interface nhỏ, các class **chỉ cần implement những interface có chức năng mà chúng cần**
- Để thiết kế một hệ thống linh hoạt, dễ thay đổi, các module của hệ thống nên **giao tiếp với nhau thông qua interface**

# Interface Segregation Principle

- Để có thể phân tách một interface lớn thành các interface nhỏ một cách hợp lý, xem lại Single Responsibility Principle
- Việc tách ra nhiều interface có thể làm *tăng số lượng interface, tăng số lượng class*, ta cần cân nhắc lợi hại trước khi áp dụng

# Interface Se

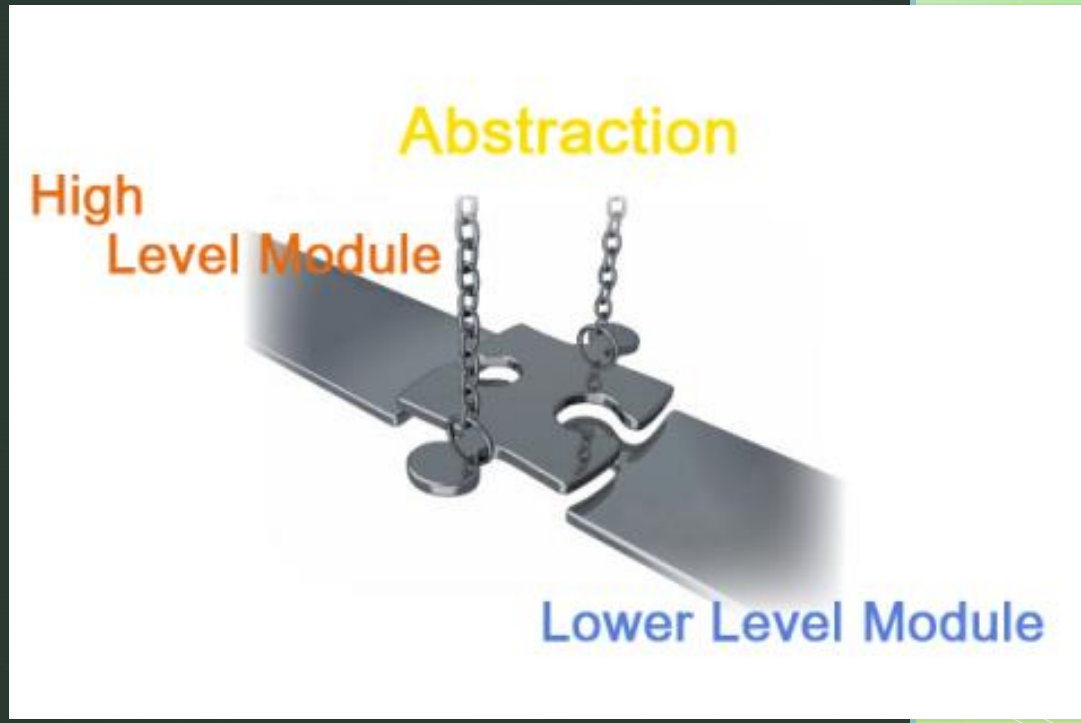
```
1  public interface IAnimal {
2    void Eat();
3    void Drink();
4    void Sleep();
5  }
6  public class Dog : IAnimal {
7    public void Eat () {}
8    public void Drink () {}
9    public void Sleep () {}
10 }
11 public class Cat : IAnimal {
12   public void Eat () {}
13   public void Drink () {}
14   public void Sleep () {}
15 }
```

```
public interface IAnimal {
  void Eat();
  void Drink();
  void Sleep();
}
public interface IBird {
  void Fly();
}
public interface IFish {
  void Swim();
}

// Các class chỉ cần kế thừa những interface có chúc năng chúng cần
public class Dog : IAnimal {
  public void Eat() {}
  public void Drink() {}
  public void Sleep() {}
}
public class FlappyBird: IAnimal, IBird {
  public void Eat() {}
  public void Drink() {}
  public void Sleep() {}
  public void Fly() {}
}
```

# Dependency Inversion Principle

- Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả 2 nên phụ thuộc vào abstraction
- Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation



# Tham khảo

- Giải thích chi tiết và ví dụ cụ thể:
- https://toidicodedao.com/2015/03/24/solid-la-gi-ap-dung-cac-nguyen-ly-solid-de-tro-thanh-lap-trinh-vien-code-cung