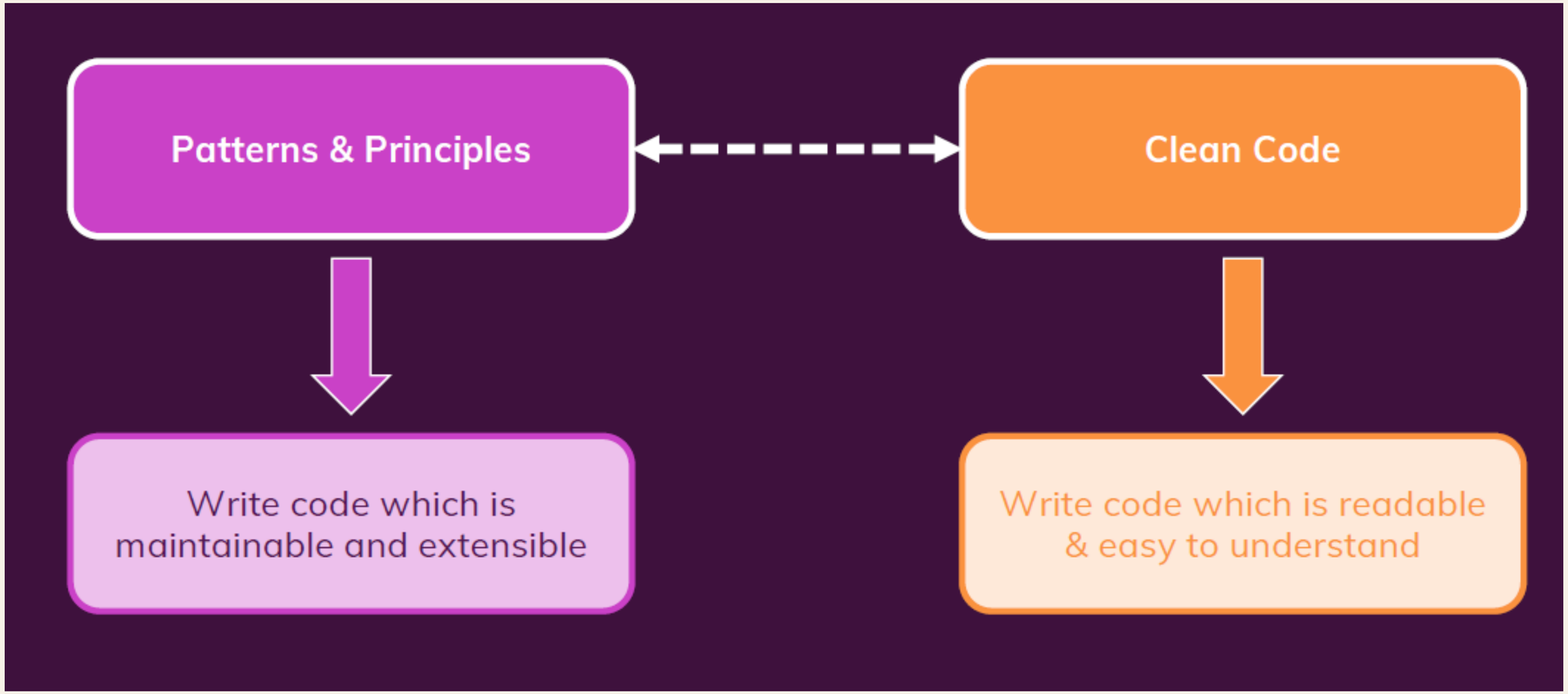
SOLID Principles

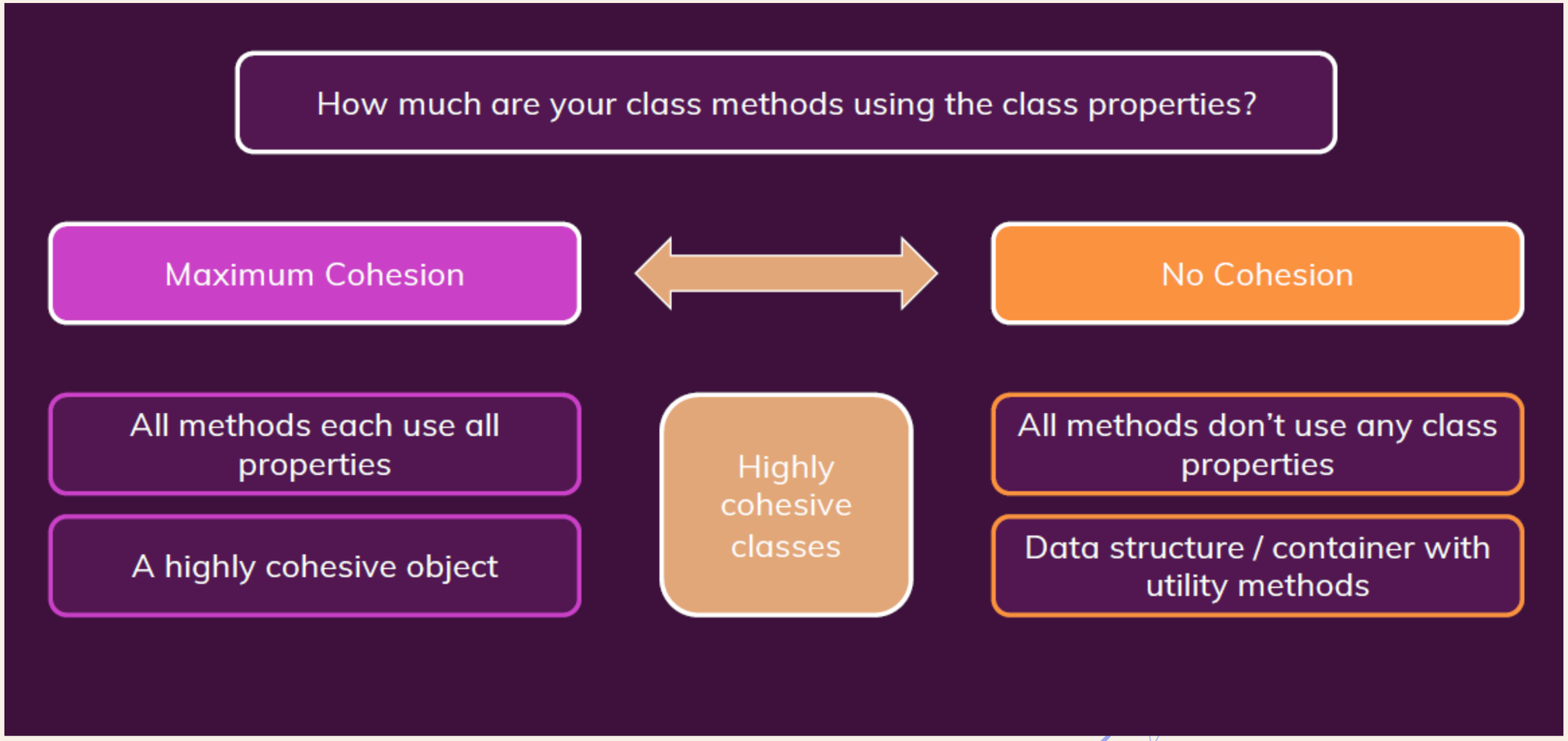
Ashik George

Clean Code and Principle Patterns

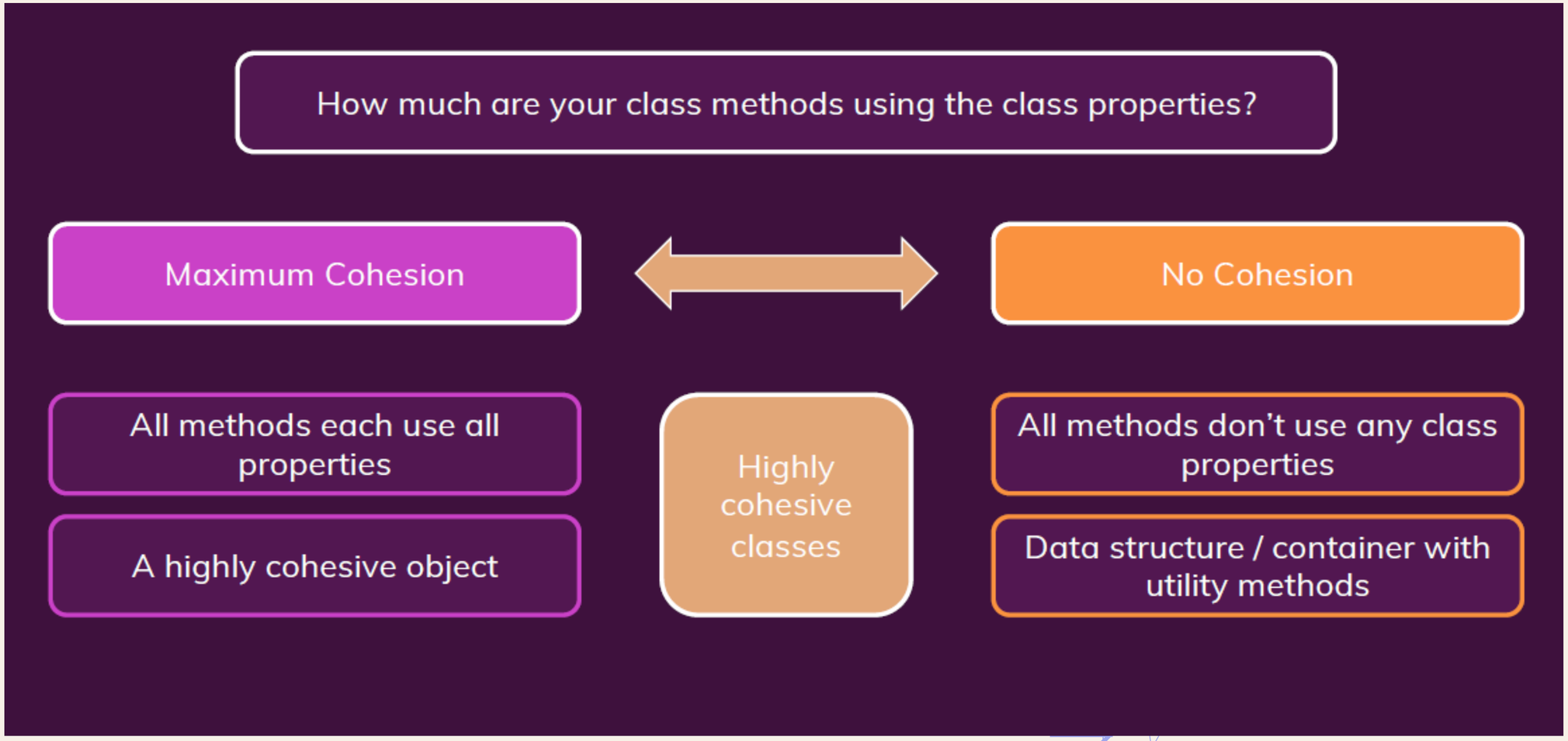


Classes should be small

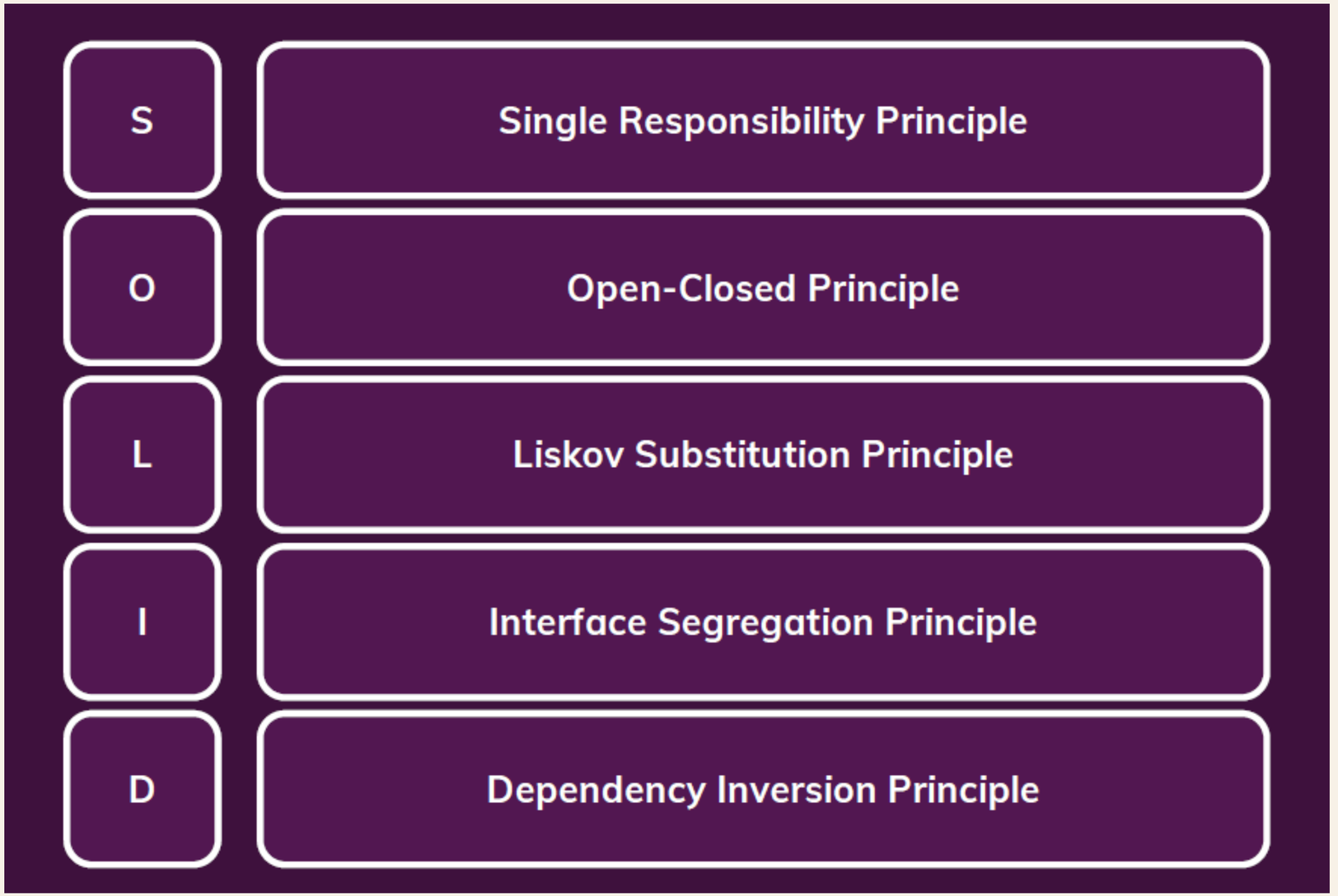
Cohesion


How much are your class methods using the class properties?
Maximum Cohesion
No Cohesion
All methods each use all properties
A highly cohesive object
Highly cohesive classes
All methods don't use any class properties
Data structure / container with utility methods

Cohesion


How much are your class methods using the class properties?
Maximum Cohesion
No Cohesion
All methods each use all properties
A highly cohesive object
Highly cohesive classes
All methods don't use any class properties
Data structure / container with utility methods

SOLID PRINCIPLES



Single Responsibility Principle

Classes should have a single responsibility – a class shouldn't change for more than one reason.

Single Responsibility Principle

```
public class Shop {
    private PlayerInventory playerInventory;

    public Shop(PlayerInventory playerInventory) {
        this.playerInventory = playerInventory;
    }
}
```

```
class Shop {
        private PlayerInventory playerInventory;

        buyDiamonds(data:any){};
        buyHearts(data:any){};
    }
```

Open Closed Principle

A class should be open for extension but closed for modification.

Open Closed Principle

```
class Shop {
        private PlayerInventory playerInventory;

        buyDiamonds(data:any){};
        buyHearts(data:any){};
    }
```

```
class Shop {
        private PlayerInventory playerInventory;
    }


class buyDiamondswithcoin extends shop implement purchase(){
        buyDiamond(data:any){
                ////
                }
}

class buyHeartswithDiamond extends shop implement purchase(){
        buyHearts(data:any){
                ////
                }
}
```

Liskov Substitution Principle

Objects should be replaceable with instances of their subclasses without altering the behavior.

Liskov Substitution Principle

```
class Shop {
        private PlayerInventory playerInventory;

        buyDiamonds(data:any){};
        buyHearts(data:any){};
    }
```

Liskov Substitution Principle

```
class Shop {
        private PlayerInventory playerInventory;
    }


class buyDiamondswithcoin extends shop implement purchase(){
        buyDiamond(data:any){
                ////
                }
}

class buyHeartswithDiamond extends shop implement purchase(){
        buyHearts(data:any){
                ////
                }
}
```

Interface Segregation Principle

Many client-specific interfaces are better than one general purpose interface.

Interface Segregation Principle

```
interface purchaseDiamond{
      buyDiamond()
}
interface purchaseHeart{
      buyHearts()
}
class buyDiamondswithcoin extends shop implement purchaseDiamond(){
        buyDiamond(data:any){
                ////
                 }
}

class buyHeartswithDiamond extends shop implement purchaseHeart(){
        buyHearts(data:any){
                ////
                 }
}
```

Dependency Inversion Principle

You should depend upon abstractions, not concretions.

Dependency Inversion Principle

```
class Shop {
        private PlayerInventory playerInventory;

        buyDiamonds(data:any){};
        buyHearts(data:any){};
    }
```

```
interface purchaseDiamond{
        buyDiamond()
}
interface purchaseHeart{
        buyHearts()
}
class buyDiamondswithcoin extends shop implement purchaseDiamond(){
          buyDiamond(data:any){
                  ////
                   }
}

class buyHeartswithDiamond extends shop implement purchaseHeart(){
          buyHearts(data:any){
                  ////
                   }
}
```