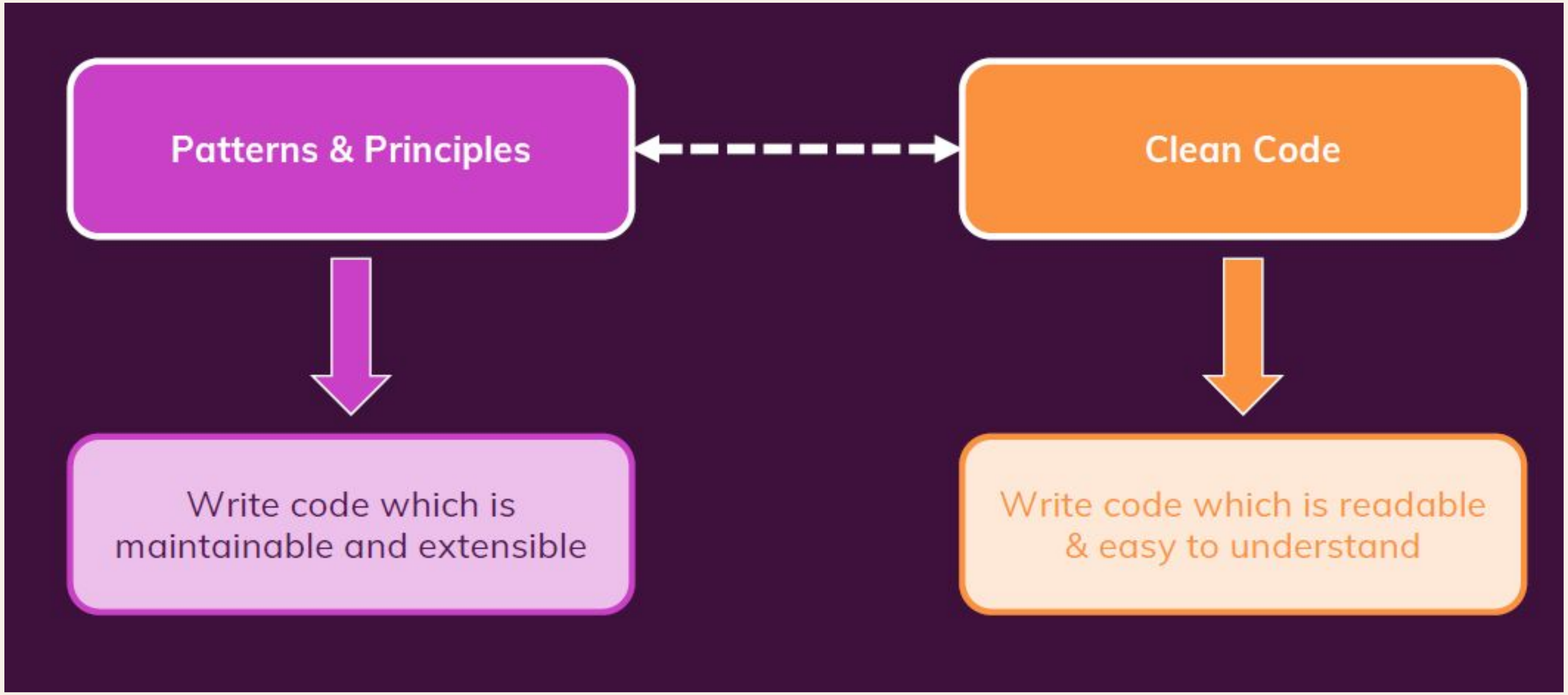
SOLID PRINCIPLES

Applying solid principles on Duolingo project

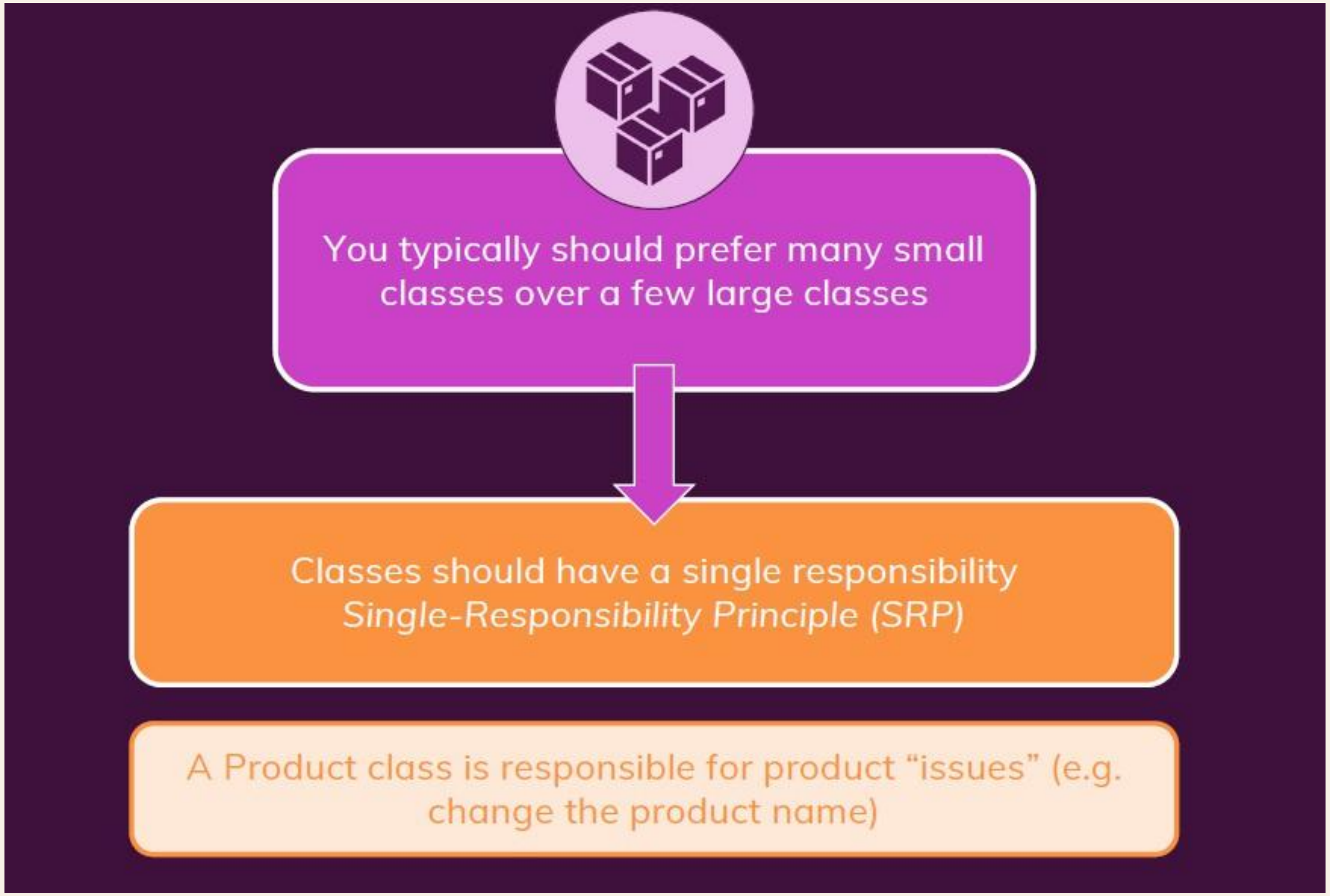
BY

JITTY TRESA THOMAS

Clean Code and Principle Patterns


Patterns & Principles
Clean Code
Write code which is maintainable and extensible
Write code which is readable & easy to understand

Classes should be small

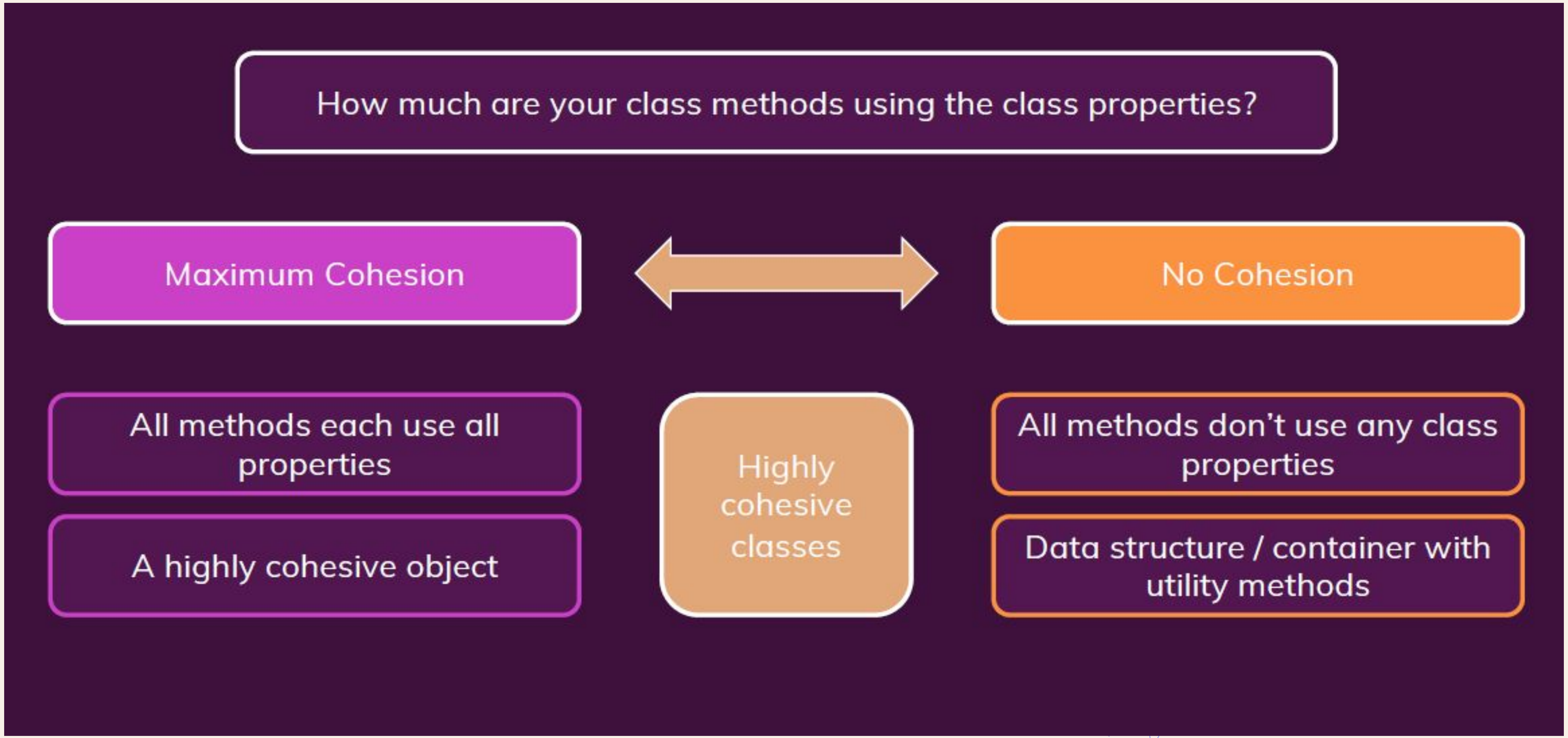


You typically should prefer many small classes over a few large classes

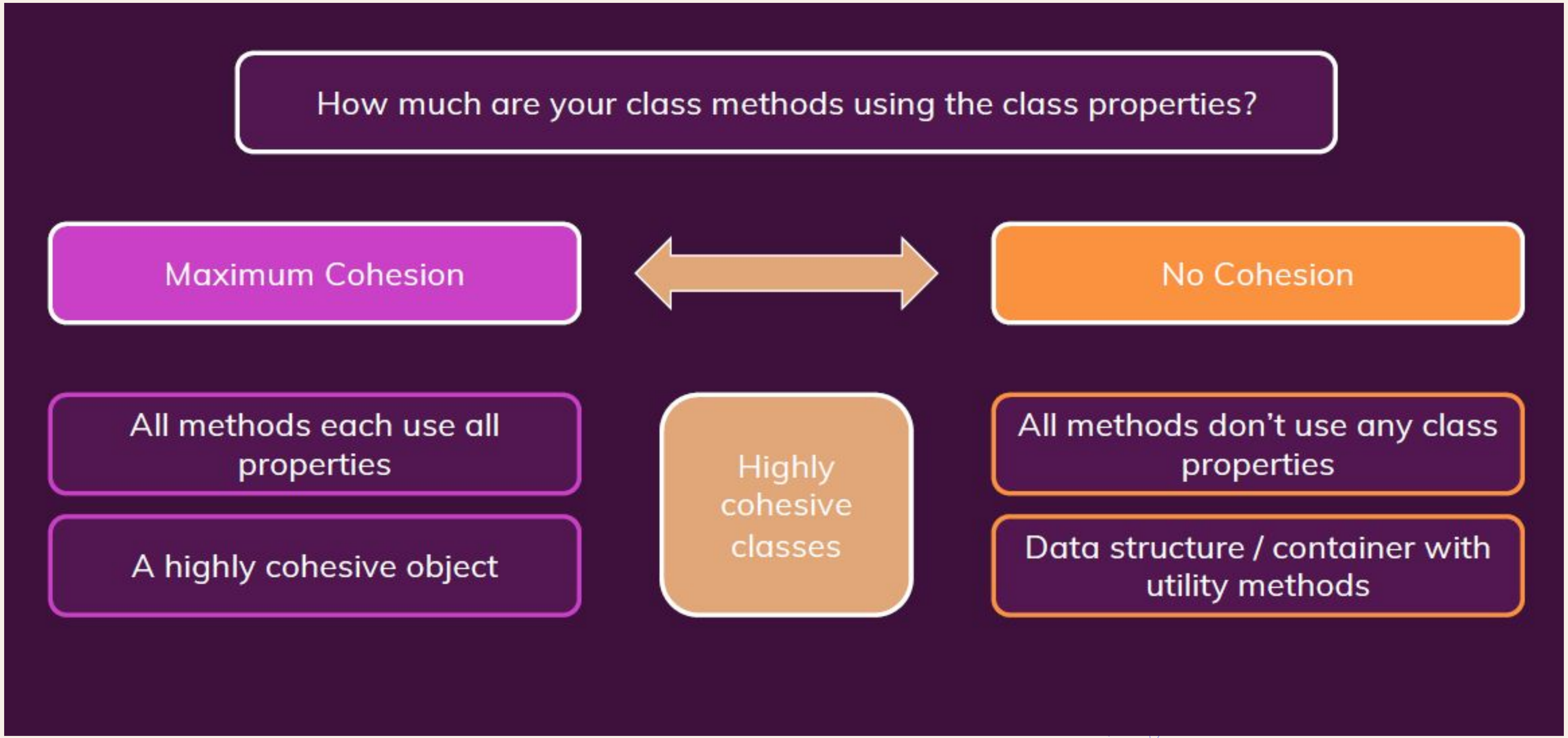
Classes should have a single responsibility
Single-Responsibility Principle (SRP)

A Product class is responsible for product “issues” (e.g. change the product name)

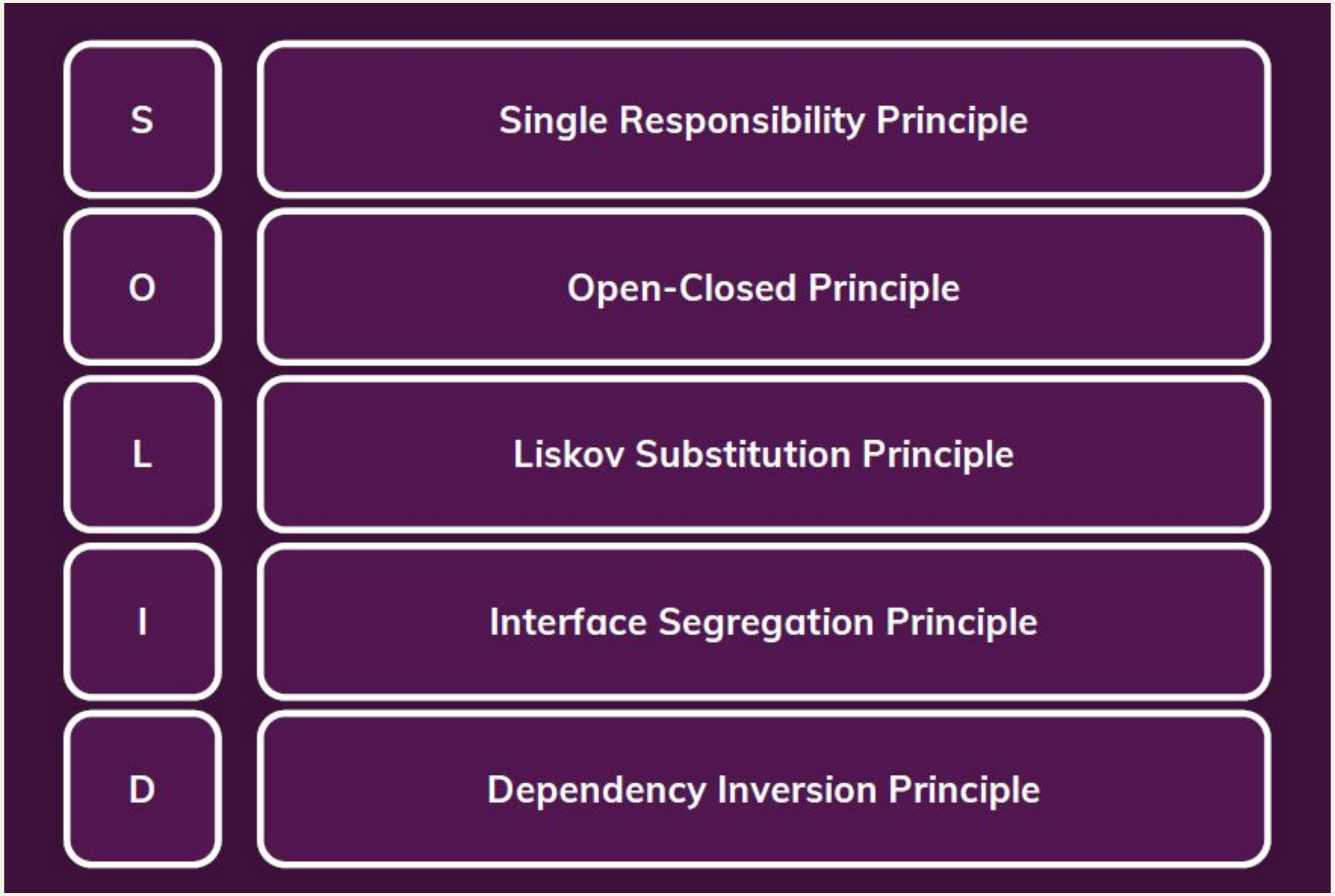
Cohesion


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

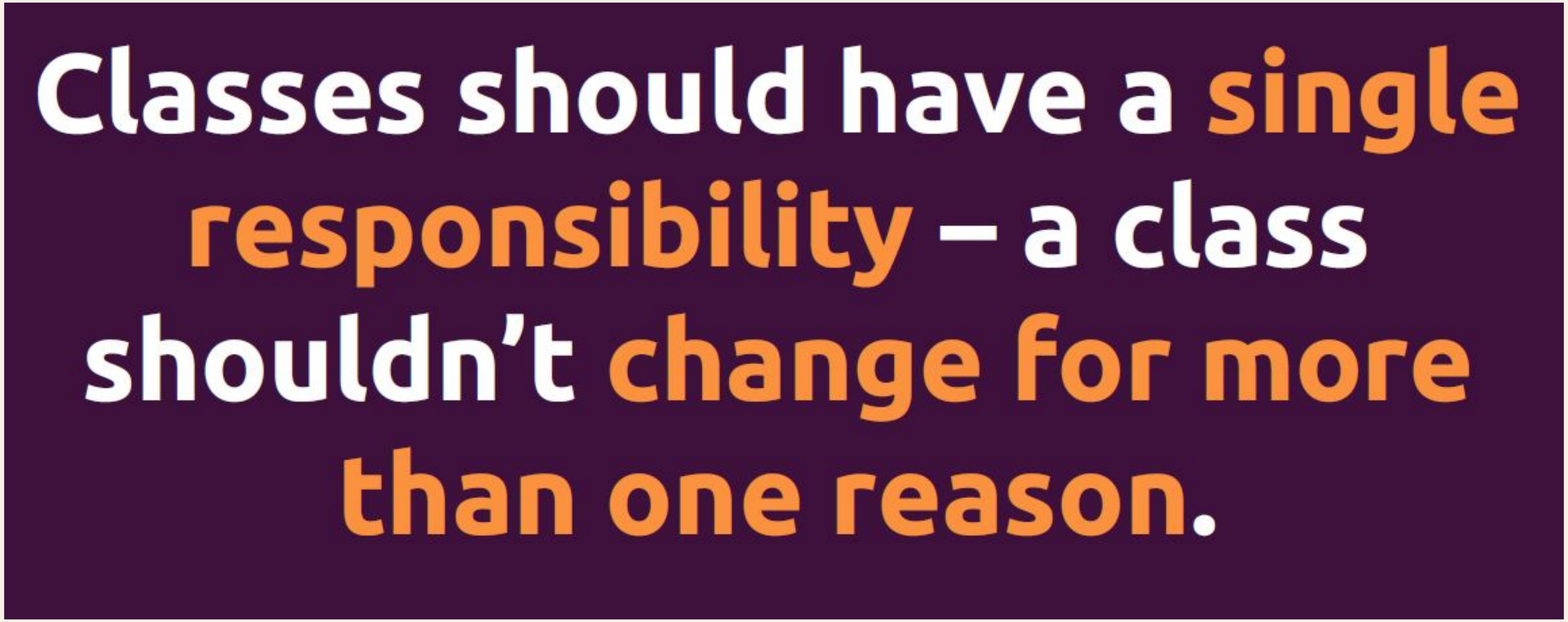

How much are your class methods using the class properties?
Maximum Cohesion
No Cohesion
All methods each use all properties
A highly cohesive object
Highly cohesive classes
All methods don't use any class properties
Data structure / container with utility methods

SOLID PRINCIPLES

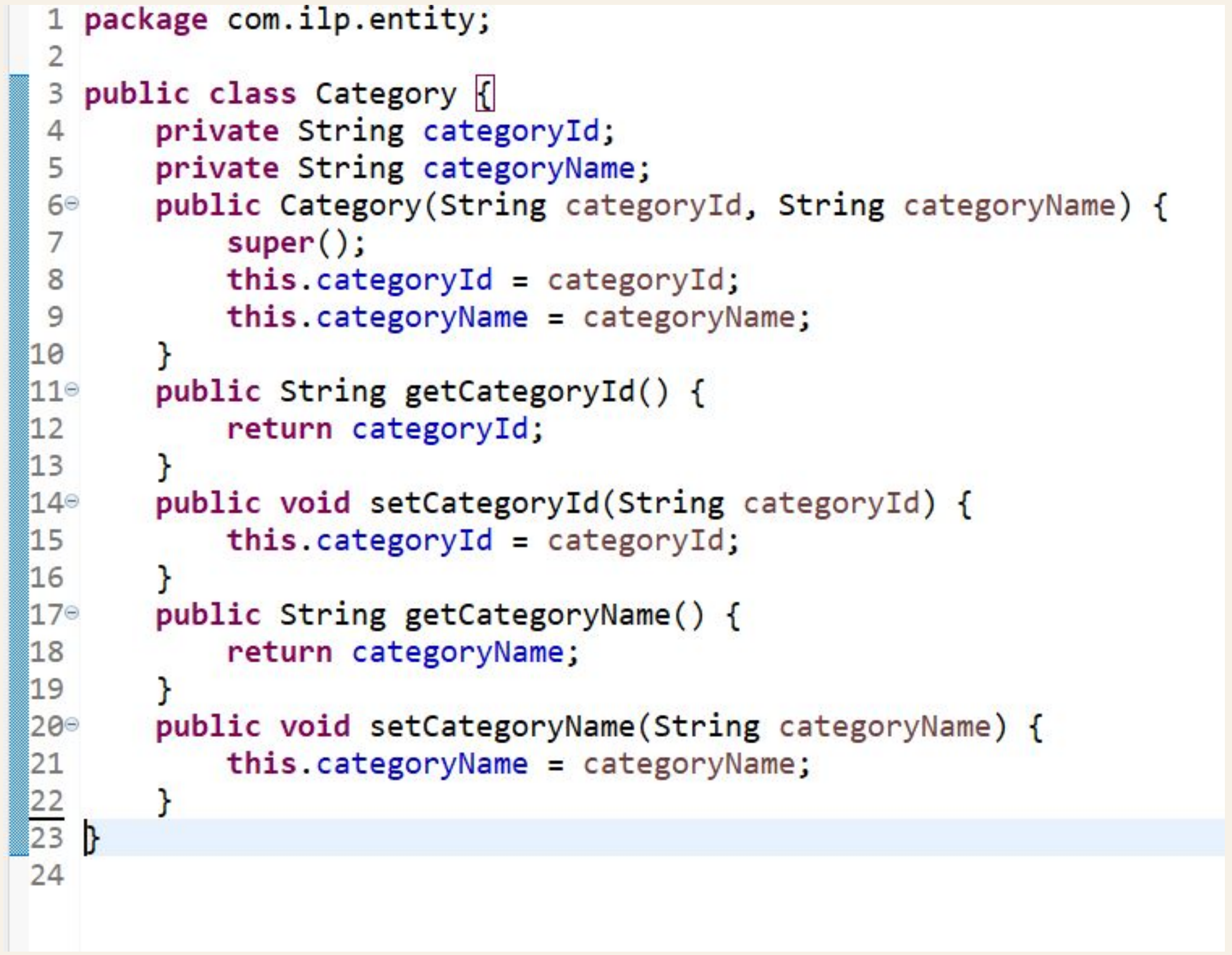


S	Single Responsibility Principle
O	Open-Closed Principle
L	Liskov Substitution Principle
I	Interface Segregation Principle
D	Dependency Inversion Principle

Single Responsibility Principle

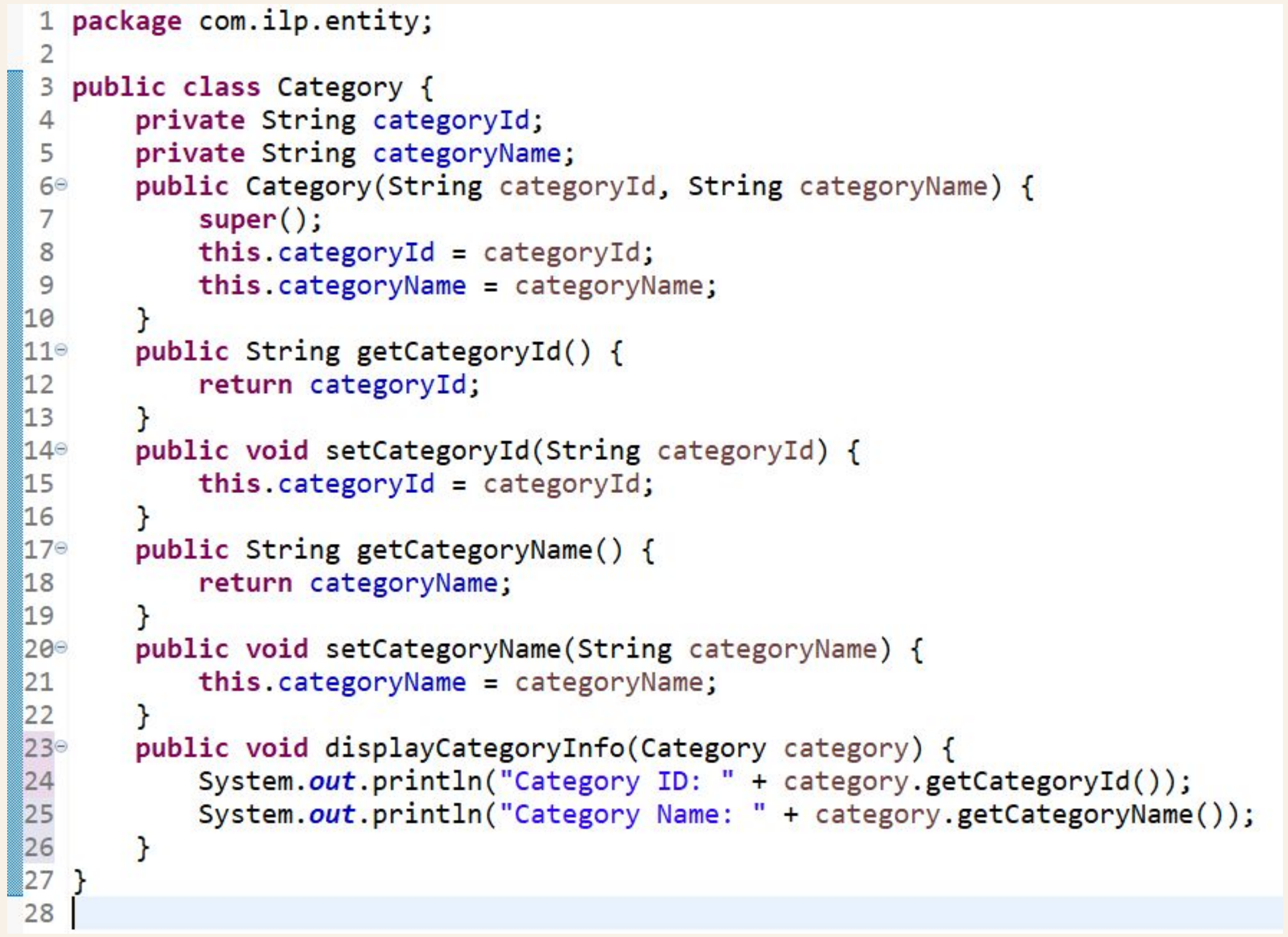

Classes should have a single responsibility – a class shouldn’t change for more than one reason.

Single Responsibility Principle



```java
package com.ilp.entity;

public class Category {
    private String categoryId;
    private String categoryName;
    public Category(String categoryId, String categoryName) {
        super();
        this.categoryId = categoryId;
        this.categoryName = categoryName;
    }
    public String getCategoryId() {
        return categoryId;
    }
    public void setCategoryId(String categoryId) {
        this.categoryId = categoryId;
    }
    public String getCategoryName() {
        return categoryName;
    }
    public void setCategoryName(String categoryName) {
        this.categoryName = categoryName;
    }
}

```

Fig 1

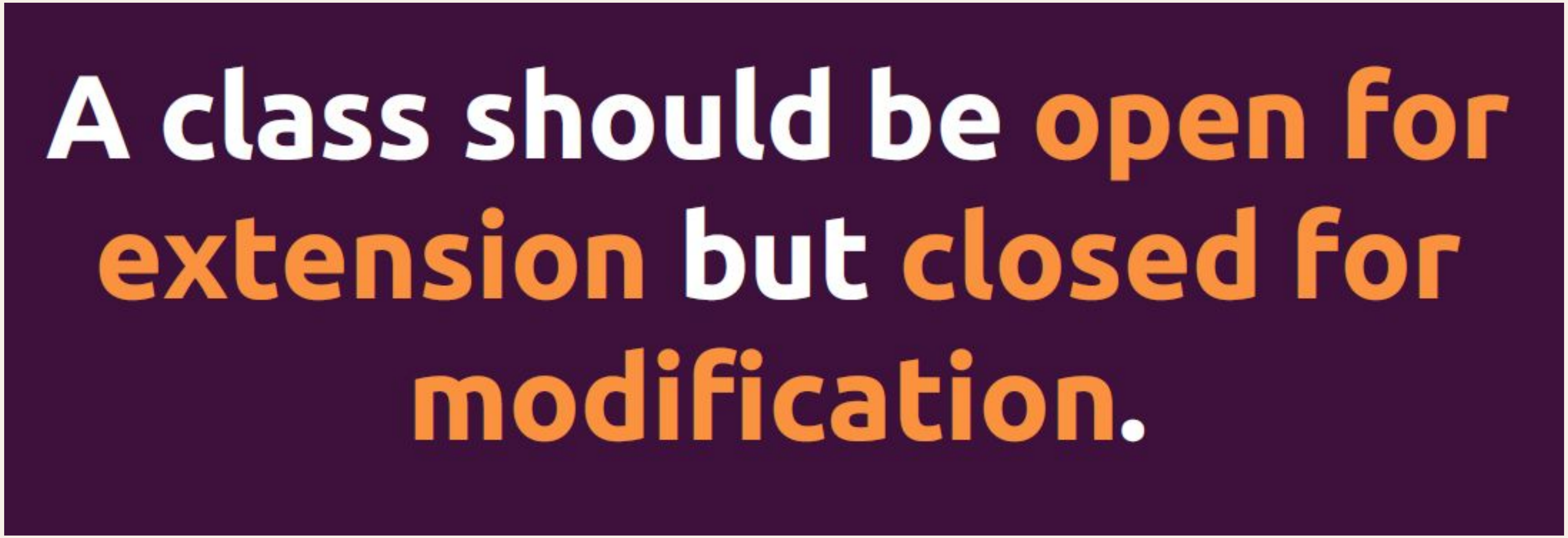


```java
package com.ilp.entity;

public class Category {
    private String categoryId;
    private String categoryName;
    public Category(String categoryId, String categoryName) {
        super();
        this.categoryId = categoryId;
        this.categoryName = categoryName;
    }
    public String getCategoryId() {
        return categoryId;
    }
    public void setCategoryId(String categoryId) {
        this.categoryId = categoryId;
    }
    public String getCategoryName() {
        return categoryName;
    }
    public void setCategoryName(String categoryName) {
        this.categoryName = categoryName;
    }
    public void displayCategoryInfo(Category category) {
        System.out.println("Category ID: " + category.getCategoryId());
        System.out.println("Category Name: " + category.getCategoryName());
    }
}

```

Fig 2

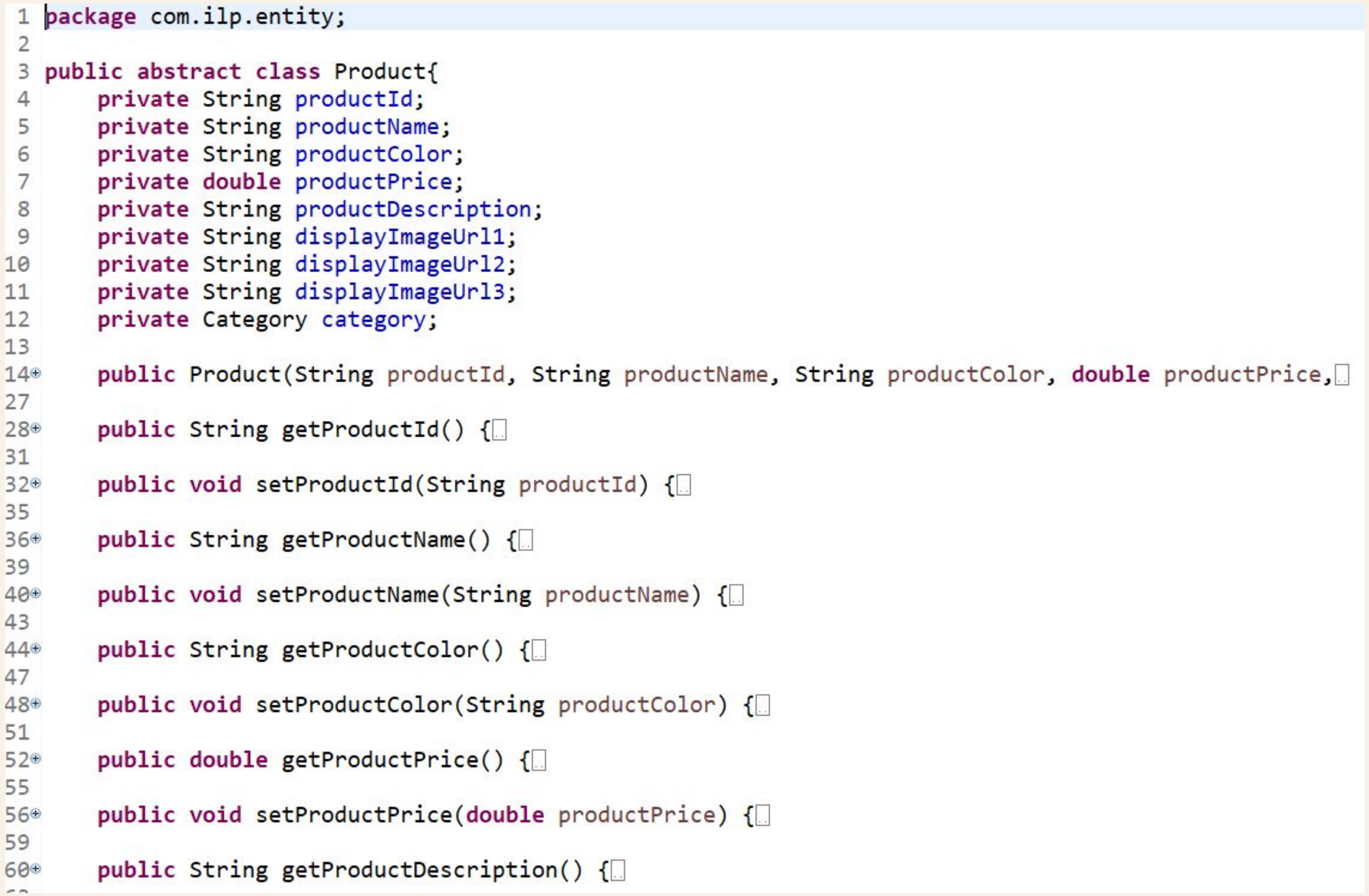
Class in Fig 2 has a function to display the category information which violates this principle. Class in Fig 1 has a single responsibility of setting the Category information.

Open Closed Principle

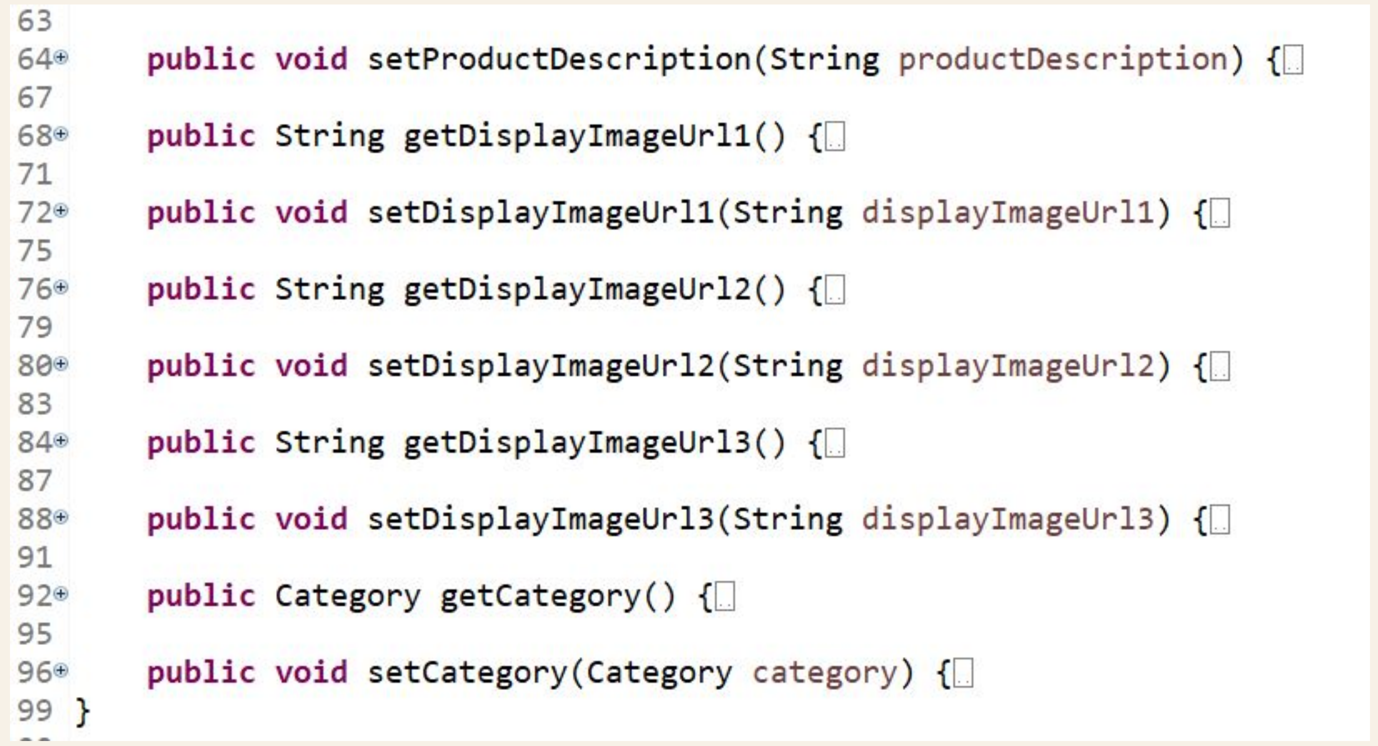


A class should be open for extension but closed for modification.

Open Closed Principle

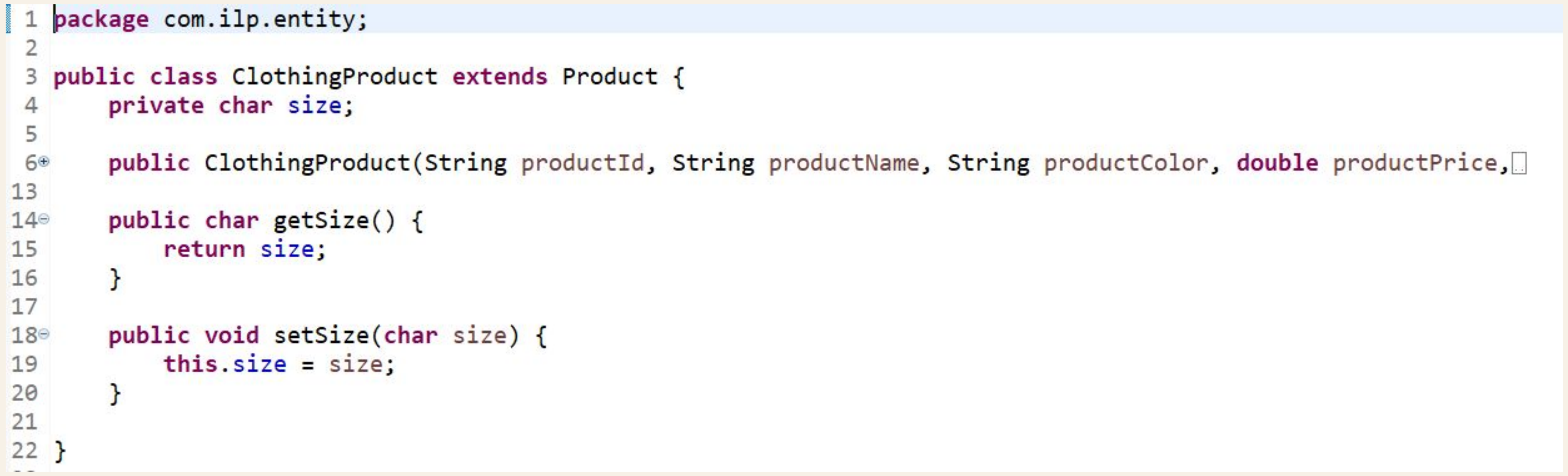


```java
package com.ilp.entity;

public abstract class Product{
    private String productId;
    private String productName;
    private String productColor;
    private double productPrice;
    private String productDescription;
    private String displayImageUrl1;
    private String displayImageUrl2;
    private String displayImageUrl3;
    private Category category;

    public Product(String productId, String productName, String productColor, double productPrice,

    public String getProductId() {

    public void setProductId(String productId) {

    public String getProductName() {

    public void setProductName(String productName) {

    public String getProductColor() {

    public void setProductColor(String productColor) {

    public double getProductPrice() {

    public void setProductPrice(double productPrice) {

    public String getProductDescription() {
```

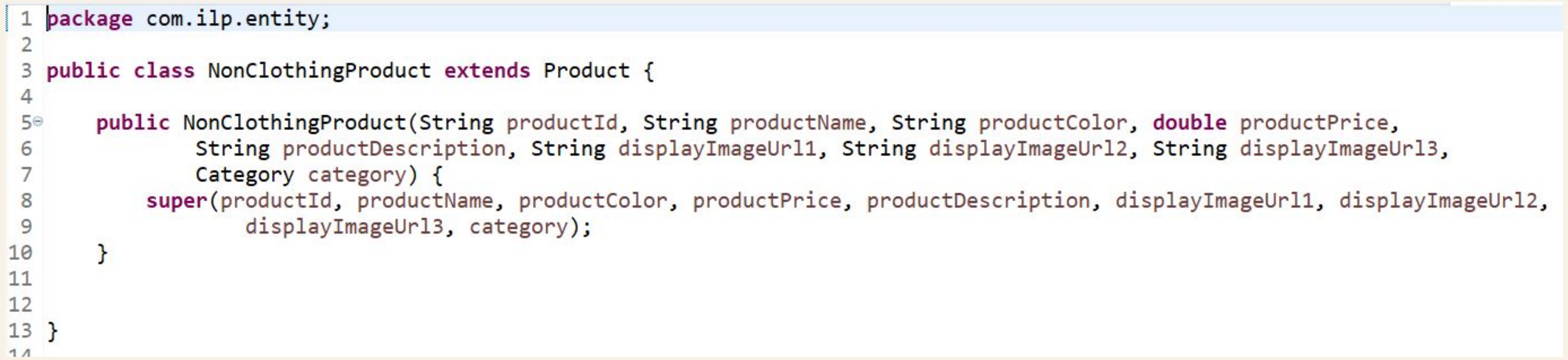


```java

    public void setProductDescription(String productDescription) {

    public String getDisplayImageUrl1() {

    public void setDisplayImageUrl1(String displayImageUrl1) {

    public String getDisplayImageUrl2() {

    public void setDisplayImageUrl2(String displayImageUrl2) {

    public String getDisplayImageUrl3() {

    public void setDisplayImageUrl3(String displayImageUrl3) {

    public Category getCategory() {

    public void setCategory(Category category) {
}
```

Open Closed Principle



```java
package com.ilp.entity;

public class ClothingProduct extends Product {
    private char size;

    public ClothingProduct(String productId, String productName, String productColor, double productPrice,

    public char getSize() {
        return size;
    }

    public void setSize(char size) {
        this.size = size;
    }

}
```

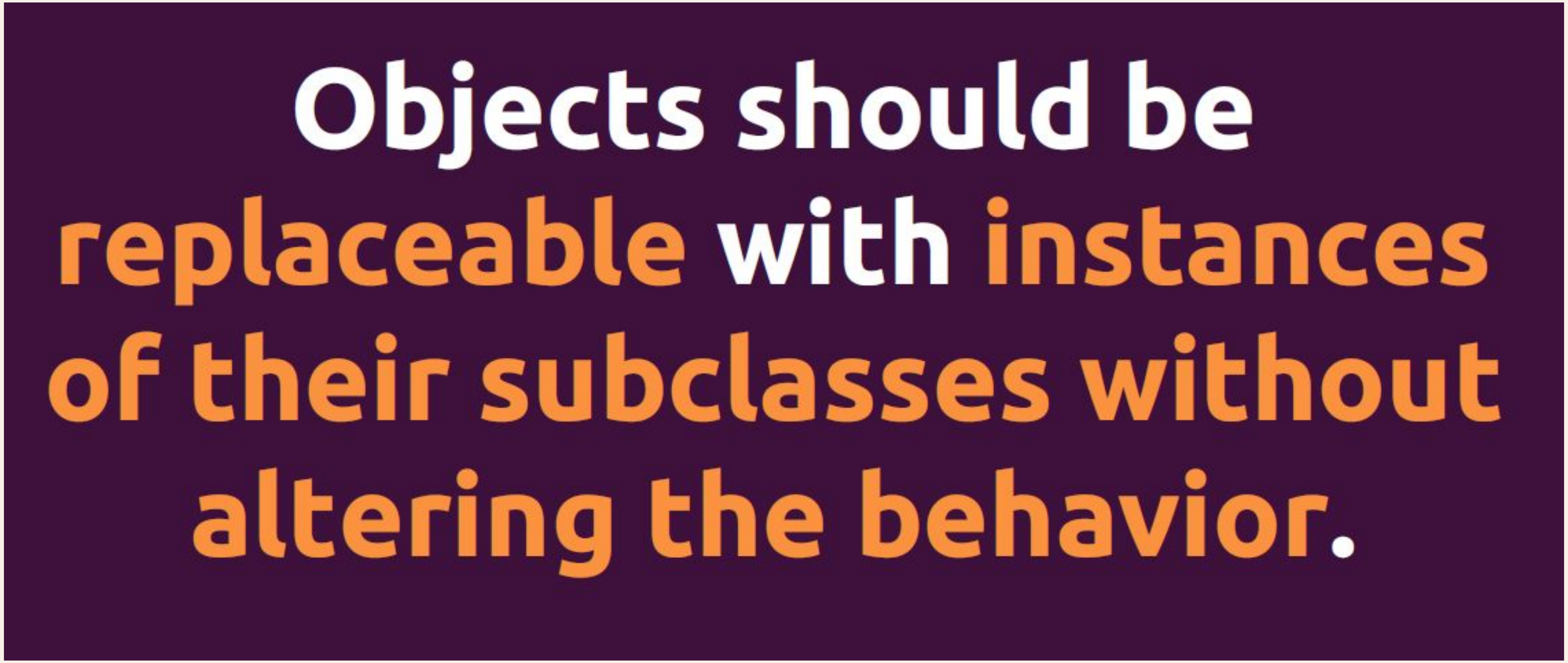


```java
package com.ilp.entity;

public class NonClothingProduct extends Product {

    public NonClothingProduct(String productId, String productName, String productColor, double productPrice,
            String productDescription, String displayImageUrl1, String displayImageUrl2, String displayImageUrl3,
            Category category) {
        super(productId, productName, productColor, productPrice, productDescription, displayImageUrl1, displayImageUrl2,
                displayImageUrl3, category);
    }


}
```

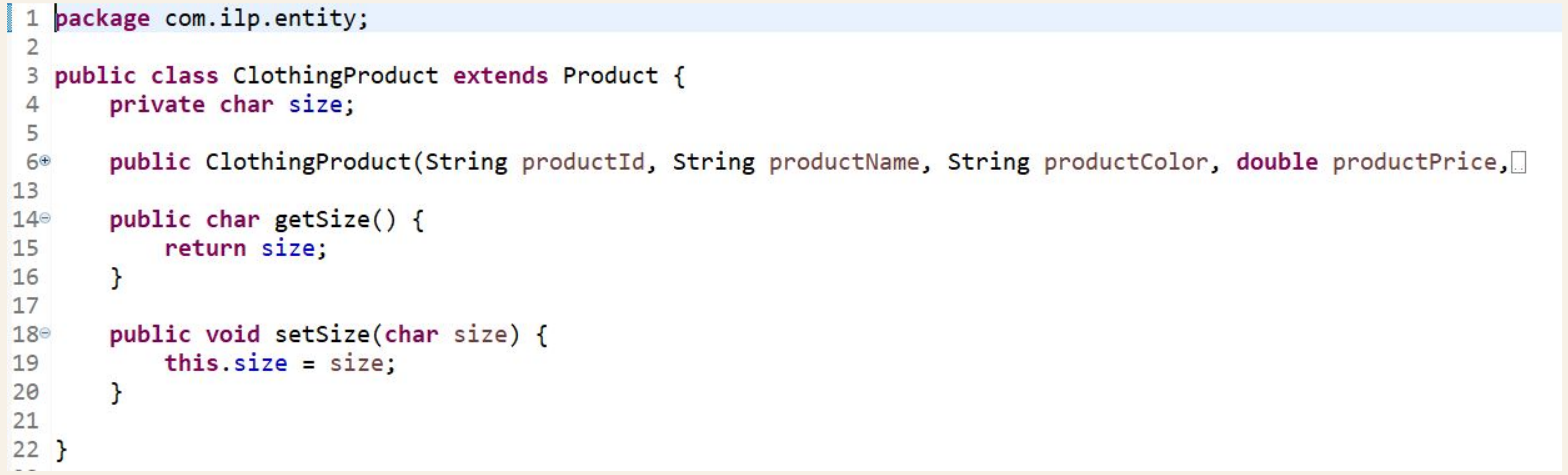
- The ClothingProduct and NonClothingProduct classes are subclasses of the Product class.
- The ClothingProduct class includes a distinct member variable named 'size.'
- This design allows for the introduction of new product types without the necessity of modifying the Product class.

- To incorporate a different type of product, one can simply extend the Product class and define any specific member variables within the new subclass.

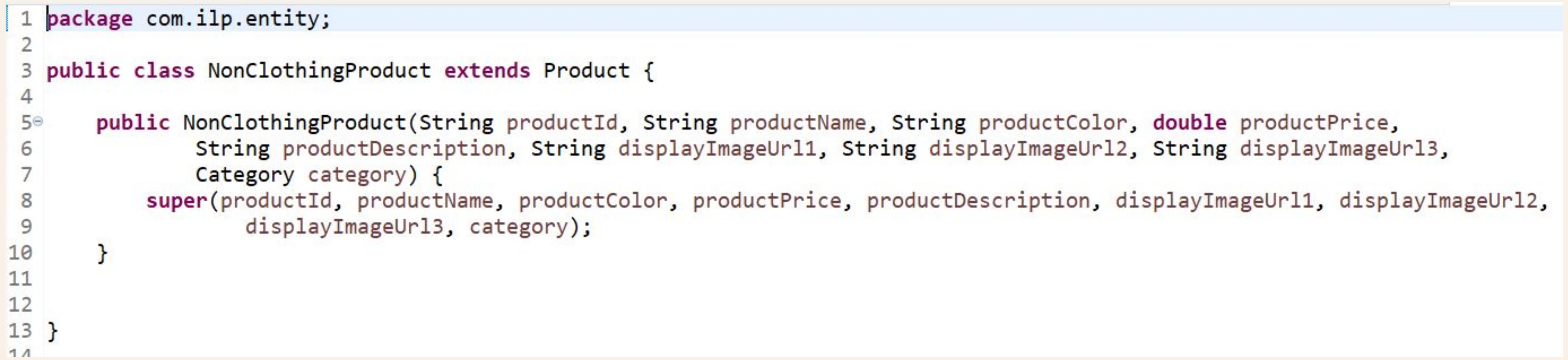
Liskov Substitution Principle


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

Liskov Substitution Principle



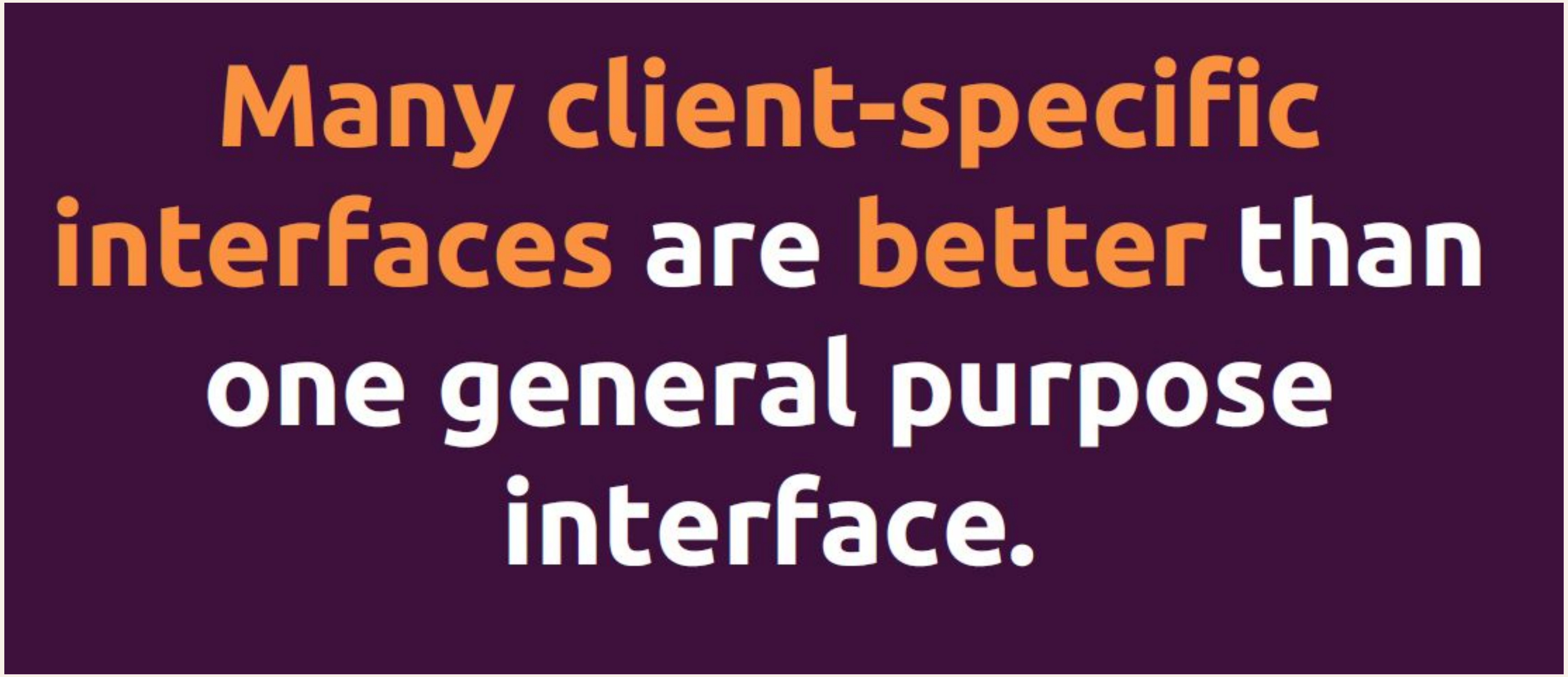
```java
package com.ilp.entity;

public class ClothingProduct extends Product {
    private char size;

    public ClothingProduct(String productId, String productName, String productColor, double productPrice,

    public char getSize() {
        return size;
    }

    public void setSize(char size) {
        this.size = size;
    }

}
```



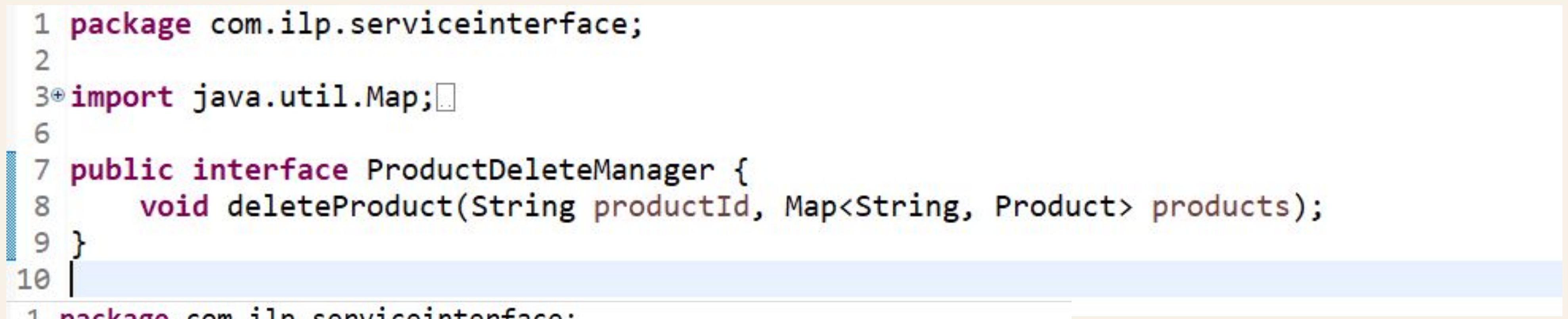
```java
package com.ilp.entity;

public class NonClothingProduct extends Product {

    public NonClothingProduct(String productId, String productName, String productColor, double productPrice,
            String productDescription, String displayImageUrl1, String displayImageUrl2, String displayImageUrl3,
            Category category) {
        super(productId, productName, productColor, productPrice, productDescription, displayImageUrl1, displayImageUrl2,
                displayImageUrl3, category);
    }


}
```

ClothingProduct and NonClothingProduct are the subclasses of Product class.

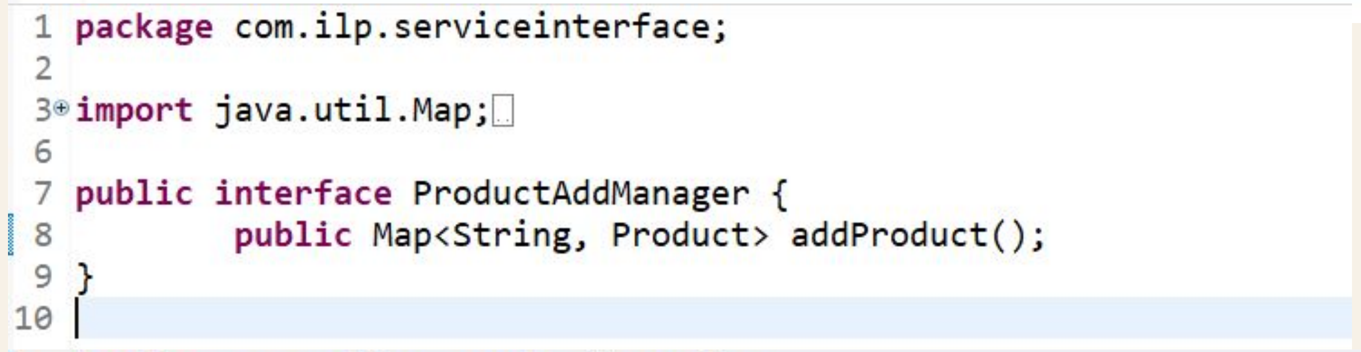
Interface Segregation Principle


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

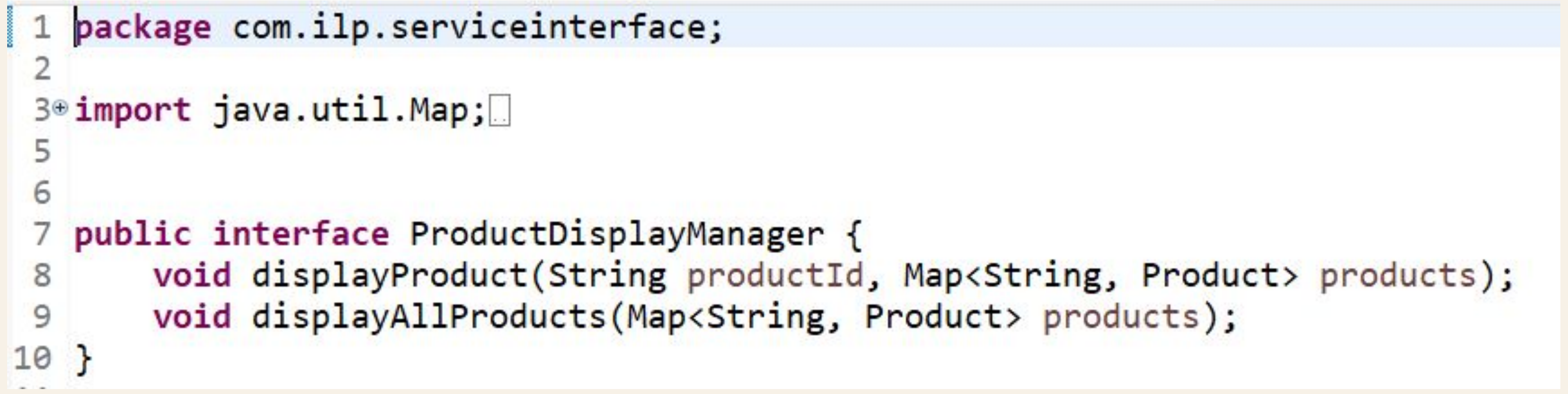
Interface Segregation Principle



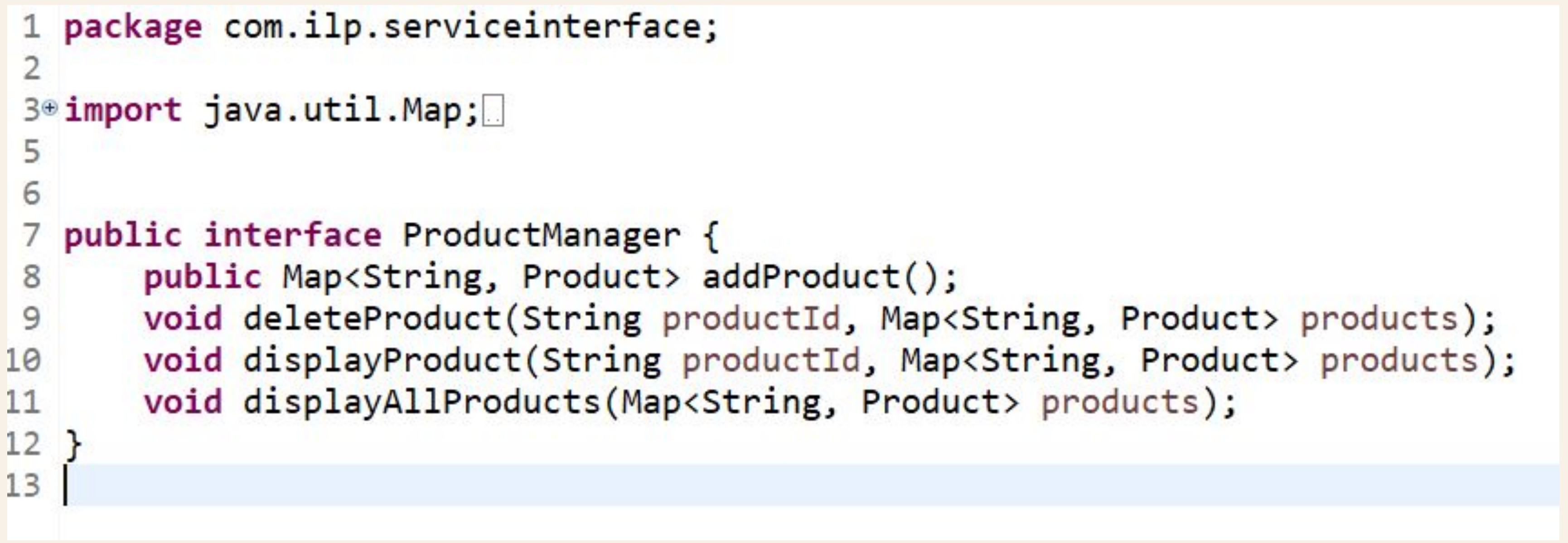
```
package com.ilp.serviceinterface;

import java.util.Map;

public interface ProductDeleteManager {
    void deleteProduct(String productId, Map<String, Product> products);
}
```



```
package com.ilp.serviceinterface;

import java.util.Map;

public interface ProductAddManager {
        public Map<String, Product> addProduct();
}
```



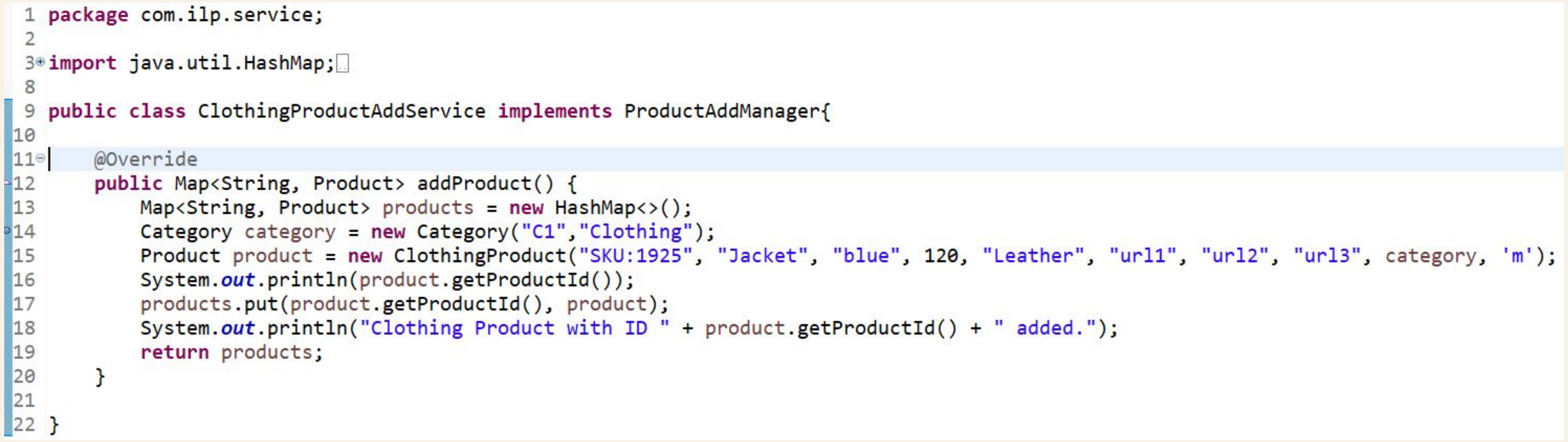
```
package com.ilp.serviceinterface;

import java.util.Map;


public interface ProductDisplayManager {
    void displayProduct(String productId, Map<String, Product> products);
    void displayAllProducts(Map<String, Product> products);
}
```



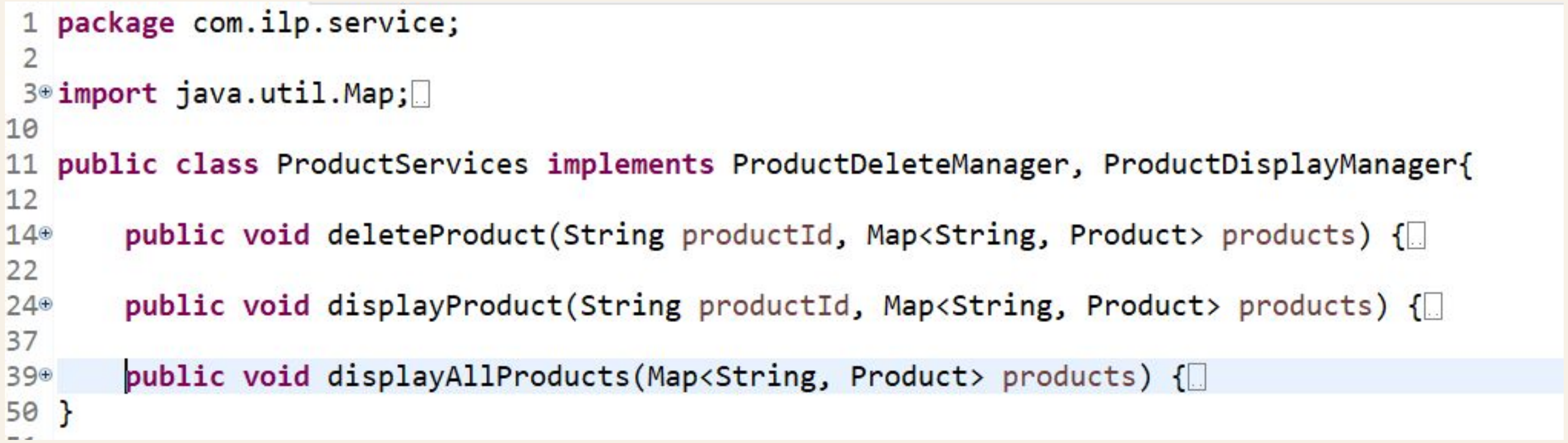
```
package com.ilp.serviceinterface;

import java.util.Map;


public interface ProductManager {
    public Map<String, Product> addProduct();
    void deleteProduct(String productId, Map<String, Product> products);
    void displayProduct(String productId, Map<String, Product> products);
    void displayAllProducts(Map<String, Product> products);
}
```

- The interface ProductManager has add, delete and display functions.
- All classes implementing this needs to override these functions
- Add function has to be implemented separately for ClothingProduct and NonClothingProduct.
- ProductManager interface alone cannot achieve this task
- So we separated this into 3 interfaces.

Interface Segregation Principle



```java
package com.ilp.service;

import java.util.HashMap;

public class ClothingProductAddService implements ProductAddManager{

    @Override
    public Map<String, Product> addProduct() {
        Map<String, Product> products = new HashMap<>();
        Category category = new Category("C1","Clothing");
        Product product = new ClothingProduct("SKU:1925", "Jacket", "blue", 120, "Leather", "url1", "url2", "url3", category, 'm');
        System.out.println(product.getProductId());
        products.put(product.getProductId(), product);
        System.out.println("Clothing Product with ID " + product.getProductId() + " added.");
        return products;
    }

}
```



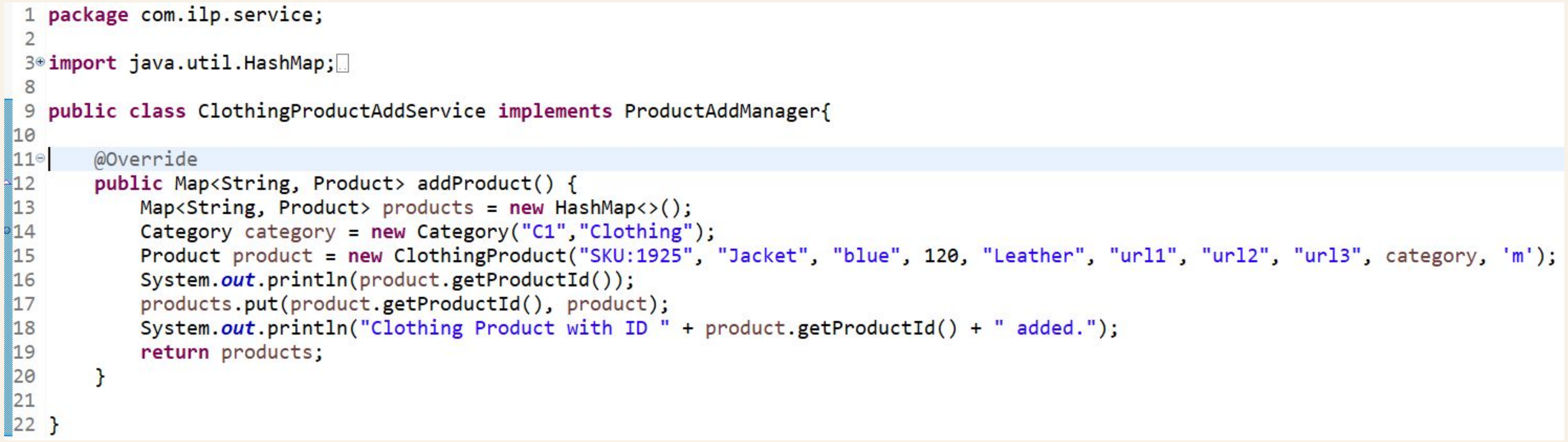
```java
package com.ilp.service;

import java.util.Map;

public class ProductServices implements ProductDeleteManager, ProductDisplayManager{

    public void deleteProduct(String productId, Map<String, Product> products) {

    public void displayProduct(String productId, Map<String, Product> products) {

    public void displayAllProducts(Map<String, Product> products) {
}
```

Implementation of ProductAddManager, ProductDeleteManager and ProductDisplayManager.

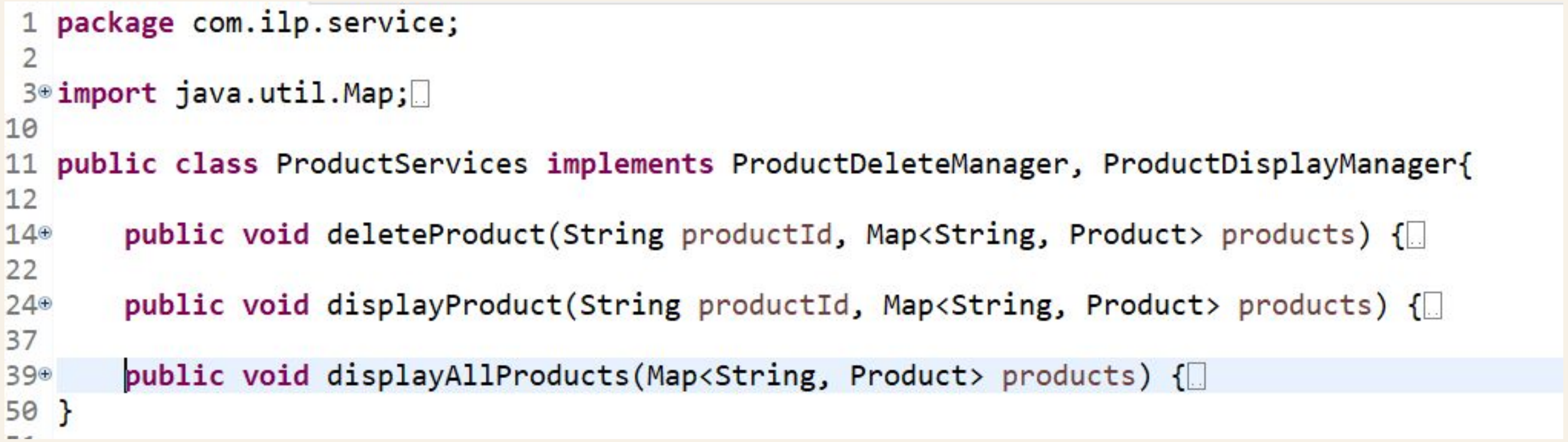
Dependency Inversion Principle

You should depend upon abstractions, not concretions.

Dependency Inversion Principle

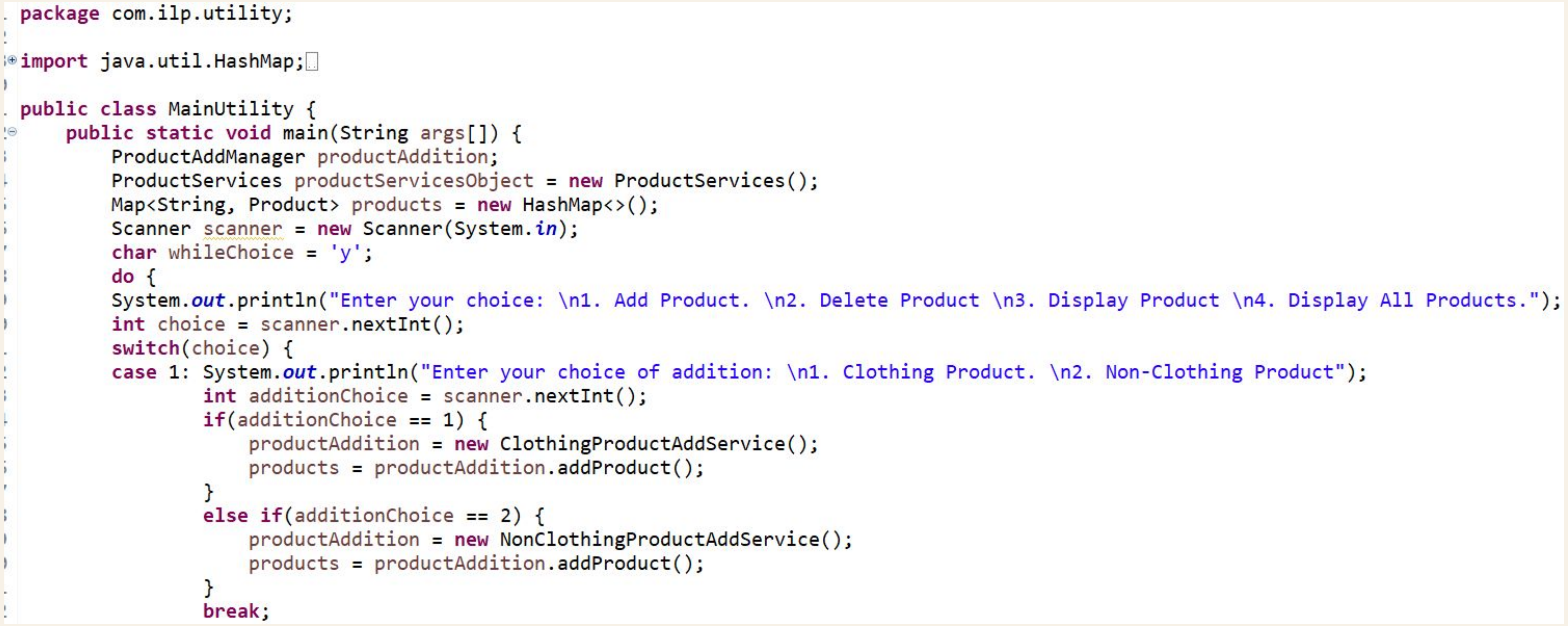


```java
package com.ilp.service;

import java.util.HashMap;

public class ClothingProductAddService implements ProductAddManager{

    @Override
    public Map<String, Product> addProduct() {
        Map<String, Product> products = new HashMap<>();
        Category category = new Category("C1","Clothing");
        Product product = new ClothingProduct("SKU:1925", "Jacket", "blue", 120, "Leather", "url1", "url2", "url3", category, 'm');
        System.out.println(product.getProductId());
        products.put(product.getProductId(), product);
        System.out.println("Clothing Product with ID " + product.getProductId() + " added.");
        return products;
    }

}
```



```java
package com.ilp.service;

import java.util.Map;

public class ProductServices implements ProductDeleteManager, ProductDisplayManager{

    public void deleteProduct(String productId, Map<String, Product> products) {

    public void displayProduct(String productId, Map<String, Product> products) {

    public void displayAllProducts(Map<String, Product> products) {
}
```

Dependency Inversion Principle



```
package com.ilp.utility;

import java.util.HashMap;

public class MainUtility {
    public static void main(String args[]) {
        ProductAddManager productAddition;
        ProductServices productServicesObject = new ProductServices();
        Map<String, Product> products = new HashMap<>();
        Scanner scanner = new Scanner(System.in);
        char whileChoice = 'y';
        do {
        System.out.println("Enter your choice: \n1. Add Product. \n2. Delete Product \n3. Display Product \n4. Display All Products.");
        int choice = scanner.nextInt();
        switch(choice) {
        case 1: System.out.println("Enter your choice of addition: \n1. Clothing Product. \n2. Non-Clothing Product");
                int additionChoice = scanner.nextInt();
                if(additionChoice == 1) {
                    productAddition = new ClothingProductAddService();
                    products = productAddition.addProduct();
                }
                else if(additionChoice == 2) {
                    productAddition = new NonClothingProductAddService();
                    products = productAddition.addProduct();
                }
                break;
```

Interface ProductAddManager is depended rather than concretions. productAddition is used to refer the objects of ClothingProductAddService & NonClothingProductAddService.

THANK YOU