Software Development Principles and Practices

Design Patterns Practice - Creational Pattern

Week 11

The best designers will use many design patterns that dovetail and intertwine to produce a greater whole

– Erich Gamma

Objectives

- Learn how to use creational patterns

Contents

- Simple practice: Creational Patterns
    - Factory Method
    - Abstract Factory
    - Builder
- Solve problem: Creational Patterns

Recap: Creational Patterns

- We are going to implement simple code with creational patterns
- Please clone the skeleton code [Link]
- We will use the following 4 android projects
  - `creationalPatterns/FactoryMethodExample`
  - `creationalPatterns/AbstractFactoryExample`
  - `creationalPatterns/BuilderExample`
  - `creationalPatterns/CreationalPatternPractice`

Practice #1: Factory Method

- Open `creationalPatterns/FactoryMethodExample` with Android Studio
- There are 4 classes
  - Client.java: Main function
  - Phone.java: Interface about Phone (Product)
  - GalaxyS22.java: Class implementing Phone (ConcreteProduct)
  - GalaxyS23.java: Class implementing Phone (ConcreteProduct)

Practice #1: Skeleton Code

```
public interface Phone {

    void info();

}

public class GalaxyS22 implements Phone{

    @Override

    public void info() {

        System.out.println("This is Galaxy S22.");

    }

}
```

```
public class GalaxyS23 implements Phone{

    @Override

    public void info() {

        System.out.println("This is Galaxy S23.");

    }

}

public static void main(String args[]){

    Phone s22 = new GalaxyS22();

    Phone s23 = new GalaxyS23();


    s22.info();

    s23.info();

}
```

7 mins

Practice #1: Implement Factory Method

- In the skeleton code, the client creates each concreteProduct directly using “new” keyword
- When there are many “new”, the client is more likely to make mistakes when writing code
- Implement the Factory Method
  - Eliminate the "new" keyword that creates the concreteProduct in the client code
  - Hint: Define the Factory interface, and create a ConcreteFactory that creates certain concreteProduct

Practice #2 : Abstract Factory

- Open `creationalPatterns/AbstractFactoryExample` with Android Studio
- There are 10 classes
  - Client.java: Main function
  - Phone.java: Interface about Phone (Product)
    - GalaxyS23.java, iPhone15.java: ConcreteProduct
  - Tablet.java: Interface about Tablet (Product)
    - GalaxyTab.java, iPad.java: ConcreteProduct
  - Laptop.java: Interface about Laptop (Product)
    - GalaxyBook.java, Macbook.java: ConcreteProduct

Practice #2: Skeleton Code (1/3)

```java
public interface Phone {

    void call();

}

public interface Tablet {

    void touch();

}

public interface Laptop {

    void

    typing();

}
```

```java
public class GalaxyS23 implements Phone{
    @Override
    public void call() {...}
}
public class GalaxyTab implements Tablet{
    @Override
    public void touch() {...}
}
public class GalaxyBook implements Laptop{
    @Override
    public void typing() {...}
}
```

Practice #2: Skeleton Code (2/3)

```java
public class iPhone15 implements Phone{
    @Override
    public void call() {...}
}
public class iPad implements Tablet{
    @Override
    public void touch() {...}
}
public class MacBook implements Laptop{
    @Override
    public void typing() {...}
}
```

Practice #2: Skeleton Code (3/3)

```java
public static void main(String[] args){
    String company = "Apple";
    Phone phone;
    Tablet tablet;
    Laptop laptop;

    if(company.equals("Apple")){
        phone = new iPhone15();
        tablet = new iPad();
        laptop = new MacBook();
    }
   else {
        phone = new GalaxyS23();
        tablet = new GalaxyTab();
        laptop = new GalaxyBook();
    }

    phone.call();
    tablet.touch();
    laptop.typing();
}
```

7 mins

Practice #2: Implement Abstract Factory

- In the skeleton code, the client creates a set of concreteProducts directly using “new” keyword
- Implement the Abstract Factory
  - Eliminate the "new" keyword
  - Manage the creation of set of concreteProducts
    - Set of Apple’s products or set of Samsumg’s products
  - Hint: Define the AbstractFactory interface, and create a ConcreteFactory that creates a set of concreteProduct

Practice #3: Builder

- Open `creationalPatterns/BuilderExample` with Android Studio
- There are 2 classes
  - Client.java: Main function
  - ModelTrainer.java

Practice #3 : Skeleton Code (1/3)

```java
public class ModelTrainer {
    String model;
    String trainDataloader;
    String validDataloader;
    String testDataloader;
    String optimizer;
    String lossFunction;
    double learningRate;
    String preProcessor;
    String postProcessor;
    String visualizer;
    int batchSize;
    int inputSize;
```

Many arguments

```java
public void info(){
   // Print information about ModelTrainer
}


public void setModel(String model) {
    this.model = model;
}
... // Other setters
```

All setters corresponding to the argument are implemented

Practice #3 : Skeleton Code (2/3)

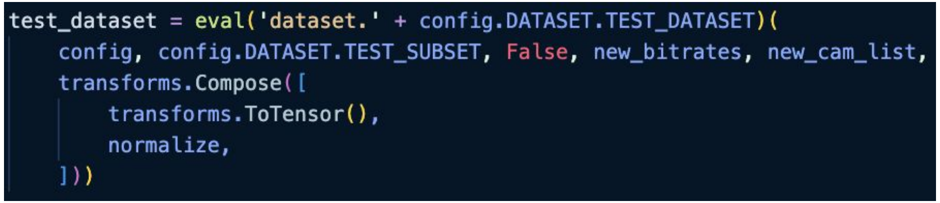
```
public ModelTrainer(String model, String trainDataloader, String testDataloader, ...){
    this.model = model;
    ... // Initialize other attributes
}

public ModelTrainer(String model, String trainDataloader, String validDataloader, String testDataloader,
...){
    this.model = model;
    ... // Initialize other attributes
}
```

Practice #3 : Skeleton Code (3/3)

```java
public static void main(String[] args){
    ModelTrainer trainer = new ModelTrainer("Yolov5",
            "TrainDataloader", "TestDataloader",
            "SGD", "MSE",
            0.001, 64, 256);
    trainer.info();
}
```

Real example



```python
test_dataset = eval('dataset.' + config.DATASET.TEST_DATASET)(
    config, config.DATASET.TEST_SUBSET, False, new_bitrates, new_cam_list,
    transforms.Compose([
        transforms.ToTensor(),
        normalize,
    ]))
```

7 mins

Practice #3 : Implement Builder

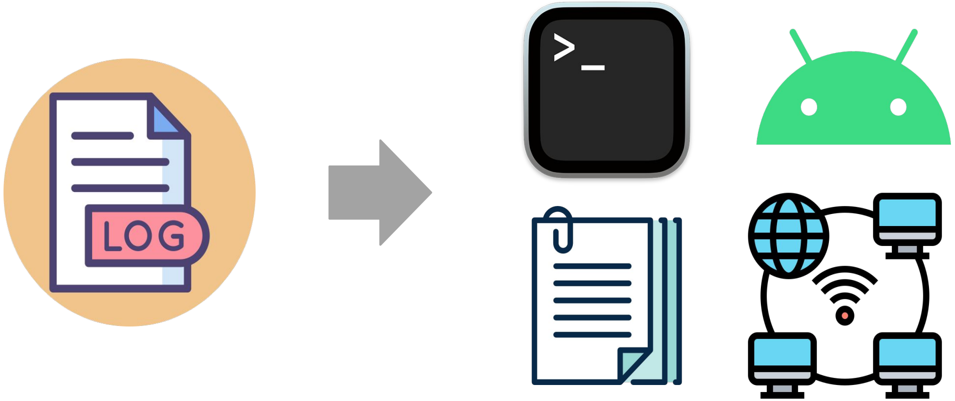
- In the skeleton code, the client creates an object with many arguments
- We actually need a variety of constructors
- Implement the Builder
  - Eliminate all implemented constructors that take a large combinations of arguments
  - Hint: Remove constructors and use setter to initialize the object
  - You don’t need to implement Director class

Solve Problem: Creational Patterns

- Now, let’s solve an open problem about creational patterns
- You should implement one of creational patterns to solve a given design problem
- Open “`creationalPatterns/CreationalPatternPractice`” with Android Studio

Problem Situation

- A logging system needs to be adaptable to various types of loggers (console, file, network, etc.)
- The code structure using multiple if-else statements to instantiate these loggers has two problems
  - It is prone to redundancy
  - It is not scalable



Problem in the Code

- When you want to add another type, you should change the `if-else` statement redundantly

```
if (loggerType.equals("MAC")){
    MACLogger logger = new MACLogger();
    logger.error(message);
} else if (loggerType.equals("Android")){
    AndroidLogger logger = new AndroidLogger();
    logger.error(message);
}
```

```
else if (loggerType.equals("File")){
    FileLogger logger = new FileLogger();
    logger.error(message);
} else if (loggerType.equals("Network")){
    NetworkLogger logger = new NetworkLogger();
    logger.error(message);
}
```

15 mins

What Pattern is Suitable?

- The current implementation for selecting a logger type is cumbersome and not maintainable
  - The code must be modified in multiple places every time a new logger type is introduced
  - The risk of errors in modification is increased if so many files uses logger
  - It violates the Open-Closed Principle
- To address this, we should utilize a design pattern that allows us to add new logger types without altering existing code in main directly

Submissions

- Submit 1 zip file on eTL
  - ZIP:
    - Zip `creationalPatterns` directory
  - ZIP file structure
    - creationalPatterns ← Zip this directory
      - FactoryMethodExample
      - AbstractFactoryExample
      - BuilderExample
      - CreationalPatternPractice
- Deadline: **11/17 23:59**

Thank You.
Any Questions?