Software Development

Dr. Hamada I. AbdulWakel[1]

[1]Computer Science Department

2022 - 2023

Ch6: Creational Design Patterns
Lec#7

Midterm Exam Solution

- SW integrity: a SW communicate with another application.
- User story: a description of a feature from a user.
- Pair programming: two programmers are developing a task.
- The violated SOLID principle: SRP
  class cityMap{ ..... }
  class citiesManaging{ ..... }
  class cityDrawing{ ..... }
  class cityCalculating{ ..... }
- The violated clean code rule: Searchable names.
  final int time = 86400000
  setTimeout(blastOff, time);

Factory Design Pattern

Problem

When dealing with class objects **based on input or events, you do not know ahead**, it becomes problematic.

Solution

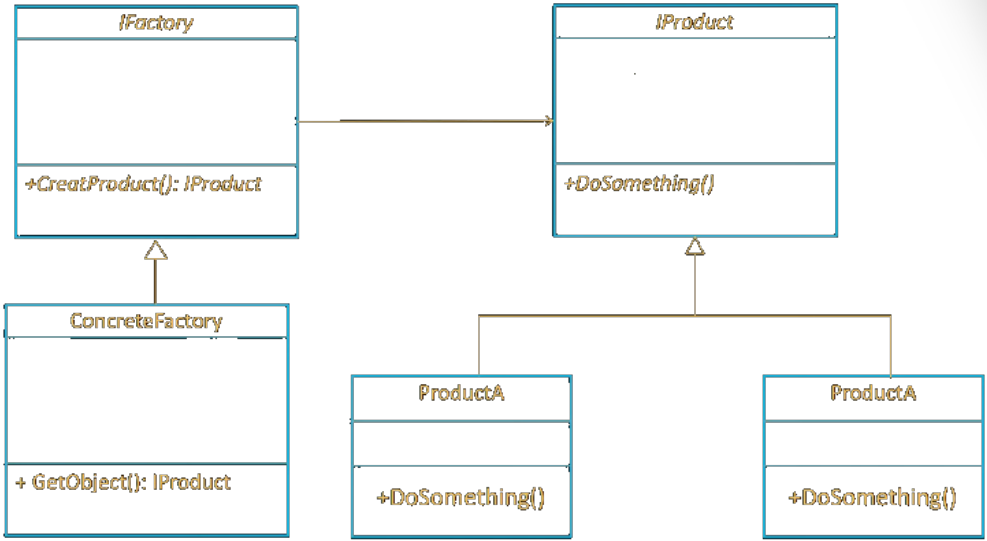
A one-class is responsible for creating these objects. Encapsulating the object creation in one place (centralized) which allow any changes.

Factory Design Pattern Examples

Factory Design Pattern Examples ...

```
public class Level_1{
  //10 levels in the game
  function getRandomEnemyinRoom_1(){
    r = random(0, 10);
    if ( 0 < r < 1) enemy = new afret();
    else if (1 < r < 2) enemy = new asd();
  }
  function getRandomEnemyinRoom_2(){
    r = random(0, 10);
    if ( 0 < r < 1) enemy = new skeleton();
    else if (1 < r < 2) enemy = new zombie();
  }
}
```

Factory Design Pattern UML



Factory Design Pattern Examples ...

Problem: violating OCP and DIP !!!!

```
Pizza orderPizza(String type) {
  Pizza pizza;

  if (type.equals("cheese"))
    pizza = new CheesePizza();
  else if (type.equals("greek"))
    pizza = new GreekPizza();
  else if (type.equals("pepperoni"))
    pizza = new PepperoniPizza();

  pizza.prepare();  pizza.bake();  pizza.cut();  pizza.box();
  return pizza;
}
```

Factory Design Pattern Examples ...

Problem: ...

```
Pizza getVIPPizza() {
   Random rand = new Random();     int r = rand.nextInt(11);
   Pizza pizza = null;

   if (r == 0)    pizza = new CheesePizza();
   else if (r == 1)    pizza = new PepperoniPizza();
   else    pizza = new ClamPizza();

   pizza.prepare();    pizza.bake();    pizza.cut();    pizza.box();
   return pizza;
}
// ...., other methods which create objects of pizza types
```

Factory Design Pattern Implementation

Solution:

```
abstract public class Pizza{
   String name, sauce;
   public String getName(){ return name;}
   prepare(){ System.out.println("Perparing " + name); }
   bake(){ .... }    cut(){ .... }    box(){ .... }
}

public class ClamPizza extends Pizza{
   public ClamPizza(){
      name = "Clam Pizza";
      sauce = "My sauce";
   }
}
```

Factory Design Pattern Implementation ...

Solution: ... only one place where objects are created

```
public class SimplePizzaFactory{
  public Pizza createPizza(String type) {
    Pizza pizza = null;
    if (type.equals("cheese"))
      pizza = new CheesePizza();
    else if (type.equals("greek"))
      pizza = new GreekPizza();
    else if (type.equals("clam"))
      pizza = new ClamPizza();
    return pizza;
  }
}
```

Factory Design Pattern Implementation ...

Solution: ...

```
// PizzaStore does not know any thing about Pizza types
public class PizzaStore{
  SimplePizzaFactory factory;
  public PizzaStore(SimplePizzaFactory factory){
    this.factory = factory;
  }
  public Pizza orderPizza(String type) {
    Pizza pizza;
    pizza = factory.createPizza(type);
    pizza.prepare();   pizza.bake();   pizza.cut();   pizza.box();
  }
}
```

Factory Design Pattern

- Advantages:
  1. Reduces hard-coded complexity.
  2. Avoiding tight coupling between the creator and the concrete products.
  3. SRP, you can move the product creation code into one place.
  4. OCP, you can introduce new types of products into the program without breaking existing client code.

- Disadvantages:
  1. The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern.

Builder Design Pattern

Problem

When dealing with **complex class object (e.g., the number of parameters)**, it becomes problematic.

Solution

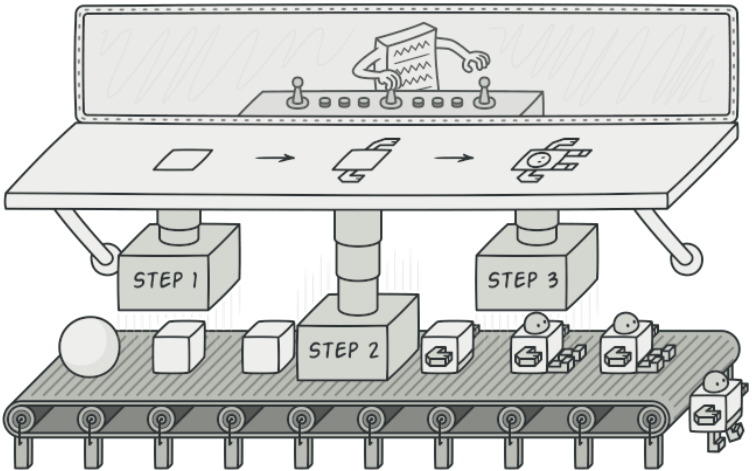
A one-class is responsible for creating the complex objects.

Builder Design Pattern Examples

Example 1

```
import pack_of_x;    import pack_of_y;    import pack_of_z;
public class MyClass {
   // these objects do not have direct relation with MyClass
   // these object will not be used any more inside MyClass
   X obj_of_x;
   Y obj_of_y;
   Z obj_of_z;
   W obj_of_w;
   U obj_of_u;
   A a = new A (obj_of_x, obj_of_y, obj_of_z, obj_of_w, obj_of_u);
}
```

Builder Design Pattern Examples ...



Builder Design Pattern Implementation

```
public class Robot{
   int id, price; static int idCount; String type, headType, bodyType, legsType;
   // creating 2^n constructors and lead to code repetition.
   Robot (String type){
   this.id = idCount++;      this.type = type;
   this.price = (type.equals("Spherical")) ? 50000 : 20000;
   this.headType = (type.equals("Spherical")) ? "Spherical" : "Cubical";
   this.bodyType = (type.equals("Spherical")) ? "Spherical" : "Cubical";
   this.legsType = (type.equals("Spherical")) ? "Spherical" : "Cubical";    }
   Robot (String type, int price){
   this.id = idCount++;      this.type = type;      this.price = price;
   this.headType = (type.equals("Spherical")) ? "Spherical" : "Cubical";
   this.bodyType = (type.equals("Spherical")) ? "Spherical" : "Cubical";
   this.legsType = (type.equals("Spherical")) ? "Spherical" : "Cubical";    } }
```

Builder Design Pattern Implementation ...

Solution 1: Constructor chain

```
public class Robot{
   int id, price;   static int idCount;   String type, headType, bodyType, legsType;
   Robot (String type){
   this.id = idCount++;      this.type = type;
   this.price = (type.equals("Spherical")) ? 50000 : 20000;
   this.headType = (type.equals("Spherical")) ? "Spherical" : "Cubical";
   this.bodyType = (type.equals("Spherical")) ? "Spherical" : "Cubical";
   this.legsType = (type.equals("Spherical")) ? "Spherical" : "Cubical";
   }
   Robot (String type, int price){
   this(type); // all objects depend on type parameter!!!
   this.price = price;
   }
}
```

Builder Design Pattern Implementation ...

Solution 2: Using setters

```
public class Robot{
  int id, price;   static int idCount;   String type, headType, bodyType, legsType;

  Robot (){ this.id = idCount++; }

  // write all setters for parameters
  public Robot setPrice(int... price){
    if (price.length > 0)
       this.price = price[0];
    else
       this.price = (this.type.equals("Spherical")) 50000 : 20000;
    return this;
  }
}
```

Builder Design Pattern Implementation ...

Solution 2: Using setters ...

```
public static void main(String[] args){
   // each set function create a part of the complete object
   // setType function must be before setPrice

   Robot r = new Robot().setType("Spherical")
                        .setPrice(1000)
                        .setBodyType("Spherical")
                        .setHeadType("Spherical")
                        .setLegsType("Spherical");
}
```

Builder Design Pattern Implementation ...

Solution 3: Class responsibility

```
public class Robot {
  //same members with one long constructor
}
public class RobotBuilder {
  String type, headType, bodyType, legsType;
  // all setters
  public RobotBuilder setPrice(String type){   this.type = type;   return this;   }
  public Robot build(){
    if(type == null)
       throw new IllegalArgumentException("Type is required");
    return new Robot(type, headType, bodyType, legsType);
  }
}
```

Builder Design Pattern Implementation ...

Solution 3: Class responsibility ...

```
public static void main(String[] args){
   RobotBuilder builder = new RobotBuilder()
                              .setType("Spherical").setPrice(1000)
                              .setBodyType("Spherical").setHeadType("Spherical")
                              .setLegsType("Spherical");
   Robot robot = builder.build();

   //Not allowed to use direct Robot class constructor
   //Solved using packages
   Robot robot2 = new Robot(..., ..., ..., ...);
}
```

Factory Pattern vs. Builder Pattern

- Both separates object creation from object use.
- Factory pattern allows you to determine object type during runtime.
- Builder pattern allows you to create complex object with full control on its steps ordering.

Ch7: Structural Design Patterns

Adapter Design Pattern ...

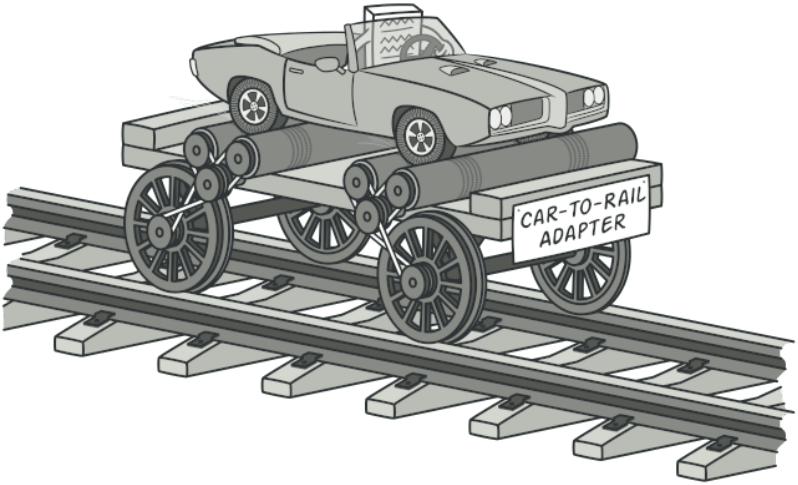
Problem

When dealing with **a third-party interface, you do not have access to edit its code**, it becomes problematic.

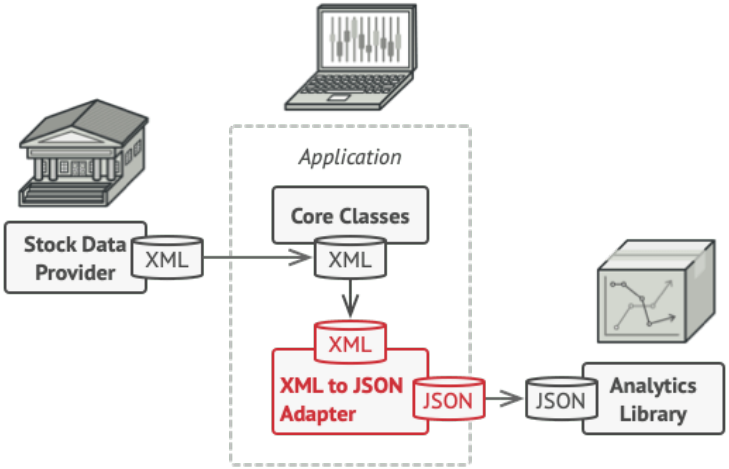
Solution

A class that is capable of adapting your interface function calls to the required one.

Adapter Design Pattern



Adapter Design Pattern Examples



Adapter Design Pattern Examples ...

```
public double clacSalary(XDocument empData){
  /* expected input
    <Employee>
       <Name> Emp name </Name>
       <BasicSalary> 1000 </BasicSalary>
    </Employee> */
}
// vendor function
public EmpJsonObject clacSalary(int empID){
  /* output format
    {"Name": "Emp name",
    "BasicSalary": 1000} */
}
```

Adapter Design Pattern Implementation

```
public interface IEnemy{   public void fire();   public void refill(); }

public class PlaneEnemy implements IEnemy{
   @Override
   public void fire(){ System.out.println("Plane is Firing Cannons"); }
   @Override
   public void refill(){ System.out.println("Plane is Filling Tank"); }
}
public class WarTankEnemy implements IEnemy{
   @Override
   public void fire(){ System.out.println("War Tank is Firing Bombs"); }
   @Override
   public void refill(){ System.out.println("War Tank is Filling Tank"); }
}
```

Adapter Design Pattern Implementation ...

```
public class MyGame{
   public static void main(String... args){
      ArrayList<IEnemy> enemies = new ArrayList<>;
      enemies.add(new PlaneEnemy());
      enemies.add(new WarTankEnemy());
      for(IEnemy enemy : enemies){
         enemy.fire();
         enemy.refill();
      }
   }
}
```

Adapter Design Pattern Implementation ...

Problem: A developer creates a game enemy called (ZombieEnemy)

```
public class ZombieEnemy{
   public void attack(){
      System.out.println("Zombie is Attacking");
   }
   public void reset(){
      System.out.println("Zombie is Sleeping");
   }
}
```

Adapter Design Pattern Implementation ...

Problem: Zombie class is incompatible with our game enemies

```
public class MyGame{
  public static void main(String... args){
    ArrayList<IEnemy> enemies = new ArrayList<>;
    enemies.add(new PlaneEnemy());
    enemies.add(new WarTankEnemy());
    enemies.add(new ZombieEnemy());
    for(IEnemy enemy : enemies){
      enemy.fire();
      enemy.refill();
    }
  }
}
```

Adapter Design Pattern Implementation ...

Solution: creating ZombieEnemyAdapter

```
public class ZombieEnemyAdapter implements IEnemy{
  ZombieEnemy zombie;
  public ZombieEnemyAdapter(ZombieEnemy zombie){
    this.zombie = zombie;
  }
  public void fire(){
    this.zombie.attack();
  }
  public void refill(){
    this.zombie.reset();
  }
}
```

Adapter Design Pattern Implementation ...

Solution: using ZombieEnemyAdapter within our game enemy

```
public class MyGame{
  public static void main(String... args){
    ArrayList<IEnemy> enemies = new ArrayList<>;
    enemies.add(new PlaneEnemy());
    enemies.add(new WarTankEnemy());
    enemies.add(new ZombieEnemyAdapter(new ZombieEnemy()));
    for(IEnemy enemy : enemies){
      enemy.fire();
      enemy.refill();
    }
  }
}
```

Adapter Design Pattern

- The Adapter pattern lets you create a middle-layer class that serves as a translator between your code and a legacy class, a 3rd-party class or any other class with a weird interface.
- Converts the interface of a class into another interface a client expects.
- Adapter lets classes work together that could not otherwise because of incompatible interfaces.
- When an outside component provides captivating functionality that we'd like to reuse, but it's incompatible with our current application. A suitable Adapter can be developed to make them compatible with each other.