Atypon Training
March – May, 2019
Software Design Patterns
(Creational Patterns)

Instructor: Dr. Fahed Jubair

fjubair@atypon.com

ATYPON WILEY

1

Motivation

- In previous lectures, we learned four design principles (abstraction, encapsulation, decomposition, and generalization) for creating modular and maintainable object-oriented software
- Using these principles, software experts have created design patterns, which are similar to cooking recipes for creating good software
- A design pattern is a practical proven solution to a recurring design problem
- Design patterns are already shown to work in practice by experts, which make them valuable for both academia and industry

2

Gang of Four's Design Patterns

- The gang of four are the four authors of the famous book: *Design Patterns: Elements of Reusable Object-Oriented Software, 1994*
  https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/
- The four authors have provided twenty-three design patterns that categorized into three groups:
  - Creational patterns
  - Structural patterns
  - Behavioral patterns

3

Creational Patterns

- Provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator
- Example of creational patterns:
  - Singleton pattern
  - Factory Method pattern

4

Singleton Pattern

- Ensure a class has only one instance that is publically accessible
- Example: a printer needs to have only one queue of jobs, i.e., if there are more than one queue, then it would create confusion
- How to ensure an object has a single instance?
  - Make the constructor private so that is not possible to instantiate an instance from the outside world
  - Use a static method to provide an internally created instance within the class

5

Singleton Pattern Example

```
public class SingletonObject {

    private static SingletonObject soloInstance = new SingletonObject();

    private SingletonObject(){

    }

    public static SingletonObject getInstance(){
        return soloInstance;
    }

}
```

A private constructor is not accessible outside the class

- Get the only instance available
- A single access point in the program
- Access is restricted to one, shared instance

6

Factory Method Pattern

- Define an interface for creating an object, but let subclasses decide which class to instantiate
- Hide creation logic from clients
- To achieve this, a client class defers objects creation to a *factory class*
- The created objects are accessed using a common interface

7

Factory Method Pattern Example

- Assume you have a computer store which sells different types of computers: PC, laptops, tablets, etc.
- We need a class to represent each kind of a computer
- We need a class to represent the computer store
- Assume in the computer store class, there is a method “getPrice” for retrieving the price of a given type of a computer
- How do you write a program for the computer store?

8

Factory Method Pattern Example Trial 1

```
public class ComputerStore {


    public int getPrice (String computerType){
        if(computerType.equals("Laptop"))
            return new Laptop().getPrice();
        else if(computerType.equals("PC"))
            return new PC().getPrice();
        else if(computerType.equals("Tablet"))
            return new Tablet().getPrice();
        return 0;
    }


}
```

Remarks:

- In this code, client class is responsible for object instantation
- What if new types of computers suddenly became available?
- What if there are other classes that use computers (e.g., ElectronicsStore, ComputerLab, etc)

Takeaways:

- Cliens' implementation need to change if new types of computer classes are added
- The is a bad code design

9

Factory Method Pattern Example Trial 2

- Let us use the Factory Method pattern
- Create an interface, called "computer", which implemented by PC, Laptop and Tablet classes (as well as any new kind of computers that added in the future)
- Create ComputerFactory class, which is responsible for creating instances of computer subclasses
- All clients (such as ComputerStore, ElectronicsStore, ComputerLab, etc) use the ComputerFactory class for instances' creation
- This way, only the ComputerFactory class needs to change when new computes are added and clients' codes remain unaffected

10

Factory Method Pattern Example Computer Classes

```
public interface Computer {
   public int getPrice();
}

public class PC implements Computer {
    public int getPrice(){ return 1100; }
}


public class Laptop implements Computer {
    public int getPrice(){ return 800; }
}

public class Tablet implements Computer {
    public int getPrice(){ return 100; }
}
```


11

Factory Method Pattern Example ComputerFactory Class

```
public class ComputerFactory {

   public Computer createComputer(String computerType){
      if(computerType.equalsIgnoreCase("Laptop"))
         return new Laptop();
      else if(computerType.equalsIgnoreCase("PC"))
         return new PC();
      else if(computerType.equalsIgnoreCase("Tablet"))
         return new Tablet();
      return null;
   }

}
```

New types of Computers are added here

12

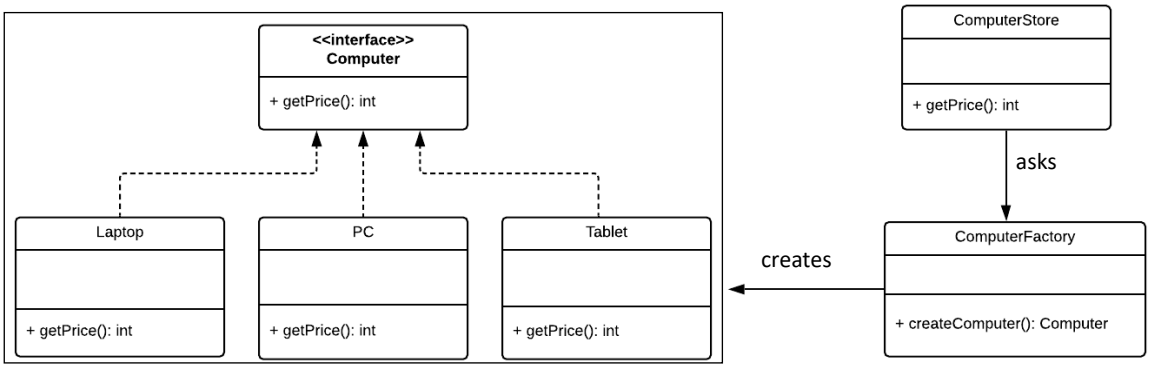
Factory Method Pattern Example ComputerStore Class

```
public class ComputerStore {

   private ComputerFactory computerFactory;

   public ComputerStore(){
      computerFactory = new ComputerFactory();
   }

   public int getPrice (String computerType){
      Computer computer = computerFactory.createComputer(computerType);
      return computer.getPrice();
   }

}
```

The creation of computer instances is decoupled from the implementation of clients

13

Factory Method Pattern Example UML Diagrams



14

Factory Method Pattern Example Test

```
public class ComputerStoreTest {

   public static void main(String[] args){

      ComputerStore computerStore = new ComputerStore();
      System.out.println(computerStore.getPrice("laptop"));
      System.out.println(computerStore.getPrice("pc"));
      System.out.println(computerStore.getPrice("tablet"));

   }
}
```


15