Nesneye Yönelik Yazılım Mühendisliği (376)

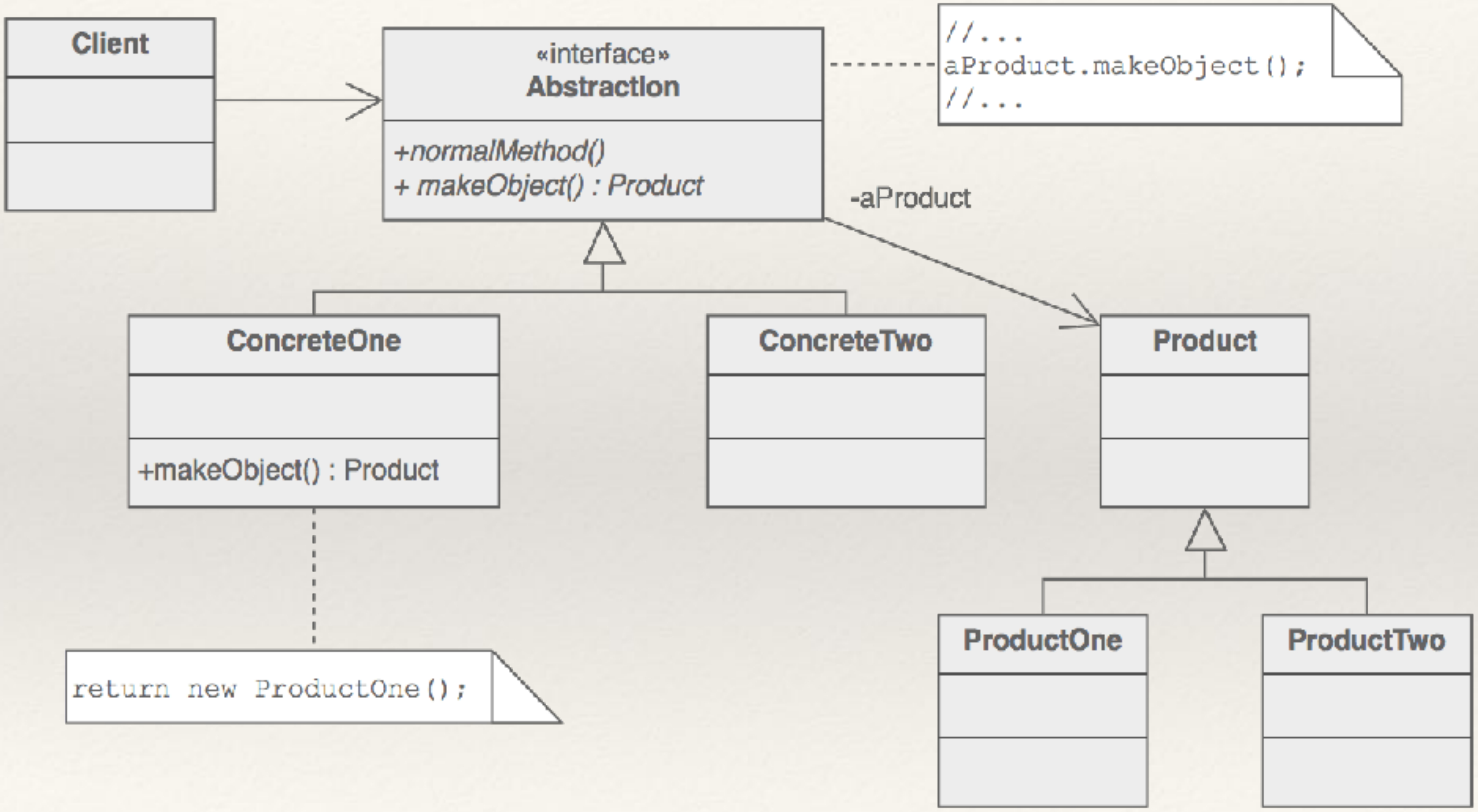
Yrd. Doç. Dr. Ahmet Arif AYDIN

Creational Design Patterns : Factory

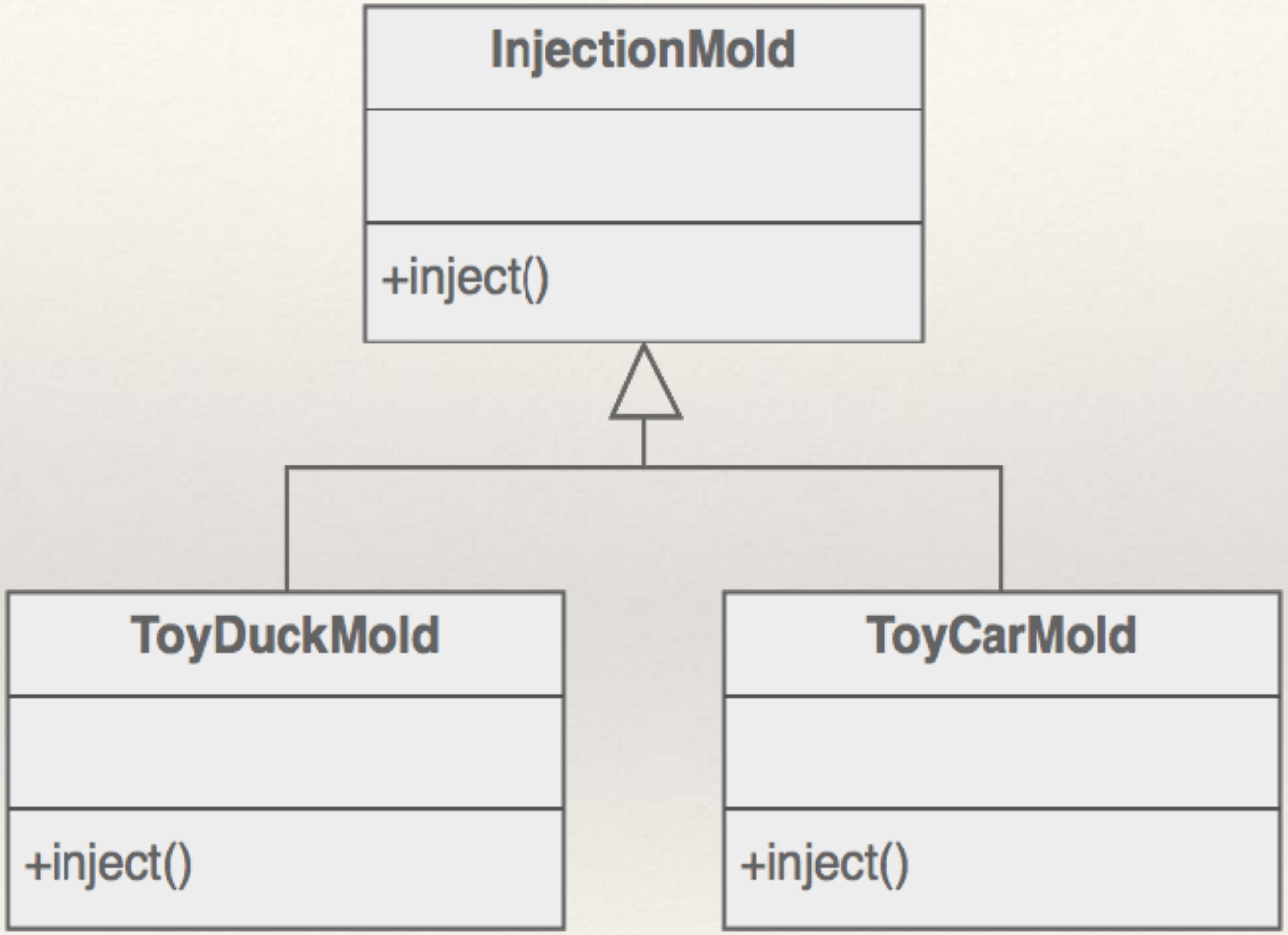
- Aynı arayüzü (interface) kullanan neslerin oluşturulması ve yönetimini sağlar (Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses)

- Kapsüllemeyi kullanarak somut klasların örneğinin oluşturulmasını sağlar .
  - The Factory Method pattern encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a factory method. The Factory Method selects an appropriate class from a class hierarchy based on the application context and other influencing factors. It then instantiates the selected class and returns it as an instance of the parent class type.

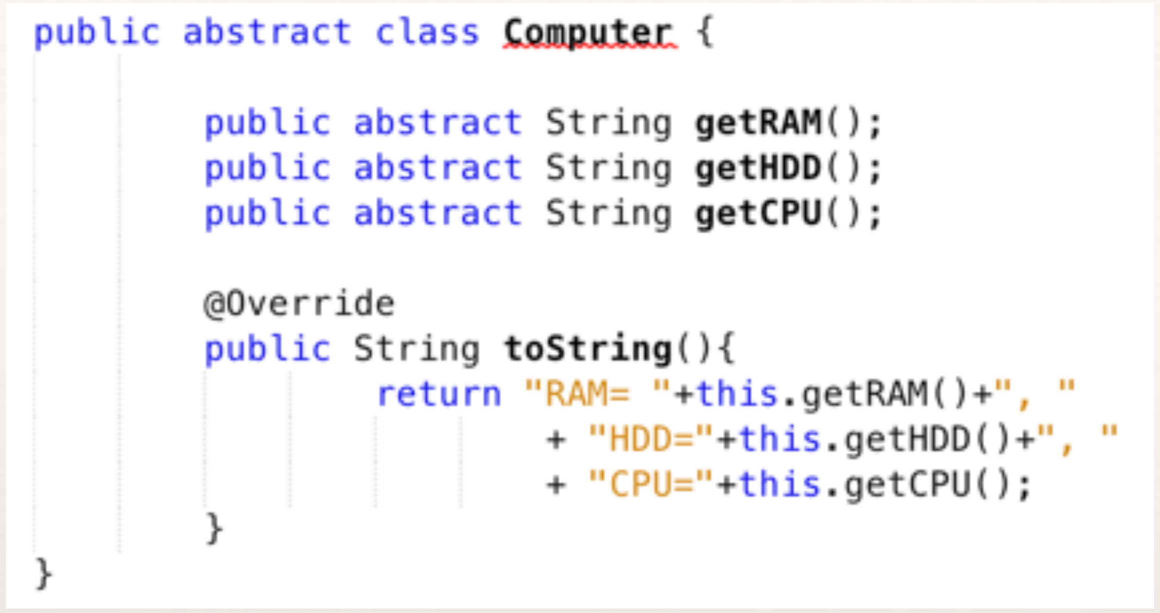
Creational Design Patterns : Factory



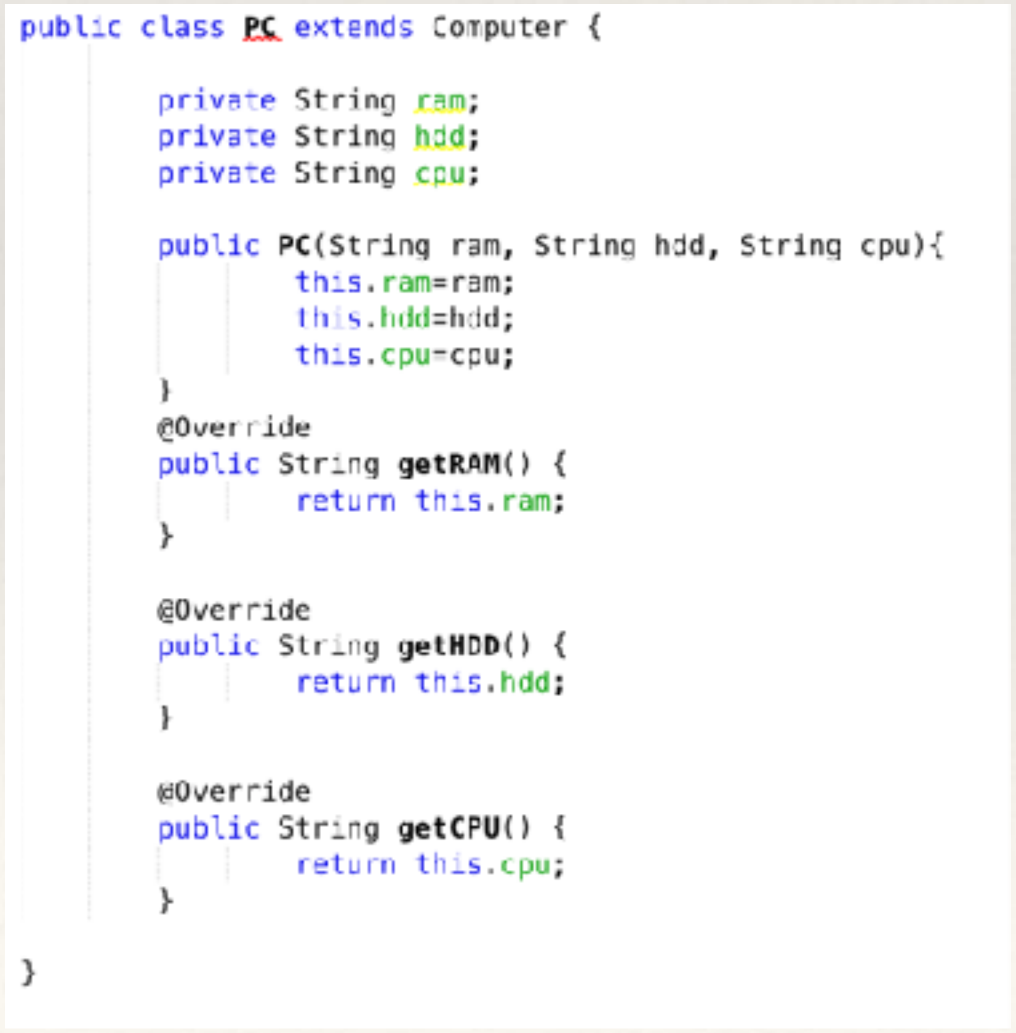
Creational Design Patterns : Factory



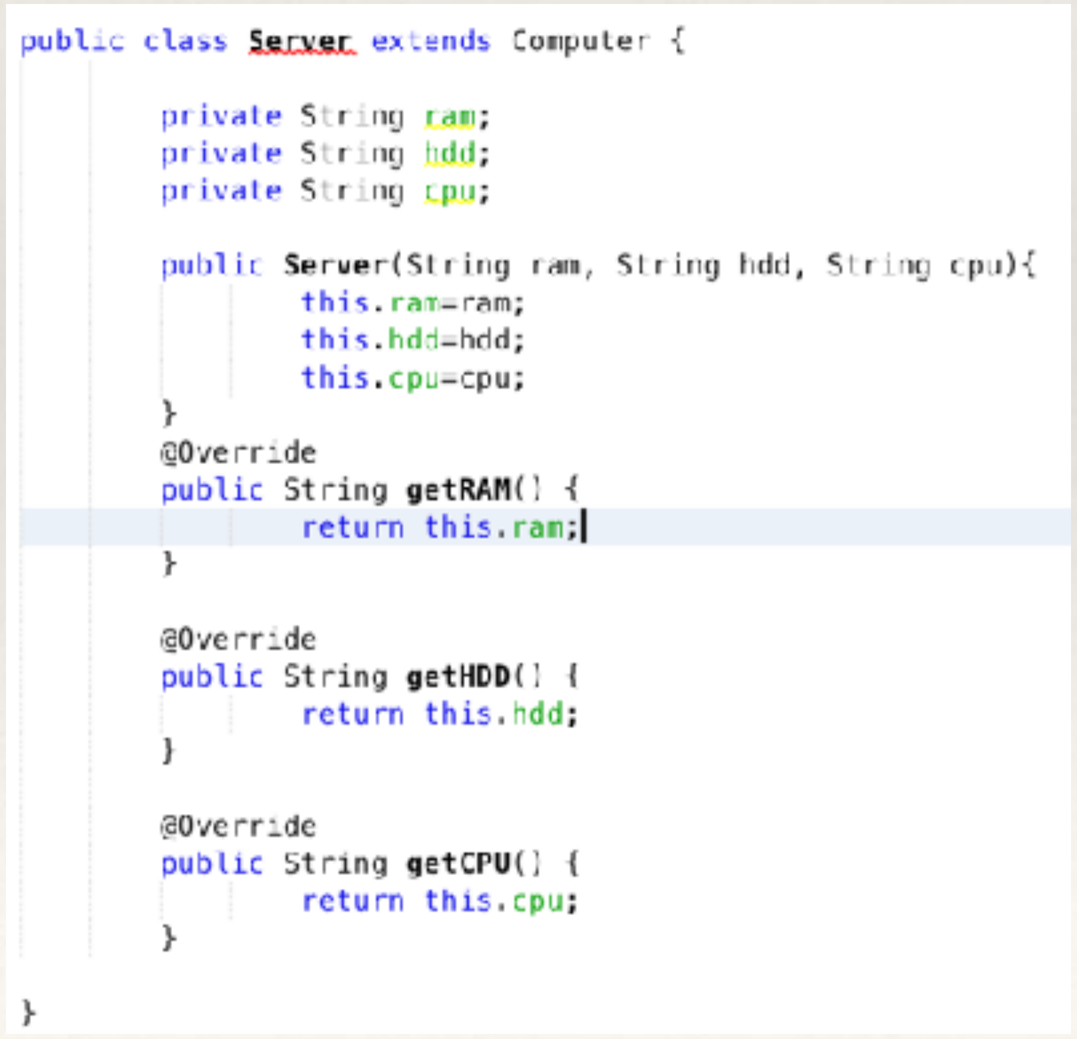
Creational Design Patterns : Factory



```java
public abstract class Computer {

        public abstract String getRAM();
        public abstract String getHDD();
        public abstract String getCPU();

        @Override
        public String toString(){
                return "RAM= "+this.getRAM()+", "
                        + "HDD="+this.getHDD()+", "
                        + "CPU="+this.getCPU();
        }
}
```

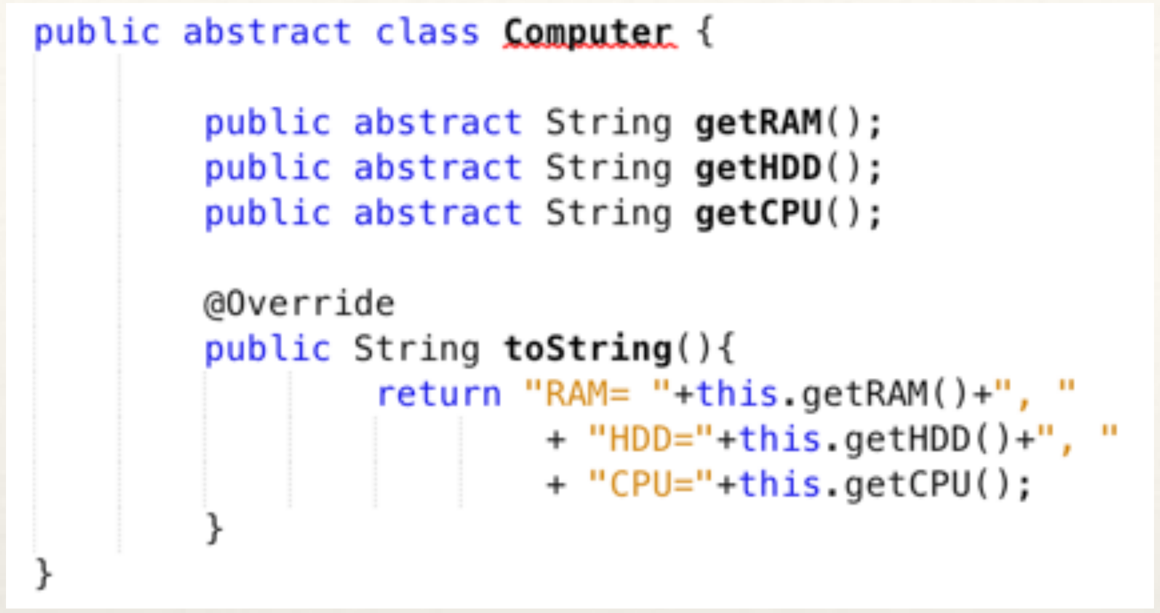


```java
public class PC extends Computer {

        private String ram;
        private String hdd;
        private String cpu;

        public PC(String ram, String hdd, String cpu){
                this.ram=ram;
                this.hdd=hdd;
                this.cpu=cpu;
        }
        @Override
        public String getRAM() {
                return this.ram;
        }

        @Override
        public String getHDD() {
                return this.hdd;
        }

        @Override
        public String getCPU() {
                return this.cpu;
        }

}
```

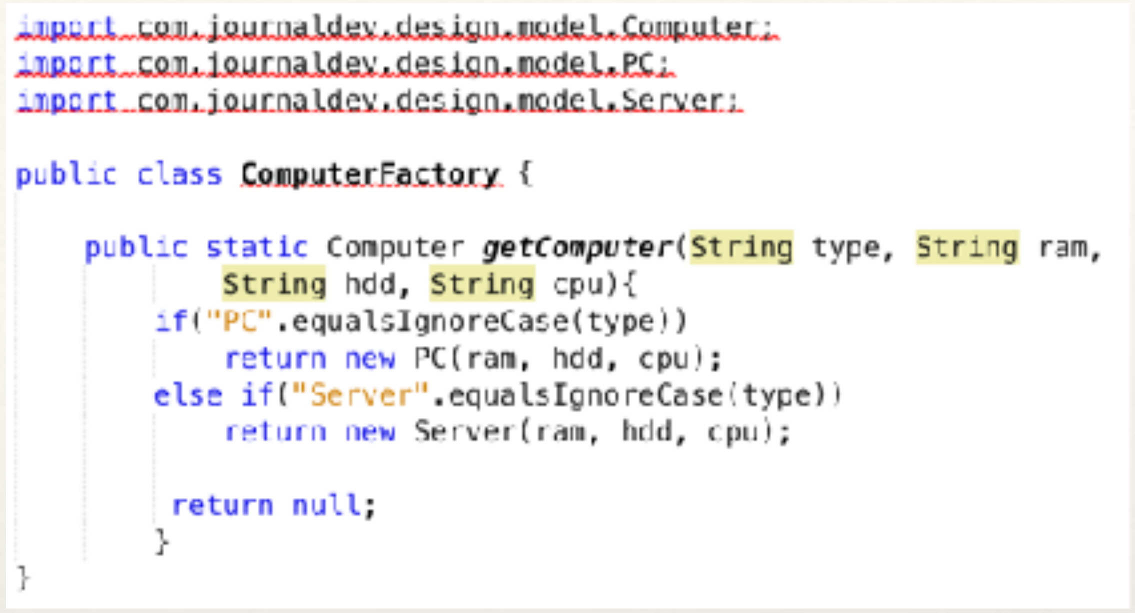


```java
public class Server extends Computer {

        private String ram;
        private String hdd;
        private String cpu;

        public Server(String ram, String hdd, String cpu){
                this.ram=ram;
                this.hdd=hdd;
                this.cpu=cpu;
        }
        @Override
        public String getRAM() {
                return this.ram;
        }

        @Override
        public String getHDD() {
                return this.hdd;
        }

        @Override
        public String getCPU() {
                return this.cpu;
        }

}
```

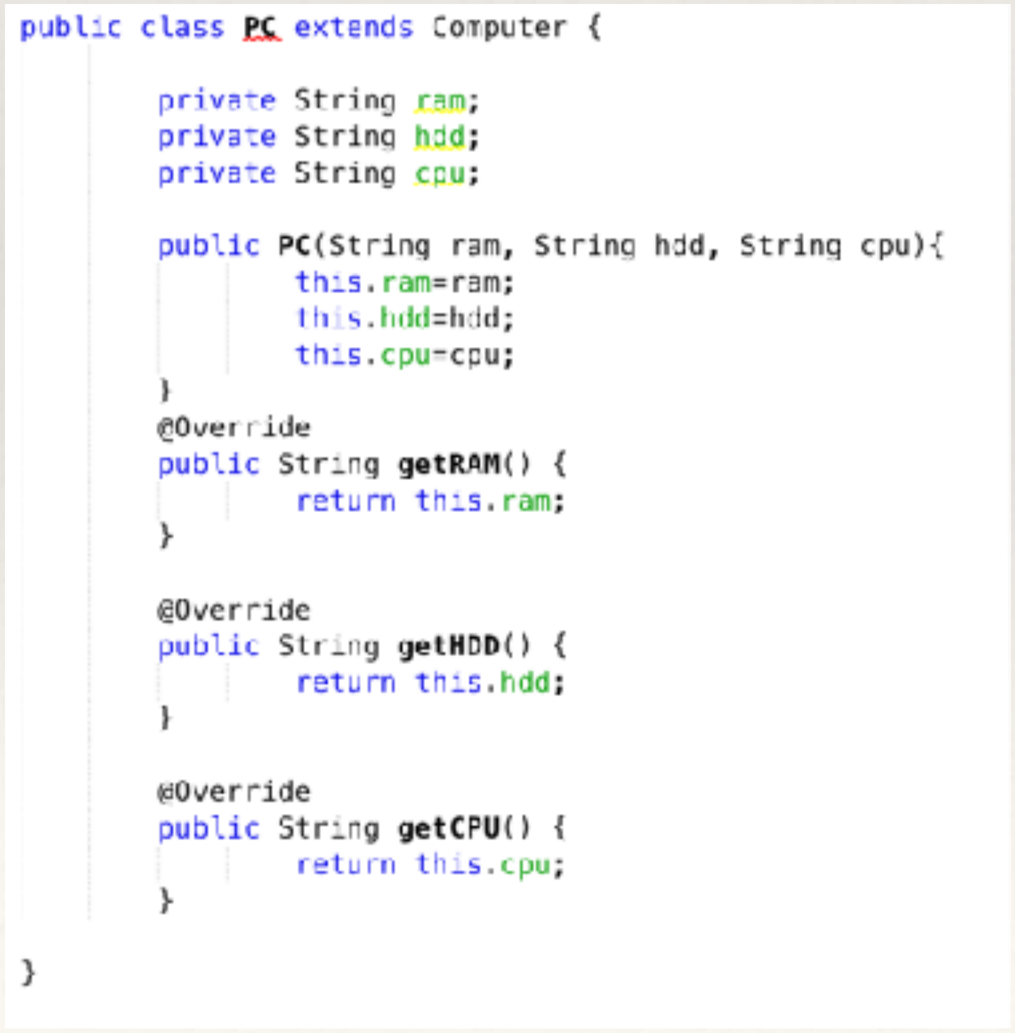
Creational Design Patterns : Factory



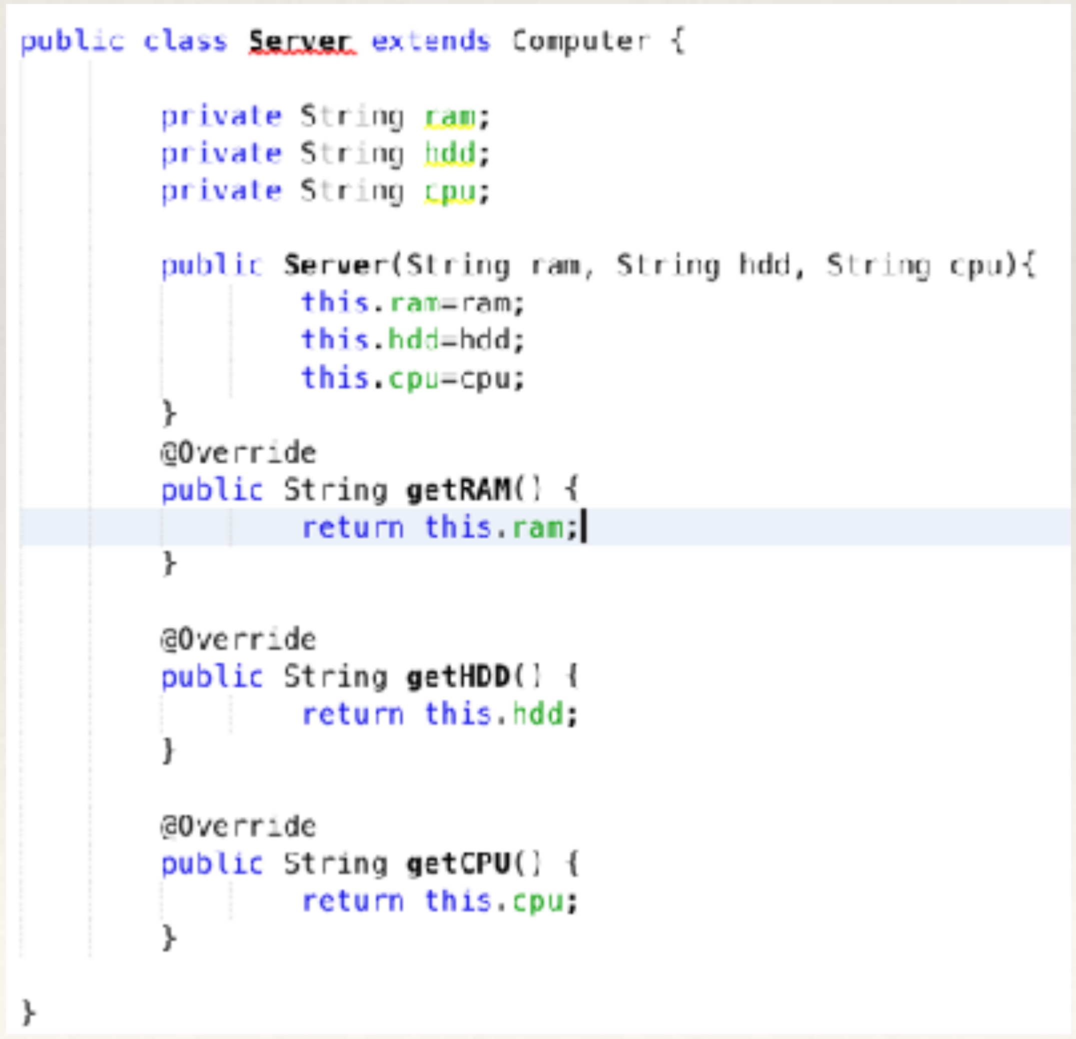
```java
public abstract class Computer {

        public abstract String getRAM();
        public abstract String getHDD();
        public abstract String getCPU();

        @Override
        public String toString(){
                return "RAM= "+this.getRAM()+", "
                        + "HDD="+this.getHDD()+", "
                        + "CPU="+this.getCPU();
        }
}
```



```java
import com.journaldev.design.model.Computer;
import com.journaldev.design.model.PC;
import com.journaldev.design.model.Server;

public class ComputerFactory {

    public static Computer getComputer(String type, String ram,
            String hdd, String cpu){
        if("PC".equalsIgnoreCase(type))
            return new PC(ram, hdd, cpu);
        else if("Server".equalsIgnoreCase(type))
            return new Server(ram, hdd, cpu);

         return null;
        }
}
```



```java
public class PC extends Computer {

        private String ram;
        private String hdd;
        private String cpu;

        public PC(String ram, String hdd, String cpu){
                this.ram=ram;
                this.hdd=hdd;
                this.cpu=cpu;
        }
        @Override
        public String getRAM() {
                return this.ram;
        }

        @Override
        public String getHDD() {
                return this.hdd;
        }

        @Override
        public String getCPU() {
                return this.cpu;
        }

}
```

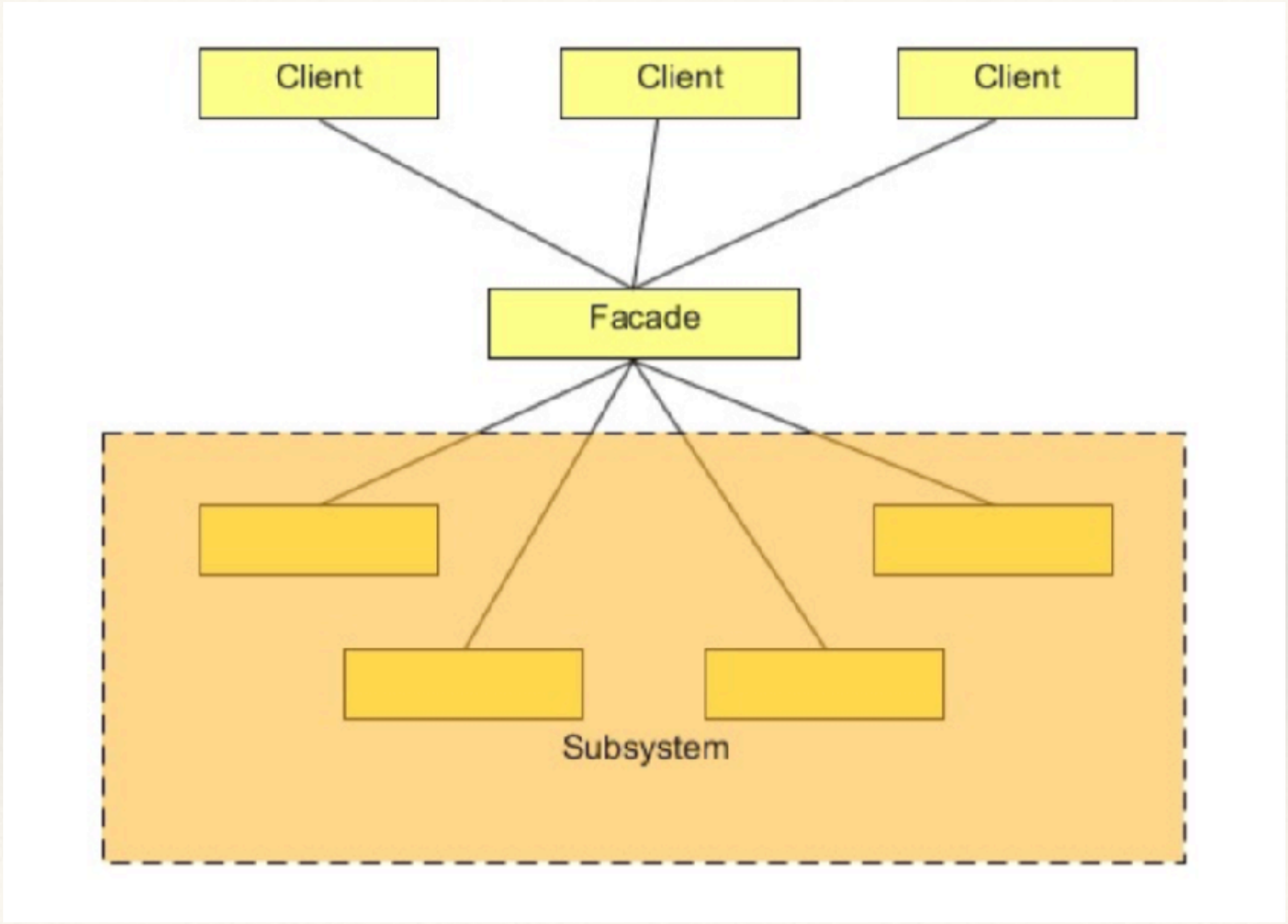


```java
public class Server extends Computer {

        private String ram;
        private String hdd;
        private String cpu;

        public Server(String ram, String hdd, String cpu){
                this.ram=ram;
                this.hdd=hdd;
                this.cpu=cpu;
        }
        @Override
        public String getRAM() {
                return this.ram;
        }

        @Override
        public String getHDD() {
                return this.hdd;
        }

        @Override
        public String getCPU() {
                return this.cpu;
        }

}
```

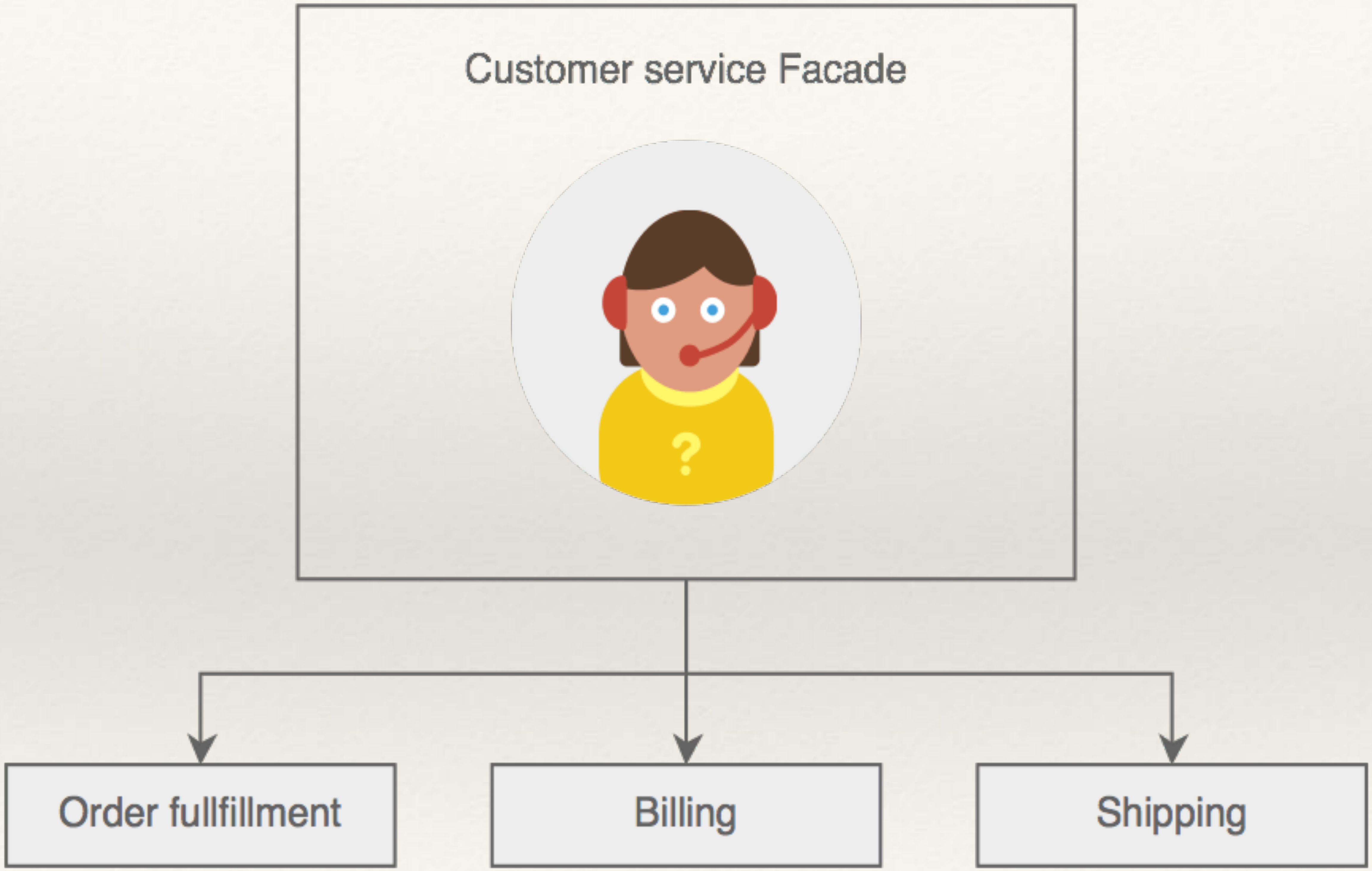
Structural Design Patterns: Facade

- Complex olan bir sistemin kullanıcılara basit ve anlaşılır olarak sunulmasını sağlayan tasarım kalıbıdır.
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
  - Wrap a complicated subsystem with a simpler interface.
  - Bir sistemde yüksek seviyeli bir arayüz oluşturup alt arayüzlerin tek bir arayüz altında gösterimini sağlar

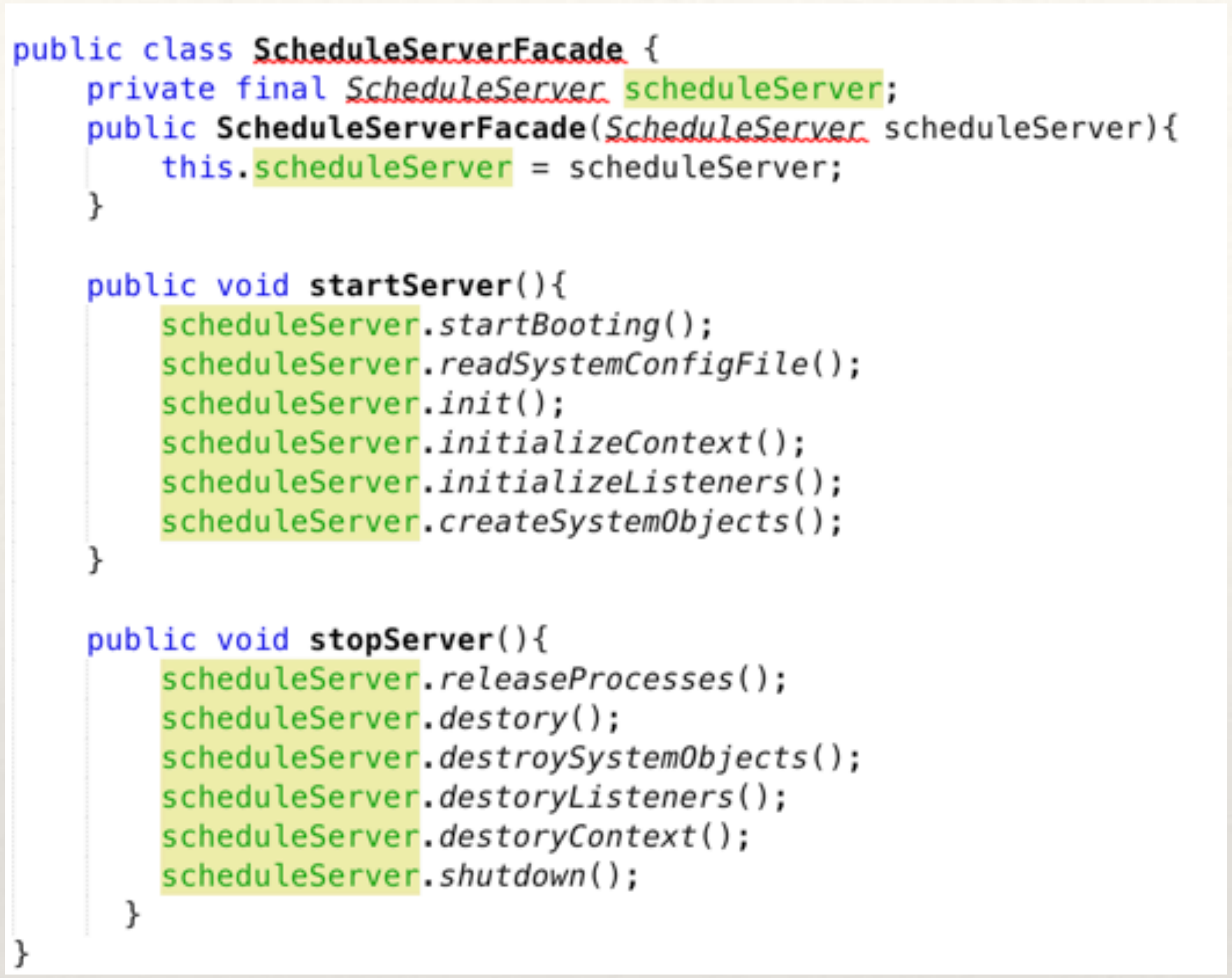
Structural Design Patterns: Facade



Structural Design Patterns: Facade

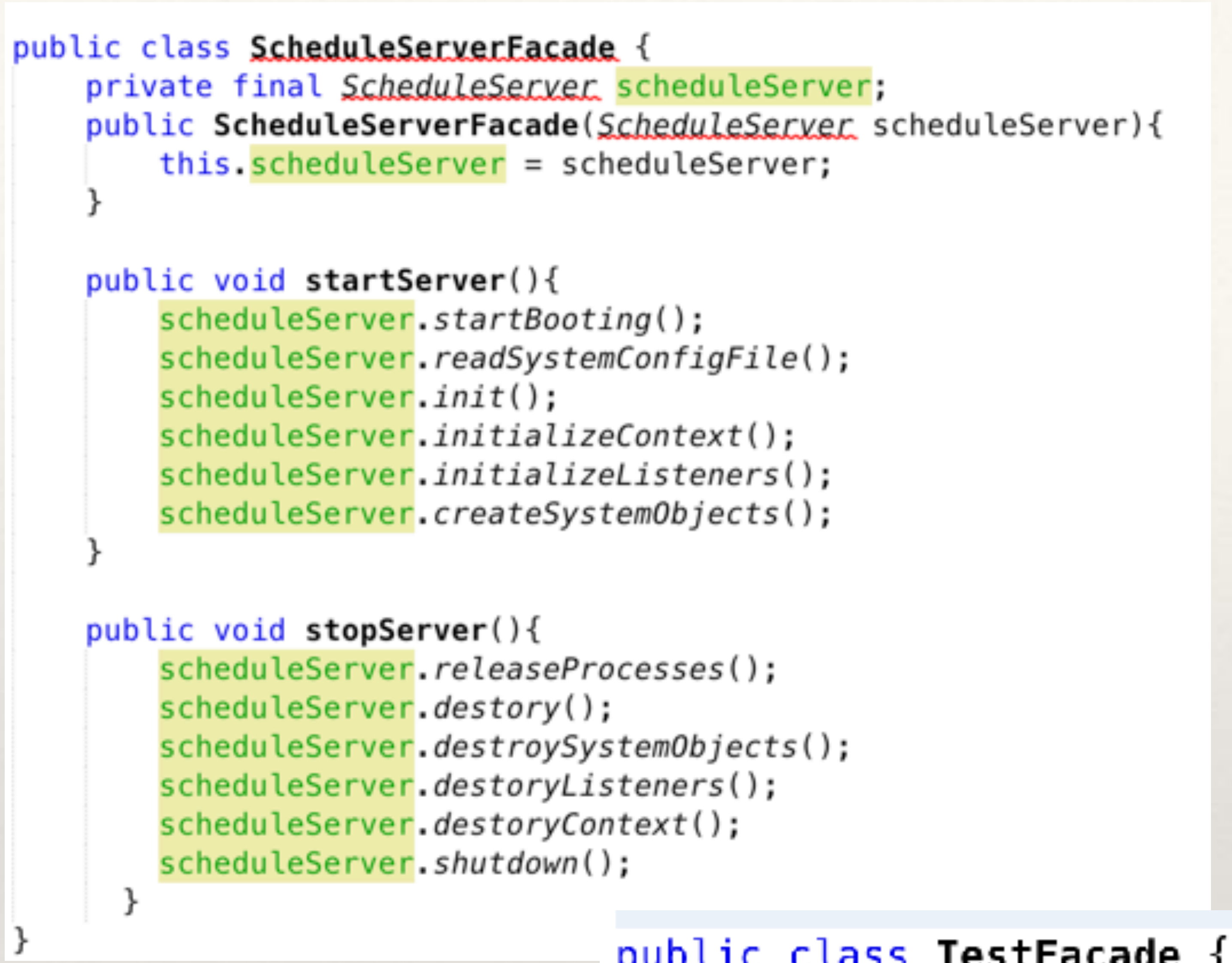


Structural Design Patterns: Facade

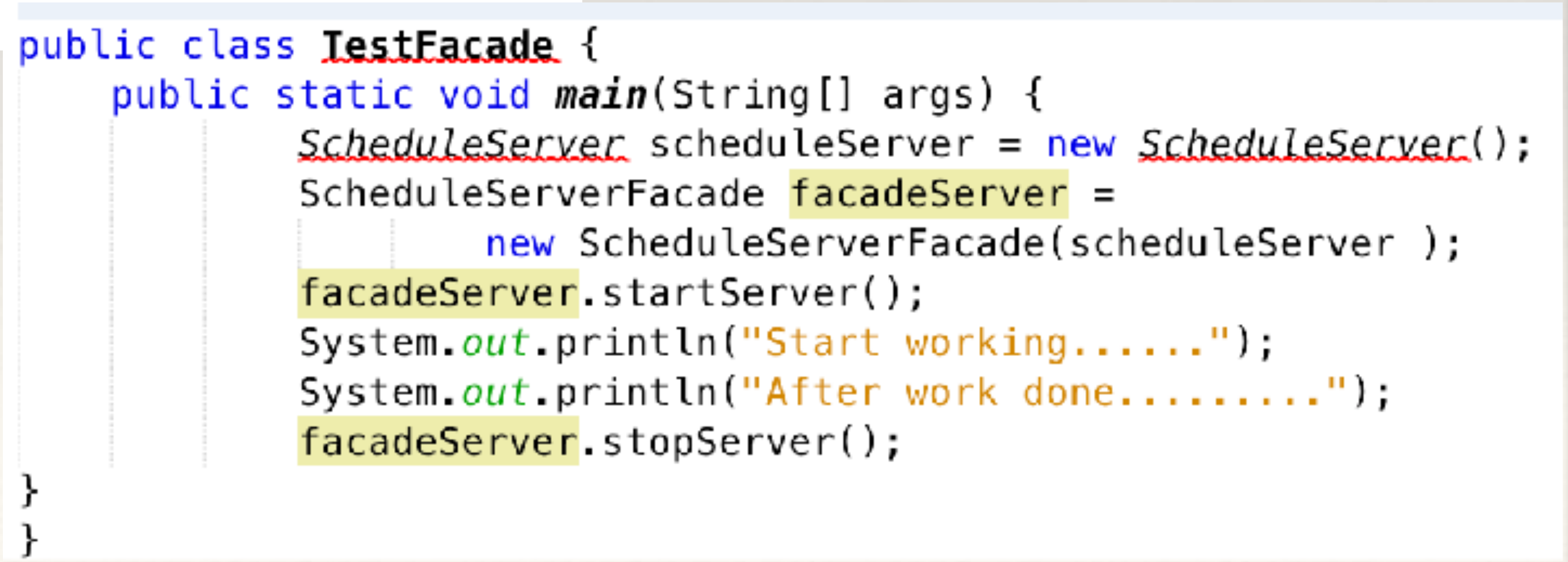


```
public class ScheduleServerFacade {
    private final ScheduleServer scheduleServer;
    public ScheduleServerFacade(ScheduleServer scheduleServer){
        this.scheduleServer = scheduleServer;
    }

    public void startServer(){
        scheduleServer.startBooting();
        scheduleServer.readSystemConfigFile();
        scheduleServer.init();
        scheduleServer.initializeContext();
        scheduleServer.initializeListeners();
        scheduleServer.createSystemObjects();
    }

    public void stopServer(){
        scheduleServer.releaseProcesses();
        scheduleServer.destory();
        scheduleServer.destroySystemObjects();
        scheduleServer.destoryListeners();
        scheduleServer.destoryContext();
        scheduleServer.shutdown();
      }
}
```

Structural Design Patterns: Facade



```java
public class ScheduleServerFacade {
    private final ScheduleServer scheduleServer;
    public ScheduleServerFacade(ScheduleServer scheduleServer){
        this.scheduleServer = scheduleServer;
    }

    public void startServer(){
        scheduleServer.startBooting();
        scheduleServer.readSystemConfigFile();
        scheduleServer.init();
        scheduleServer.initializeContext();
        scheduleServer.initializeListeners();
        scheduleServer.createSystemObjects();
    }

    public void stopServer(){
        scheduleServer.releaseProcesses();
        scheduleServer.destory();
        scheduleServer.destroySystemObjects();
        scheduleServer.destoryListeners();
        scheduleServer.destoryContext();
        scheduleServer.shutdown();
     }
}
```

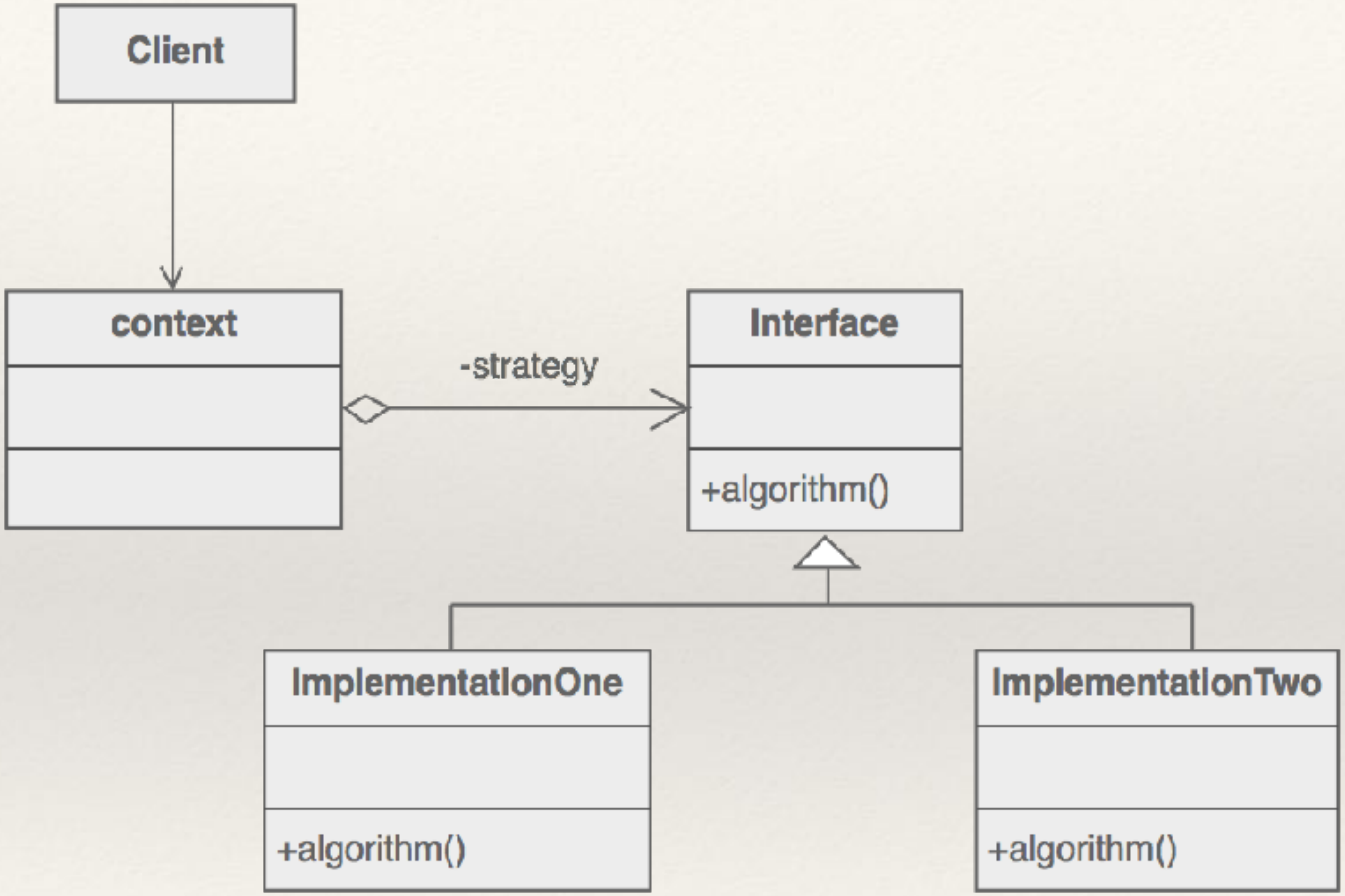


```java
public class TestFacade {
    public static void main(String[] args) {
            ScheduleServer scheduleServer = new ScheduleServer();
            ScheduleServerFacade facadeServer =
                    new ScheduleServerFacade(scheduleServer );
            facadeServer.startServer();
            System.out.println("Start working......");
            System.out.println("After work done.........");
            facadeServer.stopServer();
}
}
```

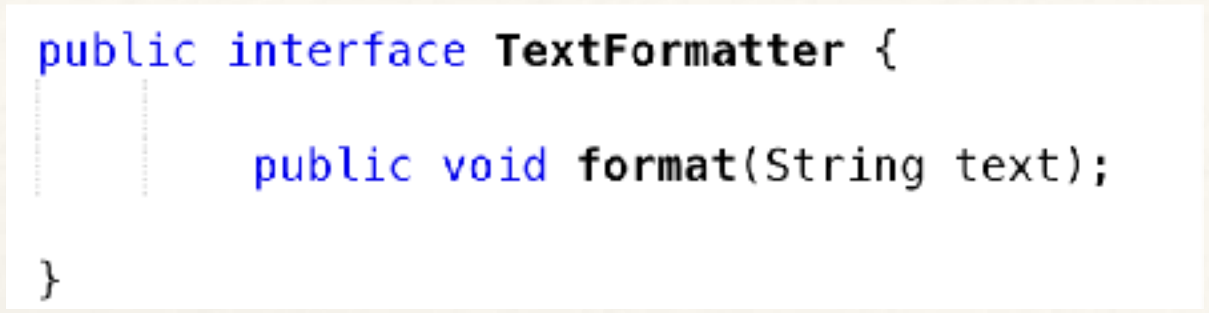
Behavioral Design Patterns: Strategy

- Birbirinin yerine dönüşümlü olarak kullanılabilecek olan algoritma veya yöntemleri tanımlamak için kullanılan tasarım kalıbıdır.
  - *defines a family of algorithms, encapsulating each one, and making them interchangeable. Strategy lets the algorithm vary independently from the clients that use it*)

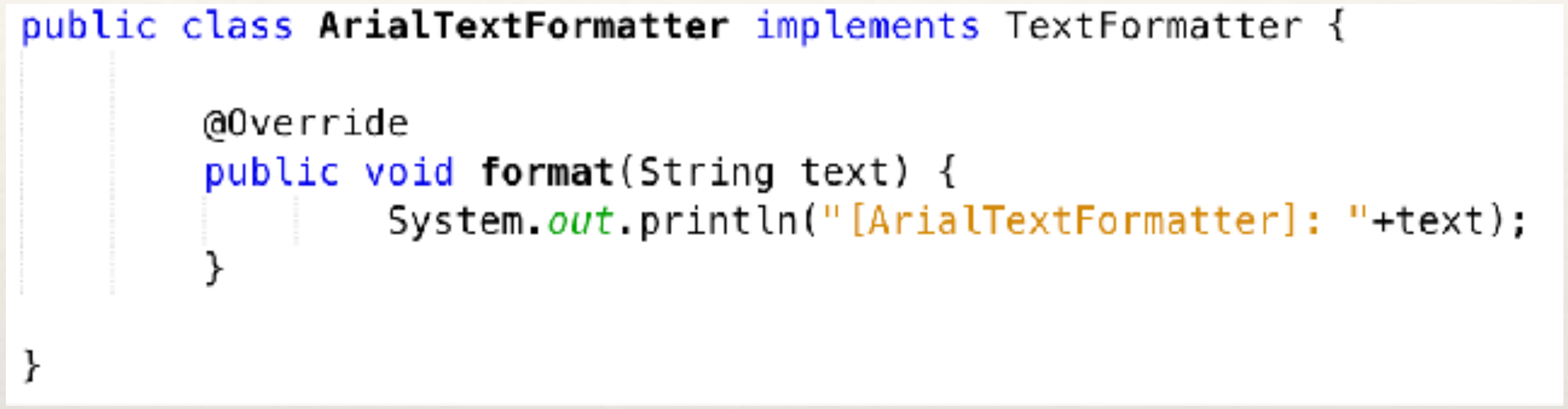
Behavioral Design Patterns: Strategy



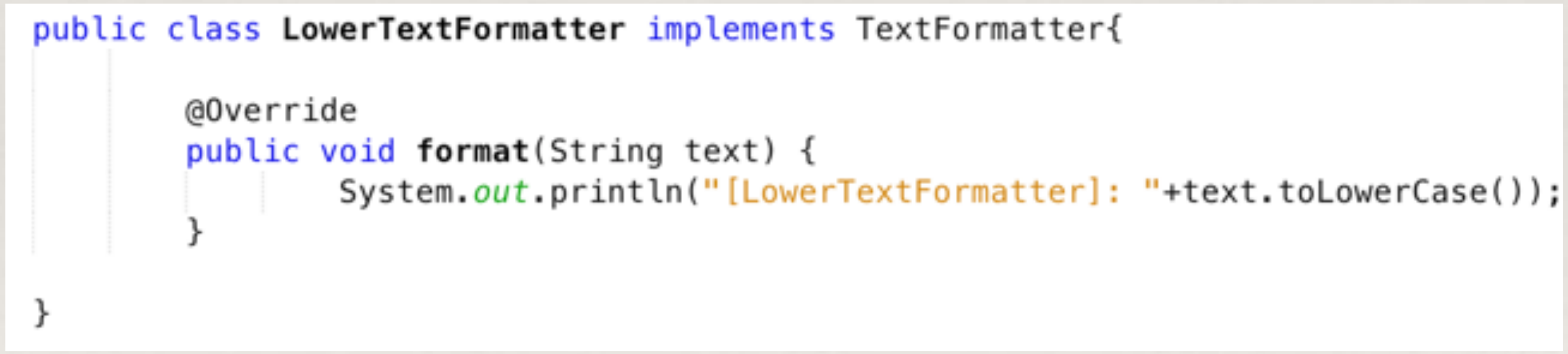
Behavioral Design Patterns: Strategy



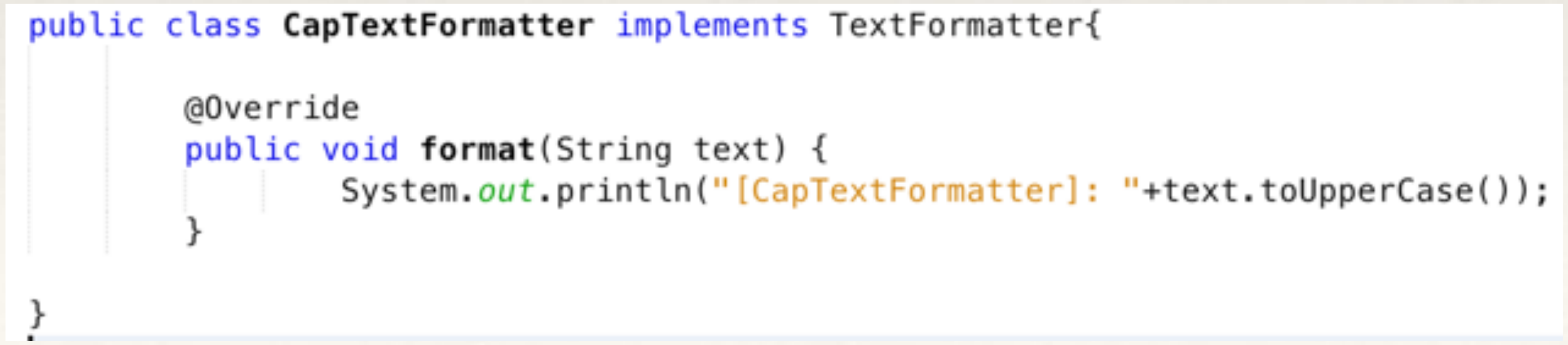
```java
public interface TextFormatter {

		public void format(String text);

}
```



```java
public class ArialTextFormatter implements TextFormatter {

	@Override
	public void format(String text) {
		System.out.println("[ArialTextFormatter]: "+text);
	}

}
```

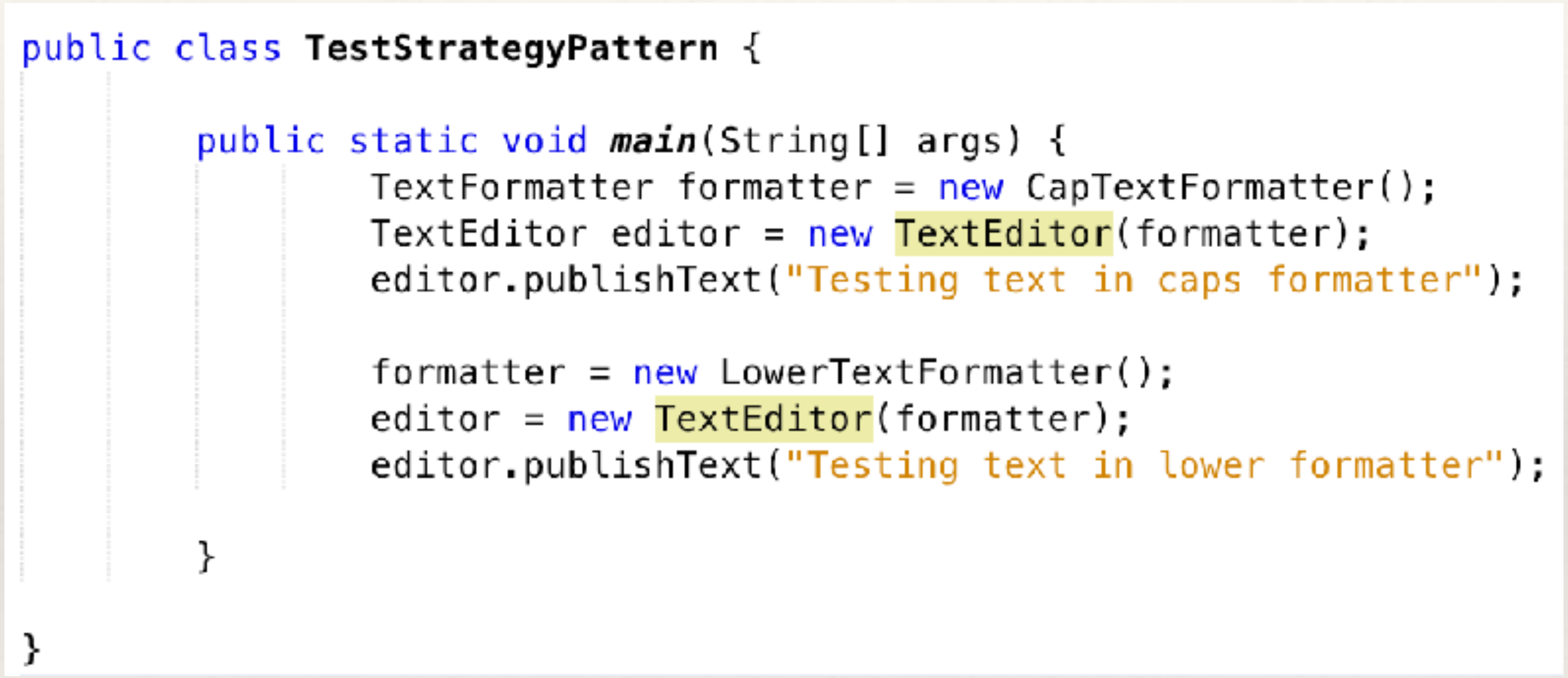


```java
public class LowerTextFormatter implements TextFormatter{

	@Override
	public void format(String text) {
		System.out.println("[LowerTextFormatter]: "+text.toLowerCase());
	}

}
```



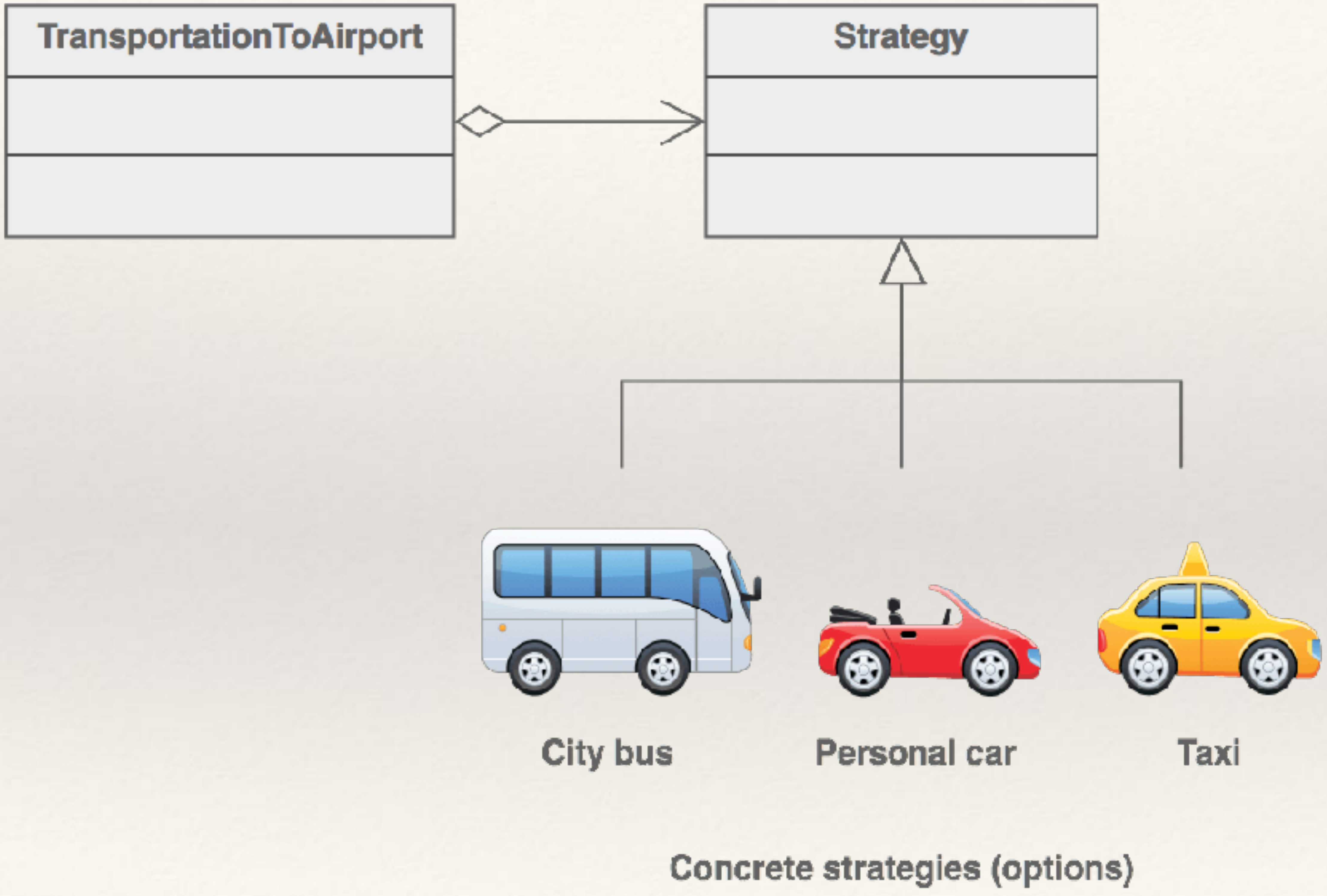
```java
public class CapTextFormatter implements TextFormatter{

	@Override
	public void format(String text) {
		System.out.println("[CapTextFormatter]: "+text.toUpperCase());
	}

}
```

Behavioral Design Patterns: Strategy

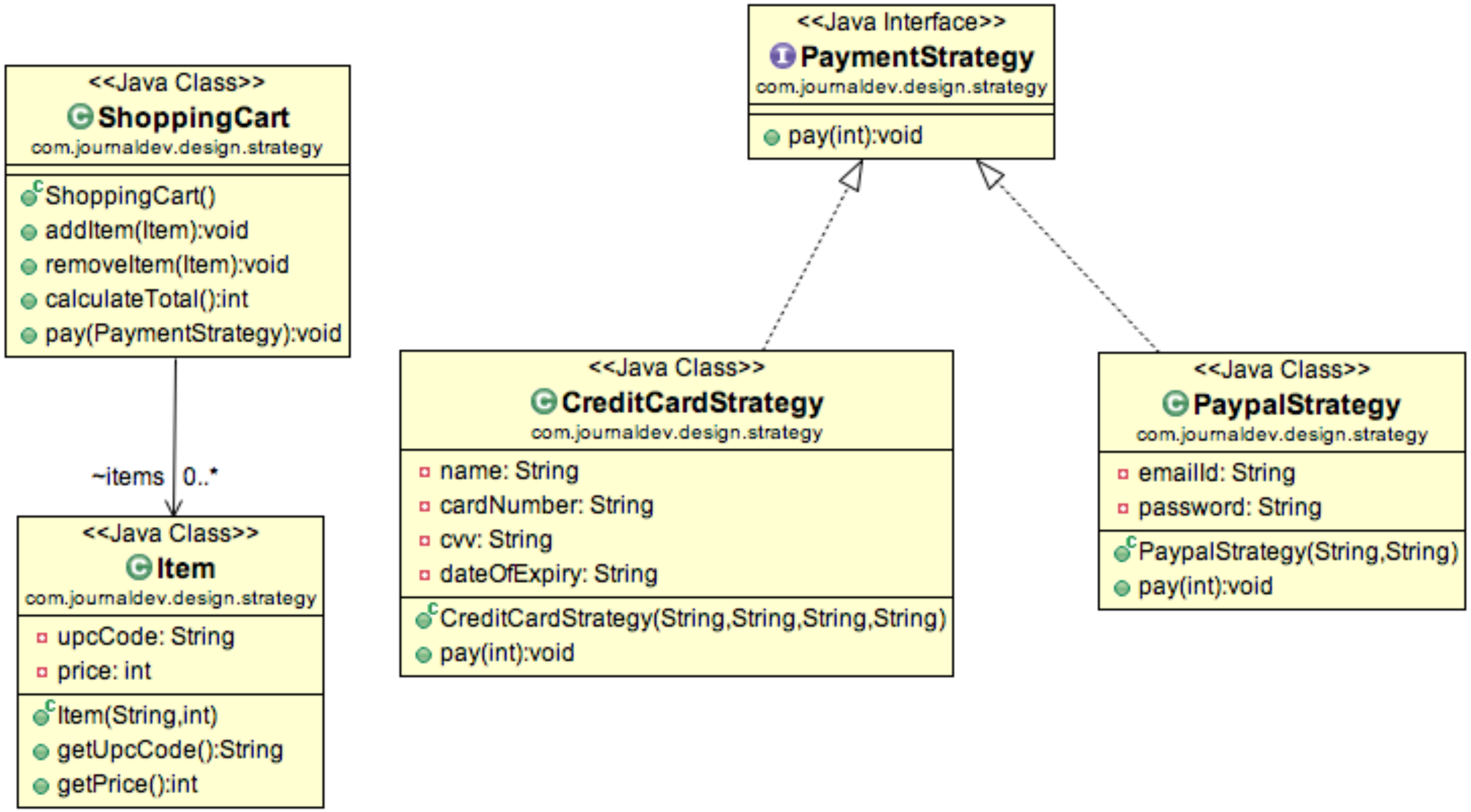


```
public class TestStrategyPattern {

    public static void main(String[] args) {
        TextFormatter formatter = new CapTextFormatter();
        TextEditor editor = new TextEditor(formatter);
        editor.publishText("Testing text in caps formatter");

        formatter = new LowerTextFormatter();
        editor = new TextEditor(formatter);
        editor.publishText("Testing text in lower formatter");

    }

}
```

Behavioral Design Patterns: Strategy



Behavioral Design Patterns: Strategy



<<Java Class>>
ShoppingCart
com.journaldev.design.strategy
ShoppingCart()
addItem(Item):void
removeItem(Item):void
calculateTotal():int
pay(PaymentStrategy):void
~items
0..*
<<Java Class>>
Item
com.journaldev.design.strategy
upcCode: String
price: int
Item(String,int)
getUpcCode():String
getPrice():int
<<Java Interface>>
PaymentStrategy
com.journaldev.design.strategy
pay(int):void
<<Java Class>>
CreditCardStrategy
com.journaldev.design.strategy
name: String
cardNumber: String
cvv: String
dateOfExpiry: String
CreditCardStrategy(String,String,String,String)
pay(int):void
<<Java Class>>
PaypalStrategy
com.journaldev.design.strategy
emailId: String
password: String
PaypalStrategy(String,String)
pay(int):void