Design Pattern

Design Pattern

- Design Pattern
  - ✓ 비슷한 목적으로 사용되는 클래스의 모범 사례를 Pattern으로 정리하려고 하는 것이 Design Pattern
  - ✓ 프로그램의 세계에는 다양한 Design Pattern이 존재
  - ✓ 유명한 것이 GoF Design Pattern 인데 GoF(고프)란 The Gang of Four의 약자이며 에리히 감마, 리처드 헬름, 랄프 존슨, 존 블리시데스의 4명이 인스턴스지향 프로그래밍에 도움이 되는 Design Pattern을 도입해 《Design Patterns: Elements of Reusable Object Oriented Software》라는 제목의 책에 정리했으며 이 책에 소개된 23가지 Design Pattern
  - ✓ Design Pattern의 분류
    - ✓ 인스턴스의 생성에 관한 Pattern
    - ✓ 프로그램의 구조에 관한 Pattern
    - ✓ 인스턴스의 행동에 관한 Pattern
  - ✓ Design Pattern은 정교한 클래스 구조의 Pattern 집합이며 잘 이해해서 올바르게 적용하면 충실하고 알기 쉬운 클래스 구성을 생성할 수 있음

Design Pattern

- Design Pattern의 장점
  - ✓ 재사용성이 높고 유연성 있는 설계가 가능
    - 프로그램을 처음부터 설계하면 만들어진 성과물의 품질이 설계자의 직관이나 경험 등에 따라 달라지지만 Design Pattern을 도입하면 초보자도 고수들의 갈고 닦은 '지혜'를 이용한 설계가 가능
  - ✓ 의사 소통이 원할
    - Design Pattern을 습득하지 않은 기술자에게 설계를 설명할 경우 '이런 클래스를 만들고, 이 클래스는 이런 역할을 갖고 있고.....'라고 끝없이 설명해야 함
    - Design Pattern을 습득하고 있는 기술자라면 '○○Pattern으로 만들어!'라는 한마디로 끝남
    - Design Pattern을 습득하고 있는 기술자끼리라면 설계에 대해 상담할 때도 Design Pattern의 이름으로 설계 개요에 대해 동의를 얻는것이 가능해 의사 소통이 원활
  - ✓ Design Pattern을 이해하고 개념을 익혀두면 자신이 직접 설계하는 경우에도 적용할 수 있어 설계의 수준을 높일 수 있음

Creational Patterns

- 생성과 관련된 패턴은 인스턴스의 생성 로직을 숨기고 new 명령어를 통하지 않고 인스턴스를 생성하는 방법
- 특정 상황에서 어떤 방법으로 인스턴스를 생성 해야할 지를 결정하는데 있어서 유연성을 제공

Creational Patterns

- Singleton Pattern
  - ✓ 하나의 인스턴스만 생성할 수 있는 클래스의 Design Pattern
  - ✓ 시스템 전체에 공통적으로 적용되는 설정 정보를 보관하는 관리자 클래스나 전역 변수를 소유한 클래스를 만들 때 이용
  - ✓ 적용 방법
    - 생성자를 private으로 생성
    - 자신의 타입으로 된 static 변수를 선언
    - static 변수가 null이면 생성해서 리턴하고 그렇지 않으면 그냥 리턴하는 static 메서드를 생성
  - ✓ jdk의 클래스 중에서 인스턴스 생성 시 생성자를 호출하지 않고 static 메서드를 이용해서 인스턴스를 리턴받는 클래스 중에는 singleton Pattern을 적용한 클래스가 있음
  - ✓ 이 클래스가 다중 스레드 환경에서 사용되는 경우 멀티 스레드 제어에 대한 부분에 주의

실습(SingletonMain.java)

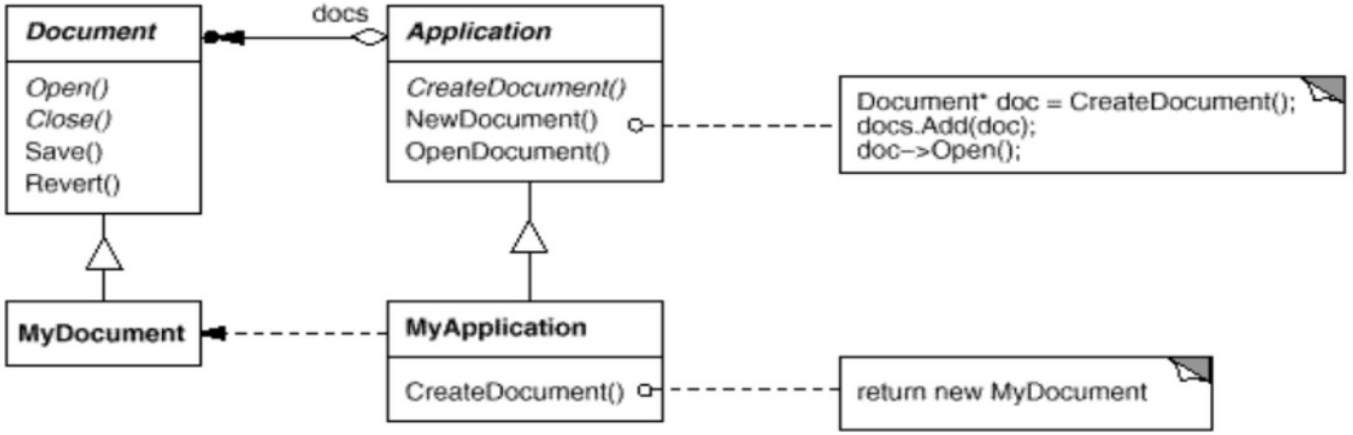
```
class OnlyOne {
        static OnlyOne obj = null;
        private OnlyOne(){
        }
        public static OnlyOne getInstance(){
                if(obj==null)
                        obj = new OnlyOne();
                return obj;
        }
}

public class SingletonMain{
        public static void main(String[] args) {
                OnlyOne obj1 = OnlyOne.getInstance();
                OnlyOne obj2 = OnlyOne.getInstance();

                System.out.println("obj1의 해시코드:" + obj1.hashCode());
                System.out.println("obj2의 해시코드:" + obj2.hashCode());
                System.out.println("obj1의 해시코드:" + System.identityHashCode(obj1));
                System.out.println("obj1의 해시코드:" + System.identityHashCode(obj2));
        }
}
```

Creational Patterns

- Factory Pattern
  - ✓ Factory Pattern은 하나의 인스턴스를 생성하는 인터페이스를 정의하는데 이때 어떤 인스턴스를 생성할 것인지에 대한 결정권을 Sub Class 에게 위임하는 Pattern
  - ✓ Factory pattern은 클래스로부터 인스턴스를 구현하는 코드를 떼어내서 코드를 더욱 탄탄하게 하고 결합도를 낮춤
  - ✓ 일관성을 유지해야 하는 관련된 일련의 인스턴스들을 생성하고자 할 때 사용하는 Pattern
  - ✓ 프레임워크의 인스턴스 생성이 주로 이 Pattern을 이용 – JDK에서는 URLConnection 클래스가 대표적

Creational Patterns

- Factory Pattern
  - ✓ 프레임워크는 일반적으로 추상클래스로 인스턴스 사이의 관계를 정의하는데 종종 인스턴스를 생성해야 할 때도 있는데 아래 다이어그램에서 관계를 맺고 있는 추상클래스들은 Document와 Application
  - ✓ 사용자가 그림 그리기 애플리케이션을 만들려면 두 추상클래스를 상속받아 DrawingApplication과 DrawingDocument 클래스를 정의해야 하고 Application클래스는 필요에 따라 Document 클래스를 생성 혹은 열기가 가능해야 함
  - ✓ 특정 문서는 특정 애플리케이션에 의존적이라는 점이 문제가 됨
  - ✓ docx는 워드에서 생성하는 문서이지 한글에서 생성하는 문서가 아닌 것과 같은 현상인데 Application의 CreateDocument()를 추상 메소드로 정의하고 이를 서브클래스에서 구현하도록 하여 각각의 Application마다 각각의 Document를 생성하도록 하는 형태로 이 경우 CreateDocument()를 팩토리 메소드라고 함



실습(FactoryPattern.java)

```
interface DB{
		public void disp();
}

class OracleDB implements DB {
		@Override
		public void disp() {
				System.out.println("Oracle Database");
		}
}

class MySQLDB implements DB{
		@Override
		public void disp() {
				System.out.println("MySQL Database");
		}
}
```

실습(FactoryPattern.java)

```
class RDBMSFactory{
        public static DB dbFactory(String env) {
                switch(env) {
                        case "Oracle":
                                return new OracleDB();
                        case "MySQL":
                                return new MySQLDB();
                        default:
                                return      null;
                }
        }
}

public class public class FactoryPattern {
        public static void main(String[] args) {
                DB db = RDBMSFactory.dbFactory("Oracle");
                db.disp();
        }
}
```

Creational Patterns

❖Abstract Factory Pattern

- ✓ 추상 팩토리 패턴은 구상클래스를 명시하지 않아도 연관있거나 의존적인 인스턴스의 그룹을 생성하기 위한 인터페이스를 제공하는 패턴
- ✓ Factory는 싱글턴 패턴을 사용하는 것이 일반적(동일한 제품을 생산하는 두개의 다른 공장을 하나의 애플리케이션에서 가지고 있을 필요는 없기 때문)
- ✓ 설정 파일에 기재된 시스템의 환경 정보에 따라 사용할 DBMS를 바꾸어 이용하기 와 같이 조건 에 따라 시스템의 동작을 변화시키는 프로그램을 생각해 보면 프로그램에서 이러한 소프트웨어에 접속하여(커넥션 객체를 생성하여) 데이터에 액세스하게 되는데,액세스 방법은 소프트웨어마다 조금씩 다른 부분이 있는데 DBMS마다 서로 다른 부분을 구현하는 방법으로 DBMS마다 별도의 클래스를 만들어 처리를 전환하는 프로그램을 만들 수 있는데 그러나 커넥션을 관리하는 클래스와 설정을 유지하는 클래스도 DBMS마다 전환해야 하는데 이런 경우 DBMS마다 연관된 인스턴스를 한꺼번에 생성하는 방법이 있으면 훨씬 편리
- ✓ 이를 실현하기 위한 수단이 바로 AbstractFactory 패턴이며 AbstractFactory 패턴은 인스턴스의 생성을 전문으로 실시하는 클래스(Factory)를 준비하여 일관성을 유지해야 하는 관련된 일련의 인스턴스들을 확실히 생성하기 위한 패턴

실습(AbstracFactoryPattern.java)

```
abstract class Connection {
        // 임의의 처리
}

abstract class Configuration {
        // 임의의 처리
}

interface Factory {
        Connection getConnection();
        Configuration getConfiguration();
}
```

실습(AbstracFactoryPattern.java)

```
class OracleConnection extends Connection {
		OracleConnection(){
			System.out.println("오라클 연결");
		}
}

class OracleConfiguration extends Configuration {
		OracleConfiguration(){
			System.out.println("오라클 환경 설정");
		}
}

class OracleFactory implements Factory {
		@Override
		public Connection getConnection() {
			return new OracleConnection();
		}
		@Override
		public Configuration getConfiguration() {
			return new OracleConfiguration();
		}
}
```

실습(AbstracFactoryPattern.java)

```
class MySQLConnection extends Connection {
		MySQLConnection(){
			System.out.println("MySQL 연결");
		}
}

class MySQLConfiguration extends Configuration {
		MySQLConfiguration(){
			System.out.println("MySQL 환경 설정");
		}
}

class MySQLFactory implements Factory {
		@Override
		public Connection getConnection() {
			return new MySQLConnection();
		}
		@Override
		public Configuration getConfiguration() {
			return new MySQLConfiguration();
		}
}
```

실습(AbstracFactoryPattern.java)

```
class DBFactory{
	public static Factory createFactory(String env) {
		switch (env) {
		case "Oracle":
			return new OracleFactory();
		case "MySQL":
			return new MySQLFactory();
		default:
			throw new IllegalArgumentException(env);
		}
	}
}

public class AbstracFactoryPattern {

	public static void main(String[] args) {
		String env = "Oracle";
		Factory factory = DBFactory.createFactory(env);
		Connection connection = factory.getConnection();
		Configuration configuration = factory.getConfiguration();

	}
}
```

Creational Patterns

❖Prototype Pattern

- ✓ 하나의 프로토타입 인스턴스를 만들고 필요할 때마다 프로토타입 인스턴스를 복사해서 인스턴스를 생성하는 패턴
- ✓ 패턴이 나오게 된 동기(Motivation): 인스턴스 생성 비용이 비싼 경우 프로토타입 원형을 만들고 클로닝 기법을 이용해서 인스턴스를 쉽게 생성하고자 하기 위해서
  - 인스턴스의 종류가 너무 많아서 각각을 별도의 클래스로 만들어야 하는 경우
  - new 키워드를 이용하는 것이 안좋은 상황일 때 – 생성자의 내용이 너무 복잡한 경우
  - 제품의 클래스 계층과 동일한 팩토리 클래스 계층을 만들고 싶지 않은 경우 – 프레임워크와의 분리
- ✓ 구성
  - Product: 추상 메소드 use 와 createClone이 선언된 인터페이스
  - Manager: createClone을 사용해서 인스턴스를 복제하는 클래스
  - MessageBox: 문자열을 테두리로 표시하는 클래스
  - UnderlinePen: 문자열에 밑줄을 표시하는 클래스
  - PrototypeMain: 실행 클래스

실습(framework.Product.java)

```
import java.lang.Cloneable;

public interface Product extends Cloneable {
    public abstract void use(String s);
    public abstract Product createClone();
}
```

실습(framework.Manager.java)

```
import java.util.*;

public class Manager {
    private HashMap<String, Product> showcase = new HashMap<>();
    public void register(String name, Product product) {
        showcase.put(name, product);
    }
    public Product create(String name) {
        Product p = (Product)showcase.get(name);
        return p.createClone();
    }
}
```

실습(MessageBox.java)

```
public class MessageBox implements Product {
    private char decochar;
    public MessageBox(char decochar) {
        this.decochar = decochar;
    }

    public void use(String s) {
        int length = s.getBytes().length;
        for (int i = 0; i < length + 4; i++) {
            System.out.print(decochar);
        }
        System.out.println("");
        System.out.println(decochar + " "  + s + " " + decochar);
        for (int i = 0; i < length + 4; i++) {
            System.out.print(decochar);
        }
        System.out.println("");
    }
```

실습(MessageBox.java)

```
    public Product createClone() {
      Product p = null;
      try {
          p = (Product)clone();
      } catch (CloneNotSupportedException e) {
          e.printStackTrace();
      }
      return p;
   }
}
```

실습(Underline.java)

```
public class UnderlinePen implements Product {
    private char ulchar;
    public UnderlinePen(char ulchar) {
        this.ulchar = ulchar;
    }
    public void use(String s) {
        int length = s.getBytes().length;
        System.out.println("₩""  + s + "₩"");
        System.out.print(" ");
        for (int i = 0; i < length; i++) {
            System.out.print(ulchar);
        }
        System.out.println("");
    }
    public Product createClone() {
        Product p = null;
        try {
            p = (Product)clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return p;
    }
}
```

실습(PrototypeMain.java)

```
public class PrototypeMain {
    public static void main(String[] args) {
        Manager manager = new Manager();
        UnderlinePen upen = new UnderlinePen('~');
        MessageBox mbox = new MessageBox('*');
        MessageBox sbox = new MessageBox('/');
        manager.register("strong message", upen);
        manager.register("warning box", mbox);
        manager.register("slash box", sbox);

        Product p1 = manager.create("strong message");
        p1.use("Hello, world.");
        Product p2 = manager.create("warning box");
        p2.use("Hello, world.");
        Product p3 = manager.create("slash box");
        p3.use("Hello, world.");
    }
}
```

Creational Patterns

- Builder Pattern
  - ✓ 생성의 흐름을 공통화하기 위한 수단이 Builder 패턴
  - ✓ Builder 패턴이란 복잡한 생산 과정이 필요한 인스턴스에 대해 그 생성 과정을 은폐함으로써 동일 과정으로 서로 다른 내부 형식의 인스턴스를 얻기 위한 패턴
  - ✓ 빌더 패턴은 팩토리 메소드 또는 추상 팩토리 패턴의 문제점을 해결하기 위해 나온 패턴
  - ✓ 팩토리 메소드 또는 추상 팩토리 패턴에는 인스턴스가 많은 속성을 가지고 있을 경우 두 가지 중요한 문제가 발생 할 수 있음
    - 클라이언트가 팩토리 클래스로 넘겨주는 인자가 많아지게 되면(팩토리가 많은 속성을 가지고 있으면 이에 대해 설정을 하기 위해 많은 인자를 받아야 함) 잘못된 인자를 넘겨주는 일도 빈번하게 일어날 뿐 아니라 관리하기도 힘듬
    - 몇몇 인자들은 optional일 수 있지만 팩토리 패턴에서는 모든 인자를 넘겨주어야 함 (optional 인자는 null로 넘김)
  - ✓ 빌더 패턴은 단계별로 인스턴스를 생성하고 마지막 인스턴스를 반환하는 메소드를 제공함으로써 위와 같은 문제들을 해결

실습(Builder.java)

```
//문서를 만들 메소드를 소유한 클래스
public abstract class Builder {
    public abstract void makeTitle(String title);
    public abstract void makeString(String str);
    public abstract void makeItems(String[] items);
    public abstract void close();
}
```

실습(Director.java)

```
//문서를 만드는 클래스
public class Director {
    private Builder builder;
    public Director(Builder builder) {        // Builder의 하위 클래스를 받아서 생성
        this.builder = builder;
    }
    public void construct() {                                    //문서 설정
        builder.makeTitle("Greeting");              // title 설정
        builder.makeString("아침과 낮에 ");          // 문자열 설정
        builder.makeItems(new String[]{             // 개별 항목 설정
            "좋은 아침입니다.",
            "안녕하세요",
        });
        builder.makeString("밤에 ");                // 별도의 문자열
        builder.makeItems(new String[]{             // 별도의 개별 항목
            "안녕하세요",
            "안녕히 주무세요",
            "안녕히 계세요",
        });
        builder.close();                        // 문서 완성
    }
}
```

실습(TextBuilder.java)

```
public class TextBuilder extends Builder {
    private StringBuilder buffer = new StringBuilder();          //문서 구축
    public void makeTitle(String title) {                        // 타이틀 설정
        buffer.append("==============================₩n");     // 장식선
        buffer.append("[" + title + "]₩n");                      // 타이틀
        buffer.append("₩n");                                     // 빈 줄
    }
    public void makeString(String str) {                        // 문자열 설정
        buffer.append('■' + str + "₩n");                        // 글머리 기호 추
        buffer.append("₩n");                                     //빈 줄
    }
    public void makeItems(String[] items) {                      // 일반 텍스트에서의 개별항목
        for (int i = 0; i < items.length; i++) {
            buffer.append("▫" + items[i] + "₩n");               // 글머리 기호 추가
        }
        buffer.append("₩n");                                     //빈 줄
    }
    public void close() {                                        // 문서 닫기
        buffer.append("==============================₩n");     // 장식선
    }
    public String getResult() {                                  // 문서 완성
        return buffer.toString();
    }
}
```

실습(HTMLBuilder.java)

```
import java.io.*;

public class HTMLBuilder extends Builder {
    private String filename;                                    // 파일명
    private PrintWriter writer;                                 //문자열을 기록할 PrintWriter
    public void makeTitle(String title) {                       // HTML 타이틀
            filename = title + ".html";                          // 타이틀로 파일명 생
        try {
            writer = new PrintWriter(new FileWriter(filename)); // PrintWriter 생성
        } catch (IOException e) {
            e.printStackTrace();
        }
        writer.println("<html><head><title>" + title + "</title></head><body>");
                                                                                    // 데이터
기록
        writer.println("<h1>" + title + "</h1>");
    }
```

실습(HTMLBuilder.java)

```
    public void makeString(String str) {                    // HTML 문자열 생성
        writer.println("<p>" + str + "</p>");
    }
    public void makeItems(String[] items) {                  // HTML 항목 생성
        writer.println("<ul>");
        for (int i = 0; i < items.length; i++) {
            writer.println("<li>" + items[i] + "</li>");
        }
        writer.println("</ul>");
    }
    public void close() {                                    // 문서 완성
        writer.println("</body></html>");
        writer.close();
    }
    public String getResult() {
        return filename;                                     // 파일명 리턴
    }

}
```

실습(BuilderMain.java)

```
public class BuilderMain {
        public static void usage() {
                System.out.println("Usage: java Main plain 일반 텍스트로 생성");
                System.out.println("Usage: java Main html  HTML 로 생성");
        }
```

실습(BuilderMain.java)

```
public static void main(String[] args) {
        if (args.length != 1) {
                usage();
                System.exit(0);
        }
        if (args[0].equals("plain")) {
                TextBuilder textbuilder = new TextBuilder();
                Director director = new Director(textbuilder);
                director.construct();
                String result = textbuilder.getResult();
                System.out.println(result);
        } else if (args[0].equals("html")) {
                HTMLBuilder htmlbuilder = new HTMLBuilder();
                Director director = new Director(htmlbuilder);
                director.construct();
                String filename = htmlbuilder.getResult();
                System.out.println(filename + "이 작성되었습니다..");
        } else {
                usage();
                System.exit(0);
        }
}
}
```

Creational Patterns

- Decorator Pattern
  - ✓ 주입을 이용해서 클래스에 기능을 추가하는 구조로 생성자에서 필요한 기능을 매개변수로 넘겨받아서 구현
  - ✓ JDK의 I/O 패키지의 클래스들이 많이 사용
  - ✓ 윈도우는 기본적인 모양을 갖고 여기에 가로 스크롤 바를 가질 수 있고 그렇지 않을 수도 있으며 세로 스크롤 바를 가질 수 있고 그렇지 않을 수도 있음
  - ✓ 상황에 따라서 생성될 수 있는 윈도우의 모양이 선택적 옵션에 따라 여러 가지인 경우 Decorator Pattern을 이용해서 구현하는 경우가 있음

실습(Decorator)

```
interface Coffee {
	public void make();
	public String getDescription();
}

//단순하게 만들어지는 클래스
class SimpleCoffee implements Coffee {
	public void make() {}
	public String getDescription() {
		return "단순한 커피";
	}
}

//다른 데코레이터를 받아서 만드는 클래스의 추상 클래스
abstract class CoffeeDecorator implements Coffee {
	protected Coffee decoratedCoffee;

	public CoffeeDecorator(Coffee decoratedCoffee) {
		this.decoratedCoffee = decoratedCoffee;
	}
}
```

실습(Decorator)

```
//데코레이터 클래스
class SugarCoffeeDecorator extends CoffeeDecorator {
	public SugarCoffeeDecorator(Coffee coffeeDecorator) {
		super(coffeeDecorator);
	}

	@Override
	public void make() {
		this.decoratedCoffee.make();
	}

	public String getDescription() {
		return decoratedCoffee.getDescription() + ", 설탕 추가";
	}
}
```

실습(Decorator)

```
//데코레이터 클래스
class CreamCoffeeDecorator extends CoffeeDecorator {
	public CreamCoffeeDecorator(Coffee coffeeDecorator) {
		super(coffeeDecorator);
	}

	@Override
	public void make() {
		this.decoratedCoffee.make();
	}

	public String getDescription() {
		return decoratedCoffee.getDescription() + ", 크림 추가";
	}
}
```

실습(Decorator)

```
public class DecoratorMain {
	public static void main(String[] args) {
		//단순 커피에 설탕을 추가하고 크림 커피로 생성
		Coffee myCoffee = new CreamCoffeeDecorator(
					new SugarCoffeeDecorator(new SimpleCoffee()));
		System.out.println(myCoffee.getDescription());
		//단순 커피
		Coffee yourCoffee = new SimpleCoffee();
		System.out.println(yourCoffee.getDescription());
	}
}
```

구조와 관련된 Pattern

- Adapter pattern
  - ✓ 현재 구현된 인터페이스의 메소드를 통해서 상속받은 클래스의 메소드를 호출하는 Pattern
  - ✓ 목적은 클라이언트가 원하는 인터페이스를 제공하기 위해서 클래스의 형태를 변형하는 것 (서로 연결을 해서 출력을 변형한다는 의미이지 실제로 변형한다는 의미가 아님)
  - ✓ 어댑터는 서로 맞물리지 않는 인터페이스들을 서로 연결시켜주는 역할
  - ✓ 기존 시스템을 재사용하여 새로운 시스템에 통합할 경우 기존 시스템의 처리는 그대로 사용할 수 있을 것이지만 새로운 시스템은 지금까지 사용하던 메서드와는 다른 인터페이스를 가지고 있다고 치면 이 경우 기존 시스템에 손을 대서 수정을 하게 되면 엄청난 변화를 강요하게 될 수 있음
  - ✓ 이 문제를 해결하기 위한 수단이 바로 Adapter 패턴으로 Adapter 패턴에서는 인터페이스에 호환성이 없는 클래스들을 조합시키는 것을 목적으로 하여 기존 시스템과 새로운 시스템의 인터페이스의 차이를 흡수하는 Adapter를 제공함으로써 적은 변경으로 기존 시스템을 새로운 시스템에 적용할 수 있도록 함
  - ✓ Adapter 패턴은 구현 방법에 따라 두 가지 방법
    - ❑ 상속을 이용하는 방법
    - ❑ 위임을 이용하는 방법 – 인스턴스 이용

클래스를 이용한 Adapter Pattern

```
class PrevSystem {
    public void prevProcess() {
      System.out.println("예전 처리 내용");
    }
}

interface PrevTarget {
    void process();
}

class Adapter extends PrevSystem implements PrevTarget {
    @Override
    public void process() {
      prevProcess();
    }
}
```

클래스를 이용한 Adapter Pattern

```
public class ClassAdapterMain {
    public static void main(String[] args) {
        PrevTarget target = new Adapter();
        target.process();
    }
}
```

인스턴스를 이용한 Adapter Pattern

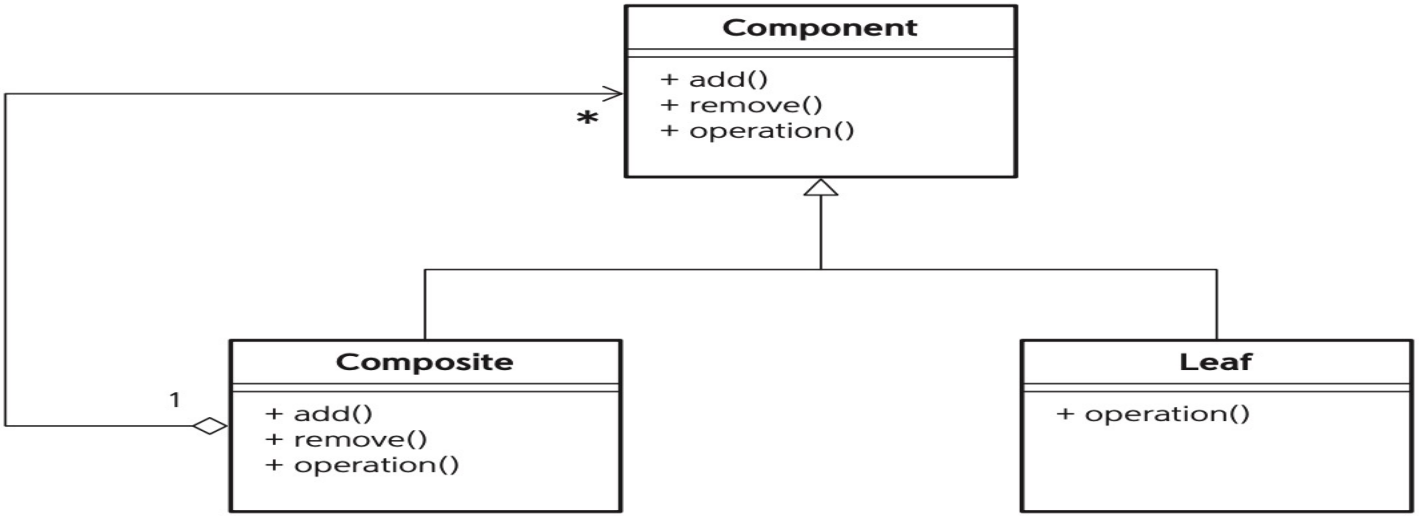
```
abstract class BeforeTarget {
    abstract void process();
}

class InstanceAdapter extends BeforeTarget {
    private PrevSystem prevSystem;

    public InstanceAdapter() {
      super();
      prevSystem = new PrevSystem();
    }


    @Override
    public void process() {
      prevSystem.prevProcess();
    }
}
```

인스턴스를 이용한 Adapter Pattern

```
public class InstanceAdapterMain {
    public static void main(String[] args) {
        BeforeTarget target = new InstanceAdapter();
        target.process();
    }
}
```

구조와 관련된 Pattern

- Composite pattern
  - ✓ 파일 시스템을 프로그램상에서 표현할 경우 일반적인 파일 시스템에서는 디렉토리 와 파일이 존재하며 디렉토리가 계층 구조를 가지고 파일은 디렉토리 안에 존재하며 여기에서 특정 디렉토리의 파일 및 디렉토리를 삭제하고자 할 때 대상이 파일인지 디렉토리인지를 일일이 의식 하지 않고 동일하게 처리할 수 있다면 보다 편리
  - ✓ 이를 실현하기 위한 수단이 Composite 패턴
  - ✓ Composite 패턴은 재귀적 구조를 쉽게 처리할 수 있도록 해주는 디자인 패턴



실습(Composite)

```
//파일과 디렉토리의 작업을 위한 인터페이스
interface Entry {
    void add(Entry entry);

    void remove();

    void rename(String name);
}
```

실습(Composite)

```
class File implements Entry {
	private String name;

	public File(String name) {
		this.name = name;
	}
	@Override
	public void add(Entry entry) {
		throw new UnsupportedOperationException();
	}
	@Override
	public void remove() {
		System.out.println(this.name + "를 삭제했다.");
	}
	@Override
	public void rename(String name) {
		this.name = name;
	}
}
```

실습(Composite)

```
class Directory implements Entry {
    private String name;

    private List<Entry> list;

    public Directory(String name) {
        this.name = name;
        this.list = new ArrayList<>();
    }

    @Override
    public void add(Entry entry) {
        list.add(entry);
    }
```

실습(Composite)

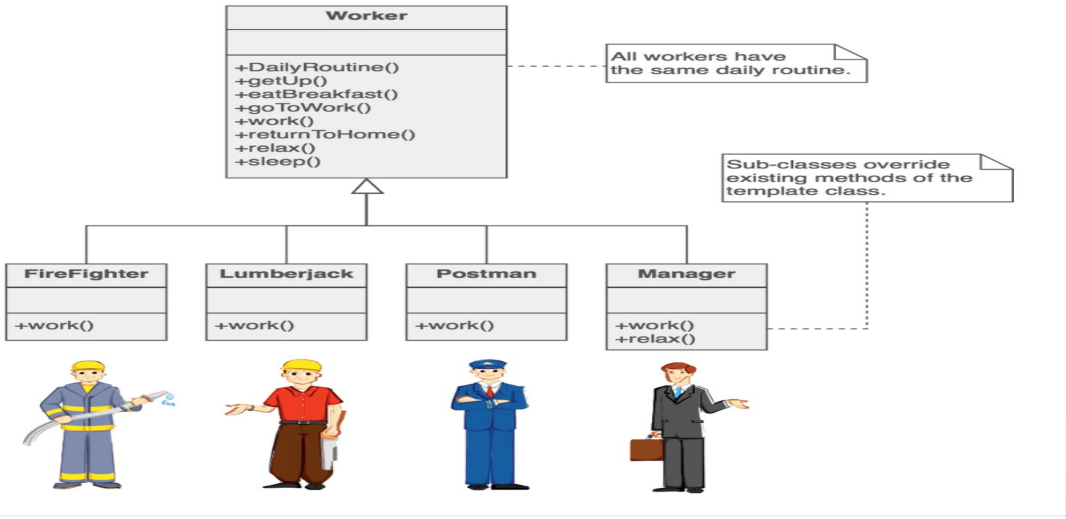
```
    @Override
    public void remove() {
        Iterator<Entry> itr = list.iterator();
        while (itr.hasNext()) {
            Entry entry = itr.next();
            entry.remove();
        }
        System.out.println(this.name + "을 삭제했다.");
    }

    @Override
    public void rename(String name) {
        this.name = name;
    }
}
```

실습(Composite)

```
public class CompositeMain {
    public static void main(String... args) {
        File file1 = new File("file1");
        File file2 = new File("file2");
        File file3 = new File("file3");
        File file4 = new File("file4");

        Directory dir1 = new Directory("dir1");
        dir1.add(file1);

        Directory dir2 = new Directory("dir2");
        dir2.add(file2);
        dir2.add(file3);

        dir1.add(dir2);
        dir1.add(file4);
        dir1.remove();
    }
}
```

행동과 관련된 Pattern

- 템플릿 메소드 Pattern
  - ✓ 추상 클래스나 인터페이스를 이용해서 알고리즘의 골격을 미리 정의해두고 하위 클래스에서 필요한 부분만 재정의 해서 사용하는 Pattern



  - ✓ 위 그림은 소방관, 목수, 우체부, 관리자 등 다양한 직업의 사람들을 인스턴스로 생성했는데 이 사람들은 공통적으로 일하는 사람들이므로 Worker라는 클래스의 하위로 만들었고 모든 사람들은 자신만의 일과 작업 순서가 있으며 이 일과 순서는 DailyRoutine()내에 정의되어있고 Worker 클래스를 이용하는 클라이언트에서는 DailyRoutine()만 호출
  - ✓ Worker가 소방관이냐, 우체부냐에 따라 work()에서 하는 일이 달라지는데 모든 사람들이 work()를 오버라이드해서 재정의
  - ✓ DailyRoutine() 메소드가 템플릿 메소드가 되는 것이고 이런 식의 Pattern을 템플릿 메소드 Pattern

실습(TemplateMain)

```
abstract class SmartPhone {
	public abstract void tel();
	public final void call(){
		tel();
	}
}

class Iphone extends SmartPhone {
	@Override
	public void tel() {
		System.out.println("아이폰에서 전화를 겁니다.");
	}
}

class Android extends SmartPhone {
	@Override
	public void tel() {
		System.out.println("안드로이드에서 전화를 겁니다.");
	}
}
```

실습(TemplateMain)

```
public class TemplateMain {
	public static void main(String[] args) {
		SmartPhone phone = new Iphone();
		phone.call();

		phone = new Android();
		phone.call();
	}
}
```

행동과 관련된 Pattern

- Iterator Pattern
  - ✓ 인스턴스 안에 여러 개의 데이터가 존재하는 경우 순서대로 전체 데이터를 검색해서 처리를 실행하기 위한 Pattern
  - ✓ 반복한다라는 의미를 갖기 때문에 한자로는 반복자라고 함
  - ✓ for-each문을 실현하기 위해 Iterator 패턴이 이용됨
  - ✓ Java API 에서의 Collection의 구현 구조
    - Iterator 인터페이스
    - Iterable 인터페이스
    - Collection 인터페이스
    - Collection 인터페이스를 구현한 클래스

행동과 관련된 Pattern

- Iterator Pattern

```
import java.util.ArrayList;
import java.util.List;

interface Iterator<E> {
        boolean hasNext();
        E next();
        void remove();
}

interface Iterable<T> {
        Iterator<T> iterator();
}

interface Collection<E> extends Iterable<E> {

}
```

행동과 관련된 Pattern

- Iterator Pattern

```
public class CollectionMain {

        public static void main(String[] args) {
                List<String> list = new ArrayList<>();
                list.add("생성 패턴");
                list.add("구조 패턴");
                list.add("행동 패턴");
                java.util.Iterator<String> itr = (java.util.Iterator<String>) list.iterator();
                while (itr.hasNext()) {
                        String str = itr.next();
                        System.out.println(str);
                }

        }

}
```

행동과 관련된 Pattern

- Iterator Pattern
  - ✓ 예제 프로그램 구성
    - Aggregate 인터페이스: 요소들이 나열된 집합체
    - Iterator 인터페이스: 하나씩 나열하면서 검색을 실행하는 인터페이스
    - Song: 노래를 나타내는 DTO 클래스
    - SongShelf: 노래 모음을 나타내는 클래스
    - SongShelfIterator: 노래 모음을 순회하는 클래스
    - ItreratorMain: 실행 클래스

Iterator

- 반복자 인터페이스 – Iterator

```
public interface Iterator {
    public abstract boolean hasNext();
    public abstract Object next();
}
```

Iterator

- 집합체 인터페이스 – Aggregator

```
public interface Aggregate {
    public abstract Iterator iterator();
}
```

Iterator

- DTO 클래스 – Song

```
public class Song {
        private String title;
        private String singer;

        public Song() {
                super();
        }
        public Song(String title, String singer) {
                super();
                this.title = title;
                this.singer = singer;
        }
```

Iterator

❖ DTO 클래스 – Song

```
        public String getTitle() {
                return title;
        }
        public void setTitle(String title) {
                this.title = title;
        }
        public String getSinger() {
                return singer;
        }
        public void setSinger(String singer) {
                this.singer = singer;
        }
        @Override
        public String toString() {
                return "Song [title=" + title + ", singer=" + singer + "]";
        }
}
```

Iterator

- Song의 집합 클래스 – SongShelf

```
import java.util.ArrayList;
import java.util.List;
public class SongShelf implements Aggregate {
    private List<Song> songs;
    private int last = 0;
    public SongShelf() {
        this.songs = new ArrayList<>();
    }
    public Song getSongAt(int index) {
        return songs.get(index);
    }
    public void appendSong(Song song) {
        this.songs.add(song);
        last++;
    }
    public int getLength() {
        return last;
    }
    public Iterator iterator() {
        return new SongShelfIterator(this);
    }
}
```

Iterator

- SongShelf의 Iterator 클래스 – SongShelfIterator

```
public class SongShelfIterator implements Iterator {
    private SongShelf songShelf;
    private int index;

    public SongShelfIterator(SongShelf songShelf) {
        this.songShelf = songShelf;
        this.index = 0;
    }

    public boolean hasNext() {
        if (index < songShelf.getLength()) {
            return true;
        } else {
            return false;
        }
    }

    public Object next() {
        Song book = songShelf.getSongAt(index);
        index++;
        return book;
    }
}
```

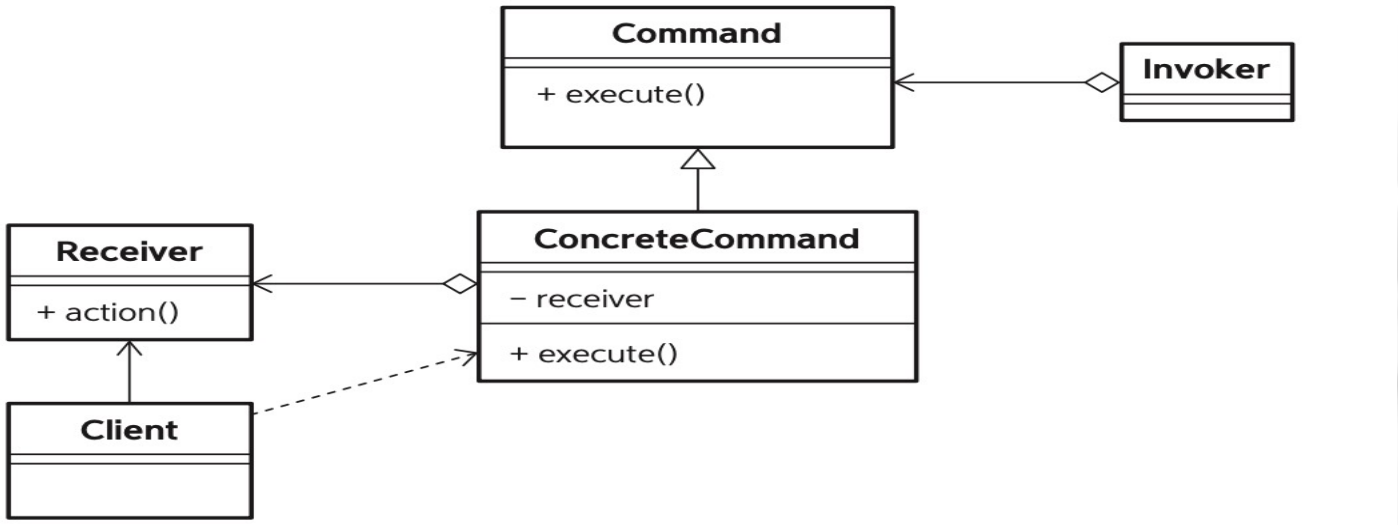
Iterator

❖ 실행 클래스 – IteratorMain

```
public class IteratorMain {
    public static void main(String[] args) {
        SongShelf songShelf = new SongShelf();
        songShelf.appendSong(new Song("달라달라", "ITZY"));
        songShelf.appendSong(new Song("빨간맛", "RedVelvet"));
        songShelf.appendSong(new Song("휘파람","BlackPink"));
        songShelf.appendSong(new Song("OOH-AHH하게", "Twice"));
        Iterator it = songShelf.iterator();
        while (it.hasNext()) {
            Song book = (Song)it.next();
            System.out.println(book.toString());
        }
    }
}
```

행동과 관련된 Pattern

- Command Pattern
  - ✓ 처리 내용이 비슷한 명령을 패턴에 따라 구분하거나 조합하거나 해서 실행하는 처리가 필한 경우 예를 들어 판매 사이트에서의 할인 계산을 생각해 보면 상품 금액에 할인율을 곱하는 것 만이라면 간단하지만 계절과 상품의 내용에 따라 할인의 패턴을 바꾸어 처리하거나 할인한 금액에 추가로 할인을 할 필요가 있을 수 있을 수 있는데 이를 실현하기 위한 수단이 Command 패턴
  - ✓ 명령을 하나의 인스턴스로 만들어서 처리하는 기법



실습(Command)

```
class Book {
	private double amount;

	public Book(double amount) {
		this.amount = amount;
	}

	public double getAmount() {
		return this.amount;
	}

	public void setAmount(double amount) {
		this.amount = amount;
	}
}
```

실습(Command)

```
abstract class Command {
	protected Book book;

	public void setBook(Book book) {
		this.book = book;
	}

	public abstract void execute();
}
```

실습(Command)

```
class DiscountCommand extends Command {
	@Override
	public void execute() {
		double amount = book.getAmount();
		book.setAmount(amount * 0.9);
	}
}
```

실습(Command)

```
class SpecialDiscountCommand extends Command {
	@Override
	public void execute() {
		double amount = book.getAmount();
		book.setAmount(amount * 0.7);
	}
}
```

실습(Command)

```
public class CommandMain {
    public static void main(String... args) {
        // 5000원의 만화책
        Book comic = new Book(5000);

        // 25000원의 기술서적
        Book technicalBook = new Book(25000);

        // 할인 가격 계산용 명령
        Command discountCommand = new DiscountCommand();

        // 특별 할인 가격 계산용 명령
        Command specialDiscountCommand = new SpecialDiscountCommand();

        // 만화책에 할인을 적용
        discountCommand.setBook(comic);
        discountCommand.execute();
        System.out.println("할인 후 금액은" + comic.getAmount() + "원");
```

실습(Command)

```
        // 기술서적에 할인을 적용
        discountCommand.setBook(technicalBook);
        discountCommand.execute();
        System.out.println("할인 후 금액은" + technicalBook.getAmount() + "원");

        // 기술서적에 추가 특별 할인을 적용
        specialDiscountCommand.setBook(technicalBook);
        specialDiscountCommand.execute();
        System.out.println("할인 후 금액은" + technicalBook.getAmount() + "원");
    }
}
```

행동과 관련된 Pattern

- Observer Pattern
  - ✓ 주기적으로 상태가 변화하는 인스턴스가 존재하는 경우 이 인스턴스의 상태가 바뀐 것을 감지하여 처리하도록 해주어야 하는 경우에 사용할 수 있는 Pattern
  - ✓ 인스턴스의 상태가 변화한 것을 관찰하고 그 인스턴스 자신이 상태의 변화를 통지하는 구조를 제공하는 Pattern

실습(Observer)

```
interface Observer {
    void update(Subject subject);
}
```

실습(Observer)

```
import java.util.ArrayList;
import java.util.List;

abstract class Subject {
	private List<Observer> observers = new ArrayList<>();

	public void addObserver(Observer observer) {
		this.observers.add(observer);
	}

	public void notifyObservers() {
		for (Observer observer : observers) {
			observer.update(this);
		}
	}

	public abstract void execute();
}
```

실습(Observer)

```
class Client implements Observer {
	@Override
	public void update(Subject subject) {
		System.out.println("통지를 수신했다.");
	}
}
```

실습(Observer)

```
class DataChanger extends Subject {
	private int status;

	@Override
	public void execute() {
		status++;
		System.out.println("상태가 " + status + "로 바뀌었다.");
		notifyObservers();
	}
}
```

실습(Observer)

```
public class ObserverMain {
	public static void main(String... args) {
		Observer observer = new Client();
		Subject dataChanger = new DataChanger();

		dataChanger.addObserver(observer);
		for (int count = 0; count < 10; count++) {
			dataChanger.execute();

			try {
				Thread.sleep(500);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

행동과 관련된 Pattern

- Strategy Pattern
  - ✓ 공통된 부분과 서로 다른 부분을 찾아내서 공통된 부분과 다른 부분을 분리시켜서 인스턴스를 생성할 때 주입(DI-Defendency Injection)을 이용해서 서로 다른 부분을 추가하는 구조
  - ✓ 처리 알고리즘을 쉽게 전환할 수 있도록 하는 패턴으로 조건에 따라 처리 알고리즘만을 전환해서 실행하려는 경우에 유용

실습(Strategy)

```
interface Market {
	public void appStore();
}
```

실습(Strategy)

```
class AndroidMarket implements Market {
	@Override
	public void appStore() {
		System.out.println("다양한 마켓 지원");
	}
}
```

실습(Strategy)

```
class IphoneMarket implements Market {
	@Override
	public void appStore() {
		System.out.println("단 하나의 마켓");
	}
}
```

실습(Strategy)

```
abstract class InternetPhone {
	private Market market;

	public void setMarket(Market market) {
		this.market = market;
	}

	public abstract void messageWeb();
	public void store(){
		market.appStore();
	}
}
```

실습(Strategy)

```
class Apple extends InternetPhone {

	@Override
	public void messageWeb() {
		System.out.println("아이폰은 메시지에 전송된 웹 주소로 바로 이동이 되지 않습니다.");
		System.out.println("보안 문제 때문입니다..");
	}
}
```

실습(Strategy)

```
class Google extends InternetPhone {

	@Override
	public void messageWeb() {
		System.out.println("안드로이드 폰은 메시지에 전송된 웹 주소로 바로 이동이 가능합니
	다.");
		System.out.println("보안에 취약합니다.");
	}
}
```

실습(Strategy)

```
public class StrategyMain {
	public static void main(String[] args) {
		Market market1 = new IphoneMarket();
		Market market2 = new AndroidMarket();

		InternetPhone android = new Google();
		InternetPhone iPhone = new Apple();

		android.setMarket(market1);
		iPhone.setMarket(market2);

		android.messageWeb();
		android.store();
		System.out.println("========================");
		iPhone.messageWeb();
		iPhone.store();
	}
}
```