Design Pattern

Creational Pattern

- Creational Pattern
  - ✓ 생성 패턴(creational Pattern)은 인스턴스를 만드는 절차를 추상화하는 패턴
  - ✓ 이 범주에 해당하는 패턴은 객체를 생성 및 합성하는 방법이나 객체의 표현 방법 과 소프트웨어 시스템을 분리해 줌
  - ✓ 클래스 생성 패턴이 인스턴스로 만들 클래스를 다양하게 만들기 위한 용도로 상속을 사용하는 반면 객체 생성 패턴은 인스턴스 화 작업을 다른 객체에게 떠넘길 수도 있음
  - ✓ 생성 패턴은 시스템이 상속보다는 복합 방법을 사용하는 쪽으로 진화되어 가면서 더 중요해지고 있기 때문에 고정된 행동 집합을 정의하는 것보다는 더 복잡한 행동을 만드는 데 필요한 구성 요소가 될 수 있는 기본적인 행동 집합을 정의하는 쪽에 더 많은 관심과 노력이 들어가고 있으므로 특정 행동을 수행하는 클래스를 만들려면 단순하게 하나의 클래스를 인스턴스화하는 일 이상의 품이 들어감
  - ✓ 생성 패턴은 시스템이 어떤 구체 클래스를 사용하는지에 대한 정보를 캡슐화
  - ✓ 생성 패턴은 이들 클래스의 인스턴스들이 어떻게 만들고 어떻게 서로 맞붙는지에 대한 부분을 완전히 가려줌
  - ✓ 생성 패턴을 이용하면 무엇이 생성되고 누가 이것을 생성하며 이것이 어떻게 생성되는지, 언제 생성할 것인지 결정하는 데 유연성을 확보할 수 있게 됨
  - ✓ 생성 패턴으로 분류되는 패턴은 여러 개인데 이런 여러 생성 패턴들은 서로 보완적 일 수도 있고 선택되기 위해 서로 경쟁적일 수도 있는데 동일한 문제 해결을 위해서 어떤 생성 패턴을 사용해야 할지 결정을 내리기 어려움

Creational Pattern

- Creational Pattern
  - ✓ 인스턴스를 생성하고 복합하는 방법에 해당하는 부분과 이들 인스턴스를 사용하는 프로그램을 분리하고자 할 때 어떤 패턴을 적용해야 하는지 판단하기는 어려운데 예를 들어 원형 패턴과 추상 팩토리 패턴 중 무엇을 선택할지 고민해야 할 때가 있음
  - ✓ 어떨 때는 또 이들이 서로 보완적일 수도 있는데 빌더 패턴은 어떤 구성 요소를 만들지 구현하는 데에 다른 생성 패턴 중 하나를 사용할 수 있음
  - ✓ 원형 패턴은 자기 자신의 구현을 위해 단일체 패턴을 사용하기도 함

Creational Pattern

- Abstract Factory Pattern
  - ✓ 추상 팩토리 패턴은 구현 클래스를 명시하지 않아도 연관있거나 의존적인 인스턴스의 그룹을 생성하기 위한 인터페이스를 제공하는 패턴으로 다른 이름으로는 Kit 라고도 함
  - ✓ 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 조건에 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 있지만 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

Creational Pattern

- Abstract Factory Pattern
  - ✓ AbstractFactoryPattern

```
package creational.abstractfactory;

abstract class Connection {
    // 임의의 처리
}

abstract class Configuration {
    // 임의의 처리
}

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

Creational Pattern

- Abstract Factory Pattern
  - ✓ AbstractFactoryPattern

```
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();
    }
}
```

Creational Pattern

- Abstract Factory Pattern
  - ✓ AbstractFactoryPattern

```
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();
    }
}
```

Creational Pattern

- Abstract Factory Pattern
  - ✓ AbstractFactoryPattern

```
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);
            }
    }
}
```

Creational Pattern

- Abstract Factory Pattern
  - ✓ AbstractFactoryPattern

```
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 Pattern

- Abstract Factory Pattern
  - ✓ 활용성
    - 객체가 생성되거나 구성 및 표현되는 방식과 무관하게 시스템을 독립적으로 만들고자 할 때
    - 여러 제품 군 중 하나를 선택해서 시스템을 설정해야 하고 한번 구성한 제품을 다른 것으로 대체할 수 있을 때
    - 관련된 제품 객체들이 함께 사용되도록 설계되었고 이 부분에 대한 제약이 외부에도 지켜지도록 하고 싶을 때
    - 제품에 대한 클래스 라이브러리를 제공하고 그들의 구현이 아닌 인터페이스를 노출시키고 싶을 때

Creational Pattern

- Abstract Factory Pattern
  - ✓ 이익 과 부담
    - 구체적인 클래스를 분리
    - 제품 군을 쉽게 대체할 수 있도록 구성
    - 제품 사이의 일관성을 증진시킴
    - 새로운 종류의 제품을 제공하기가 어려움

Creational Pattern

- 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

Creational Pattern

- Builder Pattern
  - ✓ Builder.java

```
//문서를 만들 메서드를 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();
}
```

Creational Pattern

- Builder Pattern
  - ✓ Director.java

```
//문서를 만드는 클래스
public class Director {
    private Builder builder;

    public Director(Builder builder) {        // Builder의 하위 클래스를 받아서 생성
            this.builder = builder;
    }
```

Creational Pattern

- Builder Pattern
  - ✓ Director.java

```
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
    }
}
```

Creational Pattern

- Builder Pattern
  - ✓ 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
    }
```

Creational Pattern

- Builder Pattern
  - ✓ TextBuilder.java

```
    public void close() {                                   // 문서 닫기
        buffer.append("==============================₩n");     // 장식선
    }

    public String getResult() {                             // 문서 완성
        return buffer.toString();
    }
}
```

Creational Pattern

- Builder Pattern
  - ✓ 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>");
    }
```

Creational Pattern

- Builder Pattern
  - ✓ 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
    }

}
```

Creational Pattern

- Builder Pattern
  - ✓ 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");
    }
```

Creational Pattern

- Builder Pattern
  - ✓ 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 Pattern

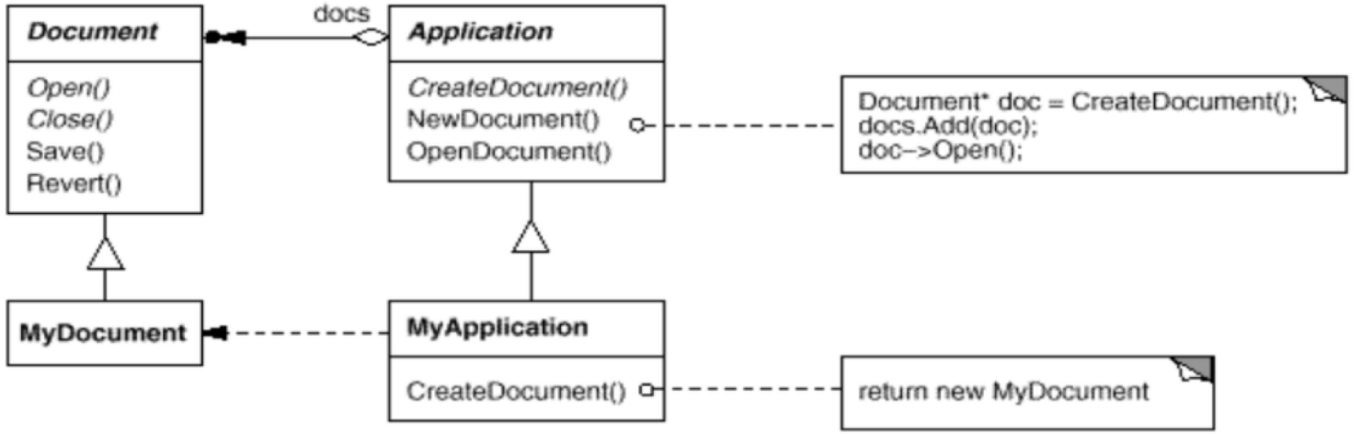
- Builder Pattern
  - ✓ 활용성
    - 복합 객체의 생성 알고리즘이 이를 합성하는 요소 객체들이 무엇인지 이들의 조립 방법에 독립적일 때
    - 합성할 객체들의 표현이 서로 다르더라도 생성 절차에서 이를 지원해야 할 때
  - ✓ 결과
    - 제품에 대한 내부 표현을 다양하게 변화할 수 있음
    - 생성과 표현에 필요한 코드를 분리
    - 복합 객체를 생성하는 절차를 조금 더 세밀하게 나눌 수 있음

Creational Pattern

- 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 Pattern

- 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



Creational Pattern

- Factory Pattern
  - ✓ 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");
    }
}
```

Creational Pattern

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

Creational Pattern

- Factory Pattern
  - ✓ 활용성
    - 어떤 클래스가 자신이 생성해야 하는 객체의 클래스를 예측할 수 없을 때
    - 생성할 객체를 기술하는 책임을 자신의 서브 클래스가 지정했으면 할 때
    - 객체 생성의 책임을 몇 개의 보조 서브 클래스 가운데 하나에게 위임하고 어떤 서브 클래스가 위임자인지에 대한 정보를 국소화시키고 싶을 때

Creational Pattern

- Prototype Pattern
  - ✓ 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

Creational Pattern

- Prototype Pattern
  - ✓ Product.java

```
import java.lang.Cloneable;

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

Creational Pattern

- Prototype Pattern
  - ✓ 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();
    }
}
```

Creational Pattern

- Prototype Pattern
  - ✓ 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("");
    }
```

Creational Pattern

- Prototype Pattern
  - ✓ MessageBox.java

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

Creational Pattern

- Prototype Pattern
  - ✓ UnderlinePen.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("");
    }
```

Creational Pattern

- Prototype Pattern
  - ✓ UnderlinePen.java

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

Creational Pattern

- Prototype Pattern
  - ✓ 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 Pattern

- Prototype Pattern
  - ✓ 활용성
    - 인스턴스화할 클래스를 런타임에 지정할 때(동적 로딩)
    - 제품 클래스 계통과 병렬적으로 만드는 팩토리 클래스를 피하고 싶을 때
    - 클래스의 인스턴스들이 서로 다른 상태 조합 중에 어느 하나일 때

Creational Pattern

- 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

Creational Pattern

- Singleton Pattern
  - ✓ 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));
    }
}
```

Structural Pattern

- Structural Pattern(구조 패턴)
  - ✓ 구조 패턴(structural Pattern)은 더 큰 구조를 형성하기 위해 어떻게 클래스와 객체를 합성하는가와 관련된 패턴
  - ✓ 구조 클래스 패턴은 상속 기법을 이용하여 인터페이스나 구현을 복합하는데 다중 상속을 통해서 둘 이상의 클래스를 하나로 만들 수 있는데 그 결과 만들어진 클래스는 상위 클래스에 정의된 모든 특성들을 혼합하여 갖게 되며 이 패턴은 서로 독립적으로 개발한 클래스 라이브러리를 마치 하나인 양 사용할 필요가 있을 때 매우 유용한 방법
  - ✓ 적응자 패턴을 사용한 클래스 형태도 있는데 일반적으로 적응자는 어떤 인터페이스가 다른 인터페이스를 따르게 만들어 서로 다른 인터페이스들의 통일된 추상을 제공하는데 이를 가능하게 하려고 적응자 클래스는 적응 대상 클래스(adaptee)에서 특성들을 상속받아 적응 대상자에 정의된 인터페이스를 마치 자신이 제공하는 것처럼 보여줌
  - ✓ 구조 객체 패턴은 인터페이스나 구현을 복합하는 것이 아니라 새로운 기능을 실현하기 위해 객체를 합성하는 방법을 제공
  - ✓ 객체 합성이 갖는 추가된 유연성은 런타임에 복합 방법이나 대상을 변경할 수 있다는 점에서 나오는데 이는 정적인 클래스 복합을 사용해서는 불가능한 일

Structural Pattern

- Adapter Pattern(적응자 패턴)
  - ✓ 클래스의 인터페이스를 사용자가 기대하는 인터페이스 형태로 적응(변환)시키는데 서로 일치하지 않는 인터페이스를 갖는 클래스들을 함께 동작시키는 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

Structural Pattern

- Adapter Pattern(적응자 패턴)
  - ✓ 클래스의 상속을 이용하는 방법 – ClassAdapterMain.java

```
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();
    }
}
```

Structural Pattern

- Adapter Pattern(적응자 패턴)
  - ✓ 클래스의 상속을 이용하는 방법 – ClassAdapterMain.java

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

Structural Pattern

- Adapter Pattern(적응자 패턴)
  - ✓ 인스턴스를 이용하는 방법 – InstanceAdapterMain.java

```
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();
    }
}
```

Structural Pattern

- Adapter Pattern(적응자 패턴)
  - ✓ 인스턴스를 이용하는 방법 - InstanceAdapterMain.java

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

Structural Pattern

- Adapter Pattern(적응자 패턴)
  - ✓ 활용성
    - 기존 클래스를 사용하고 싶은데 인터페이스가 맞지 않을 때
    - 아직 예측하지 못한 클래스나 실제 관련되지 않는 클래스들이 기존 클래스를 재사용하고자 하지만 이미 정의된 재사용 가능한 클래스가 지금 요청하는 인터페이스를 꼭 정의하고 있지 않을 때로 이미 만든 것을 재사용하고자 하나 이 재사용 가능한 라이브러리를 수정할 수 없을 때
    - 이미 존재하는 여러 개의 서브 클래스를 사용해야 하는데 이 서브 클래스들의 상속을 통해서 이들의 인터페이스를 다 개조한다는 것이 현실성이 없을 때로 객체 적응자를 써서 부모 클래스의 인터페이스를 변형하는 것이 더 바람직함

Structural Pattern

- Bridge Pattern(가교 패턴)
  - ✓ 구현에서 추상을 분리하여 이들이 독립적으로 다양성을 가질 수 있도록 하는 패턴

Structural Pattern

- Bridge Pattern(가교 패턴)
  - ✓ Color.java

```
public interface Color {
    String fill();
}
```

Structural Pattern

- Bridge Pattern(가교 패턴)
  - ✓ Blue.java

```
public class Blue implements Color {
    @Override
    public String fill() {
        return "Color is Blue";
    }
}
```

Structural Pattern

- Bridge Pattern(가교 패턴)
  - ✓ Shape.java

```
public abstract class Shape {
    protected Color color;

        public Shape(Color c){
                this.color=c;
        }

    abstract public String draw();
}
```

Structural Pattern

- Bridge Pattern(가교 패턴)
  - ✓ Square.java

```
public class Square extends Shape {

    public Square(Color color) {
        super(color);
    }

    @Override
    public String draw() {
        return "Square drawn. " + color.fill();
    }
}
```

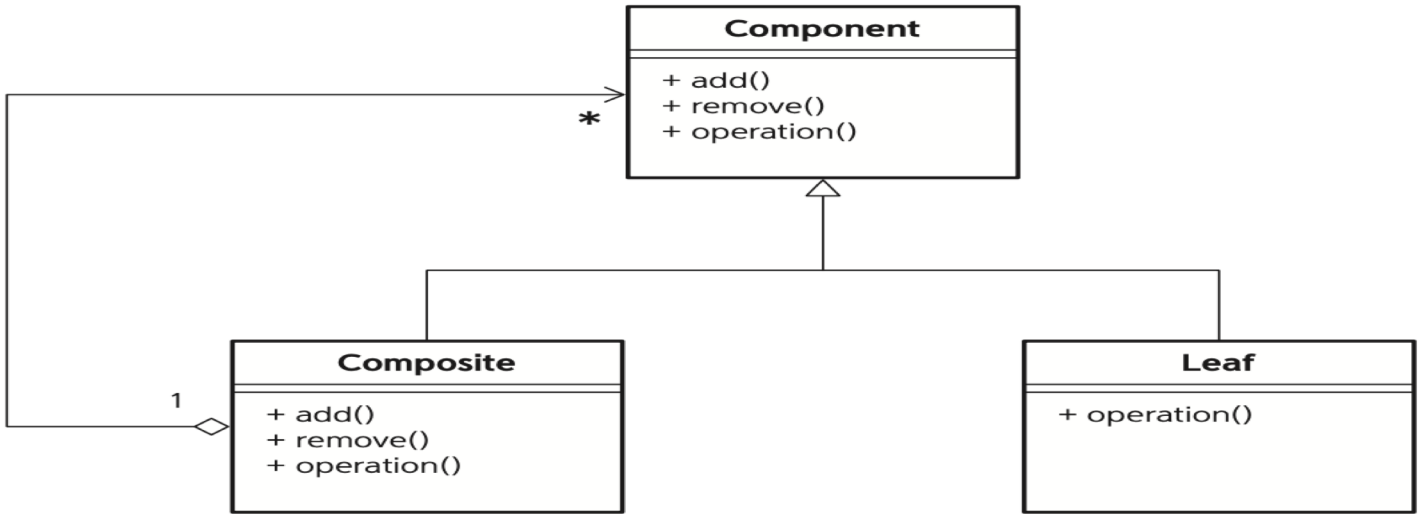
Structural Pattern

- Bridge Pattern(가교 패턴)
  - ✓ BridgeMain.java

```
public class BridgeMain {

        public static void main(String[] args) {
                Shape square = new Square(new Blue());

                System.out.println(square.draw());
        }

}
```

Structural Pattern

- Composite Pattern
  - ✓ 부분과 전체의 계층을 표현하기 위해 객체들을 모아 트리 구조로 구성
  - ✓ 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 편리한데 이를 실현하기 위한 수단이 Composite 패턴
  - ✓ Composite 패턴은 재귀적 구조를 쉽게 처리할 수 있도록 해주는 디자인 패턴



Structural Pattern

- Composite Pattern
  - ✓ CompositeMain.java

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

interface Entry {
    void add(Entry entry);

    void remove();

    void rename(String name);
}
```

Structural Pattern

- Composite Pattern
  - ✓ CompositeMain.java

```
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;
    }
}
```

Structural Pattern

- Composite Pattern
  - ✓ CompositeMain.java

```
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);
    }
```

Structural Pattern

- Composite Pattern
  - ✓ CompositeMain.java

```
        @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;
        }
}
```

Structural Pattern

- Composite Pattern
  - ✓ CompositeMain.java

```
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();
    }
}
```

Structural Pattern

- 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
  - ✓ 활용성
    - 동적으로 또한 투명하게(transparent) – 다른 객체에 영향을 주지 않고 개개의 객체에 새로운 책임을 추가하기 위해 사용
    - 제거될 수 있는 책임에 대해 사용
    - 실제 상속으로 서브 클래스를 계속 만드는 방법이 실질적이지 못할 때 사용
    - 너무 많은 수의 독립된 확장이 가능할 때 모든 조합을 지원하기 위해 이를 상속으로 해결하면 클래스 수가 폭발적으로 많아지게 되거나 클래스 정의가 숨겨지든가 그렇지 않더라도 서브 클래싱을 할 수 없게 됨

Structural Pattern

- Decorator Pattern
  - ✓ DecoratorMain.java

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

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

Structural Pattern

- Decorator Pattern
  - ✓ DecoratorMain.java

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

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

Structural Pattern

- Decorator Pattern
  - ✓ DecoratorMain.java

```
//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";
    }
}
```

Structural Pattern

- Decorator Pattern
  - ✓ DecoratorMain.java

```
//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";
    }
}
```

Structural Pattern

- Decorator Pattern
  - ✓ DecoratorMain.java

```
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());
    }
}
```

Structural Pattern

- FACADE Pattern
  - ✓ 한 서브 시스템 내의 인터페이스 집합에 대한 획일화된 하나의 인터페이스를 제공하는 패턴으로 서브 시스템을 사용하기 쉽도록 상위 수준의 인터페이스를 정의하는 패턴
  - ✓ 시스템을 서브 시스템으로 구조화하면 복잡성을 줄이는 데에 큰 도움이 되는데 공통적인 설계 목표는 서브 시스템들 사이의 의사소통 및 종속성을 최소화하려는 것
  - ✓ 퍼사드 객체는 복잡한 소프트웨어 바깥쪽의 코드가 라이브러리의 안쪽 코드에 의존하는 일을 감소시켜 주고 복잡한 소프트웨어를 사용 할 수 있게 간단한 인터페이스를 제공
  - ✓ 영화를 보기 위해 거쳐야 하는 과정
    - 음료를 준비한다.
    - TV를 켠다.
    - 영화를 검색한다.
    - 영화를 결제한다.
    - 영화를 재생한다.

Structural Pattern

- FACADE Pattern
  - ✓ RemoteControl.java

```
public class RemoteControl {

    public void Turn_On()
    {
        System.out.println("TV를 켜다");
    }
    public void Turn_Off()
    {
        System.out.println("TV를 끄다");
    }
}
```

Structural Pattern

- FACADE Pattern
  - ✓ Movie.java

```
public class Movie {

    private String name="";

    public Movie(String name)
    {
        this.name = name;
    }

    public void Search_Movie()
    {
        System.out.println(name+" 영화를 찾다");
    }

    public void Charge_Movie()
    {
        System.out.println("영화를 결제하다");
    }
```

Structural Pattern

- FACADE Pattern
  - ✓ Movie.java

```
    public void play_Movie()
    {
        System.out.println("영화 재생");
    }
}
```

Structural Pattern

- FACADE Pattern
  - ✓ Beverage.java

```
public class Beverage {

    private String name="";

    public Beverage(String name)
    {
        this.name = name;
    }

    public void Prepare()
    {
        System.out.println(name+" 음료 준비 완료 ");
    }

}
```

Structural Pattern

- FACADE Pattern
  - ✓ Facade.java

```
public class Facade {
    private String beverage_Name ="";
    private String Movie_Name="";

    public Facade(String beverage,String Movie_Name)
    {
        this.beverage_Name=beverage_Name;
        this.Movie_Name=Movie_Name;
    }
```

Structural Pattern

- FACADE Pattern
  - ✓ Facade.java

```
    public void view_Movie()
    {
        Beverage beverage = new Beverage(beverage_Name);
        RemoteControl remote= new RemoteControl();
        Movie movie = new Movie(Movie_Name);

        beverage.Prepare();
        remote.Turn_On();
        movie.Search_Movie();
        movie.Charge_Movie();
        movie.play_Movie();
    }
}
```

Structural Pattern

- FACADE Pattern
  - ✓ Viewer.java

```
public class Viewer {

    public static void main(String [] args)
    {
        Facade facade = new Facade("아메리카노","어벤져스");
        facade.view_Movie();
    }
}
```

Structural Pattern

- FLYWEIGHT Pattern
  - ✓ 공유(sharing)를 통해 많은 수의 소립(fine-grained) 객체들을 효과적으로 지원하는 패턴
  - ✓ 플라이급 객체는 공유 가능한 객체로, 여러 비슷한 상황에서 사용될 수 있지만 각각의 상황에서는 독립적인 객체로 동작

Structural Pattern

- FLYWEIGHT Pattern
  - ✓ Shape.java

```
public interface Shape {
    public void draw();
}
```

Structural Pattern

- FLYWEIGHT Pattern
  - ✓ Circle.java

```
public class Circle implements Shape {
    private String color;
    private int x;
    private int y;
    private int radius;

    public Circle(String color) {
        this.color = color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }
```

Structural Pattern

- FLYWEIGHT Pattern
  - ✓ Circle.java

```
public class Circle implements Shape {
    public void setRadius(int radius) {
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Circle [color=" + color + ", x=" + x + ", y=" + y + ", radius="
     + radius + "]");
    }
}
```

Structural Pattern

- FLYWEIGHT Pattern
  - ✓ ShapeFactory.java

```
import java.util.HashMap;

public class ShapeFactory {
   private static final HashMap<String, Circle> circleMap = new HashMap<>();

   public static Shape getCircle(String color) {
      Circle circle = (Circle)circleMap.get(color);

      if(circle == null) {
         circle = new Circle(color);
         circleMap.put(color,circle);
         System.out.println("==== 새로운 객체 생성 : " + color + "색 원 ====" );
      }

      return circle;
   }
}
```

Structural Pattern

- FLYWEIGHT Pattern
  - ✓ Main.java

```
public class Main {
    public static void main(String[] args) {
        String[] colors = {"Red","Green","Blue","Yellow"};

        for (int i = 0; i < 10; i++) {
            Circle circle = (Circle)ShapeFactory.getCircle(colors[(int)
 (Math.random()*4)]);
            circle.setX((int) (Math.random()*100));
            circle.setY((int) (Math.random()*4));
            circle.setRadius((int) (Math.random()*10));
            circle.draw();
        }
    }
}
```

Structural Pattern

- Proxy Pattern
  - ✓ 다른 객체에 대한 접근을 제어하기 위한 대리자 또는 자리채움자 역할을 하는 객체를 만드는 패턴
  - ✓ proxy는 대리인이라는 이라는 의미로 자바 코드에서 생각을 해보면 어떤 클래스의 수행을 대신 수행 하는 것으로 생각 할 수 있음
  - ✓ Proxy Pattern을 사용하는 경우는 어떤 클래스의 객체 생성이 오래 걸리는 경우 그 일을 분업을 하여 proxy 클래스에서 처리 할 수 있는 부분은 처리를 하고 proxy 클래스에서 처리 할 수 없는 작업에 대해서만 실제 클래스의 객체를 생성하고 위임하는 방식을 취함

Structural Pattern

- Proxy Pattern
  - ✓ Subject.java

```
public interface Subject {
    public void Show_Name();
    public void set_Name(String name);
    public void Complicated_Work();
}
```

Structural Pattern

- Proxy Pattern
  - ✓ RealSubject.java

```
public class RealSubject implements Subject {

    private String name;

    public void set_Name(String name)
    {
        this.name = name;
    }

    public void Show_Name()
    {
        System.out.println("my name is "+ name);
    }

    public void Complicated_Work()
    {
        System.out.println("proxy가 처리 못하는 작업을 수행합니다.");
    }

}
```

Structural Pattern

- Proxy Pattern
  - ✓ RealSubject.java

```
public class Proxy implements Subject{

    public RealSubject subject;
    public String name;

    public void set_Name(String name)
    {
        System.out.println("proxy가 대신해서 처리 할 수 있어요.");
        if(subject!= null)
        {
            subject.set_Name(name);
        }
        this.name =name;
    }

    public void Show_Name()
    {
        System.out.println("proxy가 대신해서 처리 할 수 있어요.");
        System.out.println("my name is"+ name);
    }
```

Structural Pattern

- Proxy Pattern
  - ✓ RealSubject.java

```
    public void Complicated_Work()
    {
        subject = new RealSubject();
        subject.Complicated_Work();
    }

}
```

Structural Pattern

- Proxy Pattern
  - ✓ Main.java

```
public class Main {

    public static void main(String argsp[])
    {
        Proxy proxy1 = new Proxy();

        proxy1.set_Name("홍길동");
        System.out.println("================");

        proxy1.Show_Name();
        System.out.println("================");

        proxy1.Complicated_Work();
    }

}
```

Behavioral Pattern

- Behavioral Pattern(행동 패턴)
  - ✓ 행동 패턴(behavioral pattern)은 어떤 처리의 책임을 어느 객체에 할당하는 것이 좋은지 알고리즘을 어느 객체에 정의하는 것이 좋은지 등을 다룸
  - ✓ 응용 프로그램에 따라서 행동이 다른 객체로 옮겨가거나 알고리즘이 대체될 때가 생기기 마련인데 이러한 변화의 개념을 만족할 수 있는 것이 행동 패턴
  - ✓ 행동 패턴은 객체나 클래스에 대한 패턴을 정의하는 것이 아니고 그들 간의 교류 방법에 대하여 정의하는 것으로 이러한 패턴은 런타임에 수행하기 어려운 복잡한 제어 구조를 패턴화한 것
  - ✓ 행동 패턴을 사용하면 우리는 객체 간의 제어 구조보다는 객체들을 어떻게 연결시킬 것인가에 좀더 중점을 둘 수 있음

Behavioral Pattern

- CHAIN OF RESPONSIBILITY Pattern(책임 연쇄 패턴)
  - ✓ 메시지를 보내는 객체와 이를 받아 처리하는 객체들 간의 결합도를 없애기 위한 패턴
  - ✓ 하나의 요청에 대한 처리가 반드시 한 객체에서만 되지 않고 여러 객체에게 그 처리 기회를 주려는 것
  - ✓ 여러 개의 객체 중에서 어떤 것이 요구를 처리할 수 있는지를 사전에 알 수 없을 때 사용
  - ✓ 요청 처리가 들어오게 되면 그것을 수신하는 객체가 자신이 처리 할 수 없는 경우에는 다음 객체에게 문제를 넘김으로써 최종적으로 요청을 처리 할 수 있는 객체의 의해 처리가 가능하도록 하는 패턴

Behavioral Pattern

- CHAIN OF RESPONSIBILITY Pattern(책임 연쇄 패턴)
  - ✓ Receiver.java

```
public abstract class Receiver {

    public String name;                 //리시버 이름
    private Receiver next=null;         //다음 수신 객체

    public Receiver(String name)
    {
        this.name=name;
    }

    public Receiver setNext(Receiver next) //다음에 수신 할 객체 지정
    {
        this.next = next;
        return next;
    }
```

Behavioral Pattern

- CHAIN OF RESPONSIBILITY Pattern(책임 연쇄 패턴)
  - ✓ Receiver.java

```
public final void support(int number) //처리
{
    if(resolve(number))                 //해당 수신 객체가 요청 처리를 하였는지 판단
    {
        done(number);
    }
    else if(next != null)               //처리 요청을 못하였다면 다음 수신객체에게 처리
 요청
    {
        next.support(number);
    }
    else                                //사슬에 있는 어떠한 객체도 처리 못한 경우
    {
        System.out.println("어떤 객체도 처리하지 못함");
    }
}
public abstract boolean resolve(int number);
public abstract void done(int number);
}
```

Behavioral Pattern

- CHAIN OF RESPONSIBILITY Pattern(책임 연쇄 패턴)
  - ✓ EvenReceiver.java

```
public class EvenReceiver extends Receiver{

    public EvenReceiver(String name)
    {
        super(name);
    }

    public boolean resolve(int number)    //해당 숫자가 짝수일 경우 true 반환
    {
        if(number%2==0)
            return true;
        else
            return false;
    }
    public void done(int number)
    {
        System.out.println(number +":" + name+"가 해결");
    }
}
```

Behavioral Pattern

- CHAIN OF RESPONSIBILITY Pattern(책임 연쇄 패턴)
  - ✓ OddReceiver.java

```
public class OddReceiver extends Receiver {

    public OddReceiver(String name)
    {
        super(name);
    }

    public boolean resolve(int number) //해당 숫자가 홀수일 경우 true 반환
    {
        if(number%2!=0)
            return true;
        else
            return false;
    }
    public void done(int number)
    {
        System.out.println(number +":" + name+"가 해결");
    }
}
```

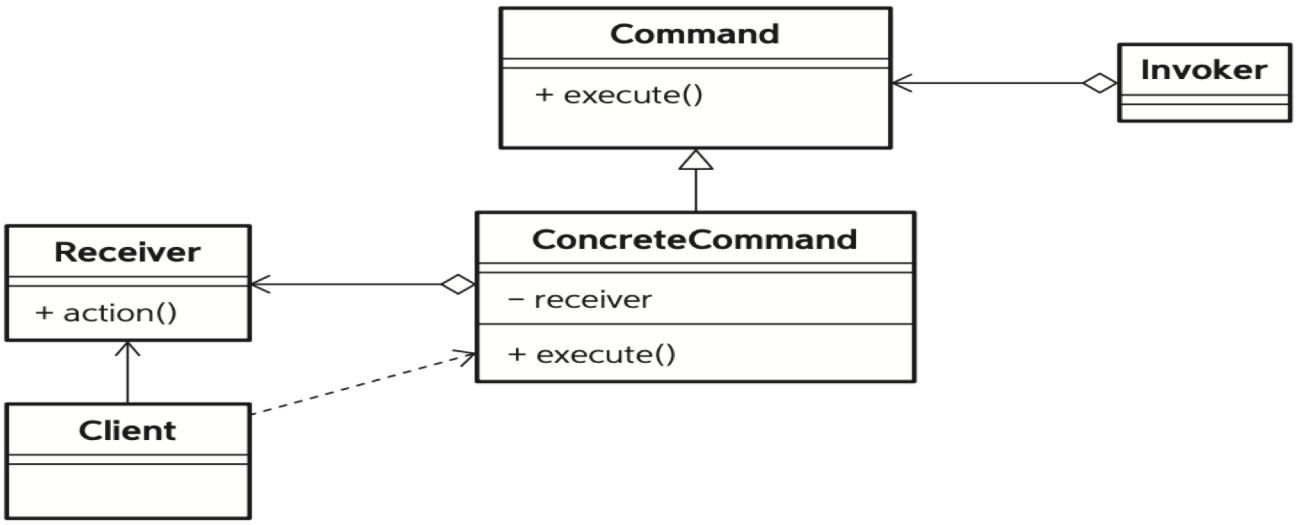
Behavioral Pattern

- CHAIN OF RESPONSIBILITY Pattern(책임 연쇄 패턴)
  - ✓ Main.java

```
public class Main {

    public static void main(String argsp[]) throws InterruptedException
    {
        Receiver odd_receiver = new OddReceiver("홀수 리시버");
        Receiver even_receiver = new EvenReceiver("짝수 리시버");

        odd_receiver.setNext(even_receiver);    //수신기의 처리 요청 순서를 지정

        for(int i=1;i<=20;i++)
        {
            odd_receiver.support(i);
        }
    }
}
```

Behavioral 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



Behavioral Pattern

- Command Pattern
  - ✓ Book.java

```
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;
    }
}
```

Behavioral Pattern

- Command Pattern
  - ✓ Command.java

```
abstract class Command {
    protected Book book;

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

    public abstract void execute();
}
```

Behavioral Pattern

- Command Pattern
  - ✓ DiscountCommand.java

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

Behavioral Pattern

- Command Pattern
  - ✓ SpecialDiscountCommand.java

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

Behavioral Pattern

- Command Pattern
  - ✓ Main.java

```
public class Main {
    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");
```

Behavioral Pattern

- Command Pattern
  - ✓ Main.java

```
                // 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");
        }

    }
```

Behavioral Pattern

- Interpreter Pattern
  - ✓ 어떤 언어의 대해, 그 언어의 문법에 대한 표현을 정의하면서 그것(표현)을 사용하여 해당 언어로 기술된 문장을 해석하는 해석자를 함께 정의
  - ✓ 프로그램을 작성할 때 프로그램의 모든 행위를 정의하지 못하는 경우도 있는데 예를 들어 브라우저를 만들 경우 사이트 디자이너가 웹 페이지를 어떻게 행동하게 하고 싶어할 지에 대해 모두 예측하는 것은 불가능하므로 자바스크립트와 같은 인터프리터 언어를 통해 브라우저에 브라우저 프로그래머가 구현하지 않은 행위를 추가할 수 있도록 함
  - ✓ Interpreter 패턴은 이러한 용도에 사용되는 인터프리터를 작성할 수 있도록 해주는 패턴으로 우선 언어의 문법을 기술하는 규칙들에 대한 형식 문법을 정의하리고 각 규칙들을 클래스를 통해 구현하는데 이들 클래스는 Context 객체를 공유하며 Context 객체로부터 입력을 받고 변수 값을 저장하는 등의 작업을 하게 됨
  - ✓ Interpreter는 실제 작업을 하는(상태 머신과 같은) 효과적인 출력 프로세서를 생성하기도 하지만 꼭 그러한 것은 아님
  - ✓ 한 클래스 내의 여러 메서드에서 일종의 묵시적 언어를 이루는 요소들을 조합하고 있다면, 그 묵시적 언어의 요소들을 각각의 클래스로 정의하고 그 객체의 조합을 통해 해석 가능한 수식을 만들어낼 수 있도록 함

Behavioral Pattern

- Interpreter Pattern
  - ✓ boolean 표현식을 평가할 수 있는 간단한 언어
    - Logic.java
      - Logic은 AbstractExpression의 역할
      - Values는 변수명과 값을 보관하는 Context 역할을 하는데 전역 정보를 관리

```
import java.util.HashMap;
import java.util.Map;

public interface Logic {
    // Logic.Values 는 Boolean 변수들을 보관하는 일종의 namespace 이다.
    public static class Values {
            static Map<String, Boolean> vars = new HashMap<>();

            // 변수명과 변수값을 할당한다.
            static void assign(String key, boolean value) {
                    if (key == null || key.length() <= 0) {
                            throw new RuntimeException("assign
failed");
                    }
                    vars.put(key, value ? Boolean.TRUE :
Boolean.FALSE);
            }
```

Behavioral Pattern

- Interpreter Pattern
  - ✓ boolean 표현식을 평가할 수 있는 간단한 언어
    - Logic.java
      - Logic은 AbstractExpression의 역할
      - Values는 변수명과 값을 보관하는 Context 역할을 하는데 전역 정보를 관리

```
        // 변수 이름으로 변수값을 찾는다.
        static boolean lookup(String key) {
                Object got = vars.get(key);
                return (Boolean) got;
        }
    }

    boolean evaluate();
}
```

Behavioral Pattern

- Interpreter Pattern
  - ✓ boolean 표현식을 평가할 수 있는 간단한 언어
    - ANDLogic.java
      - ◆ ANDLogic은 NonterminalExpression이며 AND 연산을 정의

```
public class ANDLogic implements Logic {
    Logic left, right;

    public ANDLogic(Logic left, Logic right) {
            this.left = left;
            this.right = right;
    }

    @Override
    public boolean evaluate() {
            return left.evaluate() && right.evaluate();
    }
}
```

Behavioral Pattern

- ❖ Interpreter Pattern
  - ✓ boolean 표현식을 평가할 수 있는 간단한 언어
    - ● ORLogic.java
      - ◆ ORLogic은 NonterminalExpression이며 OR 연산을 정의

```
public class ORLogic implements Logic {
    Logic left, right;

    public ORLogic(Logic left, Logic right) {
            this.left = left;
            this.right = right;
    }

    @Override
    public boolean evaluate() {
            return left.evaluate() || right.evaluate();
    }
}
```

Behavioral Pattern

- ❖ Interpreter Pattern
  - ✓ boolean 표현식을 평가할 수 있는 간단한 언어
    - ● NOTLogic.java
      - ◆ NOTLogic은 NonterminalExpression이며 NOT 연산을 정의

```
public class NOTLogic implements Logic {
    Logic value;

    public NOTLogic(Logic value) {
            this.value = value;
    }

    @Override
    public boolean evaluate() {
            return !value.evaluate();
    }
}
```

Behavioral Pattern

- ❖ Interpreter Pattern
  - ✓ boolean 표현식을 평가할 수 있는 간단한 언어
    - ● Variable.java
      - ◆ NOTLogic은 NonterminalExpression이며 이름과 값을 갖는 변수를 정의

```
public class Variable implements Logic {
    private String name;

    public Variable(String name) {
            this.name = name;
    }

    @Override
    public String toString() {
            return this.name;
    }

    @Override
    public boolean evaluate() {
            return Logic.Values.lookup(name);
    }
}
```

Behavioral Pattern

- Interpreter Pattern
  - ✓ boolean 표현식을 평가할 수 있는 간단한 언어
    - Main.java

```
public class Main {

    public static void main(String[] args) {
      Logic.Values.assign("A", true);
      Logic.Values.assign("B", false);

      Logic term1 = new ANDLogic(
          new Variable("A"), new Variable("B"));
      System.out.println(term1);
    }

}
```

Behavioral 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

Behavioral Pattern

- Iterator Pattern
  - ✓ CollectionMain.java

```
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> {

}
```

Behavioral Pattern

- Iterator Pattern
  - ✓ CollectionMain.java

```
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);
            }

    }

}
```

Behavioral 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

Behavioral Pattern

- Iterator Pattern
  - ✓ Iterator.java

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

Behavioral Pattern

- Iterator Pattern
  - ✓ Aggregate.java

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

Behavioral Pattern

- Iterator Pattern
  - ✓ Song.java

```
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;
    }
    public String getTitle() {
            return title;
    }
    public void setTitle(String title) {
            this.title = title;
    }
```

Behavioral Pattern

- Iterator Pattern
  - ✓ Song.java

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

Behavioral Pattern

- Iterator Pattern
  - ✓ SongShelf.java

```
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++;
    }
```

Behavioral Pattern

- Iterator Pattern
  - ✓ SongShelf.java

```
    public int getLength() {
        return last;
    }
    public Iterator iterator() {
        return new SongShelfIterator(this);
    }
}
```

Behavioral Pattern

- Iterator Pattern
  - ✓ SongShelfIterator.java

```
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;
        }
    }
```

Behavioral Pattern

- Iterator Pattern
  - ✓ SongShelfIterator.java

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

Behavioral Pattern

- Iterator Pattern
  - ✓ Main.java

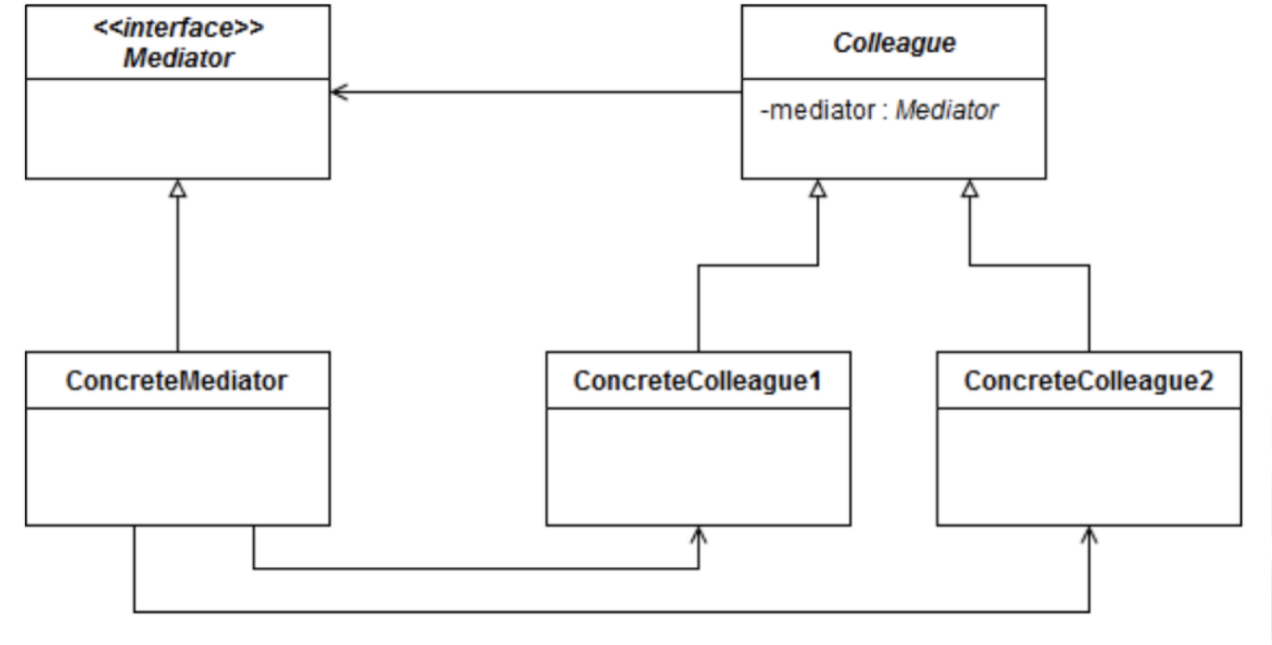
```
public class Main {
    public static void main(String[] args) {
        SongShelf songShelf = new SongShelf();
        songShelf.appendSong(new Song("달라달라", "ITZY"));
        songShelf.appendSong(new Song("빨간맛", "RedVelvet"));
        songShelf.appendSong(new Song("휘파람","BlackPink"));
        songShelf.appendSong(new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

Behavioral Pattern

- Mediator Pattern
  - ✓ 중재자 패턴은 객체의 관계를 하나의 객체로 정리하는 패턴
  - ✓ 중자재 패턴은 서로 의존적인 M:N 관계를 가진 객체를 느슨한 1:1 관계로 변경
  - ✓ 복잡한 통신과 제어를 한 곳에 집중하여 처리하는 효과가 있음
  - ✓ 다른 동료 객체에 직접 접근해서 호출하지 않고 중재자를 의존해서 다른 동료 객체를 호출
  - ✓ 중재자 패턴은 객체의 강력한 구조적 결합 문제점을 해결
  - ✓ 중재자를 이용하지 않으면 다수의 동료 객체가 서로 정보를 직접 주고받는데 중재자 패턴은 동료 객체끼리 정보를 직접 주고받지 않도록 통신 경로를 제한
  - ✓ 중재자는 하나의 객체 요청에 대해 모든 객체로 통보를 처리해야 하므로 경로의 수가 증가
  - ✓ 중재자 패턴을 설계할 때는 경로의 수가 증가함에 따라 성능이 저하되지 않도록 신경써서 구성해야 함

Behavioral Pattern

- Mediator Pattern
  - ✓ 구조



Behavioral Pattern

- Mediator Pattern
  - ✓ 구조
    - Mediator: 동료 간 통신을 위한 인터페이스
    - Collegue: 동료 간에 전달되는 이벤트를 정의하는 추상 클래스
    - ConcreteMedediator: Colleague 객체를 조정하여 협동 조작을 구현하고 동료를 유지 관리
    - ConcreteColleague: 다른 Colleague가 생성한 Mediator를 통해 받은 알림 작업을 구현

Behavioral Pattern

- Mediator Pattern
  - ✓ Mediator.java

```
public interface Mediator {

    void addColleague(Colleague colleague);

    void mediate(Colleague colleague);
}
```

Behavioral Pattern

- Mediator Pattern
  - ✓ ColleagueType.java

```
public enum ColleagueType {
    USER, SYSTEM, ADMIN
}
```

Behavioral Pattern

- Mediator Pattern
  - ✓ Colleague.java

```
public abstract class Colleague {
    private Mediator mediator;
    private String message;
    private final String name;
    private final ColleagueType type;

    protected Colleague(String name, ColleagueType type) {
        this.name = name;
        this.type = type;
    }

    public void setMediator(Mediator mediator) {
        this.mediator = mediator;
    }

    public void setMessage(String message) {
        this.message = message;
    }
```

Behavioral Pattern

- Mediator Pattern
  - ✓ Colleague.java

```
public Mediator getMediator() {
    return mediator;
}

public String getMessage() {
    return message;
}

public String getName() {
    return name;
}

public ColleagueType getType() {
    return type;
}

public void send() {
    System.out.println(this.name + " send()");
    System.out.println();
    mediator.mediate(this);
}
```

Behavioral Pattern

- Mediator Pattern
  - ✓ Colleague.java

```
        public abstract void receive(Colleague colleague);

    }
```

Behavioral Pattern

- Mediator Pattern
  - ✓ SystemConcreteColleague.java

```
public class SystemConcreteColleague extends Colleague {
    public SystemConcreteColleague(String name) {
        super(name, ColleagueType.SYSTEM);
    }

    @Override
    public void receive(Colleague colleague) {
        System.out.println("System can't receive messages");
    }
}
```

Behavioral Pattern

- Mediator Pattern
  - ✓ UserConcreteColleague.java

```
public class UserConcreteColleague extends Colleague {
    public UserConcreteColleague(String name) {
        super(name, ColleagueType.USER);
    }

    @Override
    public void receive(Colleague colleague) {
        if (ColleagueType.SYSTEM == colleague.getType()) {
            System.out.print("[SYSTEM] ");
        } else if (ColleagueType.USER == colleague.getType()) {
            System.out.print("[" + colleague.getName() + "] ");
        }
        System.out.println(colleague.getMessage());
    }
}
```

Behavioral Pattern

- Mediator Pattern
  - ✓ AdminConcreteColleague.java

```
public class AdminConcreteColleague extends Colleague {
    public AdminConcreteColleague(String name) {
        super(name, ColleagueType.ADMIN);
    }

    @Override
    public void receive(Colleague colleague) {
        System.out.println("Admin can't receive messages");
    }
}
```

Behavioral Pattern

- Mediator Pattern
  - ✓ ConcreteMediator.java

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

public class ConcreteMediator implements Mediator {
    private List<Colleague> colleagueList;

    public ConcreteMediator() {
        this.colleagueList = new ArrayList<>();
    }

    public ConcreteMediator(List<Colleague> colleagueList) {
        this.colleagueList = new ArrayList<>();
    }

    @Override
    public void addColleague(Colleague colleague) {
        this.colleagueList.add(colleague);
    }
```

Behavioral Pattern

- Mediator Pattern
  - ✓ ConcreteMediator.java

```
@Override
public void mediate(Colleague colleague) {
    for (Colleague receiverColleague : colleagueList) {
        System.out.printf("₩t[Mediating " + colleague.getName() + " to " +
 receiverColleague.getName() + "] ");
        receiverColleague.receive(colleague);
    }
}
}
```

Behavioral Pattern

- Mediator Pattern
  - ✓ Main.java

```
public class Main {
    public static void main(String [] args) {
        Mediator mediator = new ConcreteMediator();
          Colleague colleagueUser1 = new UserConcreteColleague("User1");
          Colleague colleagueUser2 = new UserConcreteColleague("User2");
          Colleague colleagueSystem = new SystemConcreteColleague("System");
          Colleague colleagueAdmin = new AdminConcreteColleague("Admin");

          colleagueUser1.setMediator(mediator);
          colleagueUser2.setMediator(mediator);
          colleagueSystem.setMediator(mediator);
          colleagueAdmin.setMediator(mediator);

          mediator.addColleague(colleagueUser1);
          mediator.addColleague(colleagueUser2);
          mediator.addColleague(colleagueSystem);
          mediator.addColleague(colleagueAdmin);
```

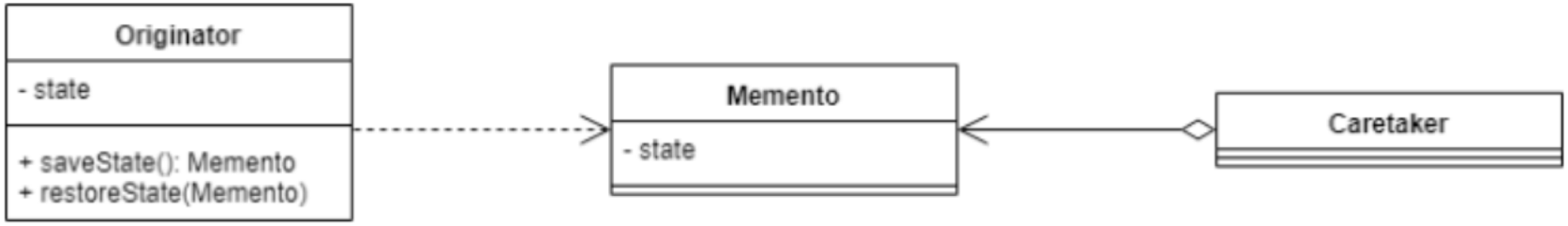
Behavioral Pattern

- Mediator Pattern
  - ✓ Main.java

```
        colleagueUser1.setMessage("안녕하세요. User1이 보낸 메시지 입니다.");
        colleagueUser1.send();

        colleagueUser2.setMessage("안녕하세요. User2가 보낸 메시지 입니다.");
        colleagueUser2.send();

        colleagueSystem.setMessage("잠시 후 20분 뒤에 점검이 있습니다.");
        colleagueSystem.send();
    }
}
```

Behavioral Pattern

- Memento Pattern
  - ✓ 메멘토 패턴은 객체의 상태 정보를 저장하고 사용자의 필요에 의하여 원하는 시점의 데이터를 복원 할 수 있는 패턴



Behavioral Pattern

- Memento Pattern
  - ✓ 상태 클래스 (TextWindowState)

```
public class TextWindowState {

    private String text;

    public TextWindowState(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

}
```

Behavioral Pattern

- Memento Pattern
  - ✓ 원조본(Originator)

```
public class TextWindow {
    private StringBuilder currentText;

    public TextWindow() {
        this.currentText = new StringBuilder();
    }

    public String getCurrentText() {
        return currentText.toString();
    }

    public void addText(String text) {
        currentText.append(text);
    }

    public TextWindowState save() {
        return new TextWindowState(currentText.toString());
    }
```

Behavioral Pattern

- Memento Pattern
  - ✓ 원조본(Originator)

```
        public void restore(TextWindowState save) {
            currentText = new StringBuilder(save.getText());
        }
    }
```

Behavioral Pattern

- Memento Pattern
  - ✓ Caretaker (케어테이커)

```
public class TextEditor {
    private TextWindow textWindow;
    private TextWindowState savedTextWindow;

    public TextEditor(TextWindow textWindow) {
        this.textWindow = textWindow;
    }

    public void write(String text) {
        textWindow.addText(text);
    }

    public String print() {
        return textWindow.getCurrentText();
    }

    public void hitSave() {
        savedTextWindow = textWindow.save();
    }
```

Behavioral Pattern

- Memento Pattern
  - ✓ Caretaker (케어테이커)

```
        public void hitUndo() {
            textWindow.restore(savedTextWindow);
        }
    }
```

Behavioral Pattern

- Memento Pattern
  - ✓ Main

```
public class Main {

    public static void main(String[] args) {
        TextEditor textEditor = new TextEditor(new TextWindow());
        textEditor.write("메멘토 패턴을 공부한다.₩n");
        textEditor.hitSave();
        textEditor.write("오늘은 집중이 안되는 날이다.₩n");
        textEditor.hitUndo();
        textEditor.write("언능 끝내고 자야겠다.₩n");
        textEditor.hitUndo();
        System.out.println(textEditor.print());
    }
}
```

Behavioral 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

Behavioral Pattern

- Observer Pattern
  - ✓ Observer.java

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

Behavioral Pattern

- Observer Pattern
  - ✓ Subject.java

```
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();
}
```

Behavioral Pattern

- Observer Pattern
  - ✓ Client.java

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

Behavioral Pattern

- Observer Pattern
  - ✓ DataChanger.java

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

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

Behavioral Pattern

- Observer Pattern
  - ✓ Main.java

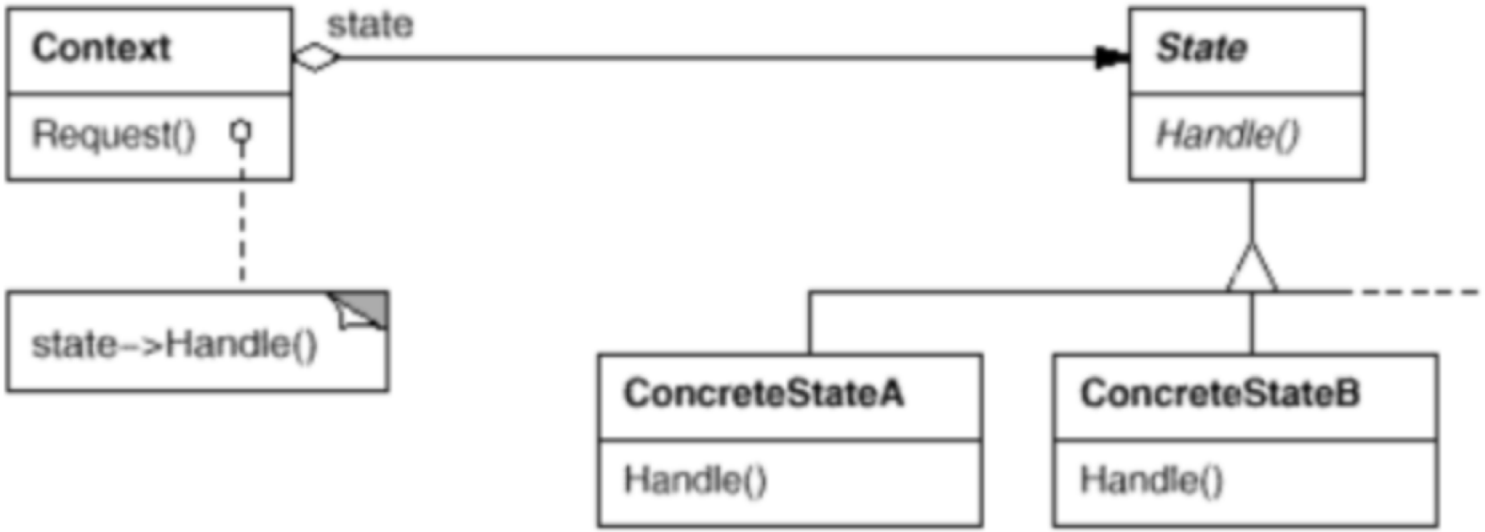
```
public class Main {
    public static void main(String... args) {
            Observer observer = new Client();
            Subject dataChanger = new DataChanger();

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

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

Behavioral Pattern

- State Pattern
  - ✓ 상태 패턴은 조건에 따른 별개의 동작을 제어문으로 사용하지 않고 대신 객체를 캡슐화하여 독립된 동작으로 구분
  - ✓ 각각의 함수 형태로 구별하는 것과 달리 객체로 동작을 분리
  - ✓ 객체의 상태에 따라 위임하는 객체를 변경
  - ✓ 상태 패턴에서는 함수 형태가 아니라 서브 클래스 형태로 분리
  - ✓ 각각의 상태를 객체로 캡슐화하기 때문에 클래스 파일이 늘어난다는 단점이 있으나 상태 패턴을 이용하지 않고 수많은 조건문을 사용하는 것보다는 유연하게 확장할 수 있음
  - ✓ 상태 패턴은 행동 패턴으로 분류되고 객체 내부 상태에 따른 동작 객체를 결정
  - ✓ 상태별로 분리된 동작 객체는 독립적이며, 각 상태값에 따라 국지화된 객체 행위를 위임

Behavioral Pattern

❖ State Pattern



- ✓ State: 상태 인터페이스
- ✓ ConcreteState: 상태의 구체화 클래스
- ✓ Context: 상태 값을 가지고 상태 값에 따른 동작을 수행하는 클래스

Behavioral Pattern

- State Pattern
  - ✓ PowerState.java

```
public interface PowerState {
    PowerState button();
}
```

Behavioral Pattern

- State Pattern
  - ✓ On.java

```
public class On implements PowerState {
    @Override
    public PowerState button() {
        System.out.println("전원이 꺼졌습니다.");
        return new Off();
    }
}
```

Behavioral Pattern

- State Pattern
  - ✓ Off.java

```
public class Off implements PowerState {
    @Override
    public PowerState button() {
        System.out.println("전원이 켜졌습니다.");
        return new On();
    }
}
```

Behavioral Pattern

- State Pattern
  - ✓ Radio.java

```
public class Radio {
    private PowerState powerState;

    public Radio() {
        System.out.println(this.getClass().getSimpleName());
        this.powerState = new Off();
    }

    public void button() {
        powerState = powerState.button();
    }
}
```

Behavioral Pattern

- State Pattern
  - ✓ TV.java

```
public class TV {
    private PowerState state;

    public TV() {
        System.out.println(this.getClass().getSimpleName());
        this.state = new Off();
    }

    public void button() {
        state = state.button();
    }
}
```

Behavioral Pattern

- State Pattern
  - ✓ Main.java

```
public class Main {

    public static void main(String[] args) {
            TV tv = new TV();
       tv.button();
       tv.button();

       Radio radio = new Radio();
       radio.button();
       radio.button();

    }

}
```

Behavioral 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

Behavioral Pattern

- Strategy Pattern
  - ✓ Market.java

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

Behavioral Pattern

- Strategy Pattern
  - ✓ IphoneMarket.java

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

Behavioral Pattern

- Strategy Pattern
  - ✓ AndroidMarket.java

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

Behavioral Pattern

- Strategy Pattern
  - ✓ InternetPhone.java

```
abstract class InternetPhone {
    private Market market;

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

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

Behavioral Pattern

- Strategy Pattern
  - ✓ Apple.java

```
class Apple extends InternetPhone {

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

Behavioral Pattern

- Strategy Pattern
  - ✓ Google.java

```
class Google extends InternetPhone {

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

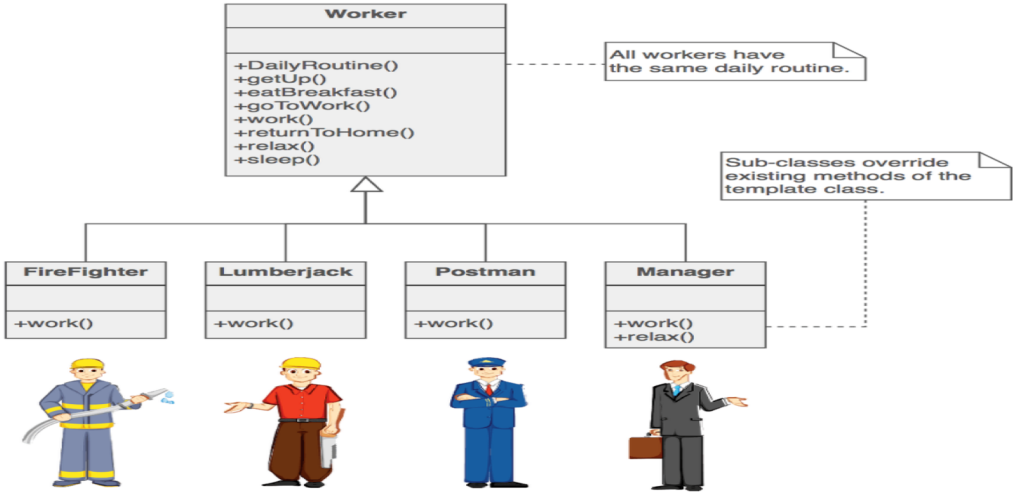
Behavioral Pattern

- Strategy Pattern
  - ✓ Main.java

```
public class Main {

    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();
    }

}
```

Behavioral Pattern

- Template Method 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 Template Method가 되는 것이고 이런 식의 Pattern을 Template Method Pattern

Behavioral Pattern

- Template Method Pattern
  - ✓ Main.java

```
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.");
    }
}
```

Behavioral Pattern

- Template Method Pattern
  - ✓ Main.java

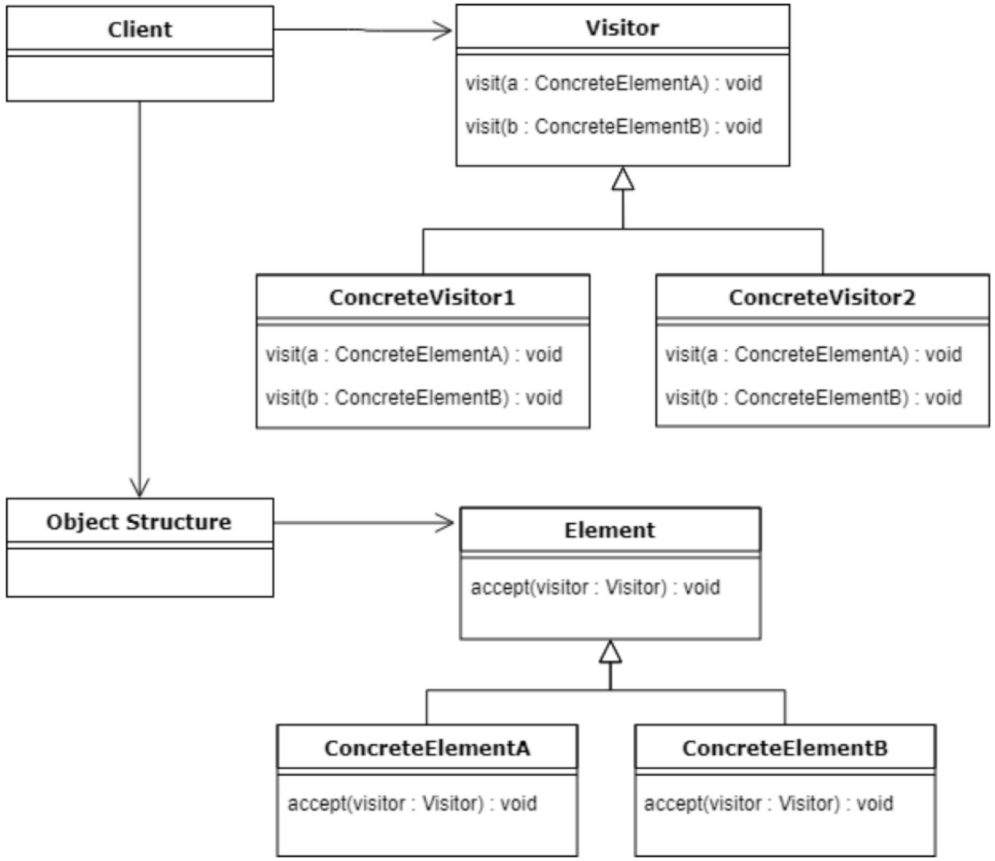
```
public class Main {
    public static void main(String[] args) {
            SmartPhone phone = new Iphone();
            phone.call();

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

Behavioral Pattern

- Visitor Pattern
  - ✓ 방문자(Visitor) 패턴은 방문자(Visitor)와 방문 공간(Visitable)을 분리하여 공통된 객체의 데이터 구조와 처리를 분리하는 패턴
  - ✓ OCP(개방 폐쇄 원칙)을 위한 디자인 패턴으로 기존 클래스를 수정하지 않고 새로운 기능을 코드에 추가하기 위해 사용하는 패턴
  - ✓ 각 Element 구조에 Visitor 클래스를 accept 함으로써 새로운 함수를 추가할 수 있음
  - ✓ 그렇게 하면 Element를 통해 ConcreteVisitor가 Element를 'visit' 하고 해당 요소에 대해 필요한 작업을 수행할 수 있음
  - ✓ 결과적으로 코드를 수정하지 않고 OCP를 준수하면서 새로운 ConcreteVisitor를 구현하여 기능을 확장할 수 있음

Behavioral Pattern

- Visitor Pattern
  - ✓ 구조



Behavioral Pattern

- ❖ Visitor Pattern
  - ✓ 구조
    - Visitor
      - ◆ 방문자 클래스의 인터페이스
      - ◆ visit(element)를 공용 인터페이스로 사용
      - ◆ element는 공용 공간
    - ConcreteVisitor: Visitor 구체 클래스
    - Element
      - ◆ 방문 공간 클래스의 인터페이스
      - ◆ accept(visitor)를 공용 인터페이스로 사용
      - ◆ visitor는 방문자
      - ◆ 내부적으로 Visitor.visit(this)를 호출
    - ConcreteElement: Element 구체 클래스

Behavioral Pattern

- Visitor Pattern
  - ✓ 구조
    - Visitor
      - 방문자 클래스의 인터페이스
      - visit(element)를 공용 인터페이스로 사용
      - element는 공용 공간
    - ConcreteVisitor: Visitor 구체 클래스
    - Element
      - 방문 공간 클래스의 인터페이스
      - accept(visitor)를 공용 인터페이스로 사용
      - visitor는 방문자
      - 내부적으로 Visitor.visit(this)를 호출
    - ConcreteElement: Element 구체 클래스

Behavioral Pattern

- Visitor Pattern
  - ✓ 구조
    - Element.java

```
public abstract class Element {
    public String uuid;

    protected Element(String uuid) {
        this.uuid = uuid;
    }

    public abstract void accept(Visitor v);
}
```

Behavioral Pattern

- Visitor Pattern
  - ✓ 구조
    - JsonElement.java

```
public class JsonElement extends Element {
    public JsonElement(String uuid) {
        super(uuid);
    }

    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

Behavioral Pattern

- ❖ Visitor Pattern
  - ✓ 구조
    - XmlElement.java

```
public class XmlElement extends Element {
    public XmlElement(String uuid) {
        super(uuid);
    }

    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

Behavioral Pattern

- Visitor Pattern
  - ✓ 구조
    - Document.java

```
import java.util.ArrayList;
import java.util.List;
public class Document extends Element {

    private final List<Element> elementList;

    public Document(String uuid) {
        super(uuid);
        this.elementList = new ArrayList<>();
    }

    @Override
    public void accept(Visitor v) {
        for (Element e : this.elementList) {
            e.accept(v);
        }
    }
```

Behavioral Pattern

- Visitor Pattern
  - ✓ 구조
    - Document.java

```
        public List<Element> getElementList() {
            return elementList;
        }
    }
```

Behavioral Pattern

- ❖ Visitor Pattern
  - ✓ 구조
    - ● Visitor.java

```
public interface Visitor {
    void visit(Element element);
}
```

Behavioral Pattern

- ❖ Visitor Pattern
  - ✓ 구조
    - ● ElementVisitor.java

```
public class ElementVisitor implements Visitor {

    @Override
    public void visit(Element element) {
        System.out.println("Element [" + element.uuid + "]");
    }

}
```

Behavioral Pattern

- Visitor Pattern
  - ✓ 구조
    - Main.java

```
import java.util.UUID;

public class Main {

        public  static void main(String [] args) {
            Visitor visitor = new ElementVisitor();
            Document document = new Document(generateUuid());
            document.getElementList().add(new JsonElement(generateUuid()));
            document.getElementList().add(new JsonElement(generateUuid()));
            document.getElementList().add(new XmlElement(generateUuid()));

            document.accept(visitor);
        }

        private static String generateUuid() {
            return UUID.randomUUID().toString();
        }

}
```