

Design Pattern With JAVA

박은종

1. 다음 코드를 상속과 Factory pattern 을 이용하여 리팩토링 하세요

```
public class Car {

    public static final String SONATA = "Sonata";
    public static final String GRANDEUR = "Grandeur";
    public static final String GENESIS = "Genesis";

    String productName;

    public Car(String productName) {
        this.productName = productName;
    }

    public String toString() {
        return productName;
    }
}

public class CarTest {

    public static void main(String[] args) {

        CarTest test = new CarTest();
        Car car = test.produceCar("Sonata");

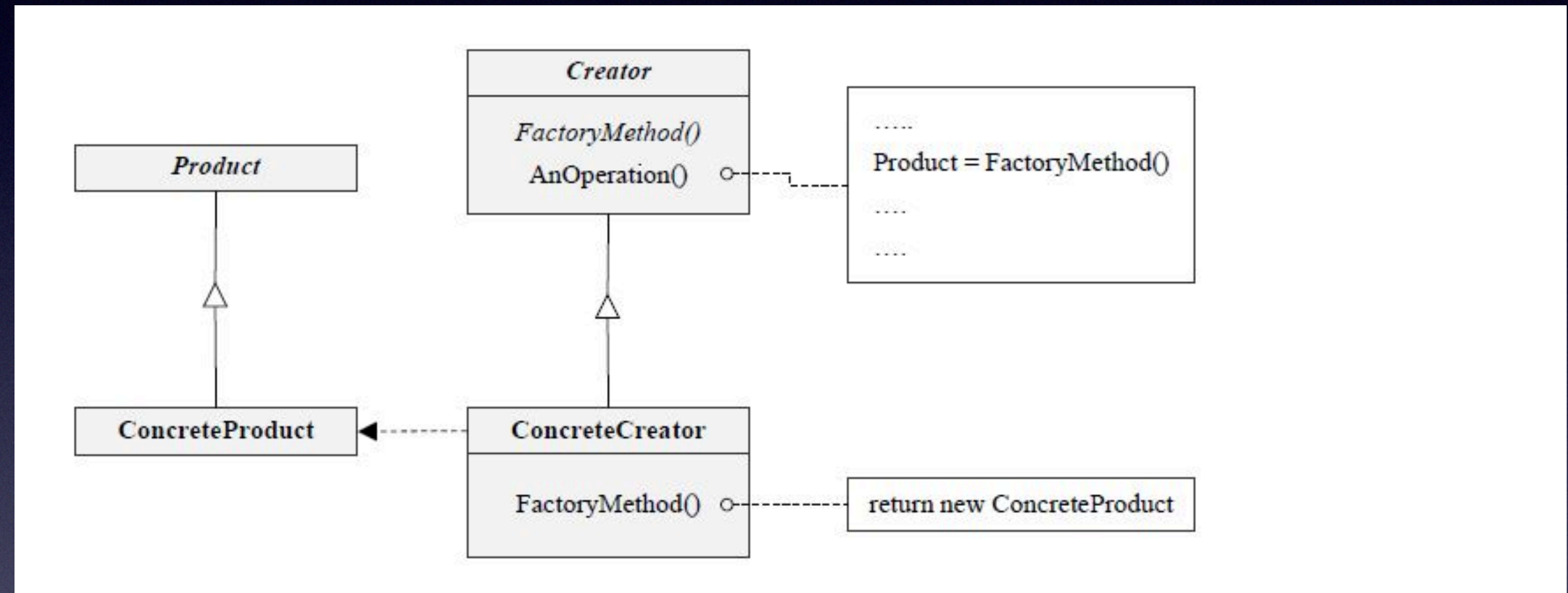
        System.out.println(car);
    }

    public Car produceCar(String name) {

        Car car = null;

        if( name.equalsIgnoreCase(Car.SONATA)) {
            car = new Car(Car.SONATA);
        }
        else if( name.equalsIgnoreCase(Car.GRANDEUR)) {
            car = new Car(Car.GRANDEUR);
        }
        else if( name.equalsIgnoreCase(Car.GENESIS)) {
            car = new Car(Car.GENESIS);
        }
        else {
            car = new Car("noname");
        }

        return car;
    }
}
```



2. 아래의 코드가 항상 true가 되도록 Singleton Pattern 으로 구현하세요

```
public class SingletonTest {  
  
    public static void main(String[] args) {  
  
        Singleton instanceA = Singleton.getInstance();  
        Singleton instanceB = Singleton.getInstance();  
  
        System.out.println(instanceA == instanceB);  
    }  
}  
  
#static
```


3. 다음 객체를 구현하는데 Decorator pattern을 활용해보세요

- 커피를 만듭니다.

커피 원두는 여러 종류가 있습니다. 이 원두를 활용하여 만드는 커피는 아메리카노, 라떼, 모카커피, 휘핑크림이 올라간 모카커피 등이 있습니다.

원두의 종류가 케냐와 에티오피아가 있다고 할 때 다음과 같이 여러 종류의 커피를 만들 수 있는 클래스 구조를 디자인 해보세요

커피에 첨가되는 장식자(Decorator)들은 다양하게 첨가되거나 바뀔 수 있습니다.

```
KenyaAmericano  
KenyaAmericano Adding Milk  
KenyaAmericano Adding Milk Adding Mocha Syrup  
EthiopiaAmericano Adding Milk Adding Mocha Syrup Adding WhippedCream
```

java I/O Stream

4. 상태에 따른 객체의 변화에 대한 코드를 구현하기

```
public class Player {

    public static final int BEGINNER_LEVEL = 1;
    public static final int ADVANCED_LEVEL = 2;
    public static final int SUPER_LEVEL = 3;

    int level;

    public Player() {
        level = BEGINNER_LEVEL;
    }

    public void jump() {

        if(level == BEGINNER_LEVEL) {
            System.out.println("Jump 할 줄 모르지롱.");
        }
        else if(level == ADVANCED_LEVEL) {
            System.out.println("높이 jump 합니다.");
        }
        else if(level == SUPER_LEVEL) {
            System.out.println("아주 높이 jump 합니다.");
        }
    }
}
```



```

public void run() {
    if(level == BEGINNER_LEVEL) {
        System.out.println("천천히 달립니다.");
    }
    else if(level == ADVANCED_LEVEL) {
        System.out.println("빨리 달립니다.");
    }
    else if(level == SUPER_LEVEL) {
        System.out.println("엄청 빨리 달립니다.");
    }
}

public void turn() {
    if(level == BEGINNER_LEVEL) {
        System.out.println("Turn 할 줄 모르지롱.");
    }
    else if(level == ADVANCED_LEVEL) {
        System.out.println("Turn 할 줄 모르지롱.");
    }
    else if(level == SUPER_LEVEL) {
        System.out.println("한 바퀴 돕니다.");
    }
}

public void play(int time) {
    run();
    for(int i =0; i<time; i++) {
        jump();
    }
    turn();
}

public void upgradeLevel(int level) {
    this.level = level;
}
}

```




```
public class MainBoard {  
    public static void main(String[] args) {  
        Player player = new Player();  
        player.play(1);  
        player.upgradeLevel(Player.ADVANCED_LEVEL);  
        player.play(2);  
        player.upgradeLevel(Player.SUPER_LEVEL);  
        player.play(3);  
    }  
}
```


5. 정책이나 알고리즘을 분리하여 대체 가능하도록하는 Strategy Pattern

학교의 학생이 수강 신청을 하게되면 과목마다 각각 성적을 받게 된다.

이때, 성적에 대한 학점을 부여하는 정책은 다음과 같이 여러가지가 있을 수 있다. 전공 관련 여부에 따라 학점이 다르게 부여되는 경우,

Pass/Fail로 만 학점을 부여하는 경우등 다양한 학점에 대한 정책이 있을때 각 성적에 대해 다양한 정책을 어떻게 구현하면 좋을까?

전공 과목인 경우의 학점 부여 방식

S	A	B	C	D	F
95~100점	90~94점	80~89점	70~79점	60~69점	60점 미만

비전공 과목인 경우의 학점 부여 방식

A	B	C	D	F
90~100점	80~89점	70~79점	55~69점	55점 미만

이름	전공과목	국어	수학	영어
Kim	수학	100	100	
Lee	국어	55	55	100

아래의 코드에 성적의 정책을 추가하여 리포트를 구현하세요

```
public class Student {  
  
    int studentID;  
    String studentName;  
    ArrayList<Subject> subjectList;  
  
    public static final int BASIC = 0;  
    public static final int MAJOR = 1;  
  
    public Student(int studentID, String studentName){  
        this.studentID = studentID;  
        this.studentName = studentName;  
  
        subjectList = new ArrayList<Subject>();  
    }  
  
    public void addSubject(String name, int score, boolean majorCode){  
        Subject subject = new Subject();  
  
        subject.setName(name);  
        subject.setScorePoint(score);  
        subject.setMajorCode(majorCode);  
        subjectList.add(subject);  
    }  
}
```



```
public class Subject {  
  
    private String name;  
    private int scorePoint;  
    private boolean majorCode;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getScorePoint() {  
        return scorePoint;  
    }  
    public void setScorePoint(int scorePoint) {  
        this.scorePoint = scorePoint;  
    }  
    public boolean isMajorCode() {  
        return majorCode;  
    }  
    public void setMajorCode(boolean majorCode) {  
        this.majorCode = majorCode;  
    }  
}
```



```

public class StudentTest {

    public static void main(String[] args) {
        Student studentLee = new Student(1001, "Lee");

        studentLee.addSubject("국어", 100, false);
        studentLee.addSubject("수학", 100, true);

        Student studentKim = new Student(1002, "Kim");

        studentKim.addSubject("국어", 55, true);
        studentKim.addSubject("수학", 55, false);
        studentKim.addSubject("영어", 100, false);

        studentLee.showGradeInfo();
        System.out.println("=====");
        studentKim.showGradeInfo();
    }
}

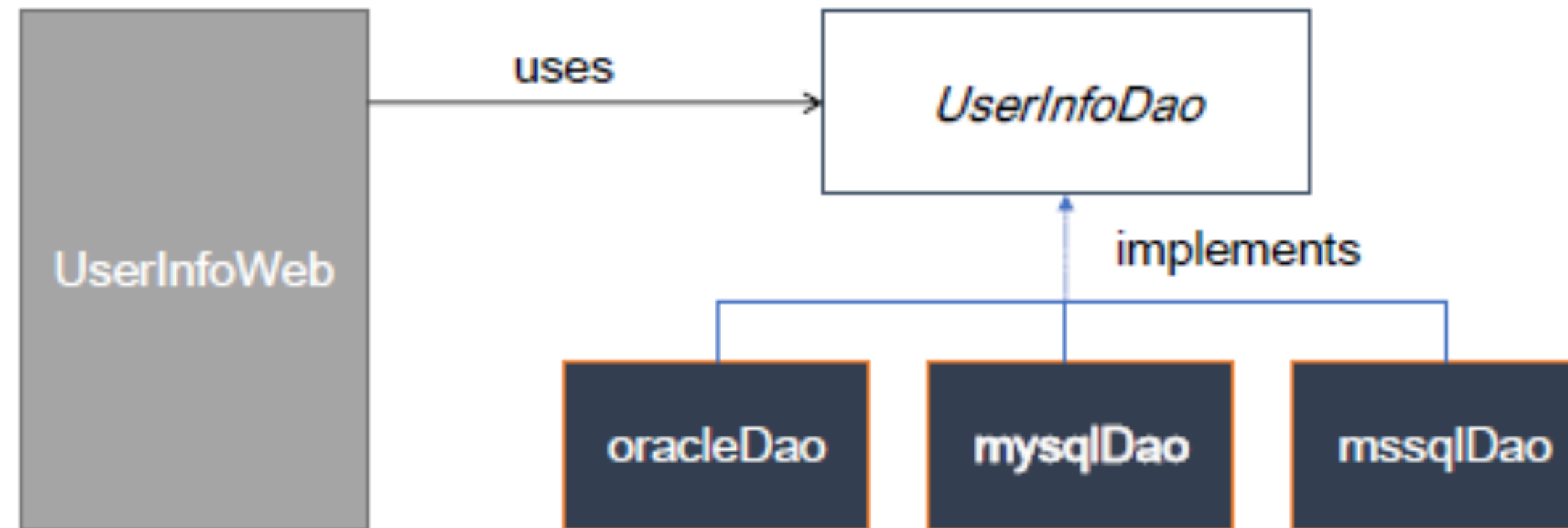
```

```

학생 Lee의 국어 과목 성적은 100점 이고, 학점은 A 입니다.
학생 Lee의 수학 과목 성적은 100점 이고, 학점은 S 입니다.
=====
학생 Kim의 국어 과목 성적은 55점 이고, 학점은 F 입니다.
학생 Kim의 수학 과목 성적은 55점 이고, 학점은 D 입니다.
학생 Kim의 영어 과목 성적은 100점 이고, 학점은 A 입니다.

```


6. DataBase 가 변경될 때 적용될 수 있는 코드 작성하기 Strategy Pattern



```
▲ [?] > ch13
  ▲ [?] > domain.userinfo
    ▲ [?] > dao
      ▲ [?] > mysql
        ▸ [?] UserInfoMySqlDao.java
      ▲ [?] > oracle
        ▸ [?] UserInfoOracleDao.java
        ▸ [?] UserInfoDao.java
        ▸ [?] UserInfo.java
    ▲ [?] > userinfo.web
      ▸ [?] UserInfoClient.java
```



```

public class UserInfo {

    private String userId;
    private String passwd;
    private String userName;

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public String getPasswd() {
        return passwd;
    }

    public void setPasswd(String passwd) {
        this.passwd = passwd;
    }

    public String getUserName() {
        return userName;
    }

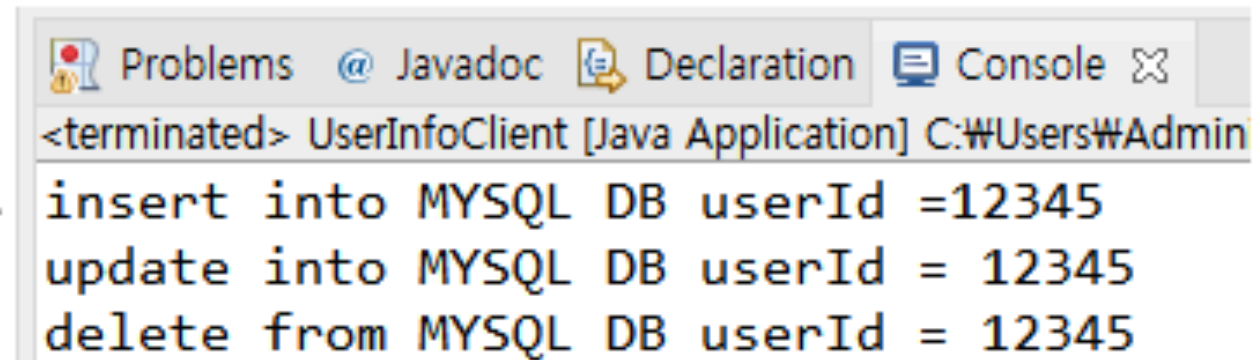
    public void setUserName(String userName) {
        this.userName = userName;
    }
}

```

db.properties 환경파일이 MYSQL 일때

DBTYPE=MYSQL

실행결과



```

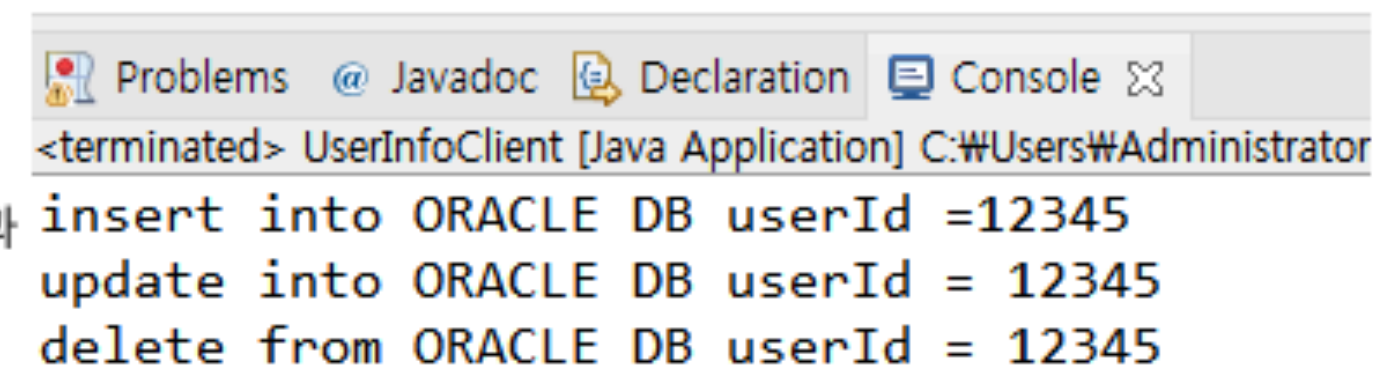
<terminated> UserInfoClient [Java Application] C:\Users\Admin
insert into MYSQL DB userId =12345
update into MYSQL DB userId = 12345
delete from MYSQL DB userId = 12345

```

db.properties 환경파일이 ORACLE 일때

DBTYPE=ORACLE

실행결과



```

<terminated> UserInfoClient [Java Application] C:\Users\Administrator
insert into ORACLE DB userId =12345
update into ORACLE DB userId = 12345
delete from ORACLE DB userId = 12345

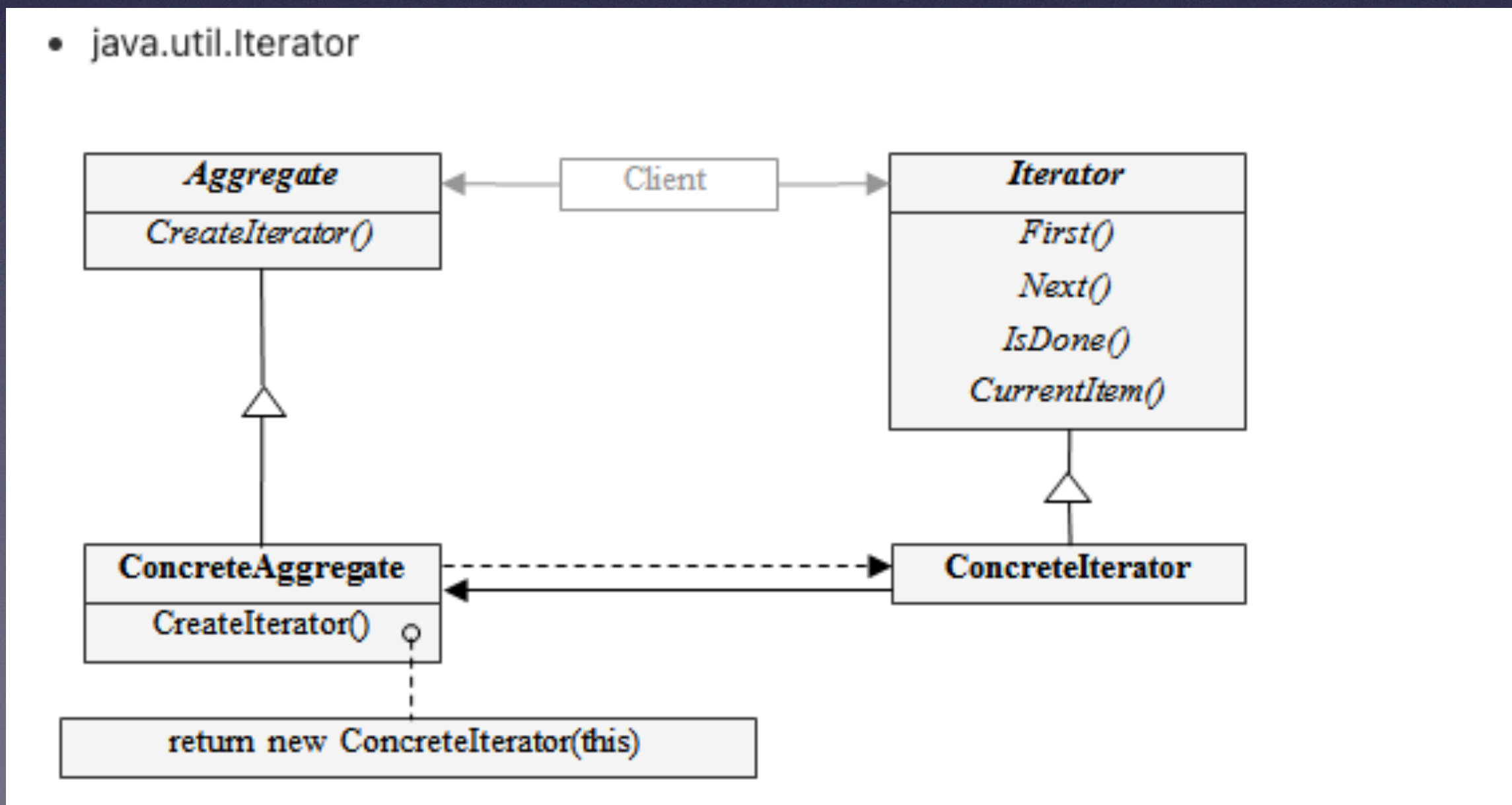
```


7. 모든 개체를 순회하는 방법 - Iterator 패턴

객체를 일괄적으로 관리하는 방법? 배열, 연결리스트, 등등. 집합체 - Aggregator

객체 내부 구현을 외부로 노출하지 않고 접근하게 하기위한 설계 - Iterator

단일 역할의 원칙 : 어떤 클래스가 하나 이상의 역할을 제공하게 되면 변경의 이유도 하나 이상이 된다. 클래스가 변경되는 이유는 하나 뿐이어야 한다. 집합체가 순회에 대한 구현까지 한다면 집합체의 기능과 순회의 기능을 모두 제공하는 것이다.




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

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

```
public class Book {  
  
    private String name;  
    public Book(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```


- Iterator 인터페이스를 구현한 BookShelfIterator 만들기

```
public static void main(String[] args) {
    BookShelf bookShelf = new BookShelf(4);
    bookShelf.appendBook(new Book("Around the World in 80 Days"));
    bookShelf.appendBook(new Book("Bible"));
    bookShelf.appendBook(new Book("Cinderella"));
    bookShelf.appendBook(new Book("Daddy-Long-Legs"));

    Iterator it = bookShelf.iterator();
    while (it.hasNext()) {
        Book book = (Book)it.next();
        System.out.println("" + book.getName());
    }
}
```

- 역순으로 순회하는 ReverseIterator 만들기

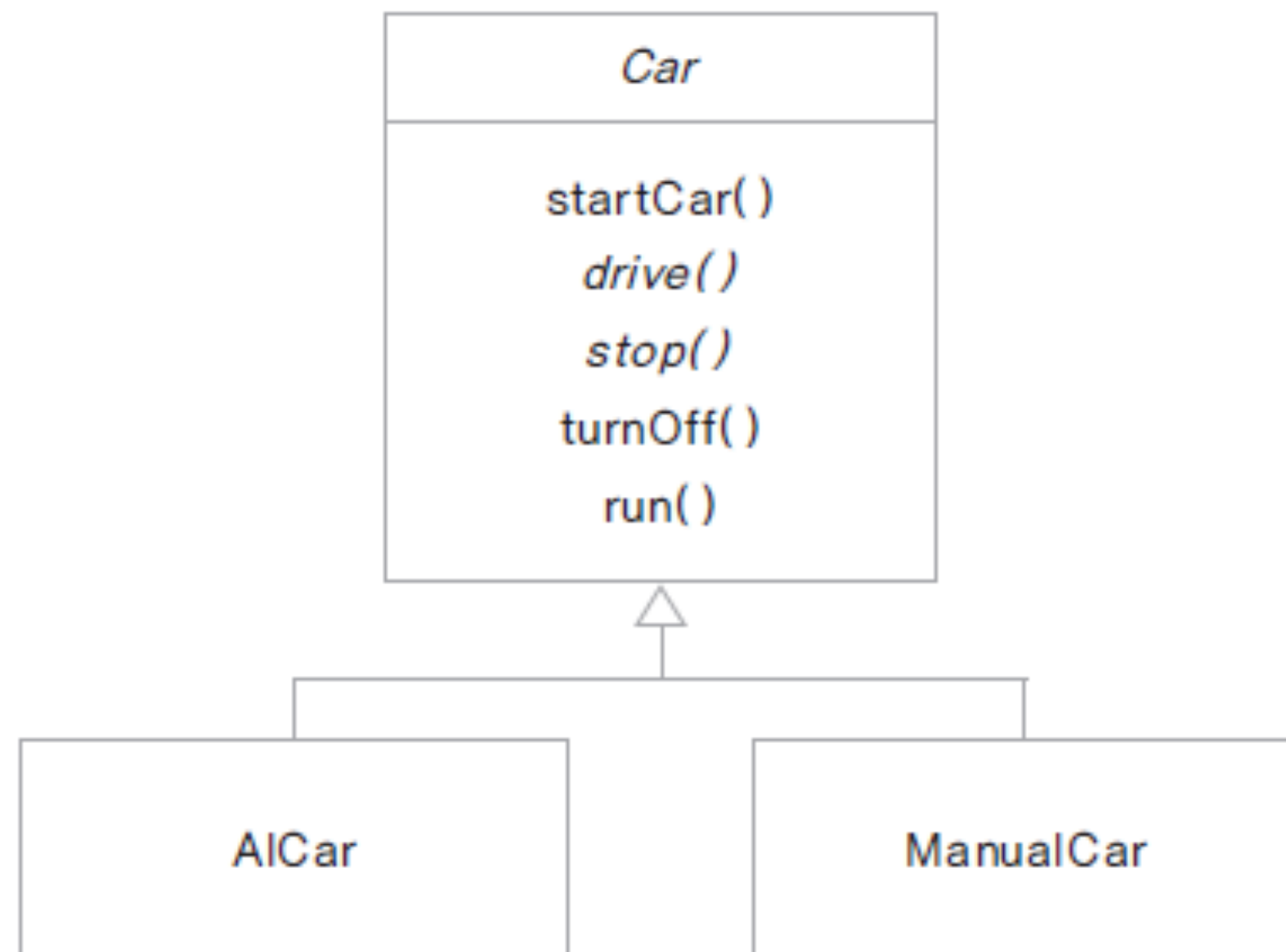
- 다양한 Iterator가 만들어 지는 Factory를 만들어 적용해보기

```
public abstract class Factory {

    public final Iterator create(Aggregate list, int type) {
        Iterator p = createProduct(list, type);
        return p;
    }

    protected abstract Iterator createProduct(Aggregate list, int type);
}
```


7. 추상 클래스와 Template Method



CarTest.java

```
public class CarTest {

    public static void main(String[] args) {

        Car aiCar = new AIcar();
        aiCar.run();
        System.out.println("=====");
        Car manualCar = new ManualCar();
        manualCar.run();

    }

}
```

Problems @ Javadoc Declaration Console

<terminated> CarTest [Java Application] C:\Users\Administrato

시동을 켭니다.
자율 주행합니다.
자동차가 스스로 방향을 바꿉니다.
스스로 멈춥니다.
시동을 끕니다.
=====
시동을 켭니다.
사람이 운전합니다.
사람이 핸들을 조작합니다.
브레이크를 밟아서 정지합니다.
시동을 끕니다.

#추상클래스 #추상메서드. #Hook메서드. #final키워드

8. 변화를 다른 객체들! 에게 알려주는 - Observer 패턴

느슨한 결합 (Loose Coupling) : 상호 작용하는 객체 사이에는 가능한 느슨한 결합이 중요

객체 사이에 일대 다의 의존 관계가 있고, 어떤 객체의 상태가 변하면 그 객체에 의존성을 가진 다른 객체들에게 변화를 통지 (notify or update) 하여 자동으로 갱신하게 됨

날씨(data)와 이것을 보여주는 여러 개의 그래프, 대시보드(view)

Log의 내용을 기록하는 Handler 가 여러 개인 경우

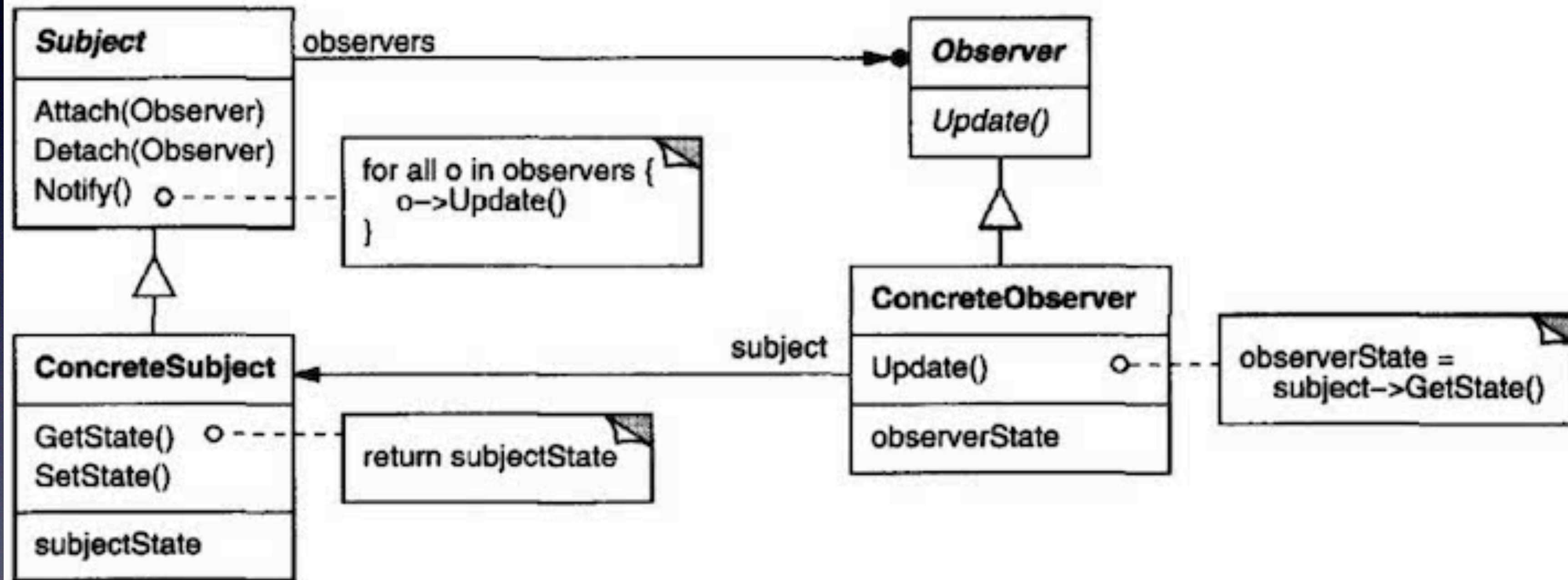
이때 중요한 것 변화에 관심을 가지고 그 정보를 업데이트 해야 하는 객체가 몇개냐와는 상관없이 일괄적으로 메시지가 전송되어야 함

변화 되는 객체 (Subject) - 이 데이터를 활용하는 객체들 (Observer) 의 관계는 느슨한 결합이어야 함

Pull or Push

MVC. Document- View

8. 변화를 다른 객체들! 에게 알려주는 - Observer 패턴




```
public abstract class NumberGenerator {
    private List<Observer> observers = new ArrayList<Observer>();
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    public void deleteObserver(Observer observer) {
        observers.remove(observer);
    }
    public void notifyObservers() {

        // ToDo

    }
    public abstract int getNumber();
    public abstract void execute();
}
```



```
public class RandomNumberGenerator extends NumberGenerator {
    private Random random = new Random();
    private int number;
    public int getNumber() {
        return number;
    }
    public void execute() {
        //ToDo
    }
}

public interface Observer {
    public abstract void update(NumberGenerator generator);
}
```

옵저버들을 만들고 Main이 수행되도록 해보세요


```
public class Main {  
    public static void main(String[] args) {  
        NumberGenerator generator = new RandomNumberGenerator();  
        Observer observer1 = new DigitObserver();  
        Observer observer2 = new GraphObserver();  
  
        // ToDo  
    }  
}
```

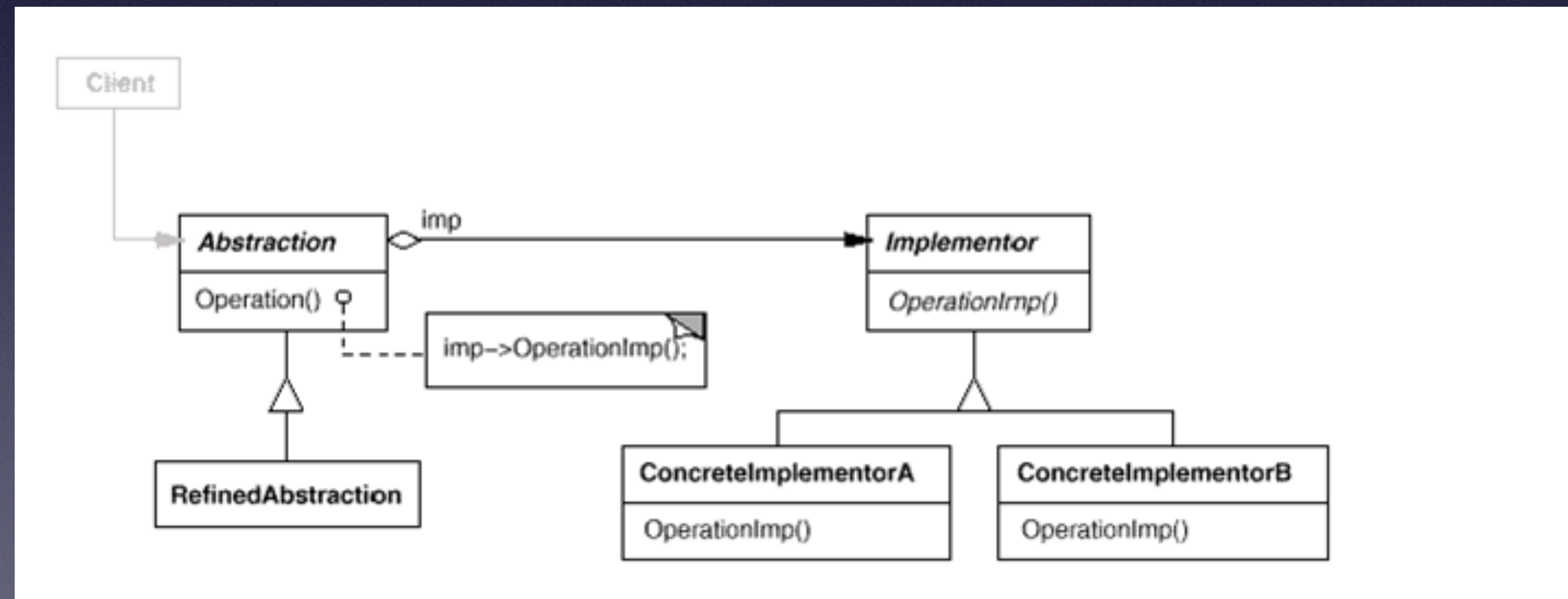

9. 기능의 확장과 구현의 확장을 분리 - Bridge

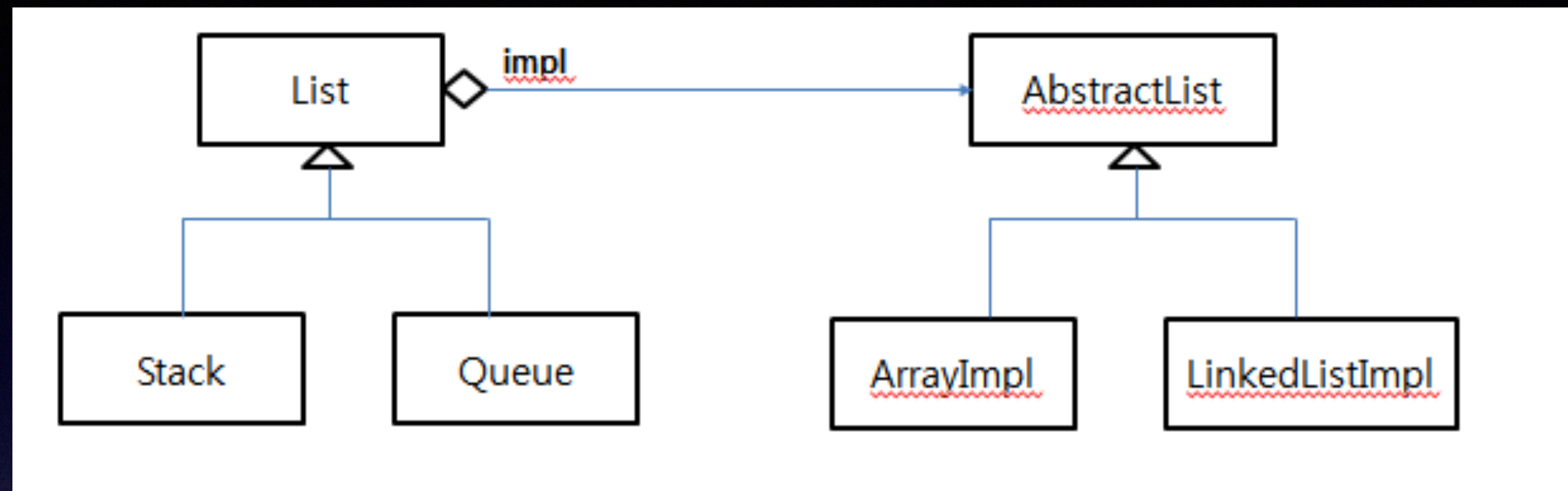
기능(추상적인 개념)의 확장과 구현의 확장을 분리한다

하나의 기능에 대한 다양한 구현이 적용될 수 있음 - 구현부에 대한 참조를 가진다

기능과 구현의 확장이 혼재 하면 상속관계가 복잡해 짐

실행 중에 구현을 선택할 수도 있고, 사용하는 코드(클라이언트)에서는 기능 인터페이스를 사용하므로 구현에 대한 부분을 숨길 수 있음





List는 선형 자료의 추상적인 개념 List 하위에 Stack과 Queue라는 개념이 있음

List의 구현은 배열, 연결리스트 모두 가능


```
public class List<T>{

    AbstractList<T> impl;

    public List(AbstractList<T> list) {
        impl = list;
    }

    public void add(T obj) {
        impl.addElement(obj);
    }
    public T get(int i) {
        return impl.getElement(i);
    }
    public T remove(int i) {
        return impl.deleteElement(i);
    }
    public int getSize() {
        return impl.getElementSize();
    }

}
```



```
import impl.AbstractList;

public class Queue<T> extends List<T> {

    public Queue(AbstractList<T> list) {
        super(list);
        System.out.println("Queue를 구현합니다.");
    }

    public void enqueue(T obj) {
        impl.addElement(obj);
    }

    public T dequeue() {
        return impl.deleteElement(0);
    }

}
```



```
public interface AbstractList<T> {

    public void addElement(T obj);
    public T deleteElement(int i);
    public int insertElement(T obj, int i);
    public T getElement(int i);
    public int getElementSize();
}
```

```
public class BridgeTest {

    public static void main(String[] args) {

        Queue<String> arrayQueue = new Queue<String>(new ArrayImpl<String>());

        arrayQueue.enqueue("aaa");
        arrayQueue.enqueue("bbb");
        arrayQueue.enqueue("ccc");

        System.out.println(arrayQueue.dequeue());
        System.out.println(arrayQueue.dequeue());
        System.out.println(arrayQueue.dequeue());
        System.out.println("=====");

        Queue<String> linkedQueue = new Queue<String>(new LinkedListImpl<String>());
        linkedQueue.enqueue("aaa");
        linkedQueue.enqueue("bbb");
        linkedQueue.enqueue("ccc");

        System.out.println(linkedQueue.dequeue());
        System.out.println(linkedQueue.dequeue());
        System.out.println(linkedQueue.dequeue());
        System.out.println("=====");

        Stack<String> arrayStack = new Stack<String>(new ArrayImpl<String>());
        arrayStack.push("aaa");
        arrayStack.push("bbb");
        arrayStack.push("ccc");

        System.out.println(arrayStack.pop());
        System.out.println(arrayStack.pop());
        System.out.println(arrayStack.pop());
        System.out.println("=====");

        Stack<String> linkedStack = new Stack<String>(new LinkedListImpl<String>());
        linkedStack.push("aaa");
        linkedStack.push("bbb");
        linkedStack.push("ccc");

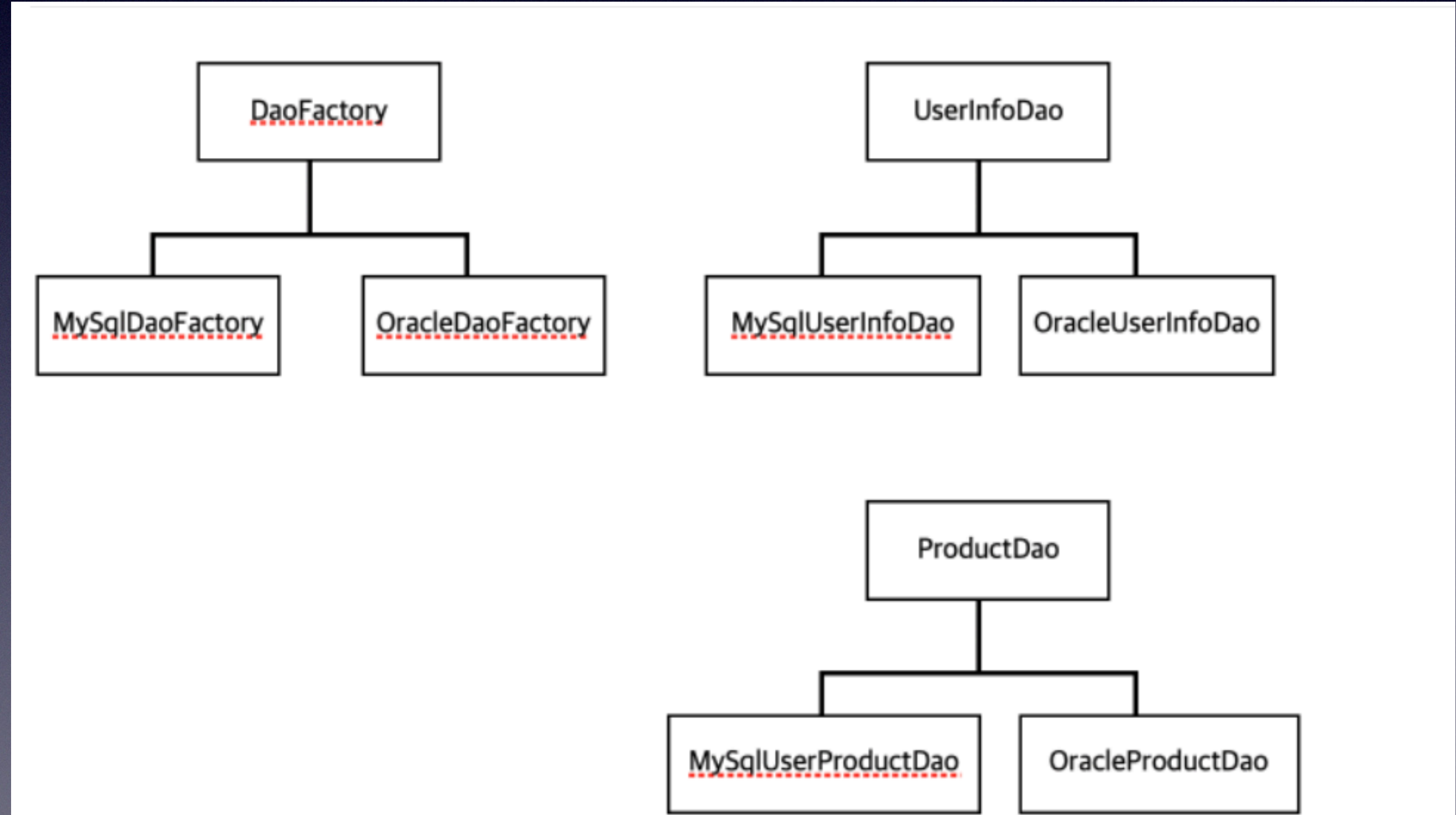
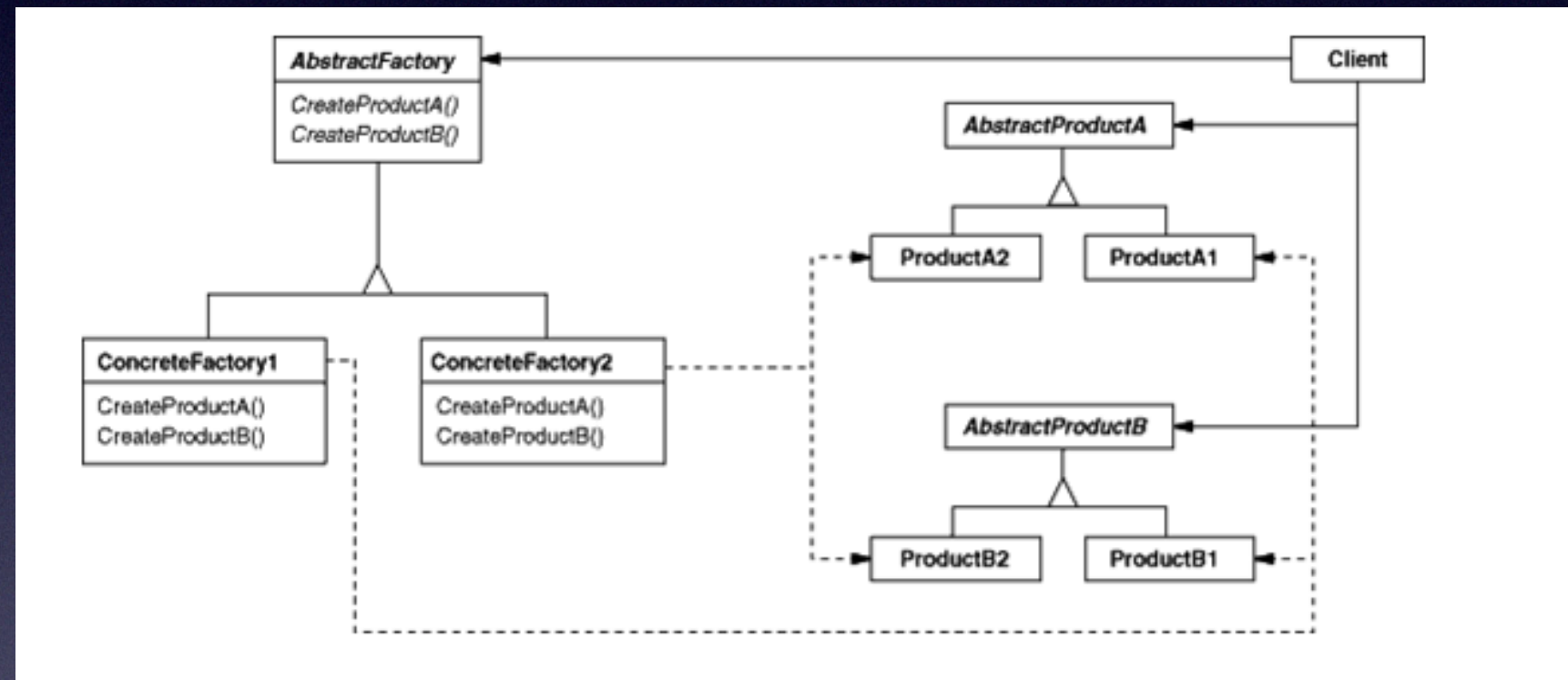
        System.out.println(linkedStack.pop());
        System.out.println(linkedStack.pop());
        System.out.println(linkedStack.pop());
        System.out.println("=====");

    }

}
```


10. 서로 연관있는 인스턴스를 한꺼번에 생성-Abstract Factory

서로 연관성 있는 여러 인스턴스를 한꺼번에 생성하는 패턴
추상화된 팩토리가 제공되고 그 상황에 맞는 인스턴스 세트가 생성되도록 한다.
예) DAO , Widget




```
public abstract class DaoFactory {  
  
    public abstract UserDao createUserInfoDao();  
  
    public abstract ProductDao createProductDao();  
  
}
```

```
public class MySqlDaoFactory extends DaoFactory{  
  
    @Override  
    public UserDao createUserInfoDao() {  
        return new UserInfoMySqlDao();  
    }  
  
    @Override  
    public ProductDao createProductDao() {  
        return new ProductMySqlDao();  
    }  
}
```



```
public class UserInfoClient {

    public static void main(String[] args) throws IOException {

        FileInputStream fis = new FileInputStream("db.properties");

        Properties prop = new Properties();
        prop.load(fis);

        String dbType = prop.getProperty("DBTYPE");

        UserInfo userInfo = new UserInfo();
        userInfo.setUserId("12345");
        userInfo.setPasswd("!@#$%");
        userInfo.setUserName("Tomas");

        Product product = new Product();
        product.setProductId("0011AA");
        product.setProductName("TV");

        DaoFactory daoFactory = null;
        UserInfoDao userInfoDao = null;
        ProductDao productDao = null;
```



```
        if(dbType.equals("ORACLE")){

            daoFactory = new OracleDaoFactory();

        }

        else if(dbType.endsWith("MYSQL")){

            daoFactory = new MySqlDaoFactory();

        }

        else{

            System.out.println("db support error");

            return;

        }

        userInfoDao = daoFactory.createUserInfoDao();

        productDao = daoFactory.createProductDao();

        System.out.println("==USERINFO TRANSACTION==");

        userInfoDao.insertUserInfo(userInfo);

        userInfoDao.updateUserInfo(userInfo);

        userInfoDao.deleteUserInf(userInfo);

        System.out.println("==PRODUCT TRANSACTION==");

        productDao.insertProduct(product);

        productDao.updateProduct(product);

        productDao.deleteProduct(product);

    }

}
```


12. 복합 객체와 단일 객체를 동일하게 처리하는 패턴 - Composite

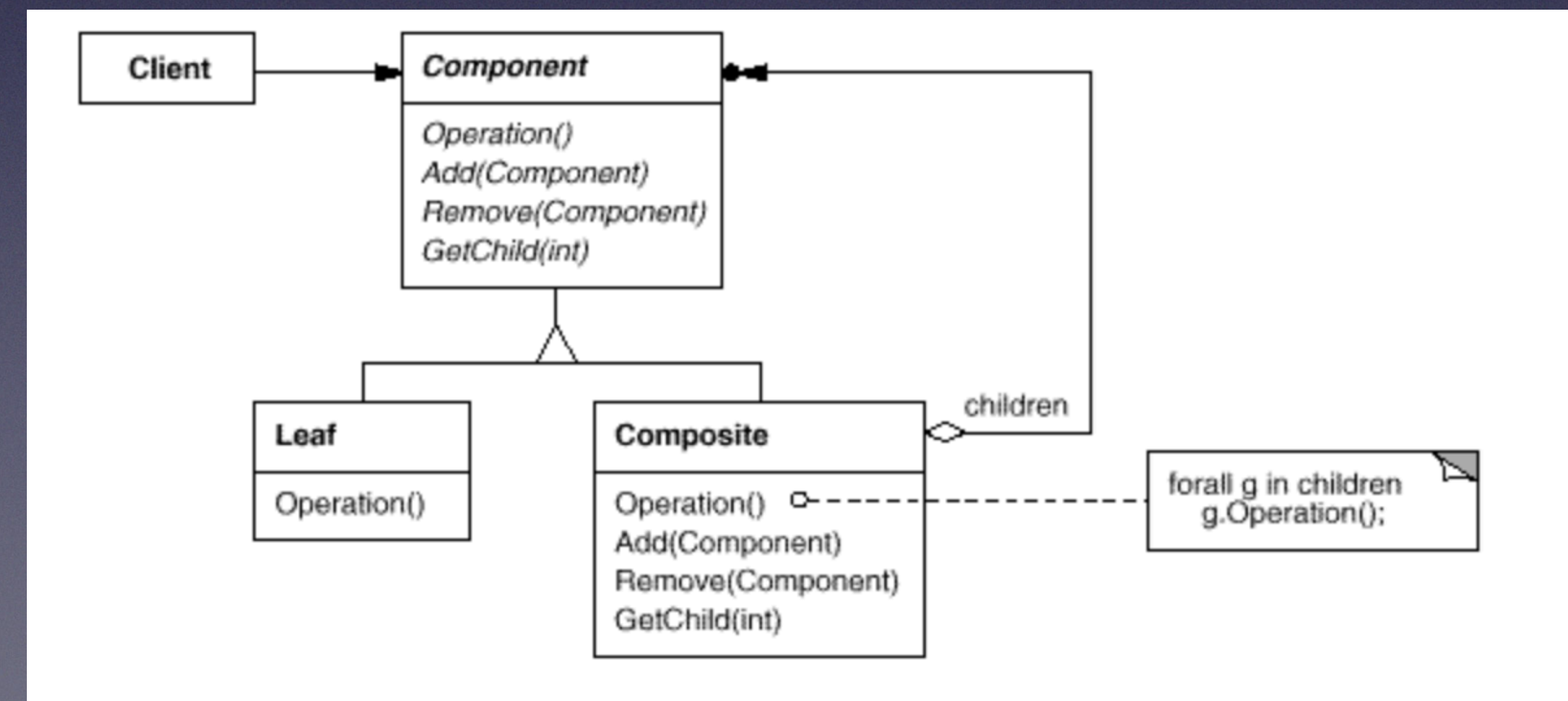
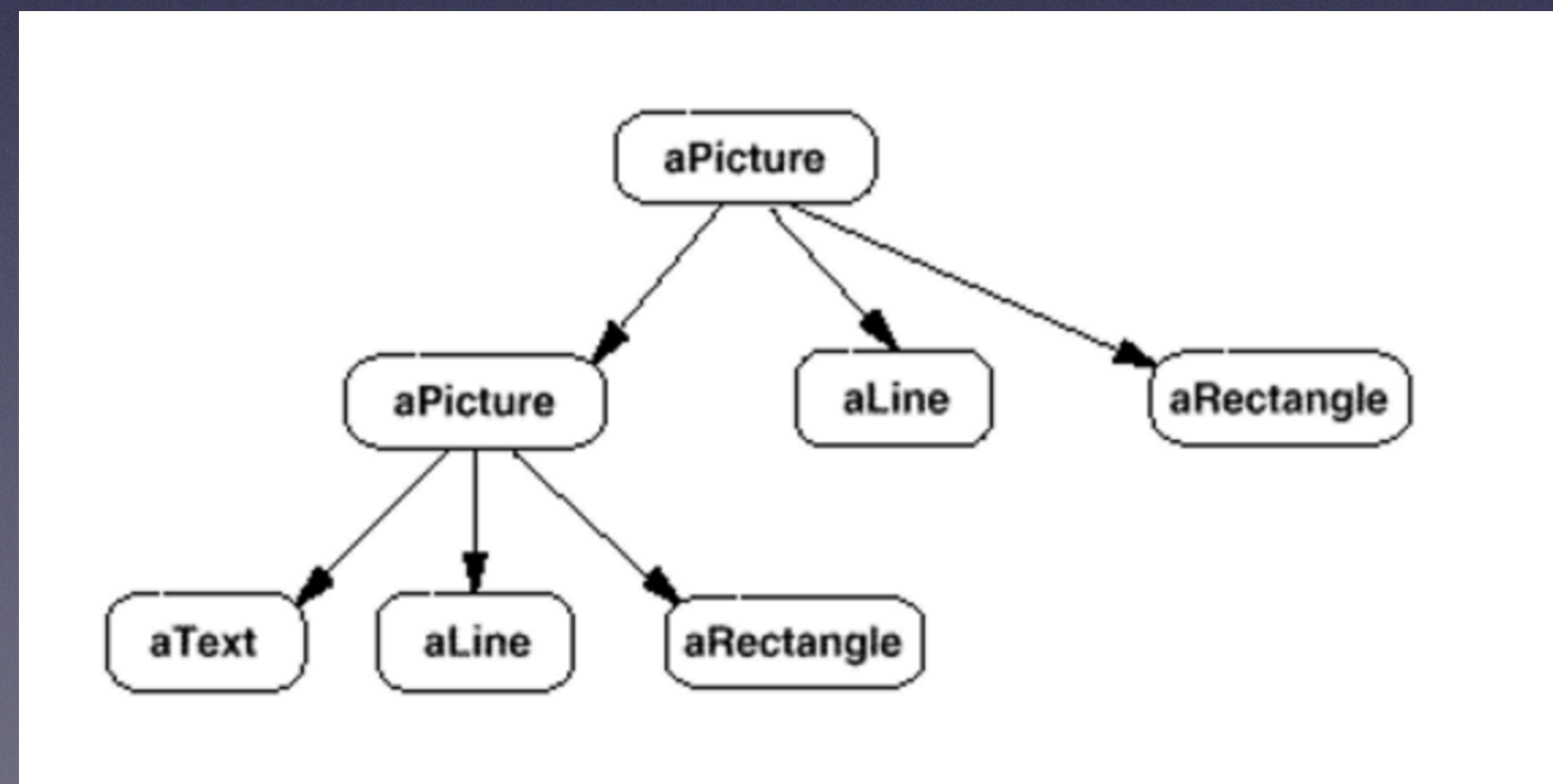
트리 구조일 때 많이 사용 됨

재귀적인 구조

단일 객체는 복합 객체에 포함되고, 복합 객체 역시 또다른 복합 객체에 포함이 되는 구조

복합 객체와 단일 객체를 구분하여 (if, instanceof 등을 사용) 구현하지 않는 편의성

객체의 단일성 vs. 클라이언트의 편의성




```
public abstract class ProductCategory {
    int id;
    String name;
    int price;

    public ProductCategory(int id, String name, int price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public abstract void addProduct(ProductCategory product);
    public abstract void removeProduct(ProductCategory product);
    public abstract int getCount();
    public abstract String getName();
    public abstract int getPrice();
    public abstract int getId();

}
```



```
public class CategoryClient {

    public static void main(String[] args) {
        ProductCategory womanCategory = new Category(1234, "Woman", 0);
        ProductCategory manCategory = new Category(5678, "Man", 0);

        ProductCategory clothesCategoryW = new Category(2345, "Clothes", 0);
        ProductCategory bagCategoryW = new Category(3456, "Bag", 0);
        ProductCategory shoesCategoryW = new Category(9876, "Shoes", 0);

        womanCategory.addProduct(clothesCategoryW);
        womanCategory.addProduct(bagCategoryW);
        womanCategory.addProduct(shoesCategoryW);

        ProductCategory clothesCategoryM = new Category(23450, "Clothes", 0);
        ProductCategory bagCategoryM = new Category(34560, "Bag", 0);
        ProductCategory shoesCategoryM = new Category(98760, "Shoes", 0);

        manCategory.addProduct(clothesCategoryM);
        manCategory.addProduct(bagCategoryM);
        manCategory.addProduct(shoesCategoryM);

        ProductCategory shoes1 = new Product(121, "Nike", 100000);
        ProductCategory shoes2 = new Product(122, "ADIDAS", 200000);
        ProductCategory shoes3 = new Product(123, "GUCCI", 300000);
        ProductCategory shoes4 = new Product(124, "BALENCIA", 400000);
        ProductCategory shoes5 = new Product(125, "PRADA", 500000);
        ProductCategory shoes6 = new Product(126, "BALLY", 600000);
    }
}
```



```
shoesCategoryW.addProduct(shoes1);
shoesCategoryW.addProduct(shoes2);
shoesCategoryW.addProduct(shoes3);
```

```
shoesCategoryM.addProduct(shoes4);
shoesCategoryM.addProduct(shoes5);
shoesCategoryM.addProduct(shoes6);
```

```
ProductCategory bag1 = new Product(121, "HERMES", 500000);
ProductCategory bag2 = new Product(122, "LOUISVUITTON", 500000);
ProductCategory bag3 = new Product(123, "GUCCI", 500000);
ProductCategory bag4 = new Product(124, "BALENCIA", 500000);
ProductCategory bag5 = new Product(125, "PRADA", 500000);
ProductCategory bag6 = new Product(126, "MULBERRY", 500000);
```

```
bagCategoryW.addProduct(bag1);
bagCategoryW.addProduct(bag2);
bagCategoryW.addProduct(bag3);
```

```
bagCategoryM.addProduct(bag4);
bagCategoryM.addProduct(bag5);
bagCategoryM.addProduct(bag6);
```

```
System.out.println(womanCategory.getCount());
System.out.println(womanCategory.getPrice());
System.out.println(manCategory.getCount());
System.out.println(manCategory.getPrice());
```

```
}
```

```
}
```


13. 명령을 캡슐화 하기- Command

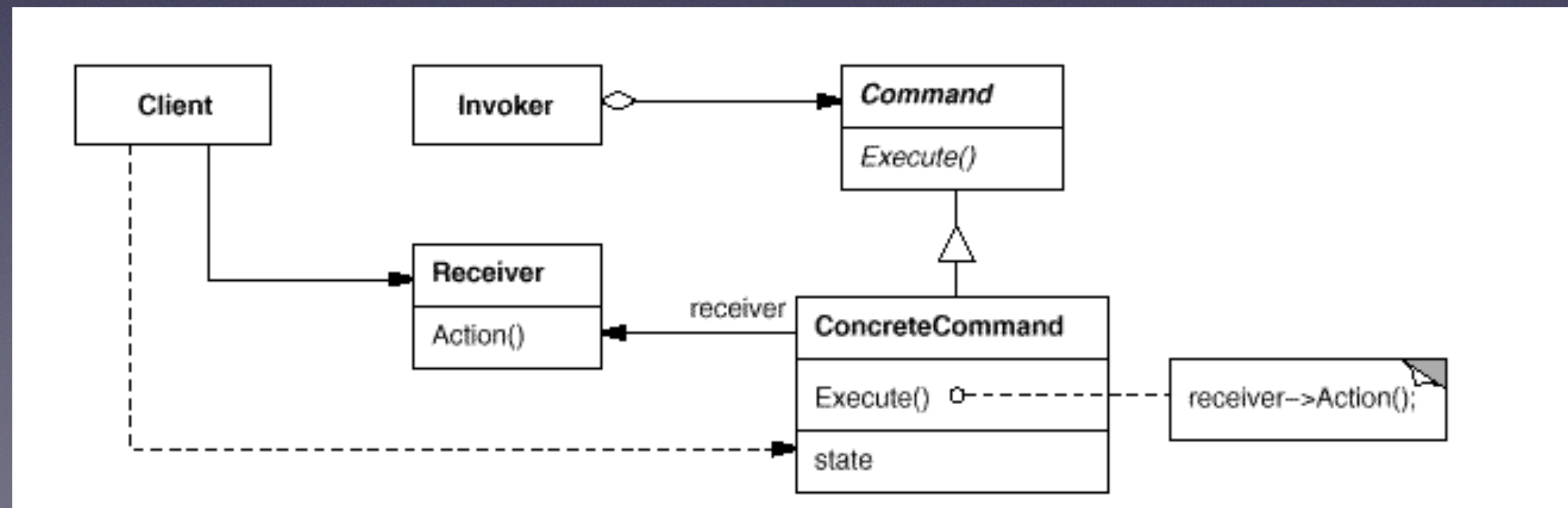
다양하게 처리되어야 하는 메뉴, 요청이 있을 때 이를 동일한 방식으로 처리될 수 있도록 함
명령을 보내는 쪽과 명령을 처리하는 쪽을 분리한다 (느슨한 구조)

여러 다른 명령 기능이 추가 되더라도 처리하는 방식이 동일하다.

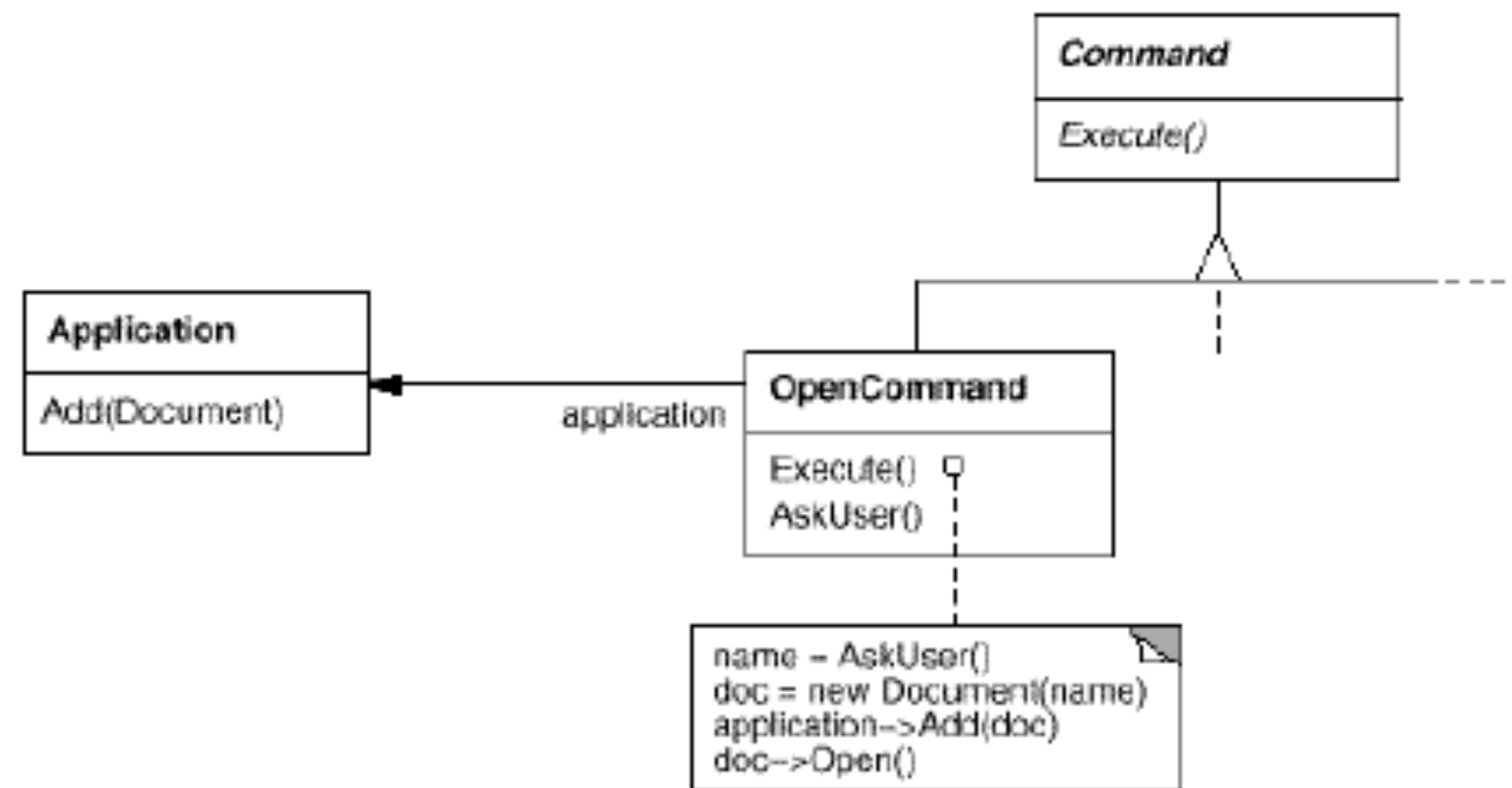
Command 내부에서 요청을 처리할 대상 (receiver)에 대한 정보를 가지고 있어야 한다.

처리된 명령에 대한 히스토리가 유지되면 롤백할 수 있다.

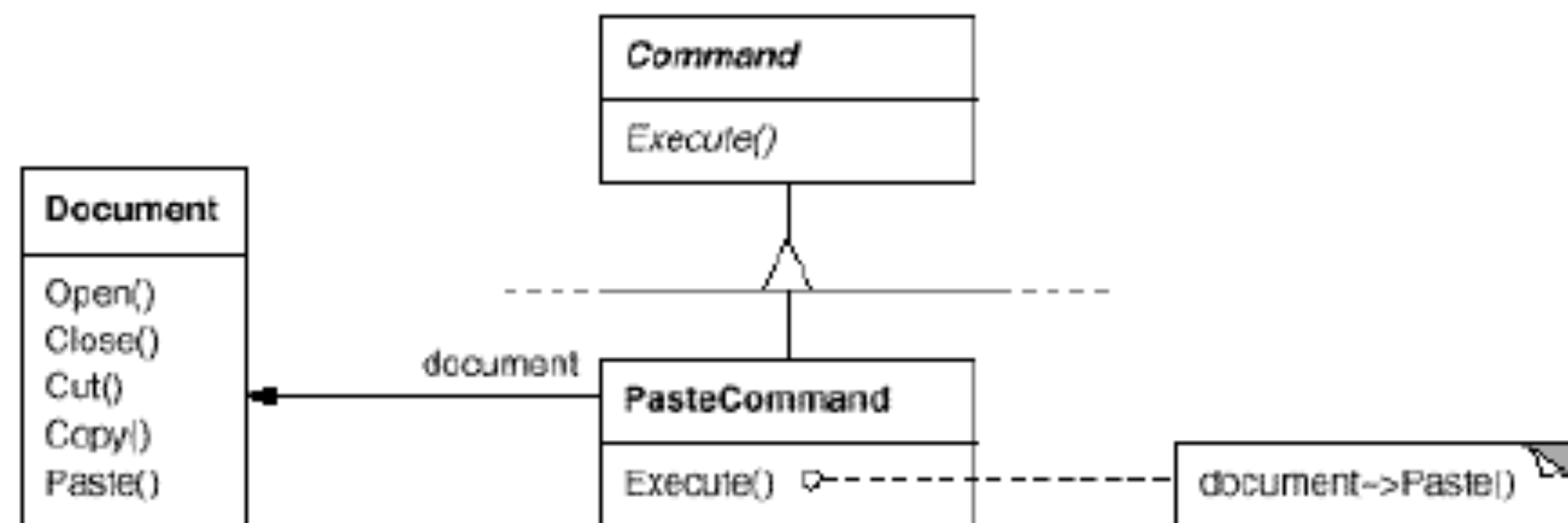
여러개의 요청을 한꺼번에 일괄적으로 처리 할 수도 있다. (macro command)



- OpenCommand



- PasteCommand



리모컨의 버튼들은 (invoker) 각각 다른 대상(receiver)에 대한 수행을 요구한다고 하면 이를 어떻게 처리할 것인가?

0번 버튼 눌리면 (ON) : 불을 켜다

0번 버튼 올라오면 (OFF) : 불을 끈다

1번 버튼 눌리면 (ON) : 에어컨이나 히터를 켜다

1번 버튼 올라오면 (OFF) : 에어컨이나 히터를 끈다

2번 버튼 눌리면 (ON) : 음악을 켜다

2번 버튼 올라오면 (OFF) : 음악을 끈다

3번 버튼 눌리면 (ON) : 문을 연다

3번 버튼 올라오면 (OFF) : 문들 닫는다

```
public interface Command {  
    public void execute();  
}
```



```
public class Light {
    String location = "";

    public Light(String location) {
        this.location = location;
    }

    public void on() {
        System.out.println(location + " light is on");
    }

    public void off() {
        System.out.println(location + " light is off");
    }
}

public class GarageDoor {
    String location;

    public GarageDoor(String location) {
        this.location = location;
    }

    public void up() {
        System.out.println(location + " garage Door is Up");
    }

    public void down() {
        System.out.println(location + " garage Door is Down");
    }

    public void stop() {
        System.out.println(location + " garage Door is Stopped");
    }

    public void lightOn() {
        System.out.println(location + " garage light is on");
    }

    public void lightOff() {
        System.out.println(location + " garage light is off");
    }
}
```



```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```



```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Light livingRoomLight = new Light("Living Room");
        CeilingFan ceilingFan= new CeilingFan("Living Room");
        GarageDoor garageDoor = new GarageDoor("Garage");
        Stereo stereo = new Stereo("Living Room");

        LightOnCommand livingRoomLightOn =
            new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
            new LightOffCommand(livingRoomLight);

        CeilingFanOnCommand ceilingFanOn =
            new CeilingFanOnCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
            new CeilingFanOffCommand(ceilingFan);

        GarageDoorUpCommand garageDoorUp =
            new GarageDoorUpCommand(garageDoor);
        GarageDoorDownCommand garageDoorDown =
            new GarageDoorDownCommand(garageDoor);

        StereoOnWithCDCommand stereoOnWithCD =
            new StereoOnWithCDCommand(stereo);
        StereoOffCommand stereoOff =
            new StereoOffCommand(stereo);
    }
}
```



```
remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);  
remoteControl.setCommand(1, ceilingFanOn, ceilingFanOff);  
remoteControl.setCommand(2, stereoOnWithCD, stereoOff);  
remoteControl.setCommand(3, garageDoorUp, garageDoorDown);
```

```
System.out.println(remoteControl);
```

```
remoteControl.onButtonWasPushed(0);  
remoteControl.offButtonWasPushed(0);  
remoteControl.onButtonWasPushed(1);  
remoteControl.offButtonWasPushed(1);  
remoteControl.onButtonWasPushed(2);  
remoteControl.offButtonWasPushed(2);  
remoteControl.onButtonWasPushed(3);  
remoteControl.offButtonWasPushed(3);  
remoteControl.onButtonWasPushed(4);  
remoteControl.offButtonWasPushed(4);
```

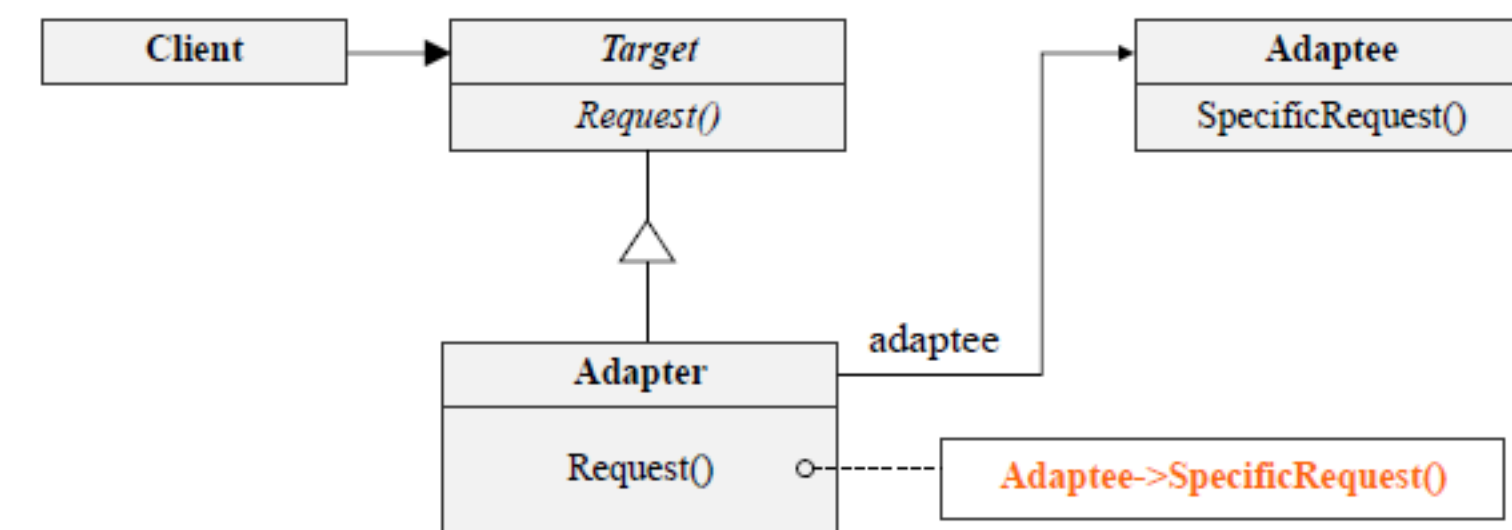
```
}
```

```
}
```

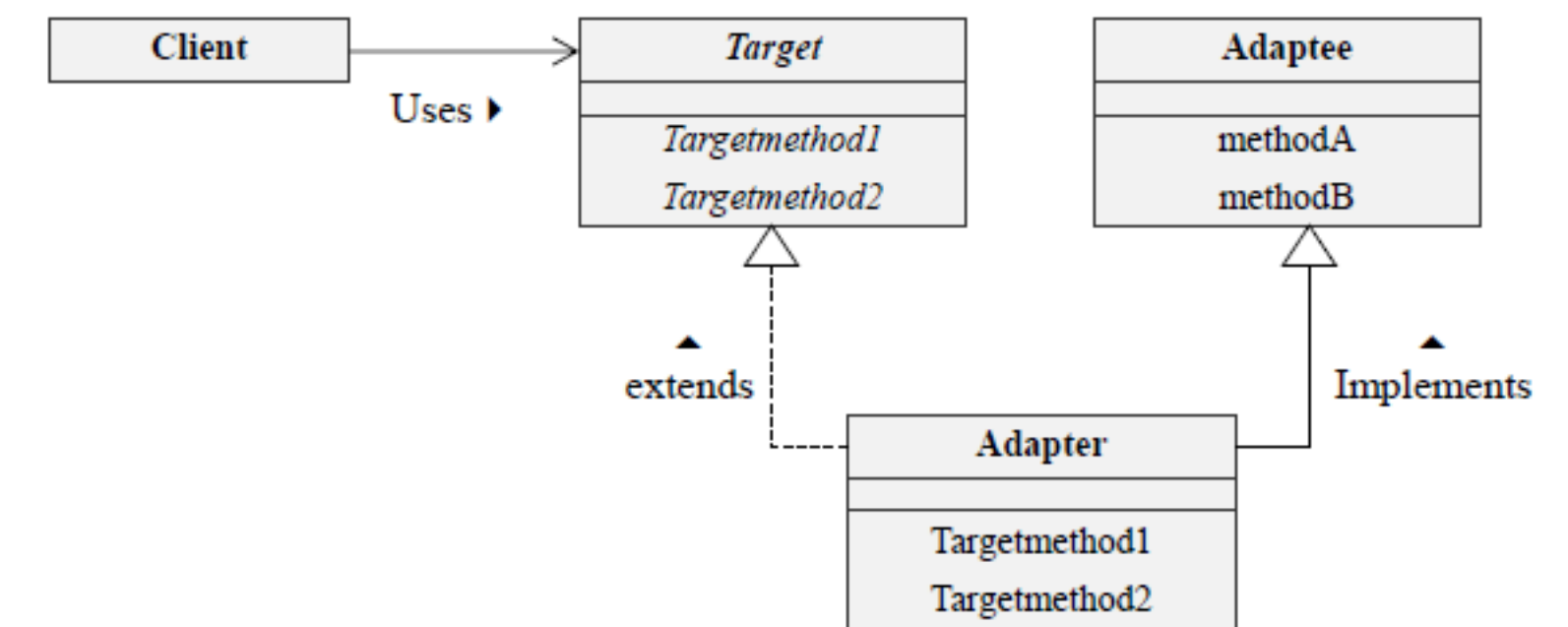

14. 호환 하기 - Adapter

서로 다른 인터페이스를 연결함
클라이언트가 사용하던 방식대로 조정해 주는 객체
안드로이드에서 View를 위한 Adapter들...

- 합성 (composition) 으로 구현하기



- 상속 (inheritance) 로 구현하기




```
public interface Print {
    public abstract void printWeak();
    public abstract void printStrong();
}

public class Banner {
    private String string;
    public Banner(String string) {
        this.string = string;
    }
    public void showWithParen() {
        System.out.println("(" + string + ")");
    }
    public void showWithAster() {
        System.out.println("*" + string + "*");
    }
}

public class Main {
    public static void main(String[] args) {
        Print p = new PrintBanner("Hello");
        p.printWeak();
        p.printStrong();
    }
}
```

PrintBanner를 상속과 합성의 방법으로 구현해보세요

15. 객체를 방문하여 기능을 처리함- visitor

클래스에 다양한 기능이 추가 되거나 삭제되는 일이 자주 일어나면 클래스를 자주 변경해야 하는 번거로움이 있다.

객체가 가지는 기능들을 클래스로 분리한다.

기능을 구현한 클래스를 Visitor라고 하고 각 객체는 이 Visitor를 accept 함으로써 기능이 수행된다.

모든 클래스에 적용하기 위한 기능을 추가할 때는 각 클래스를 수정하지 않고 Visitor를 추가하면 되지만,
클래스의 캡슐화에 위배되는 구현 방식이므로 클래스의 캡슐화가 크게 중요하지 않는 경우에 적용해야 하는 패턴임
따라서 클래스 구조의 변화는 거의 없고 기능의 추가 삭제가 자주 발생할때 사용하는것이 좋음

복합 객체인 경우 단일 객체와 동일한 기능이 구현될 때 이를 Visitor로 분리하고 Visitor가 각 객체를 순회하는 방식으로 사용할 수 있음

Visitor.java

```
public abstract class Visitor {  
    public abstract void visit(File file);  
    public abstract void visit(Directory directory);  
}
```

Acceptor.java

```
public interface Acceptor {  
    public abstract void accept(Visitor v);  
}
```

Entry.java

```
public abstract class Entry implements Acceptor {  
    public abstract String getName();                // 이름을 얻는다.  
    public abstract int getSize();                   // 사이즈를 얻는다.  
    public Entry add(Entry entry) throws FileTreatmentException { // 엔트리를 추가  
        throw new FileTreatmentException();  
    }  
    public Iterator iterator() throws FileTreatmentException {    // Iterator의 생성  
        throw new FileTreatmentException();  
    }  
    public String toString() {                                // 문자열 표현  
        return getName() + " (" + getSize() + ")";  
    }  
}
```



```

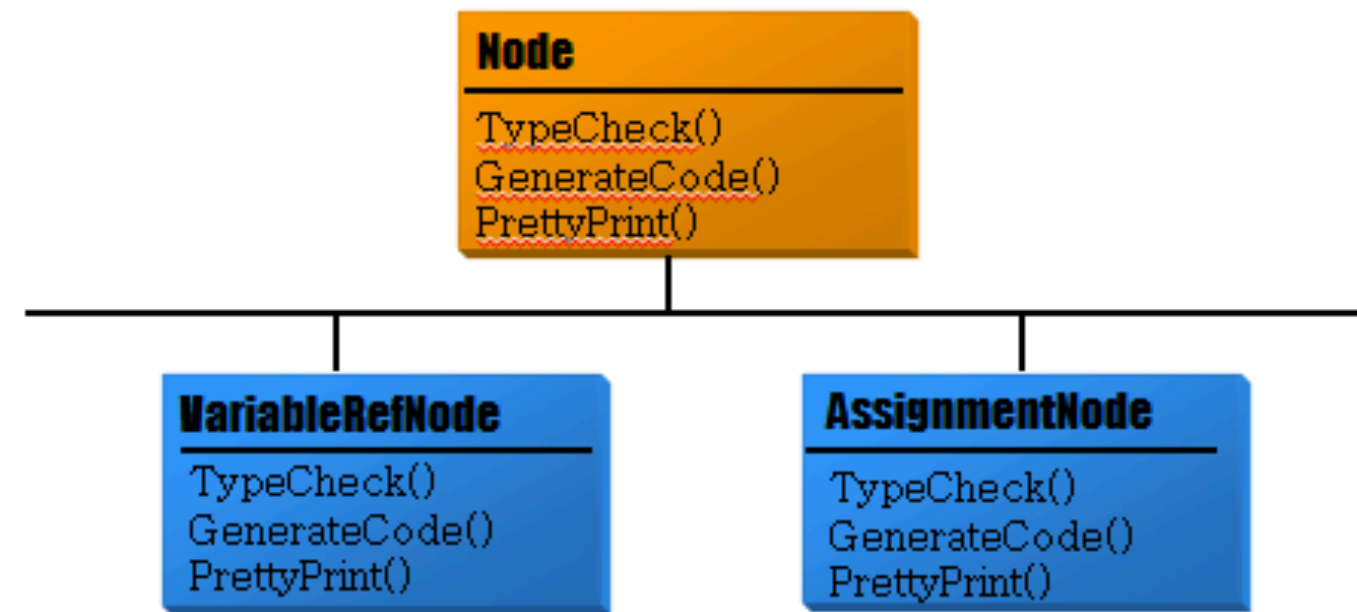
public class Main {
    public static void main(String[] args) {
        try {
            System.out.println("Making root entries...");
            Directory rootdir = new Directory("root");
            Directory bindir = new Directory("bin");
            Directory tmpdir = new Directory("tmp");
            Directory usrdir = new Directory("usr");
            rootdir.add(bindir);
            rootdir.add(tmpdir);
            rootdir.add(usrdir);
            bindir.add(new File("vi", 10000));
            bindir.add(new File("latex", 20000));
            rootdir.accept(new ListVisitor());

            System.out.println("");
            System.out.println("Making user entries...");
            Directory Kim = new Directory("Kim");
            Directory Lee = new Directory("Lee");
            Directory Kang = new Directory("Kang");
            usrdir.add(Kim);
            usrdir.add(Lee);
            usrdir.add(Kang);
            Kim.add(new File("diary.html", 100));
            Kim.add(new File("Composite.java", 200));
            Lee.add(new File("memo.tex", 300));
            Kang.add(new File("game.doc", 400));
            Kang.add(new File("junk.mail", 500));
            rootdir.accept(new ListVisitor());
        } catch (FileTreatmentException e) {
            e.printStackTrace();
        }
    }
}

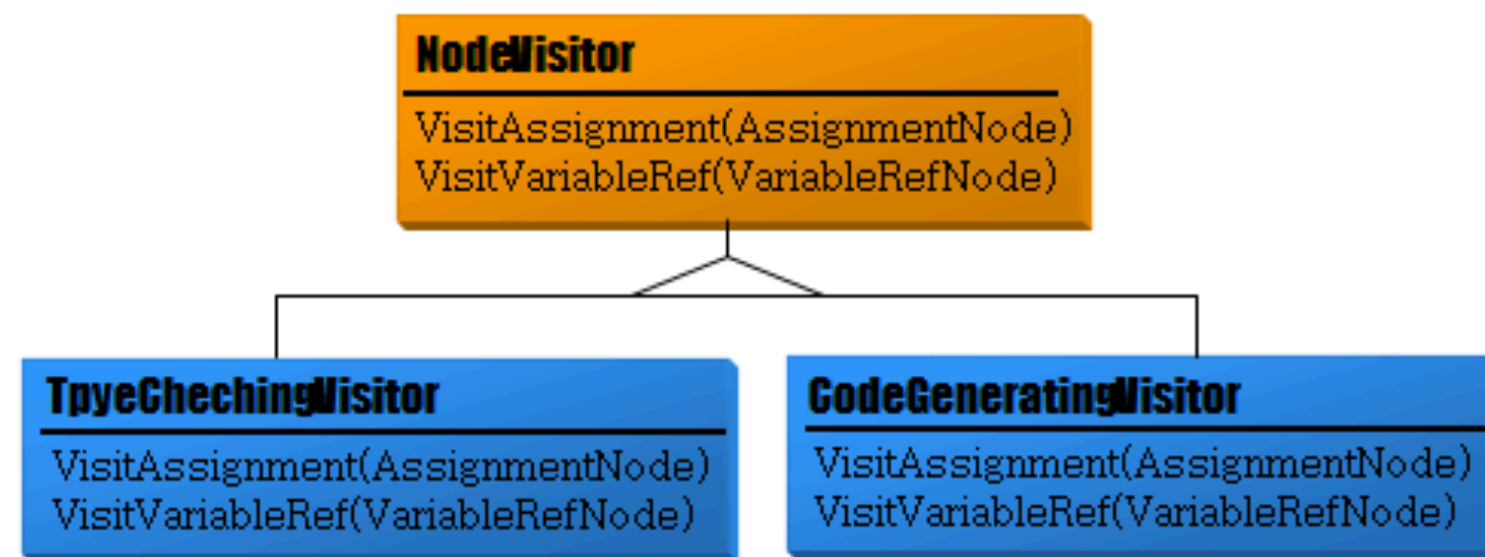
```

Entry를 상속받은 File과 Directory를 구현하고, ListVisitor를 만들어서 main()을 수행하세요

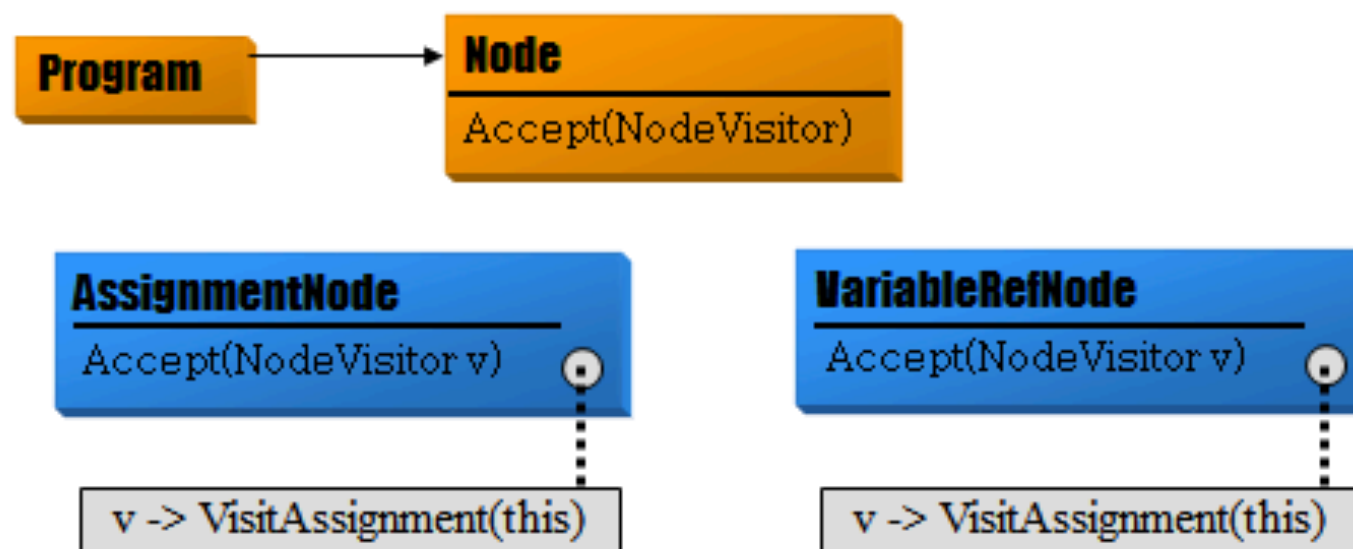
추상 구문 트리 예



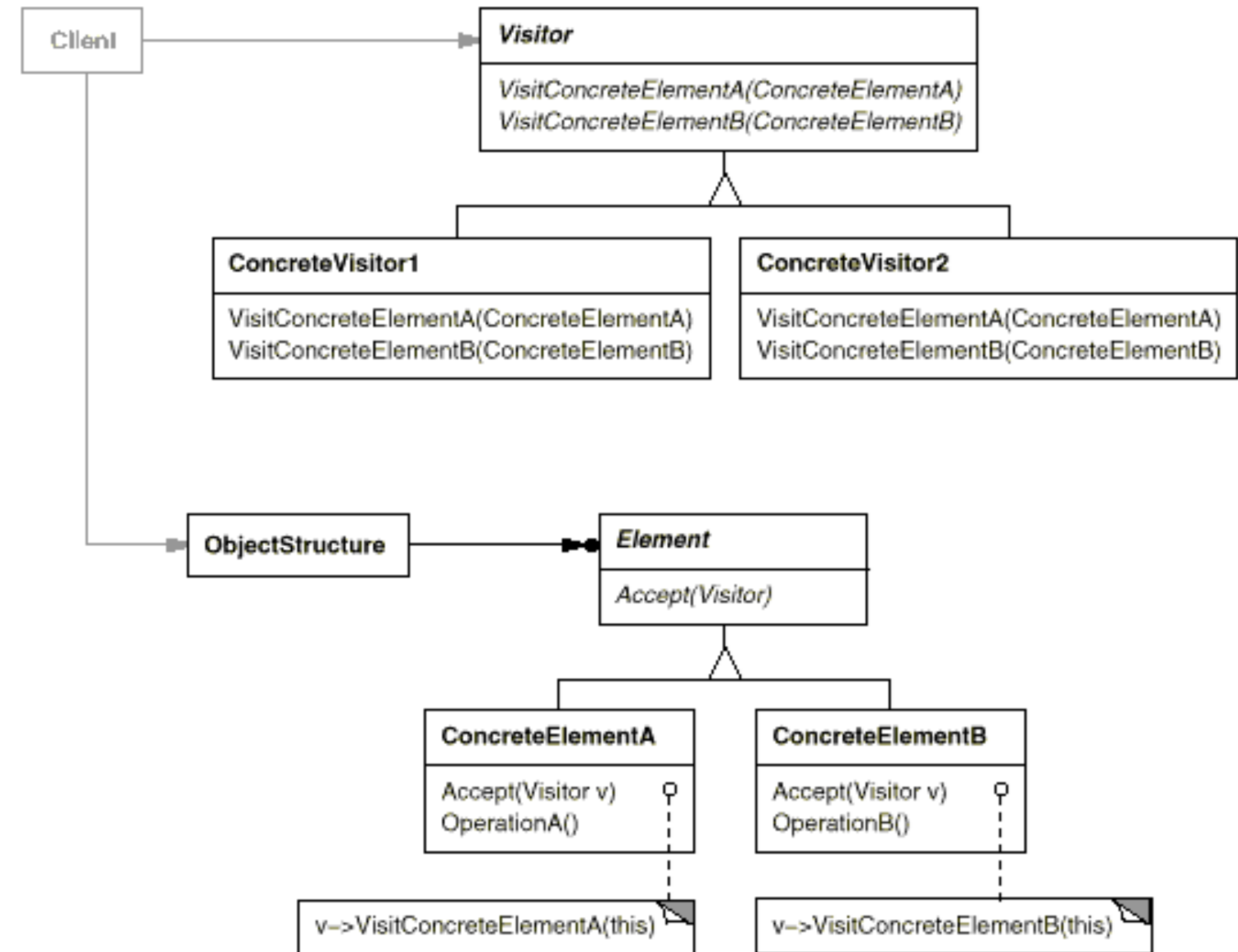
- 문법적 처리를 위해 각 노드마다 수행해야 하는 기능을 따로 정의해야 함
- 유사한 오퍼레이션들이 여러 객체에 분산되어 있어서 디버깅이 어렵고 가독성이 떨어짐
- 각 클래스에서 유사한 오퍼레이션들을 클래스로 따로 모아 Visitor를 만든다.



- 각 노드 클래스는 visitor가 방문하면 accept 하여 해당 노드에 대한 기능이 수행되도록 함



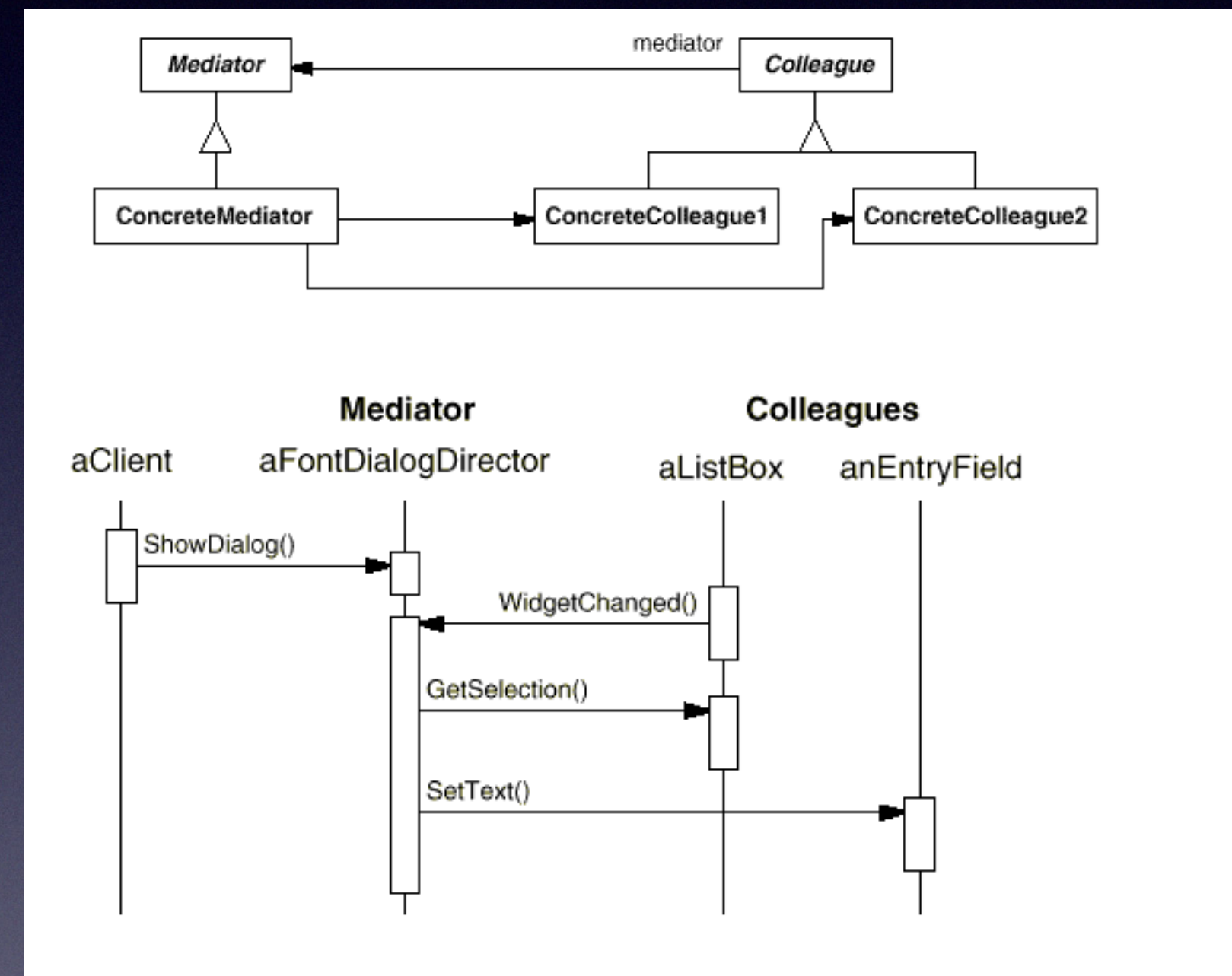
클래스 다이어그램



16. 소통은 한군데로 집중해서 - Mediator

여러 객체(Colleague)간의 상호 작용이 많고 이 작용들이 다른 Colleague에도 영향을 줄 때
주로 UI 프로그램에서 많이 사용하는 방법

Widget 상호간에 주과 받아야하는 메시지가 많은 경우, 이를 한 객체(Mediator)에서 전담하여 메시지를 처리하도록 함
N:N의 관계가 1:N으로 줄어드는 효과 (counselor)



Colleague : Mediator의 참조자를 가지고 있고, 자신의 상태가 변하거나 다른 객체와의 협력이 필요할때 Mediator에게 알림

ConcreteMediator : Colleague간의 이루어지는 협력을 구현하고, 자신의 Colleague들을 관리한다.

Mediator.java

```
public interface Mediator {  
    public abstract void createColleagues();  
    public abstract void colleagueChanged(Colleague colleague);  
}
```

Colleague.java

```
public interface Colleague {  
    public abstract void setMediator(Mediator mediator);  
    public abstract void setColleagueEnabled(boolean enabled);  
}
```

ColleagueButton.java

```
public class ColleagueButton extends Button implements Colleague {  
    private Mediator mediator;  
    public ColleagueButton(String caption) {  
        super(caption);  
    }  
    public void setMediator(Mediator mediator) {                // Mediator를 보관  
        this.mediator = mediator;  
    }  
    public void setColleagueEnabled(boolean enabled) {          // Mediator가 유효/무효를 지시한다.  
        setEnabled(enabled);  
    }  
}
```



```
public class LoginFrame extends Frame implements ActionListener, Mediator {
    private ColleagueCheckbox checkGuest;
    private ColleagueCheckbox checkLogin;
    private ColleagueTextField textUser;
    private ColleagueTextField textPass;
    private ColleagueButton buttonOk;
    private ColleagueButton buttonCancel;

    // 생성자
    // Colleague들을 생성해서 배치한 후에 표시를 실행한다.
    public LoginFrame(String title) {
        super(title);
        setBackground(Color.lightGray);
        // 레이아웃 매니저를 사용해서 4*2의 그리드를 만든다.
        setLayout(new GridLayout(4, 2));
        // Colleague들의 생성
        createColleagues();
        // 배치
        add(checkGuest);
        add(checkLogin);
        add(new Label("Username:"));
        add(textUser);
        add(new Label("Password:"));
        add(textPass);
        add(buttonOk);
        add(buttonCancel);
        // 유효/무효의 초기지정
        colleagueChanged(checkGuest);
        // 표시
        pack();
        show();
    }
}
```



```
// Colleague들을 생성한다.
public void createColleagues() {
    // 생성
    CheckboxGroup g = new CheckboxGroup();
    checkGuest = new ColleagueCheckbox("Guest", g, true);
    checkLogin = new ColleagueCheckbox("Login", g, false);
    textUser = new ColleagueTextField("", 10);
    textPass = new ColleagueTextField("", 10);
    textPass.setEchoChar('*');
    buttonOk = new ColleagueButton("OK");
    buttonCancel = new ColleagueButton("Cancel");
    // Mediator의 세트
    checkGuest.setMediator(this);
    checkLogin.setMediator(this);
    textUser.setMediator(this);
    textPass.setMediator(this);
    buttonOk.setMediator(this);
    buttonCancel.setMediator(this);
    // Listener의 세트
    checkGuest.addItemListener(checkGuest);
    checkLogin.addItemListener(checkLogin);
    textUser.addTextListener(textUser);
    textPass.addTextListener(textPass);
    buttonOk.addActionListener(this);
    buttonCancel.addActionListener(this);
}
```



```

// colleague로부터의 통지로 각 Colleague의 유효/무효를 판정한다.
public void colleagueChanged(Colleague c) {
    if (c == checkGuest || c == checkLogin) {
        if (checkGuest.getState()) { // Guest 모드
            textUser.setColleagueEnabled(false);
            textPass.setColleagueEnabled(false);
            buttonOk.setColleagueEnabled(true);
        } else { // Login 모드
            textUser.setColleagueEnabled(true);
            userpassChanged();
        }
    } else if (c == textUser || c == textPass) {
        userpassChanged();
    } else {
        System.out.println("colleagueChanged:unknown colleague = " + c);
    }
}

// textUser 또는 textPass의 변경이 있었다.
// 각 Colleague의 유효/무효를 판정한다.
private void userpassChanged() {
    if (textUser.getText().length() > 0) {
        textPass.setColleagueEnabled(true);
        if (textPass.getText().length() > 0) {
            buttonOk.setColleagueEnabled(true);
        } else {
            buttonOk.setColleagueEnabled(false);
        }
    } else {
        textPass.setColleagueEnabled(false);
        buttonOk.setColleagueEnabled(false);
    }
}

public void actionPerformed(ActionEvent e) {
    System.out.println("" + e);
    System.exit(0);
}
}

```

Main.java

```

public class Main {
    static public void main(String args[]) {
        new LoginFrame("Mediator Sample");
    }
}

```


17. 대리자 - Proxy

실제 객체에 대한 대리자를 둔다.

- 대리자의 종류

원격 프록시 (Remote Proxy) : 서로 다른 주소 공간에 존재하는 원격 객체를 대리하는 로컬 객체, Java RMI 기능

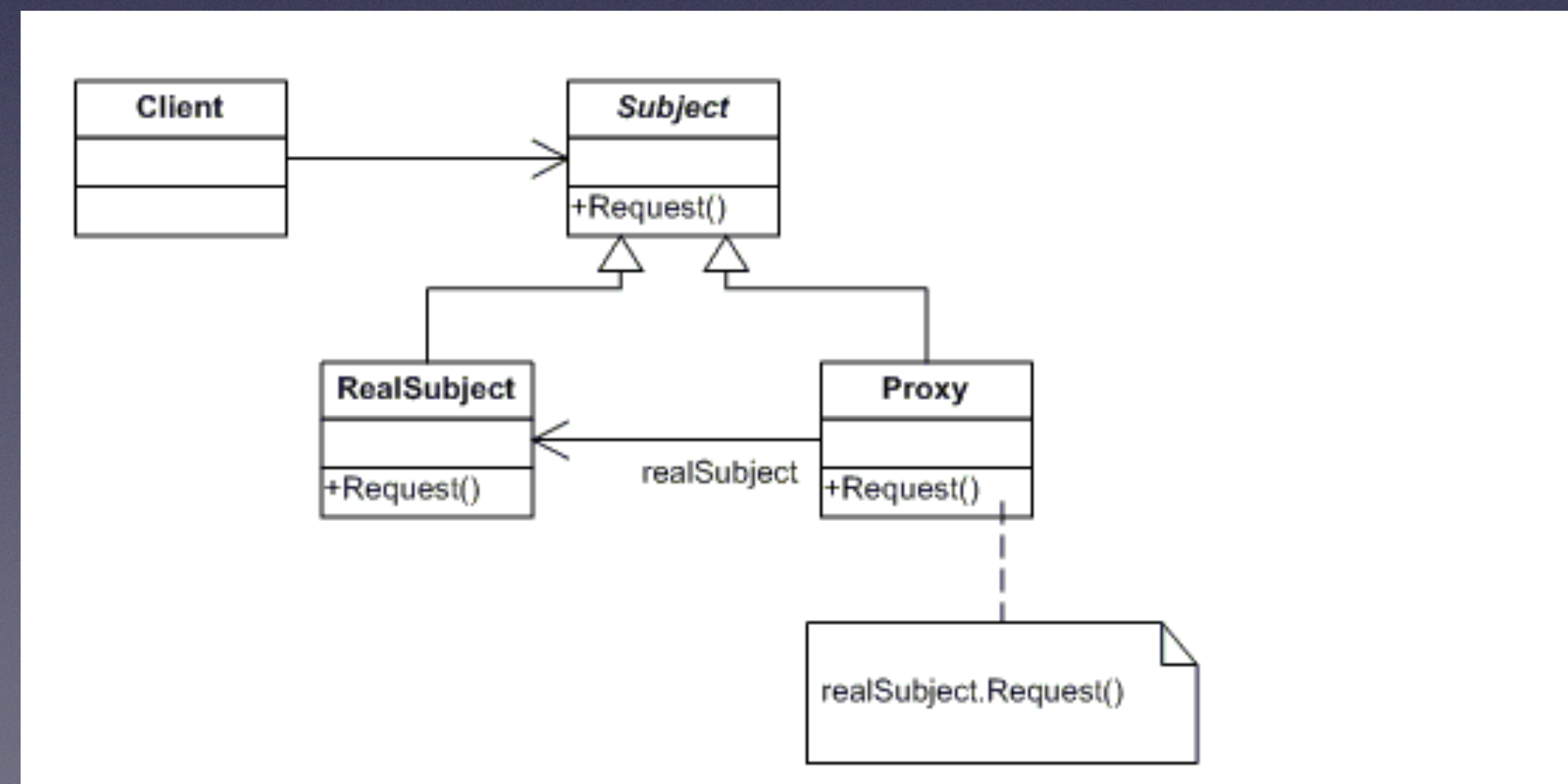
가상 프록시 (virtual proxy) : 실제 객체가 모두 생성되기 전에 대신해주는 프록시

보호 프록시 (Protection Proxy) : 실제 객체에 대한 접근 권한을 제어하기 위한 경우 사용

실제 객체와 프록시는 동일한 메서드를 제공함

클라이언트가 호출 하는 메서드는 프록시의 메서드

프록시에서 처리하는 경우도 있고, 실제 객체 전달되어 처리 되는 경우도 있음




```
public interface Printable {
    public abstract void setPrinterName(String name);    // 이름의 설정
    public abstract String getPrinterName();            // 이름의 취득
    public abstract void print(String string);           // 문자열 표시(프린트아웃)
}
```

```
public class PrinterProxy implements Printable {
    private String name;                // 이름
    private Printer real;                // "본인"
    public PrinterProxy() {
    }
    public PrinterProxy(String name) {    // 생성자
        this.name = name;
    }
    public synchronized void setPrinterName(String name) { // 이름의 설정

        //Todo
    }
    public String getPrinterName() {      // 이름의 취득
        System.out.println("proxy : getPrinterName()");
        return name;
    }
    public void print(String string) {    // 표시
        // Todo
    }
    private synchronized void realize() { // "본인"을 생성
        //Todo
    }
}
```



```
public class Printer implements Printable {
    private String name;
    public Printer() {
        heavyJob("Printer의 인스턴스를 생성중");
    }
    public Printer(String name) {                // 생성자
        this.name = name;
        heavyJob("Printer의 인스턴스(" + name + ")를 생성중");
    }
    public void setPrinterName(String name) {    // 이름의 설정
        System.out.println("real : setPrinterName()");
        this.name = name;
    }
    public String getPrinterName() {            // 이름의 취득
        System.out.println("real : getPrinterName()");
        return name;
    }
    public void print(String string) {          // 이름을 붙여서 표시
        System.out.println("=== " + name + " ===");
        System.out.println(string);
    }
    private void heavyJob(String msg) {         // 무거운 작업
        System.out.print(msg);
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            System.out.print(".");
        }
        System.out.println("완료");
    }
}
```



```

public class Main {
    public static void main(String[] args) {
        Printable p = new PrinterProxy("Alice");
        System.out.println("current printer name is" + p.getPrinterName() );
        p.setPrinterName("Bob");
        System.out.println("current printer name is" + p.getPrinterName() );
        p.print("Hello, world.");

        p.print("test");
        p.setPrinterName("Tomas");
        System.out.println("current printer name is"+ p.getPrinterName() );

    }
}

```

```

proxy : getPrinterName()
current printer name isAlice
proxy : setPrinterName()
proxy : getPrinterName()
current printer name isBob
Printer (Bob) constructor call .....
=== Bob ===
Hello, world.
=== Bob ===
test
real : setPrinterName()
proxy : setPrinterName()
proxy : getPrinterName()
current printer name isTomas

```


17. Flyweight

작은 객체들을 효과적으로 공유하여 사용하기 위한 패턴

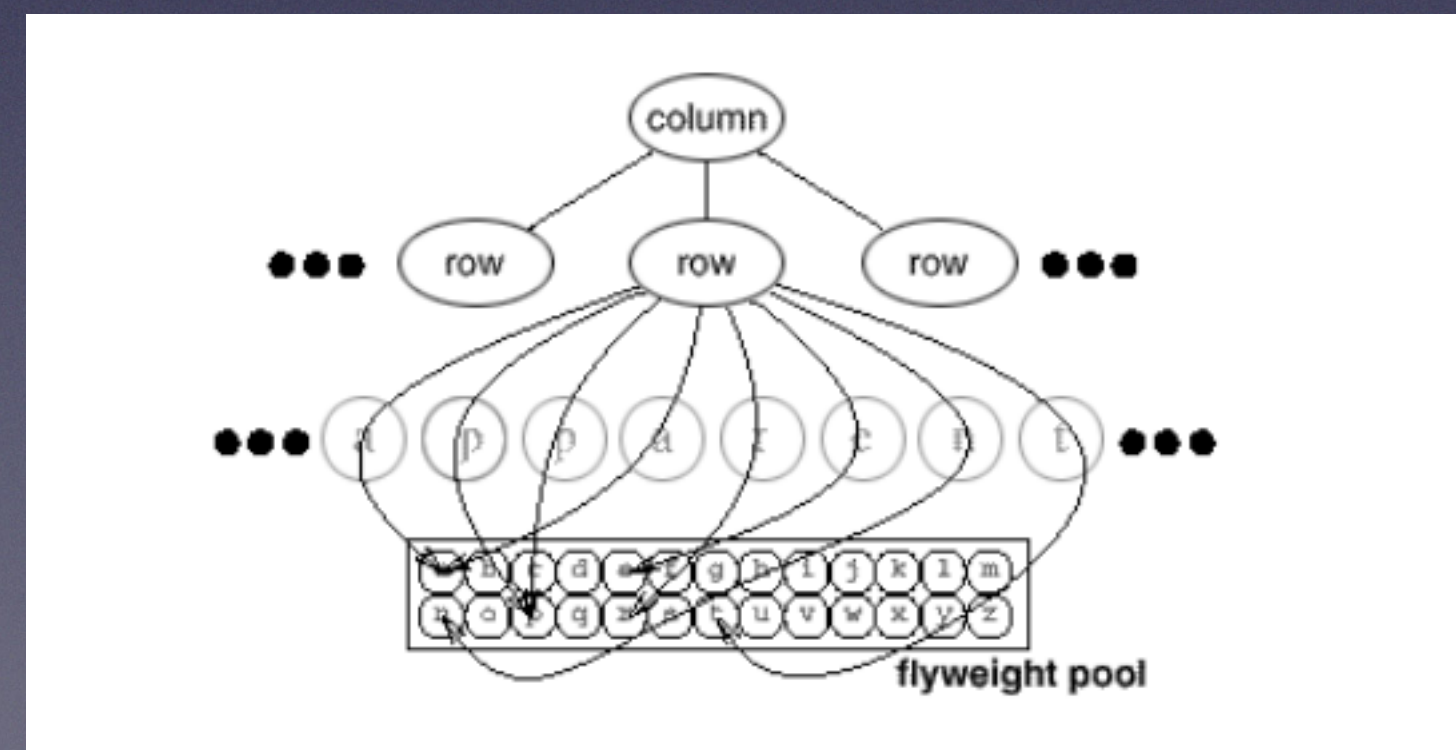
반복적으로 쓰이는 작은 객체들 (예를 들어 문자 같은)은 개별 인스턴스로 생성하는 것이 아닌

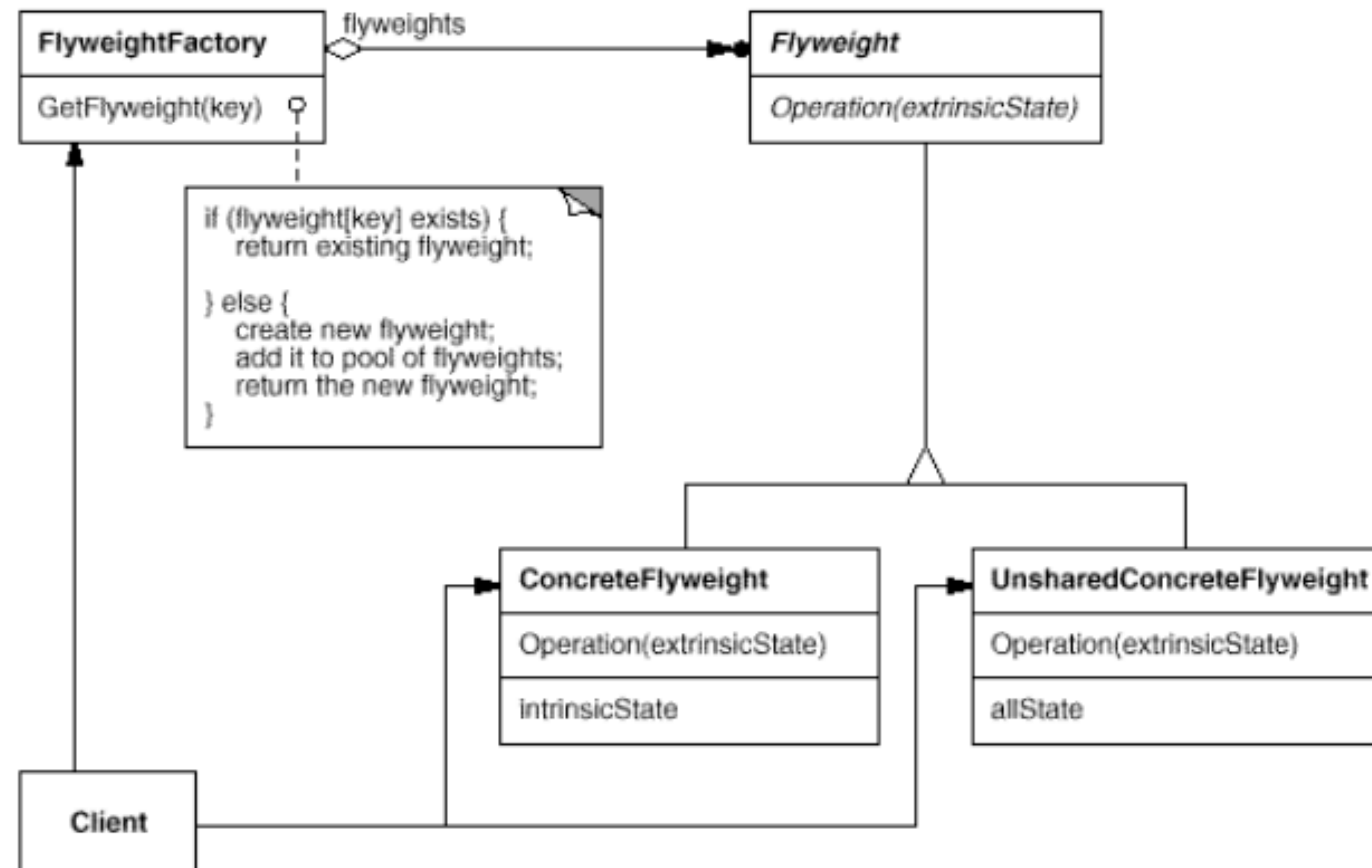
한번 생성하고 반복적으로 사용할 수 있도록 pool을 관리함

여러개의 객체생성에 대한 초기 비용이 있음

언제 사용하나?

- 객체마다 인스턴스 값이 다르지 않은 경우가 대부분이고 이런 경우 Singleton pattern으로 생성함
- 텍스트를 만든다고 할때 각 단어를 객체로 생성하는 것이 아닌 문자를 생성하고 이를 공유하도록 함
이때 필요한 추가적인 정보 (글꼴, 크기등은 부가 정보로 따로 저장함)






```
public class BigString {
    // "큰 문자"의 배열
    private BigChar[] bigchars;
    // 생성자
    public BigString(String string) {
        bigchars = new BigChar[string.length()];
        BigCharFactory factory = BigCharFactory.getInstance();
        for (int i = 0; i < bigchars.length; i++) {
            bigchars[i] = factory.getBigChar(string.charAt(i));
        }
    }
    // 표시
    public void print() {
        for (int i = 0; i < bigchars.length; i++) {
            bigchars[i].print();
        }
    }
}

public class Main {
    public static void main(String[] args) {

        BigString bs = new BigString("123abc123");
        bs.print();
    }
}
```

출력 예

[illegible]

파일에서 숫자를 읽어 BigInt를 만들고 BigIntFactory도 만들어서 출력하세요

18. Others...

Builder

Facade

Interpreter

Chain of Responsibility

19. 객체지향 프로그래밍의 원리

객체 지향 디자인 원칙 (Object Oriented Design Principle)

- 애플리케이션의 달라지는 부분을 찾아내고, 달라지지 않는 부분과 분리한다. 새로운 요구사항이 있을때 마다 달라지는 부분은 분리해야 함
- 구현보다는 인터페이스에 맞춰서 프로그래밍 한다. (Program to an interface not an implementation.)
- 상속보다는 합성을 사용한다. (Favor object composition over class inheritance.)
- Abstract class vs. Concrete class
- Class Inheritance vs. Object composition
- Interface Inheritance vs. Implementation Inheritance Etc...

SOLID 원칙

- 단일 책임의 원칙 (Single Responsibility Principle) 하나의 클래스는 하나의 기능만을 구현하도록 한다. 즉, 어떤 클래스를 변경하는 이유는 하나이어야 한다 한 클래스에서 여러 기능을 제공하게 되면 유지보수가 어려움
- 개방 폐쇄의 원칙 (Open-Closed Principle)
객체 자신의 수정에 대해서는 유연하고, 다른 클래스가 수정될 때는 영향을 받지 않는다.
인터페이스나 추상클래스를 통해 접근 하도록 함
- 리스코프 치환 원칙 (Liskov Substitution Principle)
하위 클래스는 항상 상위 클래스로 교체 될 수 있어야 한다.
즉, 상위 클래스에 제공되는 여러 기능은 하위 클래스가 모두 사용가능 해야 한다.
IS-A 관계, "is a kind of" 관계
- 의존 역전 원칙 (Dependency Inversion Principle)
의존 관계는 구체적인 것보다는 추상적인 것에 의존한다.
구체적인 것은 이미 구현이 되어있고 변하기 쉬운것
추상적인 것은 인터페이스나 추상 클래스(상위 클래스)
- 인터페이스 분리 원칙 (Interface Segregation Principle)
제공하는 기능에 대한 인터페이스에만 종속적이어야 함
만약 하나의 객체가 여러 기능을 제공해야 한다면 (단일 책임 원칙에 위배) 이때 클라이언트가
사용할 수 있는 여러 인터페이스로 분리하여 제공하면 클라이언트가 사용하지 않는 기능에 종속적이지 않을 수 있음

디자인 패턴은 규칙이 아닙니다.

- 언어에 종속적이지 않음
- 프레임 워크 개발에 적용될 수 있음
- 특정 영역에 종속적이지 않고 일반적으로 활용할 수 있음
- 좋은 설계에 대한 제안

왜 학습 하는가?

- 객체 지향을 위한 디자인 패턴은 소프트웨어의 중요한 요소 (reuse, flexibility, extensibility, modularity) 를 향상 시킴
- 소프트웨어 개발의 communication에 도움이 됨
- 좋은 설계는 좋은 소프트웨어나 오픈소스에 대해 학습하거나, 많은 경험과 연습에 의해 훈련될 수 있기에 디자인 패턴을 공부함으로써 이미 증명된 스킬과 경험을 배울수 있음
- 높은 결합도를 가지거나 알고리즘 종속성, 객체의 표현이나 구현에 종속적으로 구현된 소프트웨어의 리팩토링을 가능하게 함
- 결국 좋은 설계를 유도하여 소프트웨어의 유지보수에 들어가는 비용을 절약할 수 있음