

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");

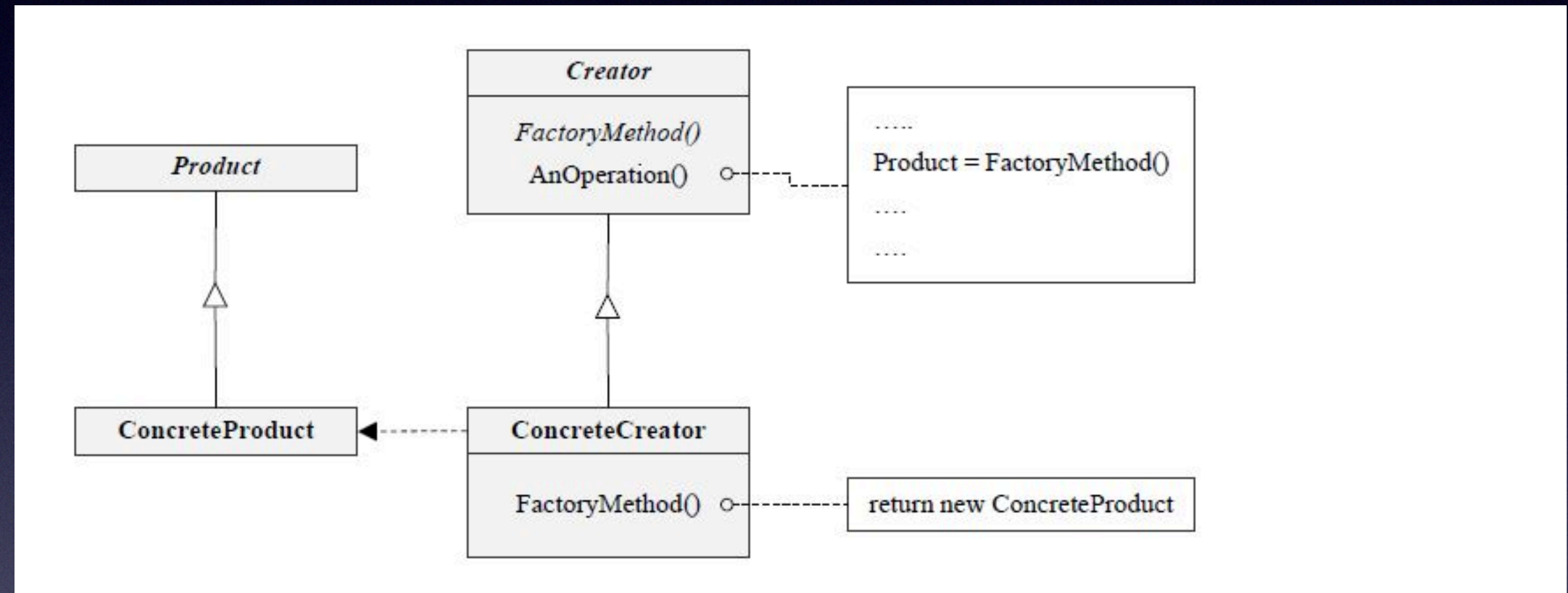
        System.out.println(car);
    }

    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();
    }
}

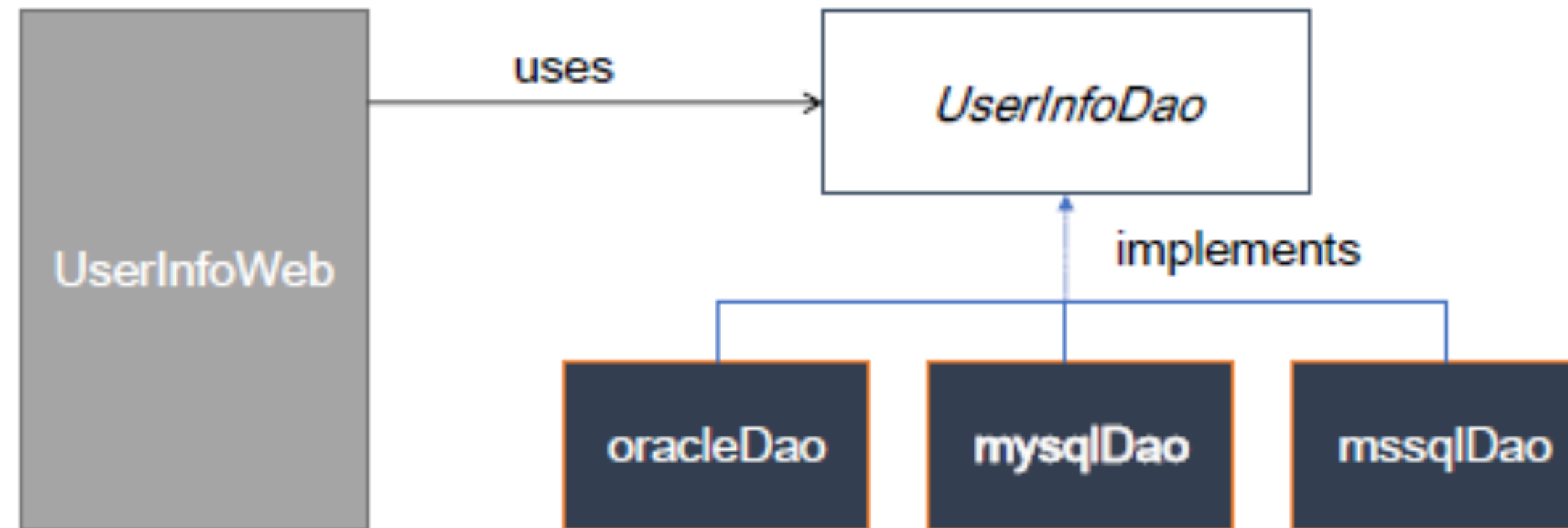
```

```

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;
    }

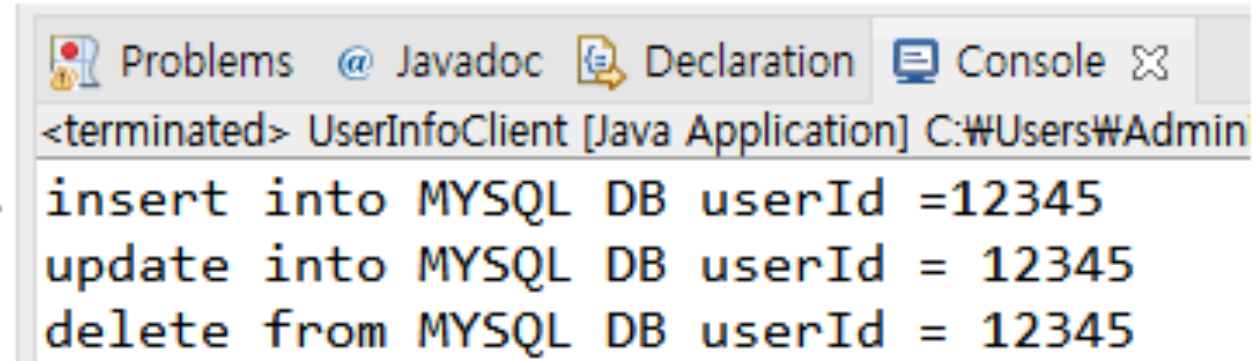
    public void setUserName(String userName) {
        this.userName = userName;
    }
}

```

db.properties 환경파일이 MYSQL 일때

DBTYPE=MYSQL

실행결과



```

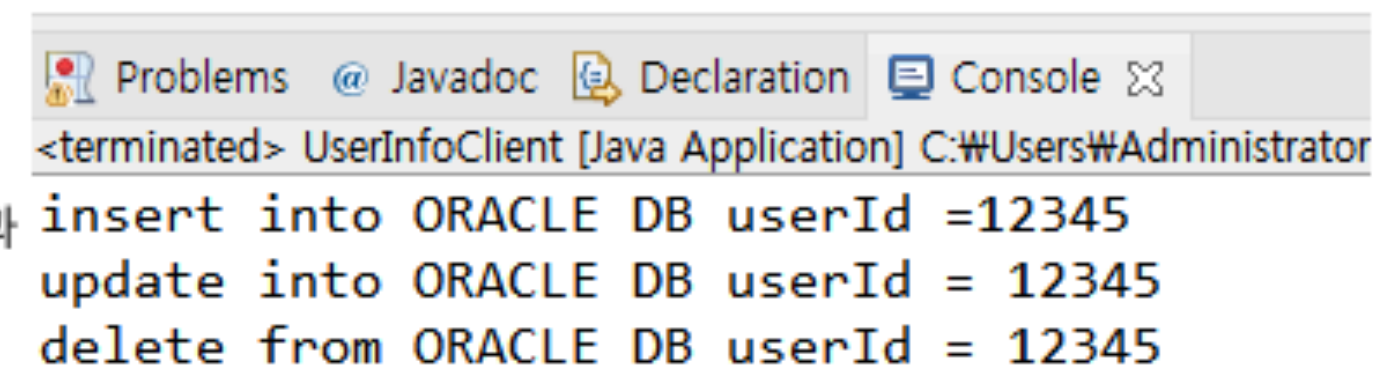
<terminated> UserInfoClient [Java Application] C:\Users\Admin
insert into MYSQL DB userId =12345
update into MYSQL DB userId = 12345
delete from MYSQL DB userId = 12345

```

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

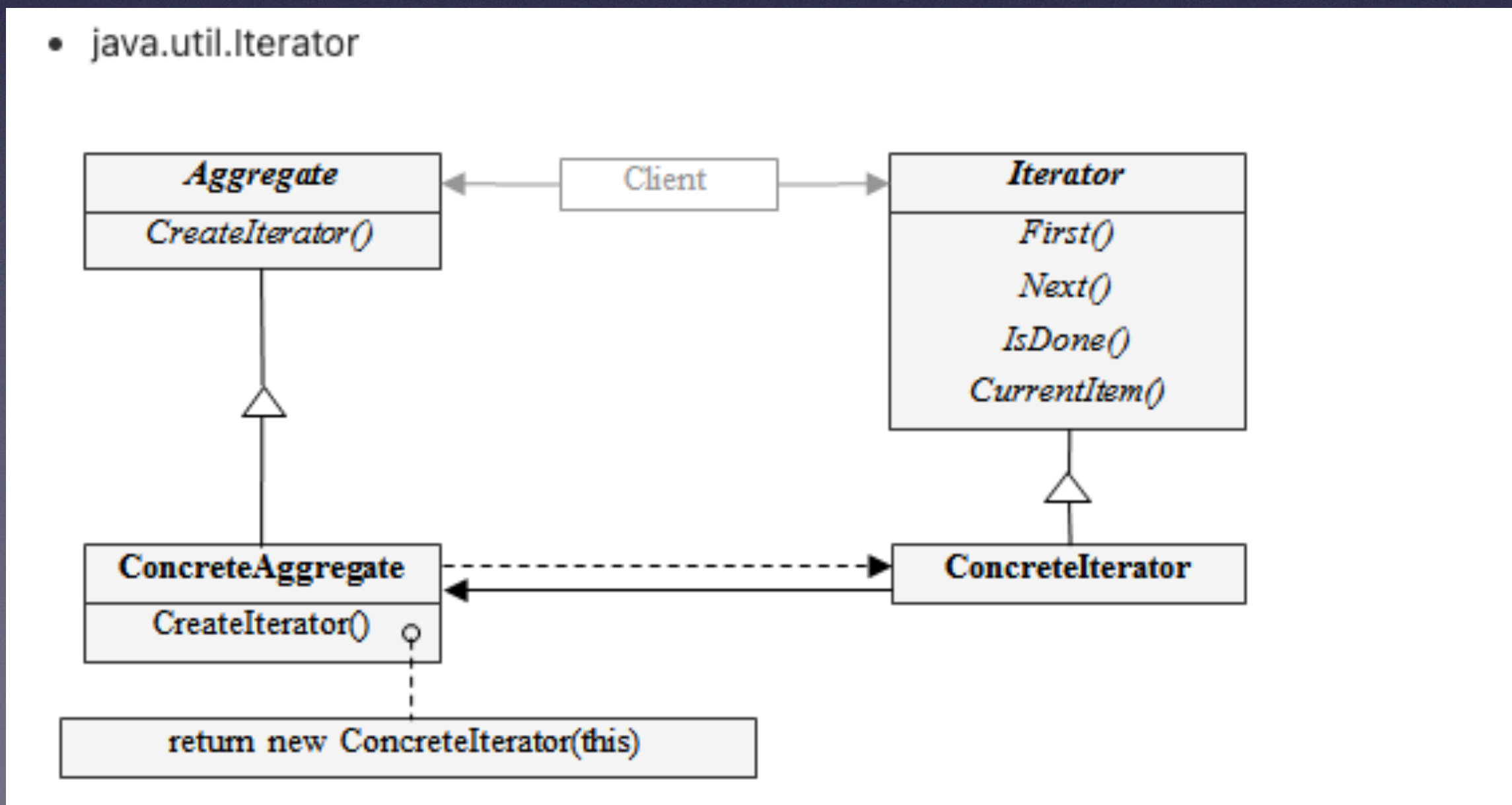
```


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

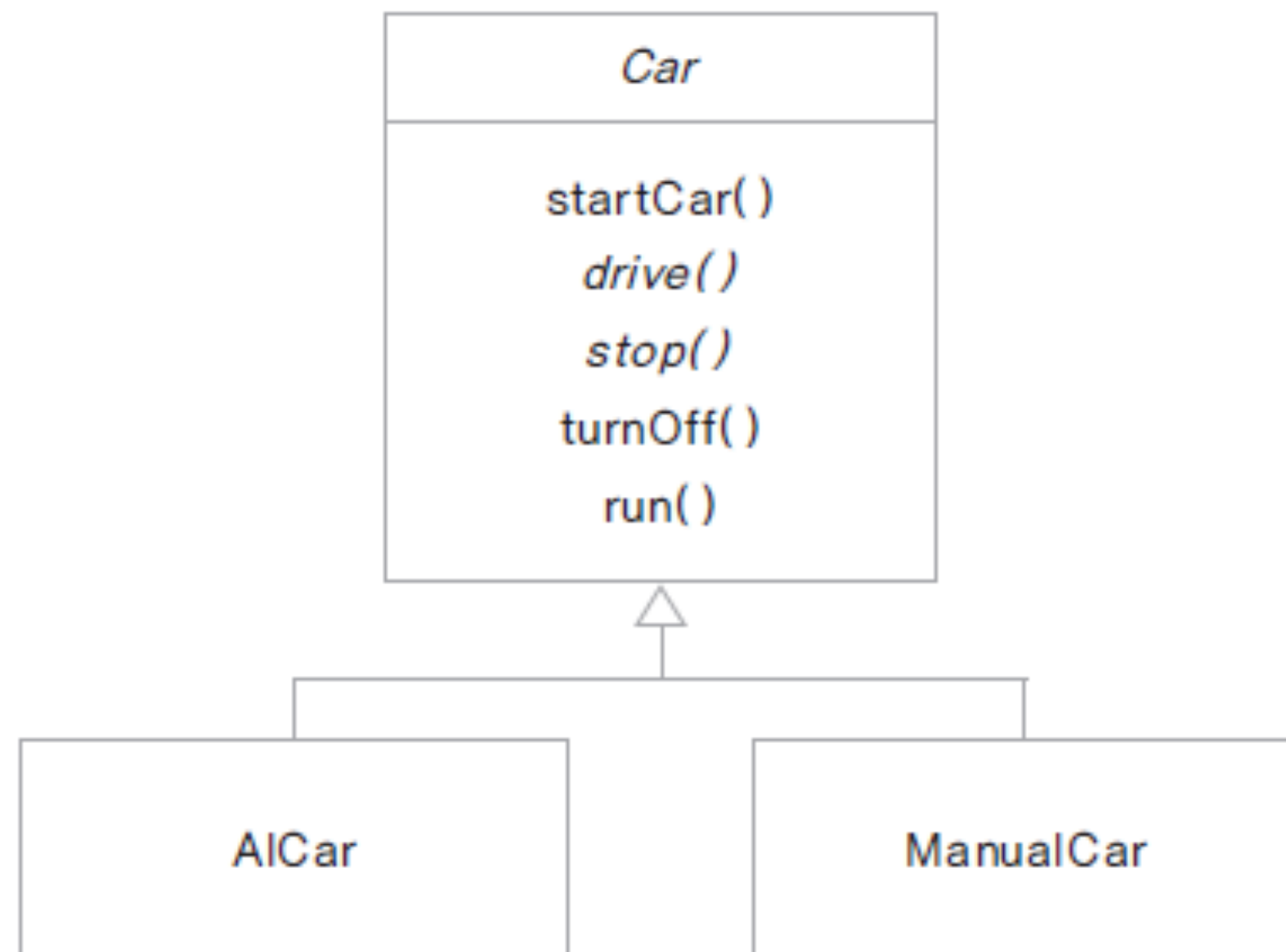
- 다양한 Iterator가 만들어 지는 Factory를 만들어 적용해보기

```
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

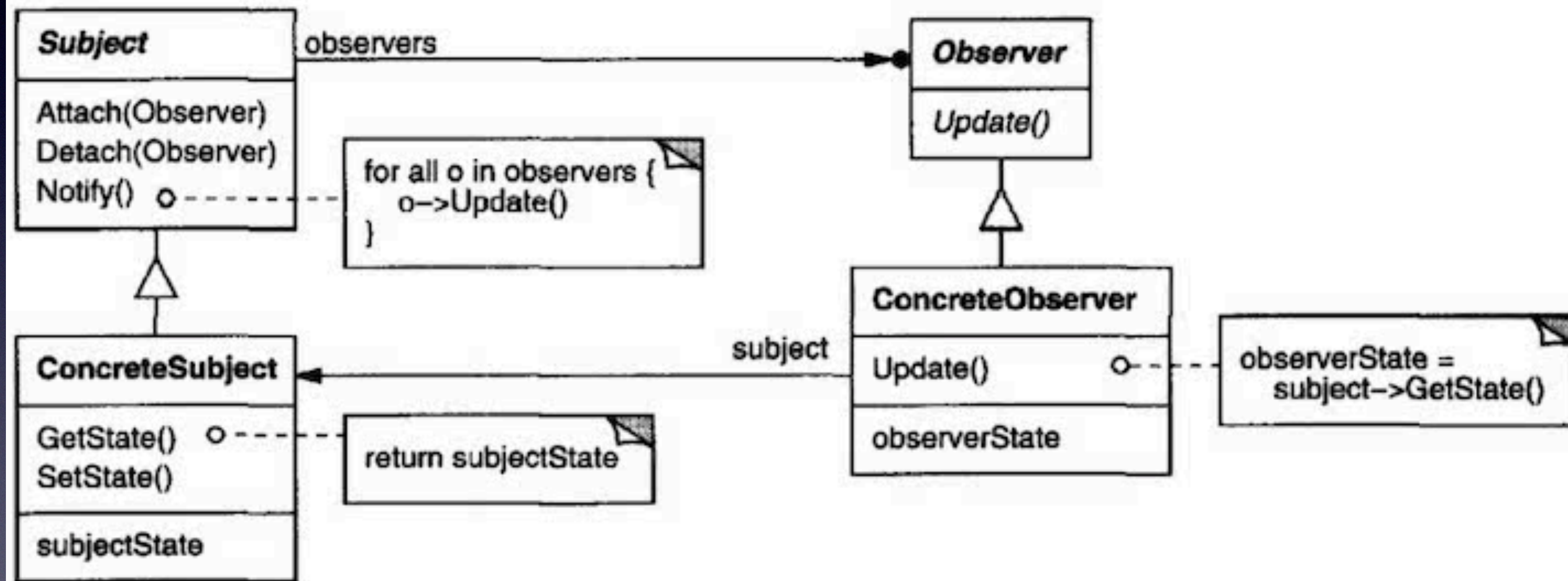
이때 중요한 것 변화에 관심을 가지고 그 정보를 업데이트 해야 하는 객체가 몇개냐와는 상관없이 일괄적으로 메시지가 전송되어야 함

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  
    }  
}
```

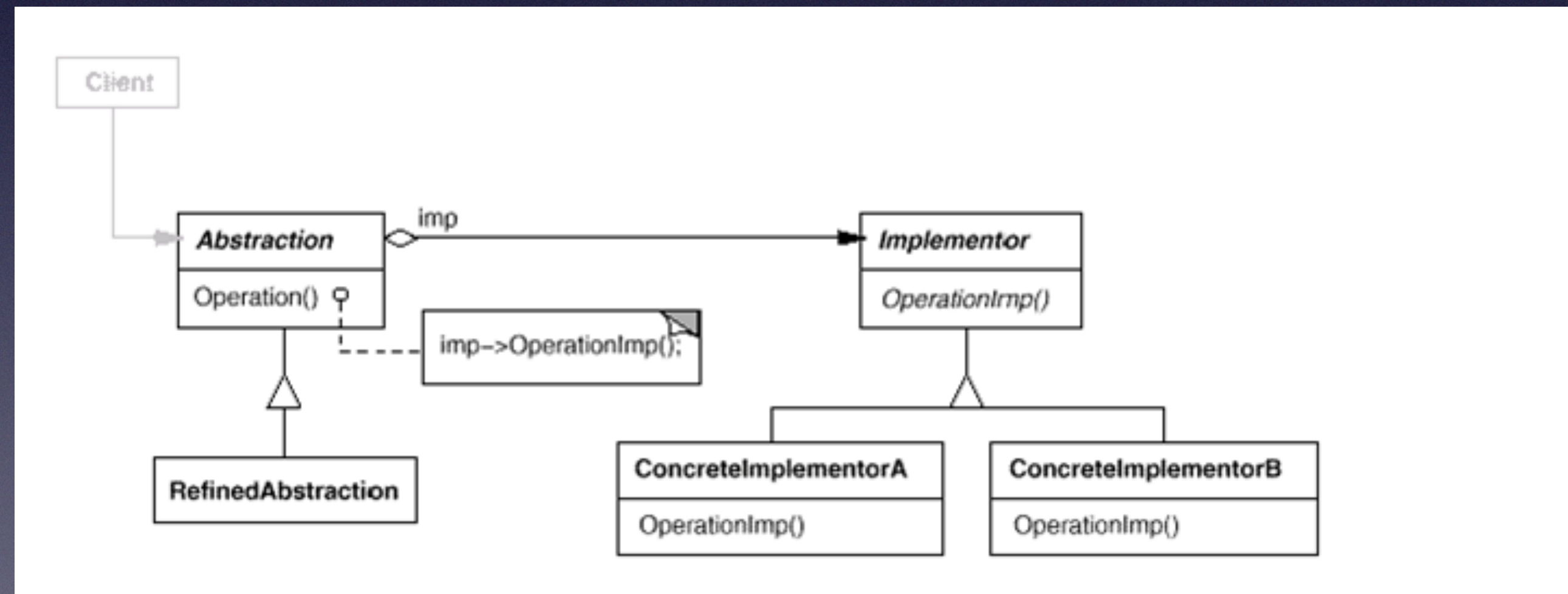

9. 기능의 확장과 구현의 확장을 분리 - Bridge

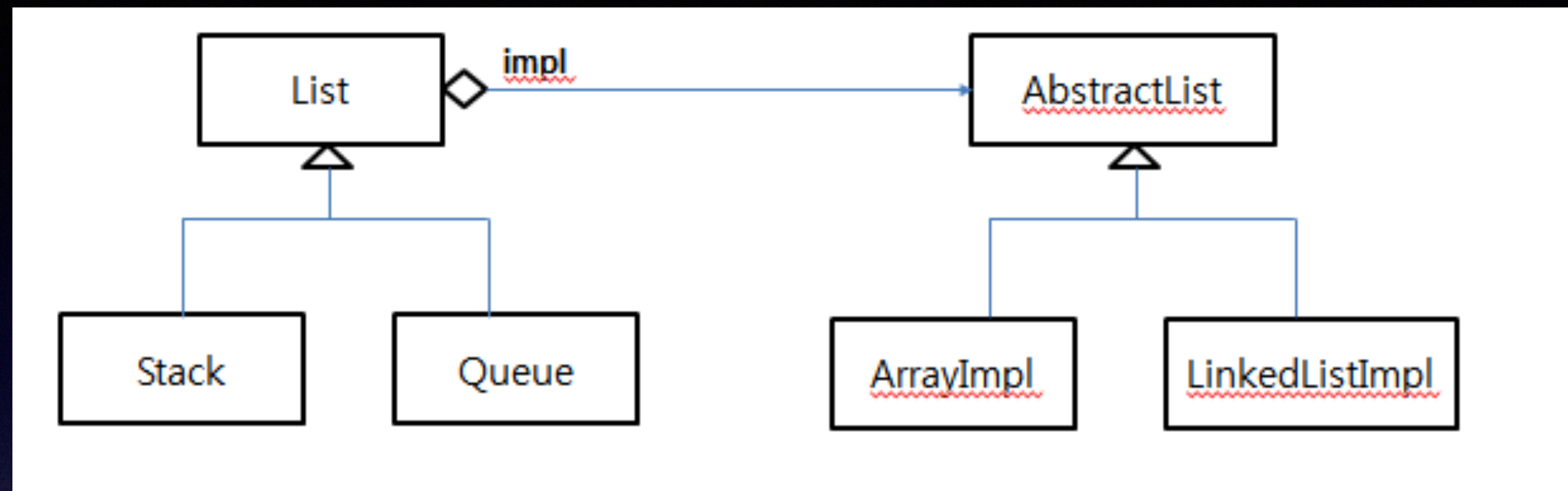
기능(추상적인 개념)의 확장과 구현의 확장을 분리한다

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("=====");

    }

}
```