디자인 패턴

디자인 패턴

❖비슷한 목적으로 사용되는 클래스의 모범 사례를 패턴으로 정리하려고 하는 것이 디자인 패턴이다.

❖프로그램의 세계에는 다양한 디자인 패턴이 존재

❖유명한 것이 'GoF 디자인 패턴'이다. GoF(고프)란 'The Gang of Four'의 약자이며, 에리히 감마, 리처드 헬름, 랄프 존슨, 존 블리시데스의 4명을 가리킨다.

❖객체지향 프로그래밍에 도움이 되는 디자인 패턴을 도입해 《Design Patterns: Elements of Reusable Object Oriented Software》라는 제목의 책에 정리했다. 이 책에 소개된 23가지 디자인 패턴을 GoF 디자인 패턴이라고 한다.

❖디자인 패턴의 분류

- ✓ 객체의 '생성'에 관한 패턴
- ✓ 프로그램의 '구조'에 관한 패턴
- ✓ 객체의 '행동'에 관한 패턴

❖디자인 패턴은 정교한 클래스 구조의 패턴집합이며, 잘 이해해서 올바르게 적용하면 충실하고 알기 쉬운 클래스 구성을 생성할 수 있다.

디자인 패턴

❖디자인 패턴의 장점

✓ 재사용성이 높고 유연성 있는 설계가 가능하다

- 프로그램을 처음부터 설계하면 만들어진 성과물의 품질이 설계자의 직관이나 경험 등에 따라 달라진다. 하지만 디자인 패턴을 도입하면 초보자도 고수들의 갈고 닦은 '지혜'를 이용한 설계가 가능하다.

✓ 의사 소통이 쉬워진다

- 디자인 패턴을 습득하지 않은 기술자에게 설계를 설명할 경우 '이런 클래스를 만들고, 이 클래스는 이런 역할을 갖고 있고……'라고 끝없이 설명해야 한다.
- 디자인 패턴을 습득하고 있는 기술자라면 '○○패턴으로 만들어!'라는 한마디로 끝나 버린다.
- 디자인 패턴을 습득하고 있는 기술자끼리라면 설계에 대해 상담할 때도 디자인 패턴의 이름으로 설계 개요에 대해 동의를 얻는것이 가능해 의사 소통이 원활하다.

✓ 디자인 패턴을 이해하고 개념을 익혀두면 자신이 직접 설계하는 경우에도 적용할 수 있어 설계의 수준을 높일 수 있다.

생성에 관련된 패턴

❖싱글톤 패턴

- ✓ 클래스가 하나의 객체만 생성할 수 있는 디자인 패턴
- ✓ 시스템 전체에 공통적으로 적용되는 설정정보를 보관하는 관리자 클래스나 전역 변수를 소유한 클래스를 만들 때 주로 이용
- ✓ 적용 방법
  - 생성자를 private으로 만듭니다.
  - 자신의 타입으로 된 static 변수를 선언합니다.
  - static 변수가 null이면 생성해서 리턴하고 그렇지 않으면 그냥 리턴하는 static 메서드를 생성합니다.
- ✓ jdk의 클래스 중에서 객체 생성 시 생성자를 호출하지 않고 static 메서드를 이용해서 객체를 리턴받는 클래스는 singleton 패턴입니다.

실습(Singleton.java)

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

public class Singleton{
        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());
        }
}
```

생성에 관련된 패턴

❖객체 생성을 자신의 생성자를 호출하지 않고 인스턴스 생성을 전문적으로 하는 클래스의 메서드를 이용해서 인스턴스를 생성하는 패턴

❖일관성을 유지해야 하는 관련된 일련의 인스턴스들을 생성하고자 할 때 사용하는 패턴

❖프레임워크의 객체 생성이 주로 이 패턴을 이용합니다.

실습(FactoryPattern.java)

```
interface DB{}
class OracleDB implements DB {
        OracleDB(){}
}
class MySQLDB implements DB{
        MySQLDB(){}
}

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

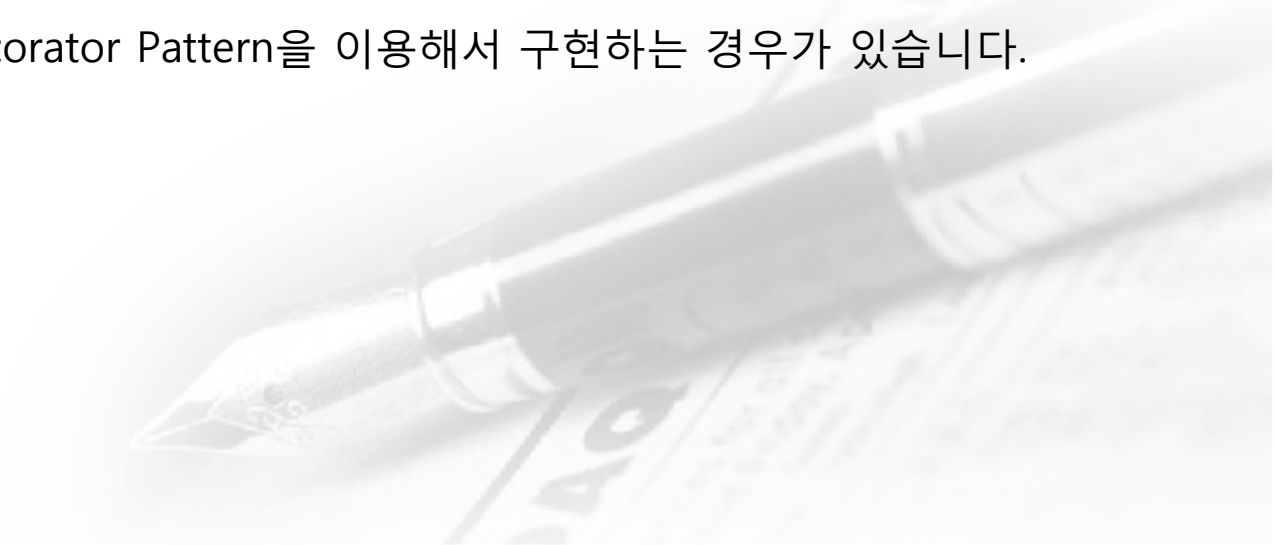
실습(FactoryPattern.java)

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

생성과 관련된 패턴

❖ Decorator Pattern

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



실습(decorator-Coffee.java)

```
package decorator;

interface Coffee {
	public void make();

	public String getDescription();
}
```

실습(decorator-SimpleCoffee.java)

```
package decorator;

class SimpleCoffee implements Coffee {
	public void make() {
	}

	public String getDescription() {
		return "단순한 커피";
	}
}
```

실습(decorator-CoffeeDecorator.java)

```
package decorator;

abstract class CoffeeDecorator implements Coffee {
	protected Coffee decoratedCoffee;

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

실습(decorator-SugarCoffeeDecorator.java)

```
package decorator;

class SugarCoffeeDecorator extends CoffeeDecorator {
	public SugarCoffeeDecorator(Coffee coffeeDecorator) {
		super(coffeeDecorator);
	}

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

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

}
```

실습(decorator-CreamCoffeeDecorator.java)

```
package decorator;

class CreamCoffeeDecorator extends CoffeeDecorator {
	public CreamCoffeeDecorator(Coffee coffeeDecorator) {
		super(coffeeDecorator);
	}

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

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

}
```

실습(decorator-DecoratorMain.java)

```
package decorator;

public class DecoratorMain {
	public static void main(String[] args) {
		Coffee myCoffee = new CreamCoffeeDecorator(
					new SugarCoffeeDecorator(new SimpleCoffee()));
		System.out.println(myCoffee.getDescription());
		Coffee yourCoffee = new SimpleCoffee();
		System.out.println(yourCoffee.getDescription());
	}
}
```

구조와 관련된 패턴

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

실습(template-SmartPhone.java)

```
package template;

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

실습(template-Iphone.java)

```
package template;

public class Iphone extends SmartPhone {

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

}
```

실습(template-Android.java)

```
package template;

public class Android extends SmartPhone {

	@Override
	public void tel() {
		System.out.println("안드로이드에서 전화를 겁니다.");
	}

}
```

실습(template-TemplateMain.java)

```
package template;

public class TemplateMain {

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

	}

}
```

구조와 관련된 패턴

- Adapter pattern
  - ✓ 현재 구현된 인터페이스의 메소드를 통해서 상속받은 클래스의 메소드를 호출하는 패턴
  - ✓ 인터페이스를 구현한 클래스의 경우 상속받은 클래스의 메소드를 직접 호출할 수 없는 경우가 있어서 인터페이스의 메소드를 이용해서 상속받은 클래스의 메소드를 호출하는 방식

구조와 관련된 패턴

- Adapter pattern
  - ✓ 현재 구현된 인터페이스의 메소드를 통해서 상속받은 클래스의 메소드를 호출하는 패턴
  - ✓ 인터페이스를 구현한 클래스의 경우 상속받은 클래스의 메소드를 직접 호출할 수 없는 경우가 있어서 인터페이스의 메소드를 이용해서 상속받은 클래스의 메소드를 호출하는 방식

실습(AdapterPattern.java)

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

interface Target {
	void process();
}

class Adapter extends OldSystem implements Target {
	@Override
	public void process() {
		oldProcess();
	}
}
```

실습(AdapterPattern.java)

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

실습(DelegatePattern.java)

```
class OldSystem1 {
	public void oldProcess() {
		System.out.println("예전 처리 내용");
	}
}

abstract class Target1 {
	abstract void process();
}

class Adapter1 extends Target1 {
	private OldSystem1 oldSystem1;

	Adapter1(){
		oldSystem1 = new OldSystem1();
	}
	@Override
	public void process() {
		oldSystem1.oldProcess();
	}
}
```

실습(AdapterPattern.java)

```
public class DelegatePattern {
	public static void main(String[] args) {
		Target1 target1 = new Adapter1();
		target1.process();
	}
}
```

구조와 관련된 패턴

- Composite pattern
  - ✓ 재귀적 구조를 쉽게 처리하기 위한 패턴
  - ✓ 파일 시스템에서 디렉토리 안에 디렉토리를 추가할 때 또는 디렉토리 안의 디렉토리를 재귀적으로 지우고자 할 때 사용할 수 있는 패턴

실습(Entry.java)

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

	void remove();

	void rename(String name);
}
```

실습(File.java)

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

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

실습(Directory.java)

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

public class Directory implements Entry {
	private String name;

	private List<Entry> list;

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

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

실습(Directory.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;
	}
}
```

실습(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();
	}
}
```

행동에 관련된 패턴

❖ Command Pattern

✓ 처리 내용이 비슷한 명령을 패턴에 따라 구분하거나 조합해서 실행하는 처리가 필요할 때 명령을 인스턴스로 취급하여 처리 조합을 하기 위한 패턴

✓ 인스턴스를 매개변수로 받아서 인스턴스에 처리를 하기 위한 목적으로 사용

실습(Book.java)

```
public class Book {
	private double amount;

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

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

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

실습(Command.java)

```
public abstract class Command {
	protected Book book;

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

	public abstract void execute();
}
```

실습(DiscountCommand.java)

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

실습(SpecialDiscountCommand.java)

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

실습(CommandMain.java)

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

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

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

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

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

실습(SpecialDiscountCommand.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");
    }
}
```

행동에 관련된 패턴

❖ Observer Pattern

✓ 주기적으로 상태가 변환하는 인스턴스가 존재하는 경우 이 인스턴스의 상태가 바뀐 것을 감지하여 처리하도록 해주어야 하는 경우에 사용할 수 있는 패턴

✓ 인스턴스의 상태가 변화한 것을 관찰하고 그 인스턴스 자신이 상태의 변화를 통지하는 구조를 제공하는 패턴

실습(Observer.java)

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

실습(Subject.java)

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

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

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

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

	public abstract void execute();
}
```

실습(Client.java)

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

실습(DataChanger.java)

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

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

실습(ObserverMain.java)

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

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

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

행동에 관련된 패턴

- Strategy Pattern
  - ✓ 공통된 부분과 서로 다른 부분을 찾아내서 공통된 부분과 다른 부분을 분리시켜서 객체를 생성할 때 주입(DI-Defendency Injection)을 이용해서 서로 다른 부분을 추가하는 구조

실습(strategy-SmartPhone.java)

```
package strategy;

public abstract class SmartPhone {
	private Market market;

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

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

실습(strategy-Market.java)

```
package strategy;

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

실습(strategy-AndroidMarket.java)

```
package strategy;

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

실습(strategy-IphoneMarket.java)

```
package strategy;

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

실습(strategy-Iphone.java)

```
package strategy;


public class Iphone extends SmartPhone {

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

실습(strategy-Android.java)

```
package strategy;

public class Android extends SmartPhone {

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

실습(strategy-StrategyMain.java)

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

		SmartPhone android = new Android();
		SmartPhone iPhone = new Iphone();

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

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