

SOILD

SRP 단일 책임 원칙

- 클래스를 수정해야하는 이유를 한가지로 구성한다.

- 클래스의 기능을 한가지로 구성

OCP 개방-폐쇄 원칙

- 기존 코드를 수정하지 않고, 확장이 가능하도록 만들어야 한다.

- 변경되지 않을 부분(parent)과 변경될 부분(child)을 분리한다.

LSP 리스코프 치환 원칙

- 자식 클래스가 부모 클래스를 대체할 수 있어야 한다.

- Class를 확장한 경우에는 적용되어야 하지만, 특화한 경우에는 적용 안됨

ISP 인터페이스 분리 원칙

- interface(API개념)을 구상할 땐, 여러 기능을 가지는 일반화된 interface가 아니라, 각 클라이언트에 특화되어 있는 interface를 사용한다. (ex. 입출력 클래스를 입력,출력 클래스로 나눔.

DIP 의존 역전 원칙

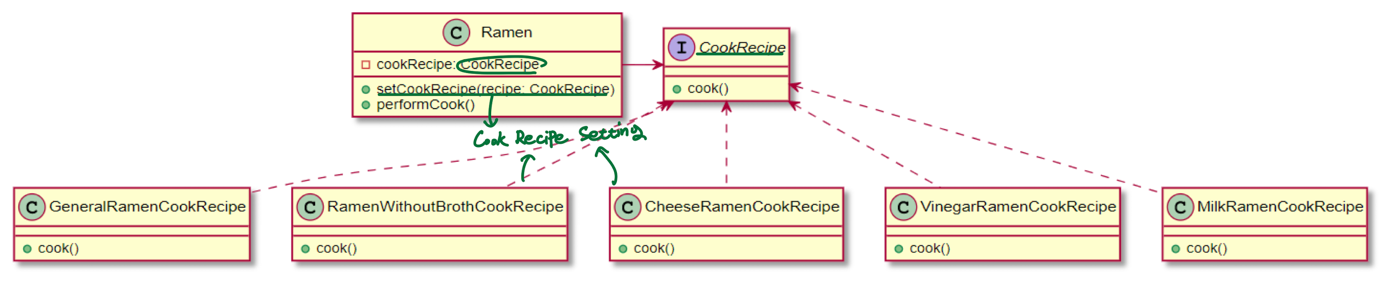
- 기능을 직접 구현한 클래스나 함수는 변경될 가능성이 높기 때문에 기능을 구현한 클래스를 상속받지 않는 것이 좋다.

Generics => ArrayList<String> 형태, data type을 정해놓고 사용하거나, 사용할 때마다 형변환 X, 처음에 data type지정.

Strategy pattern (스트래티지 패턴) MP02

알고리즘을 캡슐화하여, 클라이언트에게서 독립적으로 구현

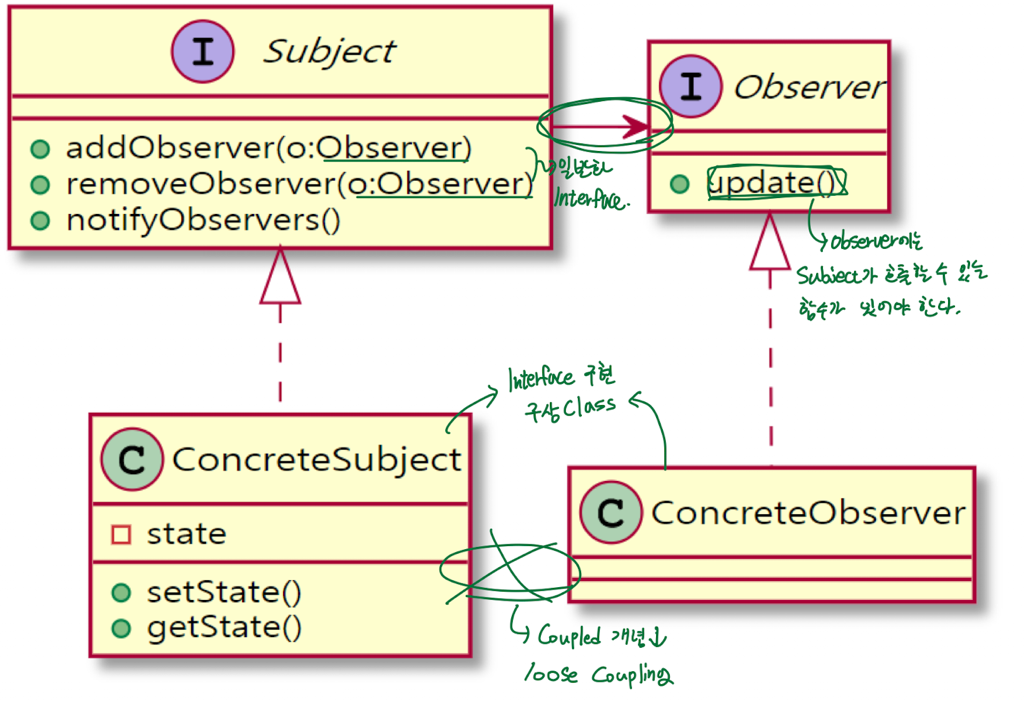
* 변화하는 알고리즘을 가진 Interface를 만들고, 이를 구현하는 여러 클래스를 만든다.
* 해당 알고리즘을 사용하는 클래스에서는 내부에서 멤버 변수로 interface 선언하고 구현된 클래스를 선언하여 내부의 함수를 사용한다.
* Setter 함수를 만들어서 Interface에 알고리즘이 들어간 클래스를 넣어줘야함.

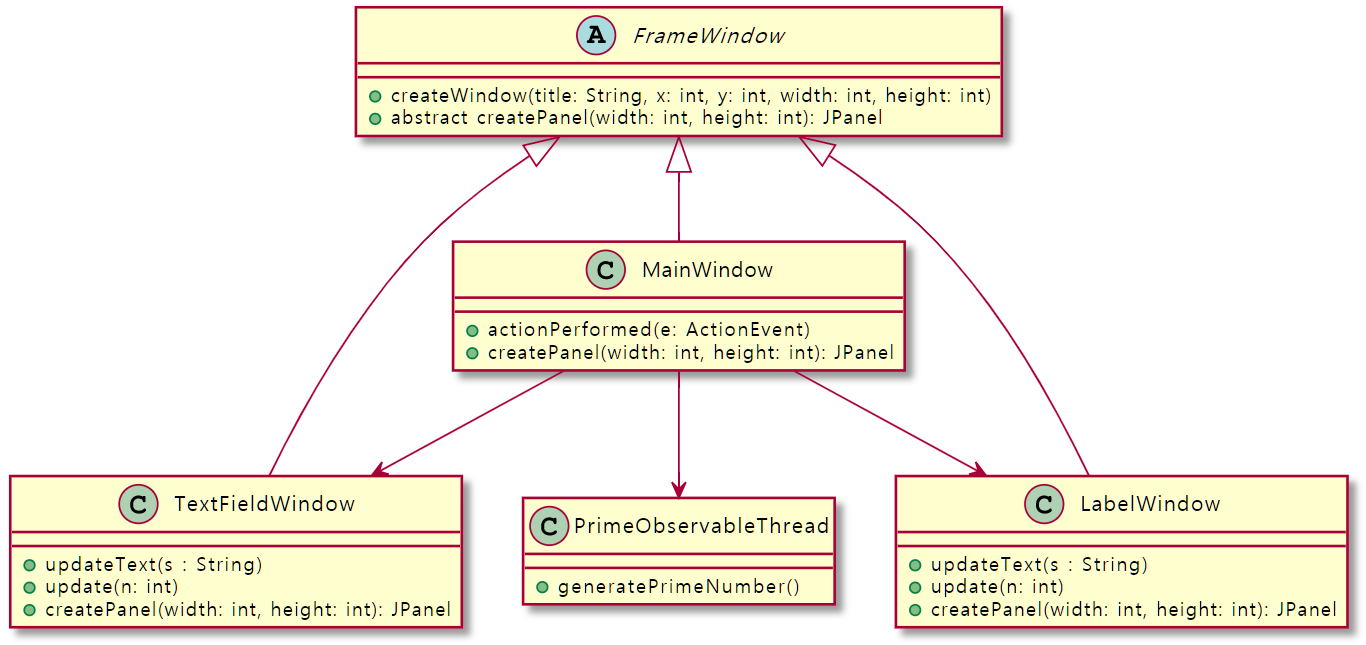


MP02 참조 – Comparble이 알고리즘 인터페이스 Comparable…이 각 기능별로 구현한 형태, sorter에서 객체를 받아서 해당 알고리즘을 활용하여 정렬하는 형태

Observer pattern (옵저버 패턴) MP03

이벤트 발생시 동작하는 Push 서비스를 구현하는 패턴 -> event driven 방식





Event 발생은 받는 방법 -> Jbutton

이벤트 발생시, 이벤트를 if문으로 확인하고, 해당 event에 해당하는 함수 호출.

**public void actionPerformed(ActionEvent e) {  
 // e.getSource()가 어떤 Jbutton에서 일어난 건지 확인하고, 그에 맞는 작업 실행.  
 if (e.getSource() == updateTextFieldObserverButton) {  
 if (updateTextFieldObserverButton.getText().equals(*TEXTFIELD\_OBSERVER\_BUTTON\_TITLE*)) {  
 updateTextFieldObserverButton.setText(*TEXTFIELD\_OBSERVER\_BUTTON\_TITLE\_REMOVE*);  
 primeThread.addObserver((Observer) textFieldWindowObserver);  
 } else {  
 updateTextFieldObserverButton.setText(*TEXTFIELD\_OBSERVER\_BUTTON\_TITLE*);  
 textFieldWindow.updateText("" + primeThread.getPrimeNumber());  
 primeThread.removeObserver((Observer) textFieldWindowObserver);  
 }  
 }**

이런형태로 구현됨,

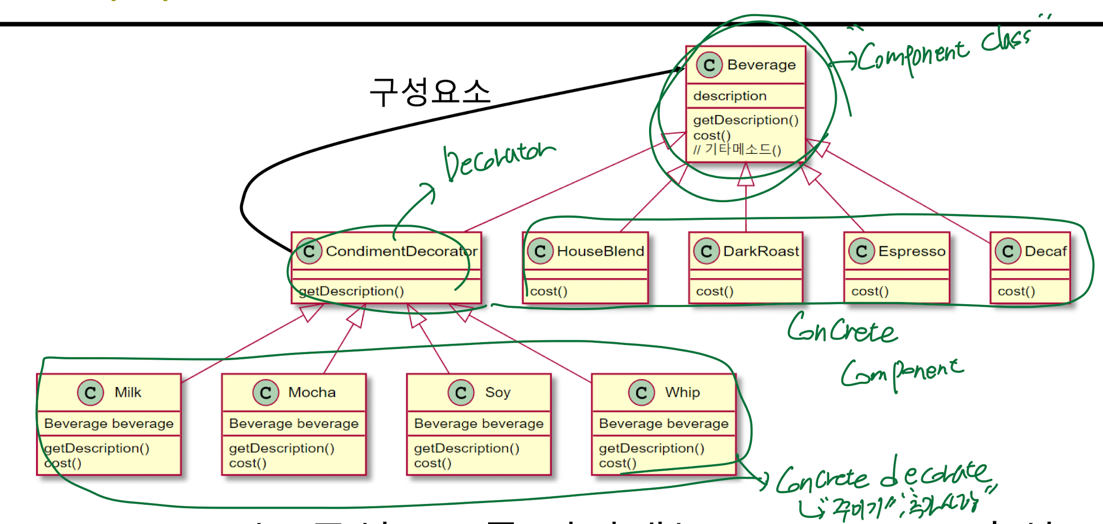
public class MainWindow extends FrameWindow implements ActionListener

Main에서 ActionListenr 상속 override actionPerformed -> event 사용시, 적용됨.

Decorator pattern (데코레이터 패턴) MP04

객체에 추가적인 기능을 Dynamic하게 추가하기 위해서 구현되는 패턴

Ex) 음식에 재료 추가하는 방식,



* Component Class는 기본적인 기능을 가지고 있다.
* Concrete Component는 Component를 상속받아 구현한 클래스들로, 가장 안쪽에 들어가는 Class라고 생각하면 된다. (메인 메뉴)
* Decorator는 Component를 구현한 Class이다. 크게 기능은 없고, Concrete Decorator들에게 상속된다.
* Concrete Decorator는 Decorator를 구현한 클래스들로, 주변을 감싸는 Class이다. 바로 자신의 값을 return하지않고, 자신이 감싸고 있는 객체의 해당 함수를 사용한 후 추가로 자신의 기능을 더한다.

텍스트이(가) 표시된 사진

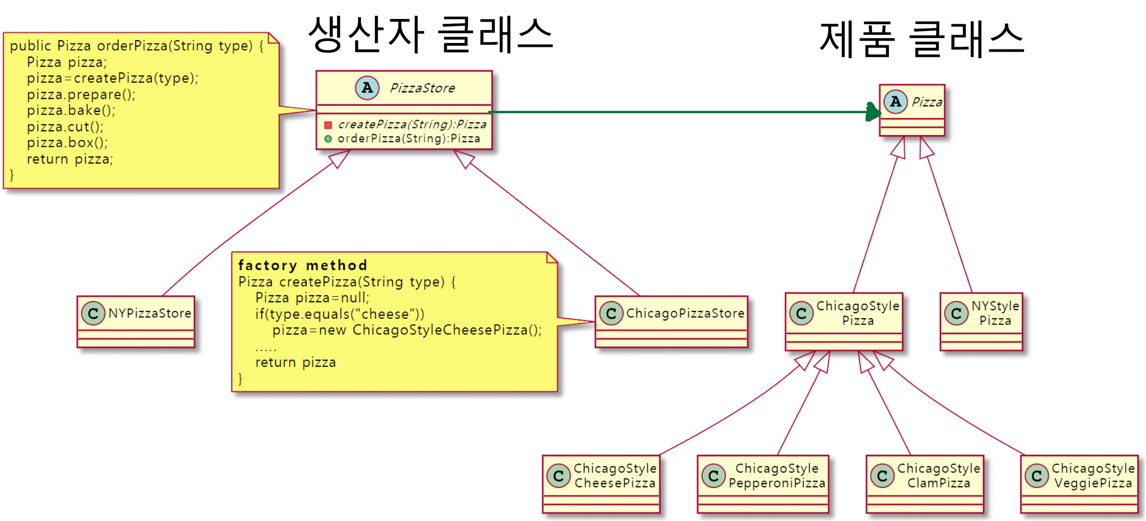
자동 생성된 설명

Return 하는 값에 해당 Class에서의 가격에 해당클래스로 감싸져 있는

클래스들의 cost 함수에서 return 받은 cost.

Abstract Factory pattern (추상 팩토리 패턴) – MP05

객체 생성을 분리하여 캡슐화 -> 객체 생성 부분을 빼서, if문으로 만들기.



위 그림처럼 생산자 Class에서 제품 Class를 만들때, 데이터를 넣어주면, if문으로 해당 데이터에 맞는 상세 제품 Class 객체를 생성하여 전달해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위의 이미지처럼 제품 객체(parent)를 만들고 내부에 상세 제품(Child)를 할당하여 Return

Singleton pattern(싱글턴 패턴)

* 하나의 Class로 하나의 객체만 만들 수 있도록 하는 것.
* 생성자를 Private으로 구성하고, getInstance()에서 생성자 실행.

텍스트이(가) 표시된 사진

자동 생성된 설명

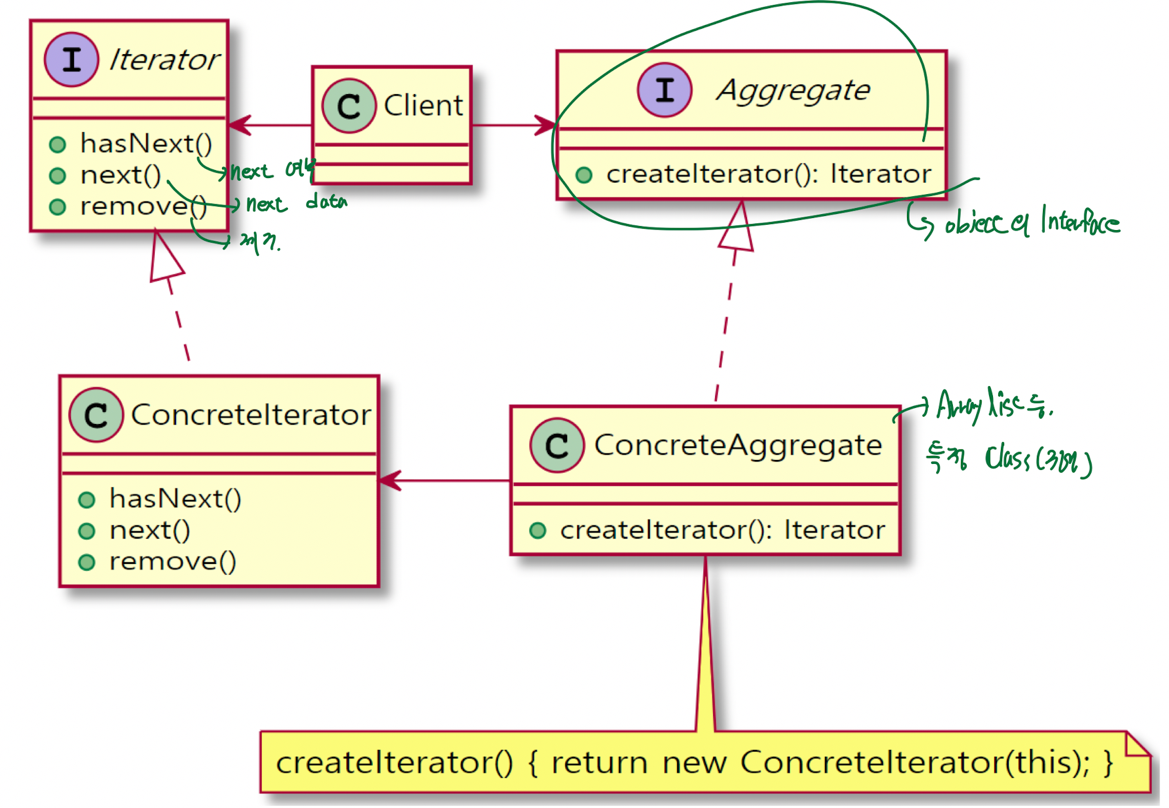
위의 이미지 처럼 Multi-Thread 상황에서 동시에 접근시 Sychronized해줘야 동시에 여러개의

객체가 생기는 것을 막을 수 있음

Ex) 초콜릿 공장…

Iterator pattern(반복자 패턴) - DynamicArray

* 여러 요소를 가지는 객체에 대한 이해 없이, 각 요소에 한번씩 접근 가능하게 도와주는 패턴
* 접근기능과 자료구조를 분리시켜서 객체화
* 서로 다른 구조를 가지고 있는 저장 객체에 대해서 접근하기 위해서 interface를 통일시키고 싶을 때 사용하는 패턴



추가사항

Abstract Factory pattern + Decorator pattern + observer pattern 으로 주문시스템 만들기.

Abstract Factory pattern의 제품 부분에 추가로 Decorator 붙이고, ConcreteDecorator 구현한 후에,

제품 객체의 Create 부분 뒤에 topping 추가하는 부분 추가 (ArrayList의 길이만큼 토핑 추가)

Ex)

If (pizzaclass가 맞는지) {

Pizza = new pizzaclass(피자 구분 data);

}

-----------------------------------------------------------------------------

for( ) {

pizza = new 토핑class(pizza, toppings[1]);

}

Return pizza