디자인 패턴

남동강

목차

디자인 패턴(Design Pattern)

정의

- 객체 지향 프로그래밍 설계를 할 때 자주 발생하는 **문제들을 피하기 위해 사용되는 패턴**
- **코드 문제 해결을 위해 사용**한 기법들을 모아 만들어 졌다
- 디자인 패턴은 알고리즘이 아니라 상황에 따라 자주 쓰이는 **설계 방법을 정리한 코딩 방법론**
- **의사소통 수단의 일종**

특징

- 여러 사람이 협업해서 개발할 때 새로운 기능을 추가해야 하는데 의도치 않은 결과나 버그를 발생하기 쉽고 최적화가 어려운데 이런 문제를 해결하는데 도움이 된다

주의점

- 디자인 패턴을 맹신하여 모든 문제를 패턴을 써서 해결하려 들지 않도록 조심
- **코드 베이스의 간결성이 중요**
- 디자인 패턴에 얽매이는 것보단 그 패턴이 왜 효율적인 방식인지를 이해가 중요

싱글톤 패턴(Singleton Pattern)

- **하나의 프로그램** 내에 **하나의 인스턴스만을 생성**해야 할 경우 사용
- **클래스 내에서 인스턴스가 단 하나뿐임을 보장**하므로, 프로그램 전역에서 해당 클래스의 인스턴스를 바로 얻을 수 있고, **불필요한 메모리 낭비를 최소화**한다
- 생성자를 다른 곳에서도 사용할 수 있으면 그 곳에서도 인스턴스를 만들 수 있기 때문에, 이 패턴에서는 **생성자를 클래스 자체만 사용할 수 있도록 반드시 private 등의 접근 제한자를 통하여 제한**하여야 한다

싱글톤 패턴(Singleton Pattern)

```
class Singleton
{
private:
    static Singleton * m_pInstance;

    Singleton() { }
Public:
    ~Singleton() { }

    static Singleton* get_instance()
    {
        if (NULL == m_pInstance) m_pInstance = new Singleton;
        return m_pInstance;
     }

    void Destroy()
    {
        if (m_pInstance)
        {
            delete m_pInstance;
            m_pInstance = NULL;
        }
    }
};
Singleton* Singleton::m_pInstance = NULL; // static 맴버 변수 초기화.

Singleton :: get_instance()->…
```

연습문제

학생 관리 문제.txt 파일을 이용하여
학생을 관리하는 **Std_manager**를
싱글톤 패턴으로 수정

옵저버 패턴(Observer Pattern)

- **일 대 다의 관계**를 가진다
- 관찰자(observer)가 **관찰하던 대상**(subject)의 상태가 **갱신**되면 **모든 관찰자**(observer)**들**은 갱신된 내용을 알고 **자신**(observer)**의 상태를 갱신**한다
- **관찰자(observer)**와 **관찰대상(subject)**으로 구성
- **데이터 전달 방식**
  - **대상에서 옵저버**에게 데이터를 **보내는** 방식**(푸시 방식)**
  - **옵저버에서 대상**의 데이터를 **가져오는** 방식**(풀 방식)**

옵저버 패턴(Observer Pattern)

```
class Observer
{
public:
    Observer() {}
    virtual ~Observer() {}

    virtual void update(…) abstract;
};

class Subject
{
private:
    std::vector<Observer*> observer;
public:
    Subject() {}
    virtual ~Subject() { }

    virtual void add(Observer*) abstract;
    virtual void remove(Observer*) abstract;
    virtual void notify() abstract;
};
```

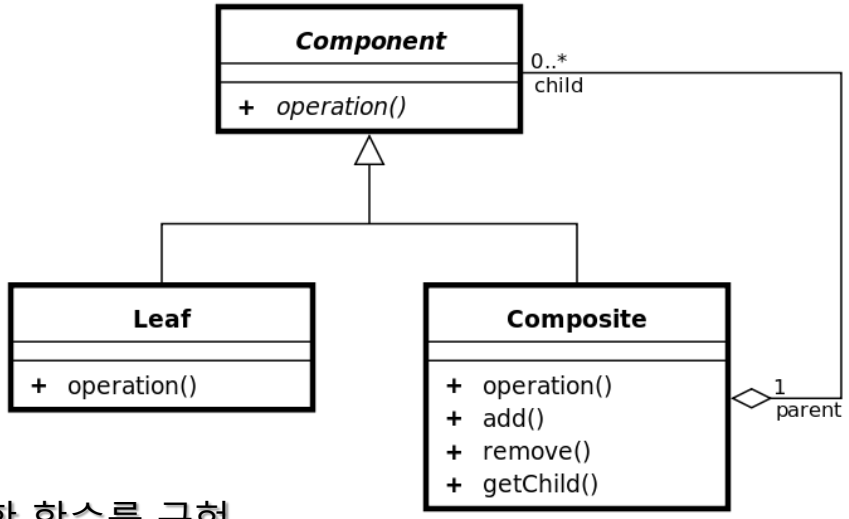
연습문제

옵저버 패턴 문제.txt 파일을 보고
옵저버 패턴 문제.exe 처럼 **출력**

컴포지트 패턴(Composite Pattern)

- 사용자가 **단일 객체와 복합 객체 모두 동일**하게 다루도록 한다
- **객체들의 관계**를 **트리 구조로 구성**하여 **부분-전체 계층을 표현**

- **Component**
  - 모든 표현할 요소들의 **추상적인 인터페이스**
- **Leaf**
  - Component로 지정된 **인터페이스를 구현**
- **Composite**
  - **Component를 지니고 관리**하며, 관리를 위함 함수를 구현



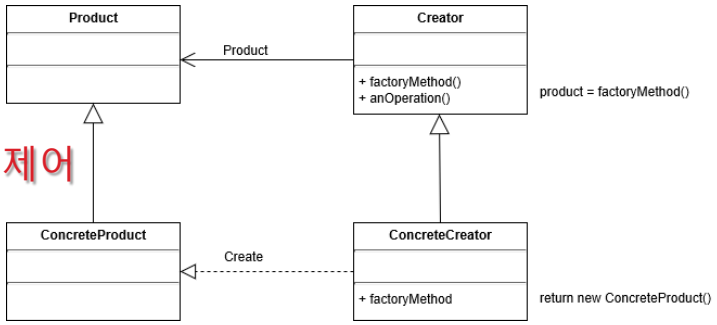
컴포지트 패턴(Composite Pattern)

예제소스/컴포지트 패턴.cpp 참고

팩토리 패턴(Factory pattern)

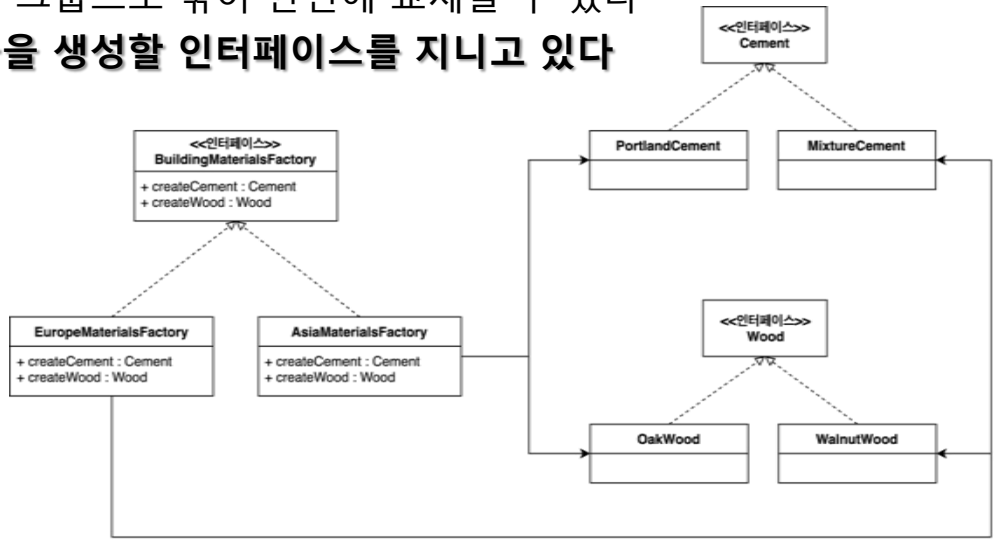
팩토리 메소드(Factory Method)

- **객체를 생성**하여 **반환**하는 함수
- 초기화 과정을 외부에서 보지 못하게 숨기고 반환 타입을 제어
- 생성 후 해야 **할 일의 공통점** 정의
- 생성해야 할 **객체가 한 종류**



추상 팩토리(Abstract Factory)

- 많은 수의 연관된 서브 클래스를 특정 그룹으로 묶어 한번에 교체할 수 있다
- 연관된 또는 독립된 **객체들의 타입 군을 생성할 인터페이스를 지니고 있다**
- 생성해야 할 **객체 군의 공통점**에 유의
- 생성해야 할 **객체가 여러 종류**



유한 상태 기계(Finite State Machine)

상태 디자인 패턴

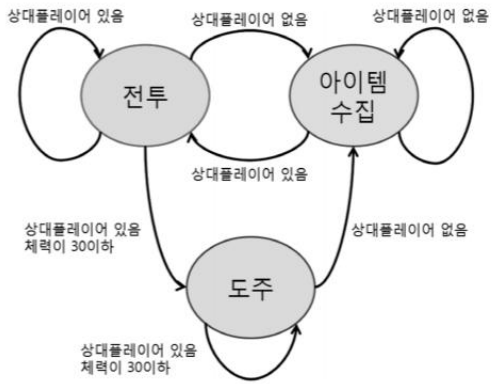
- **상태 구동형(state driven) 행동**을 구현하는 세련된 방법

유한 상태 기계(FSM)

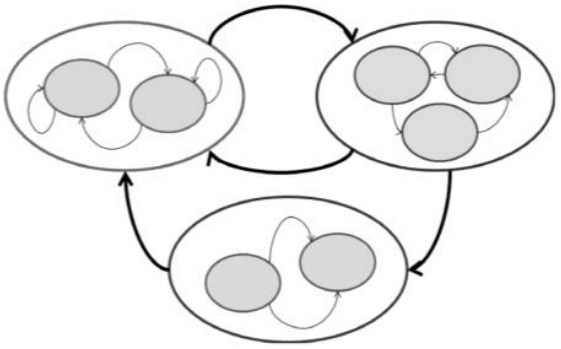
- 간단히 **상태 기계**라고 한다
- 컴퓨터 프로그램과 전자 논리 회로를 설계하는 데에 쓰이는 수학적 모델
- **유한 개의 상태**를 가지고 있는 추상 기계
- **주어지는 입력(조건)에 따라** 어떤 **상태에서 상태로 전환**시키거나 출력, 액션이 일어나게 하는 모델(기계)

특징

- 빠르고 코딩하기 쉽다
- 오류 수정이 용이하다
- 계산 부담이 없다
- 직관적이다
- 유연성이 있다



상태가 세 개인 FSM

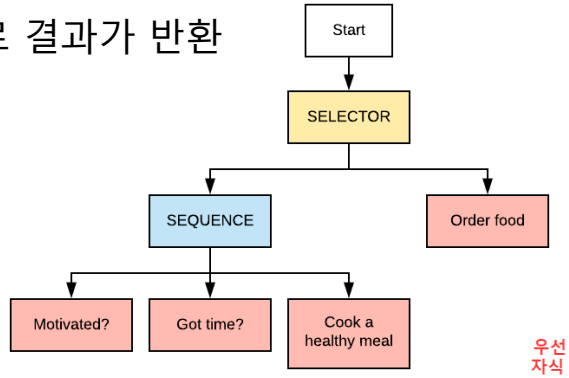


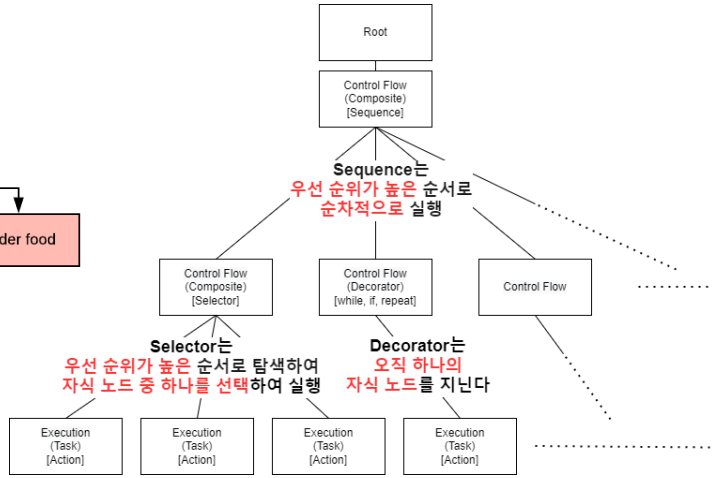
계층적 FSM

유한 상태 기계(Finite State Machine)

- **단점**
  - 규모가 커지면 설계가 복잡해진다
  - 제한된 범위의 문제에만 적용 가능하다
  - 단순 계산 작업과 같은 간단한 작업에는 부적합하다
  - 출력이 예측하기 쉽기 때문에, 출력에 있어 불확실성이 요구되는 경우 적합하지 않을 수 있다

- **행동 트리(Behavior Tree)**
  - 기존의 FSM의 문제점을 보완하여, 보다 다양한 AI 패턴을 구현하는데 최적화 되어 있다
  - 깊이 우선 탐색 구조
  - 자식 노드에서 부모 노드로 결과가 반환





연습문제

연습문제/FSM 폴더에 있는 **소스 코드**를 **참고**
자신만의 **새로운 Agent 제작**