[디자인패턴] Mediator, Singleton

- 정의

- 목적

- 다이어그램

- 사용 방법

- 활용 예시

+) 소스코드 작성, 템플릿으로 만들어두면 좋음

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

디자인 패턴

- 소프트웨어 디자인 패턴(Software Design Pattern)은 소프트웨어 공학에서 특정 문맥에서 공통적으로 발생하는 문제에 대해 재사용 가능한 해결책

- 소스나 기계 코드로 바로 전환될 수 있는 완성된 디자인은 아니며, 다른 상황에 맞게 사용될 수 있는 문제들을 해결하는데 쓰이는 서술이나 템플릿

- 프로그래머가 어플리케이션이나 시스템을 디자인할 때 공통된 문서들을 해결하는데 쓰이는 형식화된 가장 좋은 관행

창조 패턴(Creational Patterns)

구조 패턴(Structural Patterns)

행위 패턴(Behavioral Patterns)

동시실행 패턴(Concurrency Patterns)

디자인 원칙

- 애플리케이션에서 달라지는 부분을 찾아내고, 달라지지 않는 부분으로부터 분리시킨다

- 구현이 아닌 인터페이스에 맞춰서 프로그래밍한다

- 상속보다는 구성을 활용한다

객체지향의 기초

추상화, 캡슐화, 다형성, 상속

훌륭한 객체지향 디자인 : 재사용성, 확장성, 관리의 용이성

객체지향 원칙

바뀌는 부분을 캡슐화한다

상속보다는 구성을 활용한다

구현이 아닌 인터페이스에 맞춰서 프로그래밍 한다

서로 상호작용을 하는 객체 사이에서는 가능하면 느슨하게 결합하는 디자인을 사용해야 한다

클래스는 확장에 대해서는 열려 있지만 변경에 대해서는 닫혀 있어야 한다(OCP)

추상화된 것에 의존하라. 구상클래스에 의존하지 않도록 한다

친한 친구들하고만 이야기 한다

먼저 연락하지 마세요. 저희가 연락드리겠습니다

어떤 클래스가 바뀌게 되는 이유는 한가지 뿐이어야만 한다

매디에이터 패턴

- 상태가 바뀔 때마다 메디에이터에게 알려줌

- 메디에이터에서 보낸 요청에 응답을 함

alarm coffee

Mediator

calender sprinkler

Mediator

if(alarmEvent)

{ CheckCalender(); CheckShower(); CheckTemp() }

if(weekend) { checkWeather() }

if(trashDay) { resetAlarm() }

- 메디에이터 패턴의 장점 : 시스템하고 각 객체를 분리시킴으로써 재사용성을 획기적으로 향상시킬 수 있음

- 제어 로직을 한 군데 모아놨기 때문에 관리하기가 수월함

- 시스템에 들어있는 객체 사이에서 오가는 메시지의 종류를 확 줄이고 단순화시킬 수 있음

- 메디에이터 패턴의 단점 : 디자인을 잘 하지 못하면 메디에이터 객체 자체가 너무 복잡해 질 수 있음

- 매디에이터 패턴의 활용 : 서로 연관된 GUI 구성요소들을 관리하기 위한 용도

싱글턴 패턴

- 해당 클래스의 인스턴스가 하나만 만들어지고, 어디서든지 그 인스턴스에 접근할 수 있도록 하기 위한 패턴

- 클래스 인스턴스가 하나만 만들어지도록 하고, 그 인스턴스에 대한 전역 접근 제공

- 다중 스레드를 사용하는 경우 속도와 자원 문제를 파악한 후 적절하게 구현해야 함

- 클래스 로더가 여러 개 있을 경우, 싱글턴이 제대로 작동하지 않고 여러 개의 인스턴스가 생길 수 있음

singletone

static uniqueInstance

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

static GetInstance()

+)

장점 :

단점 : 클래스의 단독 인스턴스가 실제로 필요하지 않은 상황에서 불필요한 제한을 도입, 응용 프로그램에 전역 상태 도입

일반적인 용도 : 상태 객체

전역변수보다 선호되는 경우 존재

1) 불필요한 변수로 전역 네임 스페이스(또는 중첩 네임 스페이스가 있는 언어, 포함 네임 스페이스)를 오염시키지 않음

2) 지연 할당 및 초기화를 허용하는 반면 많은 언어의 전역변수는 항상 리소스를 소비함