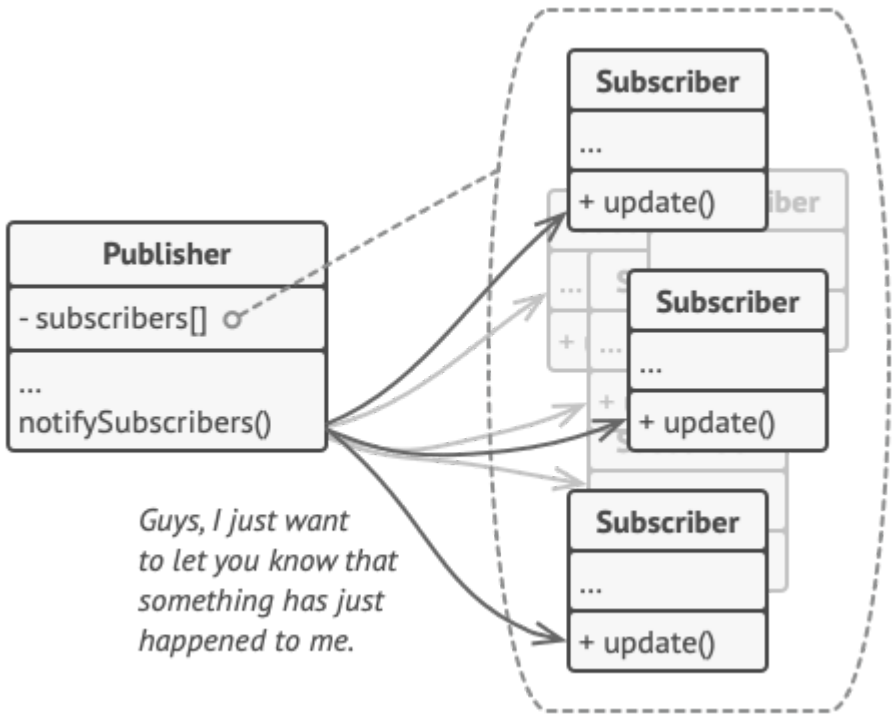
옵저버 패턴

개요

행동패턴

옵저버 패턴(Observer Pattern)은 객체 간에 일대다(one-to-many) 의존 관계를 정의하여, 어떤 객체의 상태가 변할 때 그 객체에 의존하는 다른 객체들이 자동으로 알림을 받아 상태 변화에 대응할 수 있도록 하는 디자인 패턴 중 하나입니다. 이 패턴은 주로 분산 이벤트 핸들링 시스템, MVC(Model-View-Controller) 아키텍처, GUI 프로그래밍 등에서 사용됩니다.



장점:

1. **느슨한 결합 (Loose Coupling):**
    - 주제와 옵저버는 서로 독립적으로 존재하며, 주제는 옵저버에 대한 정보를 알 필요가 없습니다. 이로써 시스템의 유연성이 증가하고, 객체 간의 결합도가 낮아져서 코드의 재사용성이 증가합니다.
2. **동적인 변경 가능성:**
    - 새로운 옵저버를 추가하거나 기존 옵저버를 제거하기가 간편하므로, 시스템의 변경에 유연하게 대응할 수 있습니다. 새로운 동작이나 기능을 추가할 때, 기존 코드를 수정하지 않고도 확장이 가능합니다.
3. **이벤트 기반 프로그래밍:**
    - 주제의 상태 변경이 이벤트처럼 다수의 옵저버에게 알려지기 때문에, 이벤트 기반 프로그래밍에 적합합니다. 이는 특히 GUI 프로그래밍이나 분산 시스템에서 유용합니다.
4. **분산 이벤트 핸들링:**
    - 여러 컴포넌트나 모듈이 분산되어 있는 환경에서 이벤트 발생 시 그에 대응하는 동작을 수행하기에 유용합니다.

단점:

1. **순서에 대한 의존성:**
    - 옵저버들 간의 실행 순서에 의존성이 있을 경우, 예측이 어려울 수 있습니다. 따라서 순서가 중요한 경우 디자인을 신중하게 해야 합니다.
2. **성능 이슈:**
    - 옵저버 패턴을 사용하면서 옵저버가 많아질 경우, 모든 옵저버에게 통지할 때까지 시간이 오래 걸릴 수 있습니다. 이로 인해 성능에 영향을 줄 수 있습니다.
3. **메모리 누수:**
    - 주제와 옵저버 간의 순환 참조가 발생할 수 있어 메모리 누수의 가능성이 있습니다. 이에 대한 관리가 필요합니다.
4. **디버깅 어려움:**
    - 어떤 옵저버가 어떤 이벤트에 반응하는지를 파악하기 어려울 수 있습니다. 이로 인해 디버깅이 어려울 수 있습니다.

예상 질문:

1. 옵저버 패턴이 어떤 상황에서 사용되나요?
2. 프록시 패턴을 사용하여 일어나는 상호작용에 대해 설명할 수 있나요?
3. 옵저버 패턴을 사용한 실제 프로젝트 경험이나 예시를 제공할 수 있나요?

출처:

https://refactoring.guru/design-patterns/observer

https://inpa.tistory.com/entry/GOF-💠-옵저버Observer-패턴-제대로-배워보자

https://youtu.be/bdcxCpB68Xs?si=JGyFIxVJWALW05tM