참고 자료

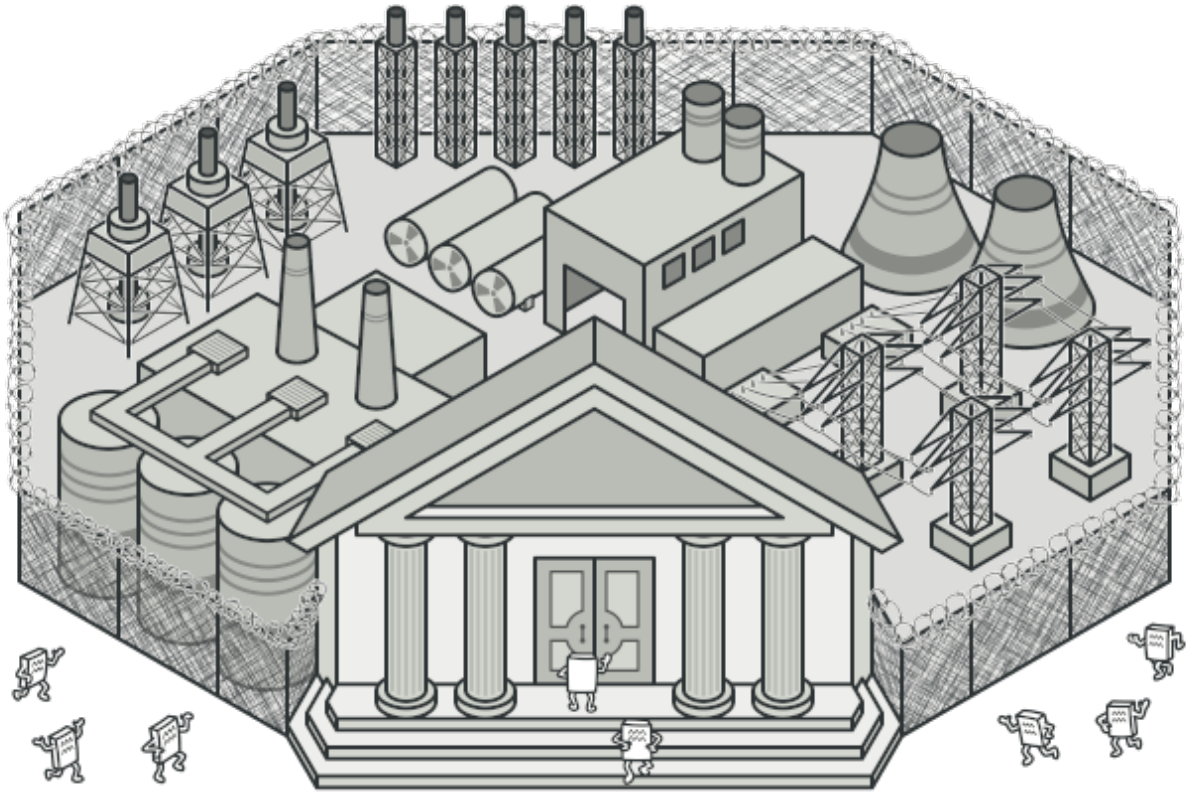
- https://refactoring.guru/design-patterns/facade

  여기서 설명하는 인터페이스는 자바의 인터페이스가 아니라 API, GUI에서 I에 해당하는 의미에 가깝습니다.

> *인터페이스*(*interface*)는 서로 다른 두 개의 시스템, 장치 사이에서 정보나 신호를 주고받는 경우의 접점이나 경계면

FacadePattern이란?

- structural design pattern
- library나 framework, 또는 다른 복잡한 클래스 집합(또는 서브시스템)에게 간소화된 인터페이스를 제공하는 패턴입니다.



상황

- Application이 여러 복잡한 라이브러리 및 프레임워크의 object들을 이용해서 작업을 해야 합니다.

문제

Application에서 직접 라이브러리와 프레임워크를 사용한다면 Application의 비즈니스 로직이 라이브러리와 매우 강하게 결합하게 됩니다.

해결방법

Application이 concrete class에 의존하고 있어 발생한 문제입니다.

따라서 Application과 다른 object간에 연결 시켜주는 class를 만듭니다. 이때, Application이 사용하고 있는 objects의 공통 인터페이스를 만들어주는 것이 핵심입니다.

해당 클래스를 구현한 오브젝트를 **facade object**라고 부르게 됩니다.

이와 같이 하면 2가지 이점이 존재합니다.

이점

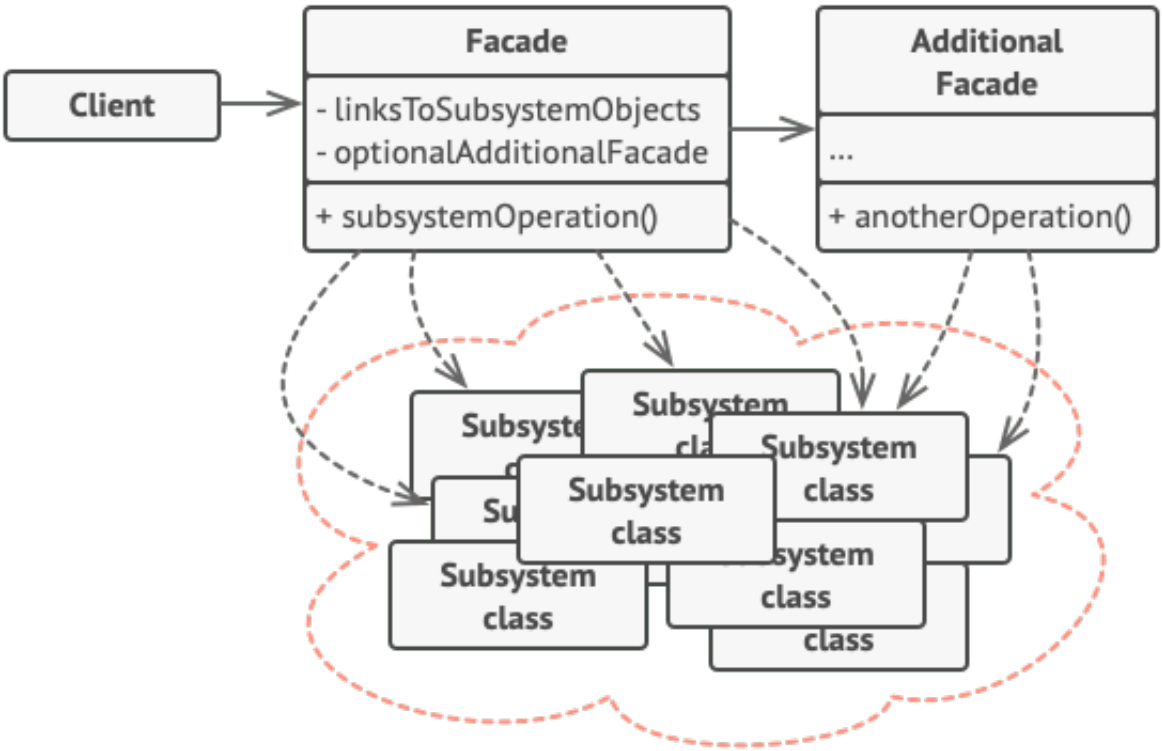
1. interface역할을 하는 클래스에 의존함으로써 결합을 낮출 수 있습니다.
2. 복잡한 로직을 interface에 숨길 수 있습니다. (bridge pattern의 asbtraction과 동일)

   예를 들어, 이미지 파일을 올리기 위해 보정하고, 압축하는 과정이 있습니다. 이를 클라이언트 관점에서 인터페이스를 간소화하여 정의하자면, '업로드할 이미지를 만들어 주세요' 라고 추상화할 수 있습니다.

   이를 ''*문맥에 따른 인터페이스*''라고 표현합니다.

   (provide a context-specific interface to more generic functionality)

구조



1. Facade

특정 서브 시스템 접근을 편리하게 도와줍니다. 그래서 클라이언트 요청을 어디에 연결시켜줄지, 그리고 서브시스템의 부분들을 어떻게 동작시킬지 알고 있습니다.

2. Additional Facade

단일 Facade가 로직을 만드는 중에 관련성이 떨어지는 features에 의해 오염되는 것을 막기 위해 만들 수 있습니다. 이는 client와 다른 Facade에 의해 사용될 수 있습니다.

3. Complex Subsystem

facade에 의해 사용되는 부분들입니다. 의미있는 작업을 위해 여러 부분들이 사용될 수 있죠. 그리고 이들은 facade에 의해 사용되는 부분들이라 subsystem은 facade의 존재를 모릅니다.

4. Client

클라이언드는 facade를 통해 subSystem object에 접근합니다.

코드 예시 작성