참고자료

- https://refactoring.guru/design-patterns/bridge
- https://en.wikipedia.org/wiki/Bridge_pattern
- Design Patterns: Elements of Reusable Object-Oriented Software

  (link) https://archive.org/details/designpatternsel00gamm/page/151

BridgePattern이란?

- structural design pattern

guru 정의

뚱뚱한 클래스나 매우 연관된 클래스를 두 개의 계층으로 분리하는 패턴

GoF-DesignPattern 정의

구현된 클래스(implementation)와 abstraction클래스가 독립적으로 변하도록 하기 위해서 두 클래스를 분리하는 패턴

(Decouple an **abstraction** from its **implementation** so that the two can vary independently.)

Abstraction과 Implementation

- abstraction과 implementation은 **로버트 마틴이 말한 의미 그대로**입니다.
- DIP에서 나온 설명에서 중 Keyboard와 Printer는 implementation에 해당하고 Copy는 abstraction에 해당한다고 볼 수 있지요.
- 즉, Copy는 high-level-policy로 어플리케이션의 내재된 추상부분에 해당하는 것이고
- copy라는 추상적인 내용을 구체적으로 수행해주는 low-level-policy는 keyboard와 printer였던 것을 떠올리시면 됩니다.
- BridgePattern은 더 나아가서 abstraction은 추상 클래스로 두어 high-level-policy의 확장까지 꾀합니다.

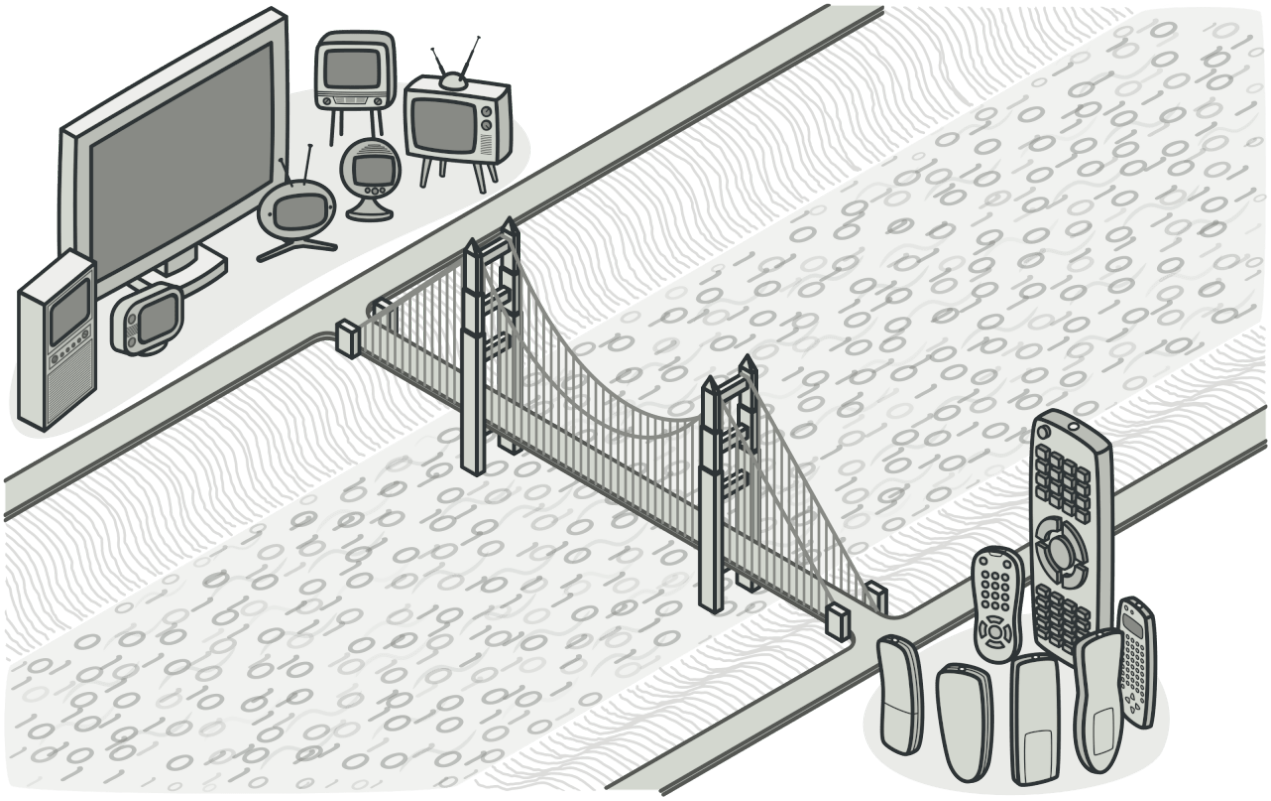
비유 - 부제 (사람은 추상적으로 생각합니다)

1. **숟가락**을 집어 **쌀**을 퍼 먹는다.
2. 숟가락 -> 어디 음식점에 있는 숟가락일 수도 있고, 집에 있는 숟가락일 수도 있습니다. 쌀 또한 특정 시간에 음식점에서 해서 나온 밥일 수 있고, 집에 남아있는 쌀을 먹을 수도 있습니다.

허기를 달래기 위해 숟가락을 집어 쌀을 퍼먹는 행위(**추상**)를 합니다.

이 행위는 집에 있는 숟가락(**구체**)를 집어 집에 남아 있는 쌀(**구체**)을 퍼먹었음으로써 달성됩니다.

이때, 어떠한 숟가락, 쌀이 나오더라도 1. 의 의미가 변하지 않지요. 왜냐하면 숟가락와 쌀은 여러 숟가락을 대표하기 때문입니다. 즉 다형성을 띈다 볼 수 있습니다.



상황

1. GUI를 통해서 API를 콜하여 사용자의 요청에 응답해주는 프로그램을 고려해봅시다.
2. 이때 API는 여러 OS에 대한 Operation 코드가 존재합니다.

   ex:) mac OS, window OS, Linux OS (GUI로 삭제버튼! 누르면 OS별로 각기 다른 로직을 수행할 것입니다.)
3. 몇 개의 다른 GUI도 생각해봅시다. 이때, GUI abstraction 클래스를 상속받아 admin, regular customer등을 구현합니다.

   ex:) admin, regular customer....

문제

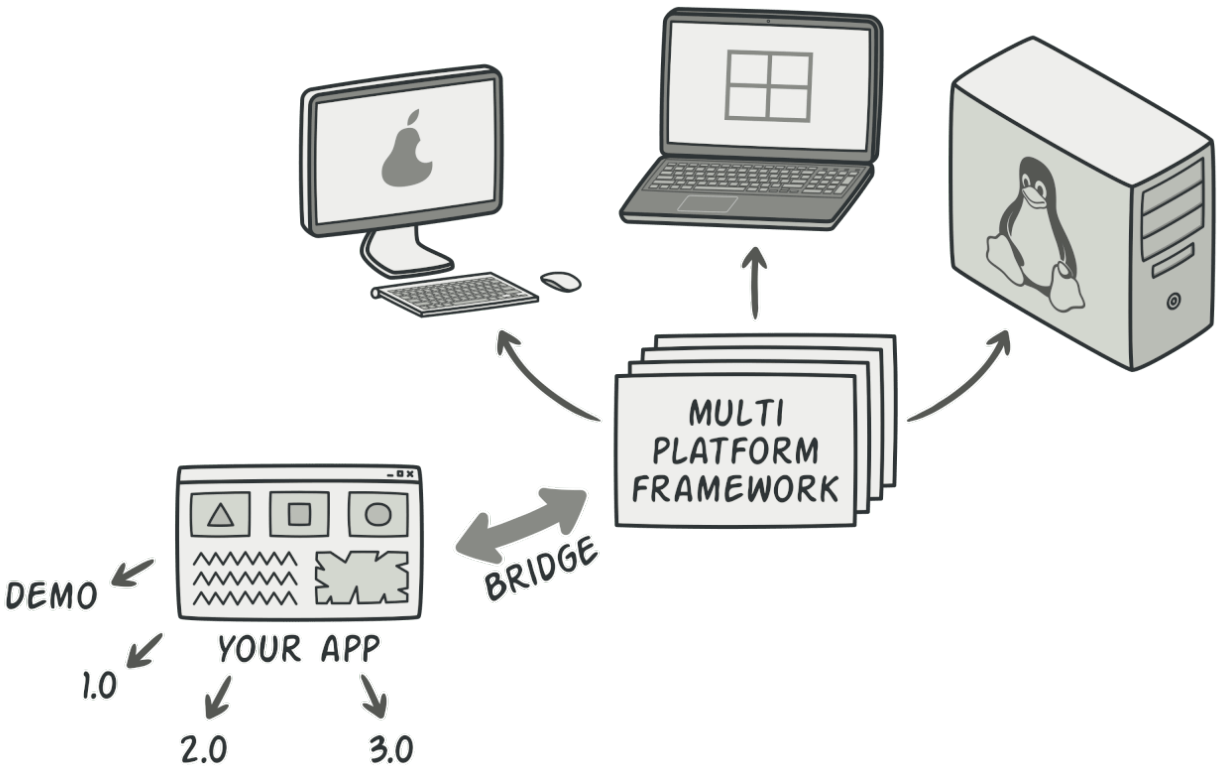
GUI에 대해 API가 작동하는 코드를 만들고 싶습니다.

일일이 다 서브클래스를 생성한다면?

GUI를 상속받아 Window에 대한 admin GUI, Linux에 대한 admin GUI, macOS에 대한 admin GUI 클래스 등 각각에 대해 많은 클래스들을 만들 수 있습니다.

이렇게 일일이 상속받는다면, 결국 GUI 종류 * 운영체제 수 만큼의 **수많은** 서브클래스를 생성해야하는 문제가 발생합니다.

해결 방법

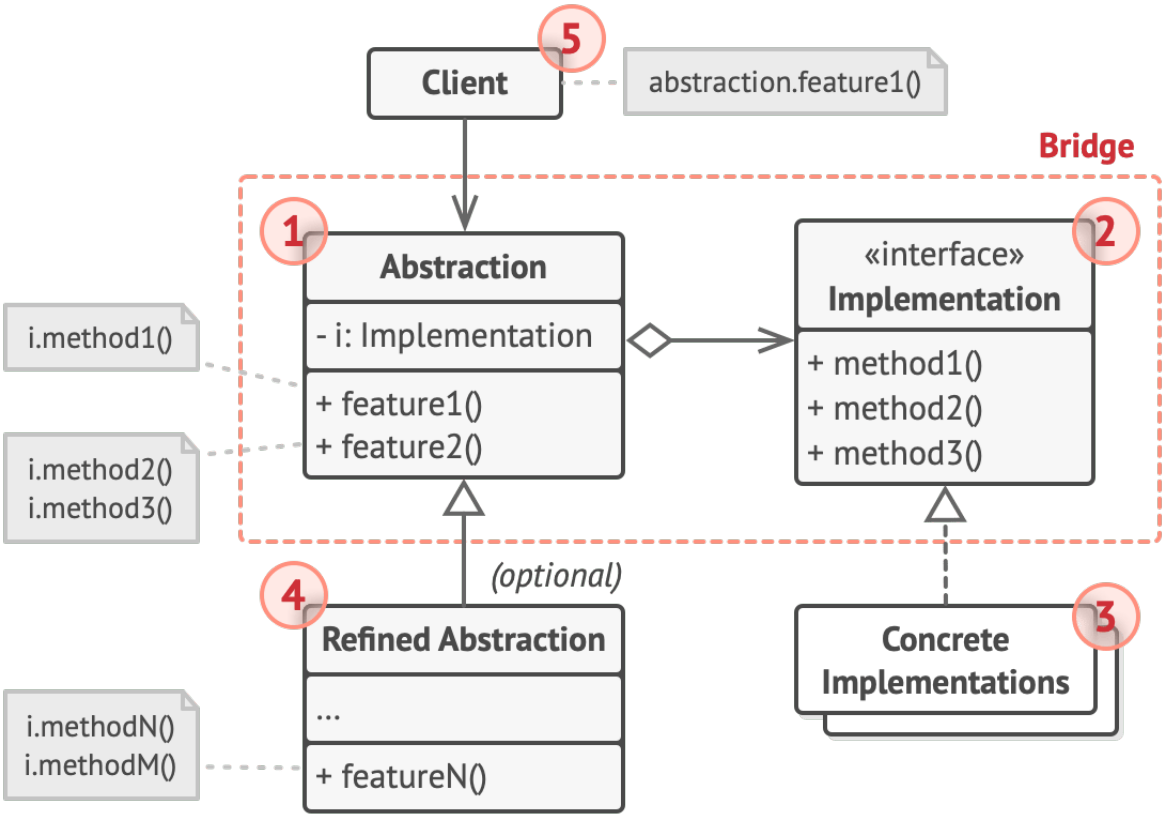


BridgePattern을 이용!

1. 어떤 API(Implementation)가 오든지 간에, GUI는 변하지 않습니다.
2. 따라서 GUI가 high-level-policy라고 말할 수 있고, API는 low-level-policy라고 말할 수 있습니다.
3. 따라서 Low-level-policy를 추상화한 클래스(interface)를 GUI가 갖도록합니다.
4. 그리고 그 GUI가 해당 interface에 의존하도록 합니다.

이렇게 한다면 GUI를 따로 개발하더라도 추상 클래스에 의존하고 있기 때문에 API로직은 변경할 필요가 없을 것이며, 다른 OS를 추가한다 하더라도 GUI를 변경할 필요가 없게 됩니다.

구조



1. Abstraction

1. high-level-policy logic을 제공합니다.
2. low-level-policy에 의존합니다.

2. Implementation

1. low-level-policy 로직에 해당하는 부분입니다.
2. abstraction은 오직 implementation에 작성되어 있는 method를 통해서만 concrete implementaion 과 소통할 수 있습니다.

3. Concrete Implementations

1. platform-specific code가 있습니다.

4. Refined Abstraction

1. 다양한 종류의 컨트롤 로직을 제공합니다.

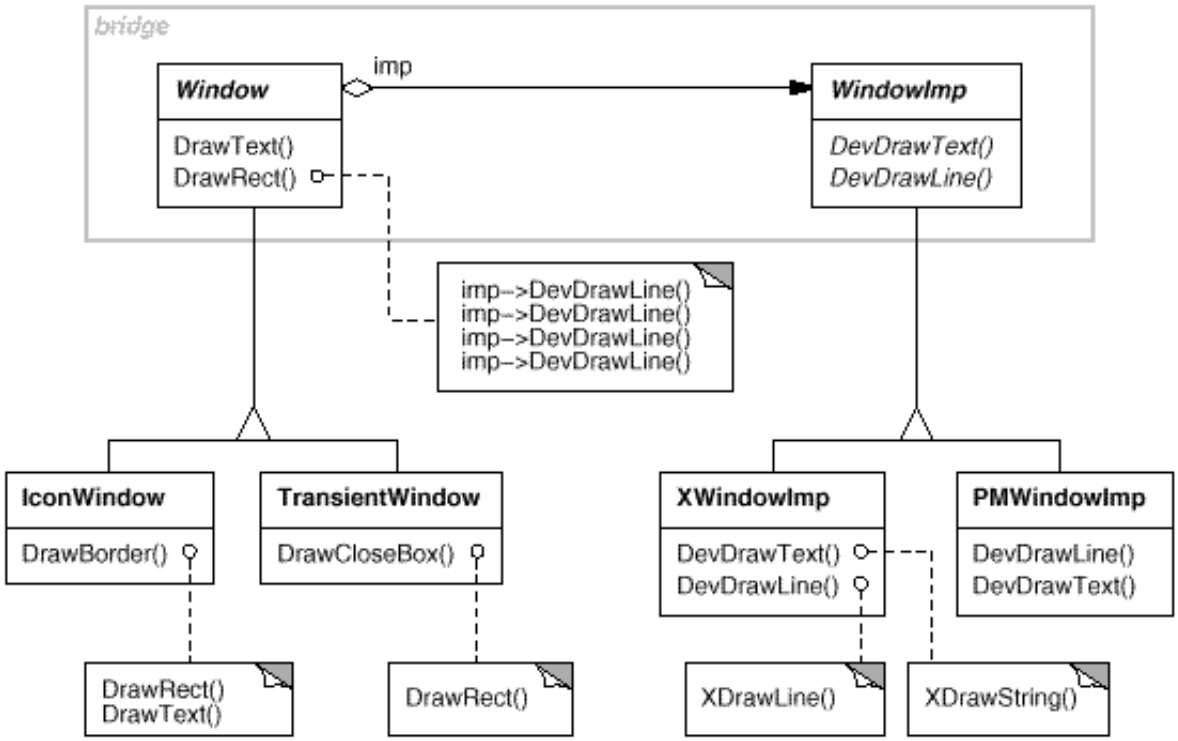
5. Client

1. client는 abstraction과만 일을 하지만, 어떤 implementation objects를 연결할 지는 클라이언트의 책임입니다.

다른 패턴과 연관성

1. Bridge패턴은 다른 두 성격의 클래스가 독립적으로 개발할 수 있도록 도와주는 패턴인 반면에, Adapter 패턴은 기존에 존재하는 코드의 인터페이스가 서로 맞지 않을 경우 적용할 수 있는 패턴입니다.

마지막으로 BridgePattern에 대한 저자의 말



> 여기서 Window는 윈도우 GUI를 의미하고, WindowImp는 해당 기능의 구현체의 인터페이스를 의미합니다. 즉, implementation이지요.

We refer to the relationship between Window and WindowImp as a **bridge**, because it bridges the abstraction and its implementation, **letting them vary independently.**