1️⃣ 단일 책임의 원칙 (SRP - Single Responsibility Principle)

정의

> 한 클래스는 하나의 책임만 가져야 한다는 원칙이다. 다르게 말하면 클래스를 수정할 이유가 오직 하나여야만 한다는 뜻이다.

책임이란?

💡 객체에 의해 정의되는 응집도 있는 행위의 집합으로 객체가 유지해야 하는 정보와 수행할 수 있는 행동에 대해 개략적으로 서술한 것을 책임이라고 한다. 즉, 객체의 책임은 무엇을 알고 있는가와 무엇을 할 수 있는가로 구성된다.

책임을 판단하는 기준

- 메소드가 증가한다고 책임이 증가하지 않는다. (같은 부류의 메소드들이 증가하는 것은 책임의 수가 변하지 않는다.)
- 메소드의 변경을 유발하는 사용자(클라이언트)가 같다면 하나의 책임으로 볼 수 있다. 즉, `변경의 근원`으로 볼 수 있다.
- 액터(동일한 역할을 수행하는 사용자 그룹을 액터라고 한다.)를 기준으로 책임을 나눈다.
- 액터는 여러 명의 사용자라고 하더라도 수행하는 역할이 같다면 같은 액터로 구분한다. (실제 유저와는 다르다. - 특정 역할을 수행할 때 액터라고 부른다.)
- 특정 액터의 요구사항을 만족시키기 위한 일련의 함수의 집합을 클래스로 분류하며 이것을 하나의 책임을 가진다 라고 표현한다. (클래스를 변경하기 위한 사유는 한 액터에

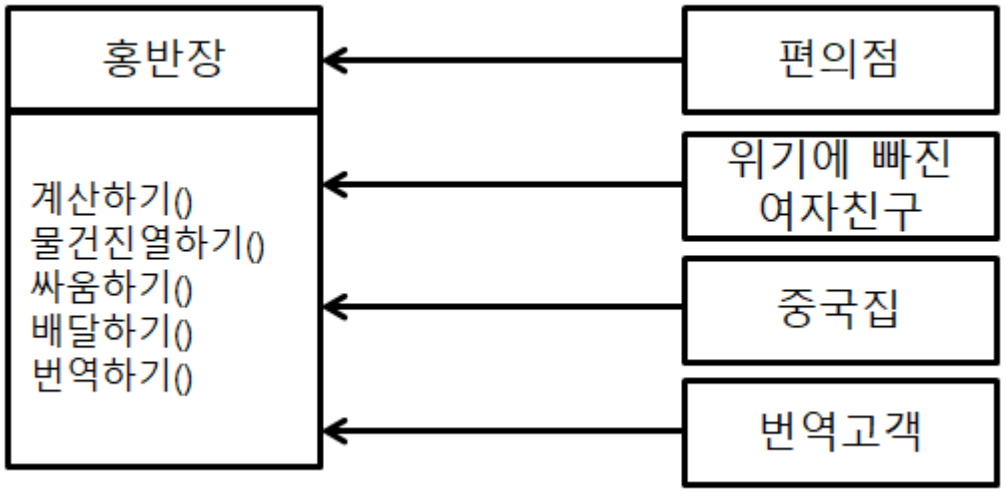
의해서여야만 한다.)

책임이 많은 클래스(Fat Class or Got Class)의 문제점

책임이 많은 클래스는 만능 엔터테이너이다.

2004년에 개봉한 영화 <어디선가 누군가에 무슨일이 생기면 틀림없이 나타난다 홍반장> 이라는 영화를 보면, 홍반장이라는 캐릭터는 동네 반장 역할을 수행하며 동네의 모든 일을 다 해결하는 만능 엔터테이너 역할을 수행한다. 편의점에서 계산하는 아르바이트를 하던, 중국집에서 배달을 하거나, 다국어에도 능통해 번역까지 할 수 있다. 또한 위기에 빠진 여자친구를 구하기 위한 싸움 능력도 탁월하다.

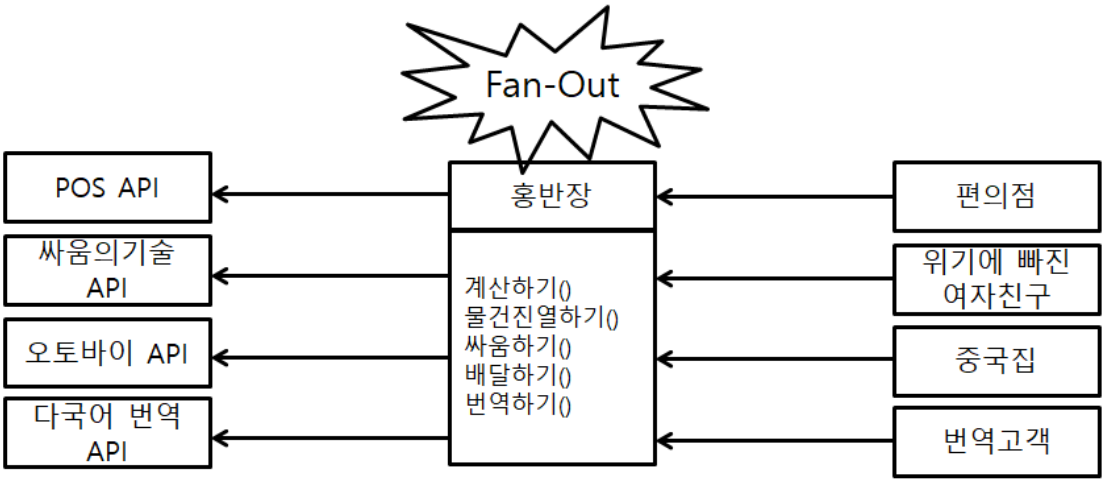
많은 클라이언트에 의존적인 클래스가 발생시키는 변경에 대한 악영향



홍반장은 만능이다. 하지만 많은 일을 할 수 있다고 해서 Fat한 역할을 수행한다고 볼 수 있는 것이 아닌, 판단의 기준은 의존하고 있는 클라이언트 수 이다. 홍반장의 역할에 의존하고 있는 클라이언트는 현재 편의점, 중국집, 번역고객, 그리고 위기에 빠진 여자친구이다. 많은 클라이언트의 요청을 처리해야 하는 많은 책임을 가지는 홍반장은 위기에 빠진 여자친구를 구하기 위해 싸움을 하다가 팔이 부러졌다. 그 영향으로 한 팔로는 오토바이 운전을 하지 못하기에 중국집 클라이언트의 배달하라는 요청을 수행하는데 문제가 발생하게 되었다. 또, 위기에 빠진 여자친구가 갑자기 고맙다며 데이트 요청을 하는 바람에 편의점 아르바이트를 하루 쉬게 되었다.

- 많은 책임을 가진 클래스는 여러 액터에 의해 변경의 근원이 있다. 따라서 한 액터의 변경 사항이 발생 할 시 변경이 발생하지 않은 액터에도 영향을 미칠 수 있다.

많은 책임을 수행하기 위해 너무 많은 정보를 알고 있어야 하는 현상

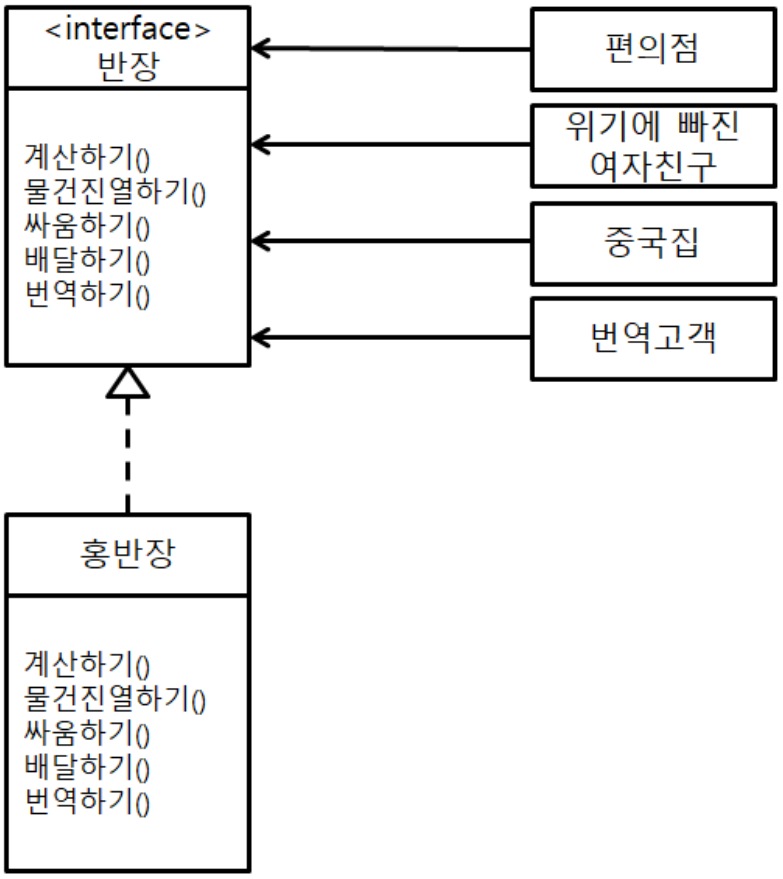


홍반장은 다양한 클라이언트의 요청에 대한 책임을 가지고 역할을 잘 수행하도록 다양한 API 활용 기술을 알고 있다. POS, 싸움의기술, 오토바이, 다국어번역 등의 API의 존재를 알고 그 것을 활용한다. 하지만 한 요청을 처리하기 위해서 불필요한 모든 내용을 알고 있기에 역할에 대한 책임을 수행할 시 잘못된 API를 호출할 가능성도 발생을 하게 된다. 예를 들어 POS API를 이용해 편의점에서 고객의 물건을 계산할 때 갑자기 싸움의 기술을 발휘할 수 도 있다는 말이다.

> 위 두 가지 문제점을 해결하기 위해 가장 좋은 방법은, 한 클라이언트(액터)의 요청의 역할을 수행하는 클래스에게 적절한 책임을 부여하는 것이다.

해결 방법 (책임 분리 과정)

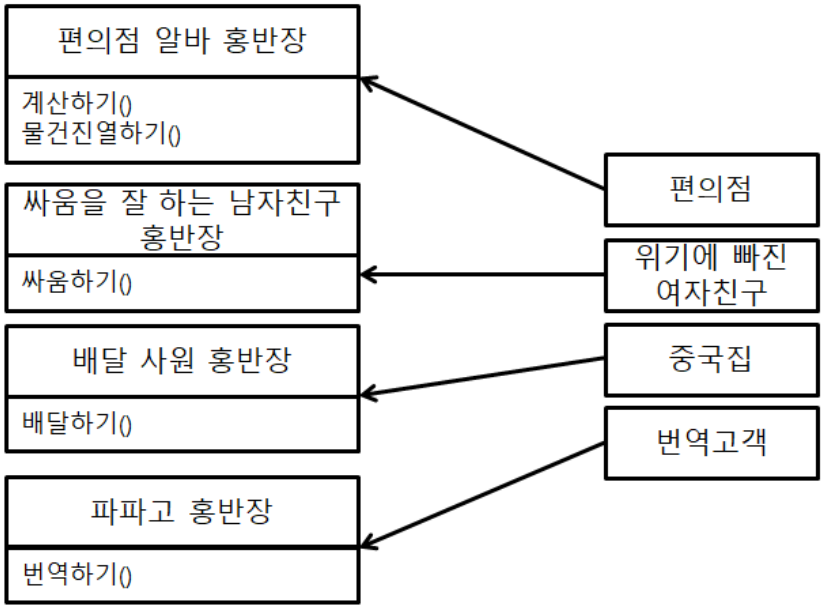
Inverted Dependency



홍반장에게 직접 의존하던 액터의 요청을 반장이라는 역할에 의존하도록 하고 홍반장은 그 역할에 들어온 요청에 대해 실제 처리를 수행하는 형태로 분리하게 되면 홍반장을 대체할 다른 누군가가 오기만 한다면 반장에 의존하던 액터들은 문제가 발생하지 않는다.

> 해결된 부분
> 클래스를 인터페이스와 클래스로 분리하여 컴파일 시점에 클래스간의 결합도를 없애고 런타임 시점에 의존하도록 해 두었다.
>
> 한계점
> 액터들이 하나의 인터페이스이 결합관계를 가지게 된다.

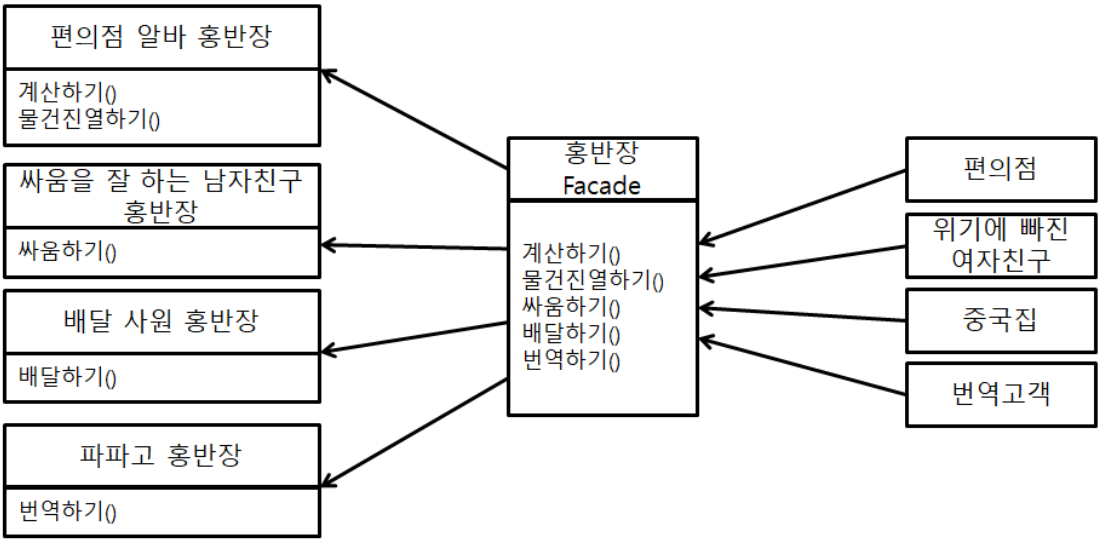
Extract Class



단일책임의 원칙을 고수하고자 잔인하게도 홍반장을 각 책임에 맞게끔 쪼개놓았다. 이렇게 책임을 따로 분리하여 설계를 하게 되면 하나의 홍반장이라는 개념이 4개로 분할이 되는 논리적 문제도 발생하고, 컴파일 시 의존성도 여전히 가지게 된다. 하나의 홍반장이라는 개념이 4개로 분할되는 경우 각 분할된 홍반장의 버젼이 맞지 않으면 문제가 발생할 수 있다.

> 해결된 부분
> 하나의 책임이 변경되어도 다른 책임에는 영향을 미치지 않는다.
>
> 한계점
> 컴파일 시 의존성이 아직 존재하며, 개념적으로 하나의 클래스를 단순 분리시켜 놓은 개념이기 때문에 한 개의 클래스 버젼이 맞지 않게 되면 문제가 발생할 수 있다.

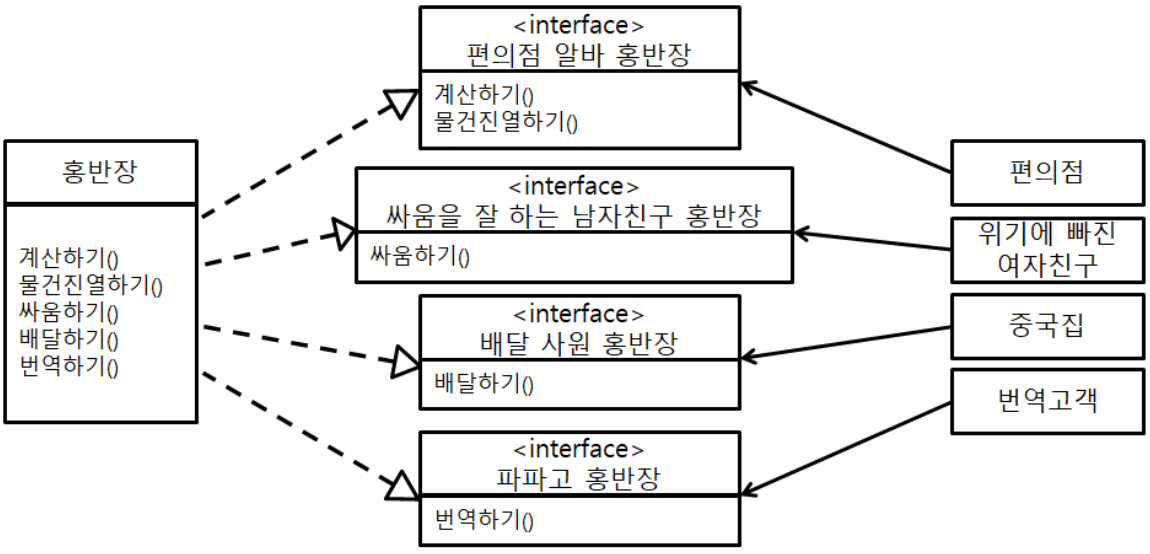
Facade



홍반장은 사실 홍길동의 후예인 것 같다. 본인의 역할을 더 잘 수행할 수 있는 분신들을 만들어서 각 책임을 수행하도록 명령을 내린다. 홍반장 Facade를 하나 두어 개념적인 홍반장은 하나로 묶고, 책임을 수행하기 위한 각 클래스를 분리하게 되면, 홍반장이 역할을 수행할 때 적절한 정보를 알고 하나의 책임을 수행하는 객체를 호출하며 단일책임의 원칙을 준수할 수 있게 되지만, 여전히 액터와 홍반장 Facade 클래스와의 직접적인 의존관계는 남아있는 상태이다.

> 해결된 부분
> 하나의 책임이 변경되어도 다른 책임에는 영향을 미치지 않으며, 홍반장이라는 개념이 같은 클래스를 하나로 관리하여 버전에 따른 홍반장이 달라지는 것을 최소화 하였다.
> 또한 하나의 책임 변경이 어느 클래스에 작성되어 있는지가 명확하기 때문에 수정할 부분을 찾기 쉬워진다.
>
> 한계점
> 컴파일 시 의존성이 아직 존재하여 변경에 취약한 형태가 된다.

Interface Segregation



분신을 이용해 액터의 요청을 수행하는 책임을 나눠서 분신들에게 주다 보니 홍반장은 여전히 액터들은 자기에게 의존하고 있다는 것을 알게 되었다. 그래서 분신을 사용하는 방법을 바꾸어서 인터페이스 분신을 두었다. 인터페이스 분신은 허상이며 실체는 홍반장이다.
액터들은 분신을 이용해서 요청을 하도록 변경하였다. 결국 모든 일처리는 허상이 아닌 실체인 홍반장이 수행하기는 하지만 그것 만큼은 어쩔 수 없었던 것 같다. 만약 홍반장이 일이 바쁘거나 한 액터가 해달라는 요청이 달라지게 되면 그 요청을 받는 분신만 없애거나 다른 분신으로 대체를 하면 되기 때문에 직접 액터들은 홍반장의 존재는 알지 못해도 상관이 없어서 홍반장은 한결 편안해졌다. 하지만 홍반장은 어떤 분신이 어떤 요청을 처리하고있는지 이제 헷갈린다. 그냥 분신을 통해 들어오는 역할만 성실히 수행할 뿐이다.

> 해결된 부분
> 위의 3가지 방식의 중재한이라고 볼 수 있다.
> 액터들과 홍반장의 컴파일 시 의존성을 제거하였다.
> 한 액터의 요청에 대해 하나의 책임을 수행하는 인터페이스를 분리하여 변경에 대한 영향을 최소화 하였다.
>
> 한계점
> 어떠한 인터페이스가 어떠한 책임을 가지고 있는지는 찾기가 어렵게 되

> 었다.
> 완벽한 분리는 하지 못하고 결국은 한 클래스의 실체에서 모든 역할을 수행하기는 하지만, 문제가 발생할 수 있는 최소한의 여건은 제거를 한 중재안이다.

한계점

완벽한 해결 방법은 존재하지 않는다. 다만 적절한 책임을 가지게 한다는 목적을 가지고 노력하는 것은 그 만한 가치를 가지게 한다.
또한 홍반장이 머리아프게 고민했듯, 액터를 도출하는 것과 유스케이스를 하나의 책임으로 분리하는 것은 매우 어렵다. 특히 워터폴 방식의 특성상 초반부터 적절하게 책임을 부여하기는 더더욱 어렵다. 소프트웨어는 지속적으로 변화하는 요구사항을 수용해야 하며, 적절하게 책임을 할당했다 하더라도 향후 변경되는 요구사항에 다시 변경해야 할 일이 생기기 때문이다.
따라서 초반부터 완벽한 책임분리를 고려한다기보다 적절한 책임 분리로 단위 테스트 코드를 작성하여 향후 변경되는 사항을 반영하더라도 테스트 영향을 덜 받을 수 있도록 분리하며 신뢰성 있는 모듈을 만들어 가는 것이 좋다.