적응자 (Adapter)

GoF의 디자인 패턴 스터디

by 까북

정의

: 클래스의 인터페이스를 **사용자가 기대하는** 다른 인터페이스로 변환하는 패턴.

→ 다른 이름으로는 **래퍼(Wrapper)**

동기

1. 재사용을 목표로 개발한 툴킷
2. 그러나 그러한 툴킷조차 실제로 재사용성을 발휘하지 못할 때가 발생
3. 응용프로그램(사용자)가 요청하는 인터페이스 **!=** 툴킷에 정의된 인터페이스

활용

- 기존 클래스를 사용하고 싶은데 **인터페이스가 맞지 않을 때** 🐶 vs 😸

- 이미 만든 것을 재사용하고자 하나 이 재사용 가능한 라이브러리를 **수정할 수 없을 때** ~~앞 사람이 만든걸 건드리기는 싫은데, 사용해야만 할...~~

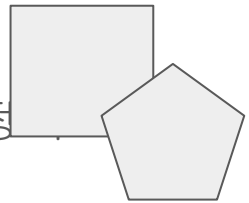
- (객체 적응자(object adapter)의 케이스 한정) 이미 존재하는 **여러 개의 서브클래스**를 사용해야 하는데.
  이 서브클래스들의 상속을 통해서 **이들의 인터페이스를 전부 개조한다는 것이 현실성이 없을 때.** ~~개조하기 싫을 때..~~ 객체 적응자를 써서 부모 클래스의 인터페이스를 변형하는 것이 더 바람직함.

예시 - 그림판

1. 그림판에는 선, 다각형, 텍스트 등을 만들 수 있음.

2. 모양을 편집할 수 있는 특징을 가지며, 화면에 그려질 수 있는 행동을 정의
   → **Shape** (추상 클래스)

3. 각각의 그래픽 요소를 **Shape**의 서브 클래스로 정
   → e.g. LineShape, PolygonShape, …



그런데.. **TextShape** 는? → (~~스포주의~~) 그래서 앞으로 우리가 만들고자 하는 것이 **TextShape**

I’m your father

TextShape

1. 텍스트 처리를 위해서는 화면 수정, 버퍼 관리 등의 복잡한 기능을 모두 구현해야..

2. But, 이미 있던 툴킷에는 **TextView** 클래스를 제공중.

3. 요것을 재사용하면 되겠다!!

그러나..

TextView != TextShape

기존에 제공되던 **TextView**는 **Shape** 클래스를 고려해서 개발된 것이 아님.. OTL → 툴킷이 제공하는 TextView 클래스를 **TextShape 클래스로 대체해서 사용할 수 없음**.

1. 즉, 이미 존재하기는 하는데..
2. 이를 사용하고자 하는 클래스(TextView)와는 아무런 연관 없이 개발될 클래스(TextShape)이거나.
3. 서로 일치하지 않는 인터페이스를 갖는 클래스들을 잘(...) 통합하여.
4. 하나의 응용프로그램을 개발해야 할 때의 아이디어가 필요. 🤔

솔루션 아이디어

1. 기존의 **TextView**의 **인터페이스를 변경**해서 사용하고자 하는 Shape의 인터페이스와 일치하게 만들자.
   → TextView의 소스코드를 가지고 있으면, 가능. 🙆🏻
   → 그런데 **소스코드가 없으면**..?! 🤦🏻
   → + 갖고 있다 하더라도 굳이..?!?! 💁🏻

2. 그럼, 기존의 툴킷 개발자에게 하나의 응용프로그램 동작을 위해 인터페이스를 변경해 달라는 요청을 하자.
   → 🙅🏻

솔루션

기존 클래스의 인터페이스를 수정할 수 없다면, Shape와 TextView 인터페이스 둘 다 맞도록 **우리가 개발한 TextShape 클래스를 조정해야 함**.

두가지 아이디어.

- Shape 인터페이스와 TextView의 구현을 모두 상속받기.

  or..
- TextView의 인스턴스를 TextShape에 포함시키고, TextView 인터페이스를 사용하여 TextShape를 구현하기.

솔루션

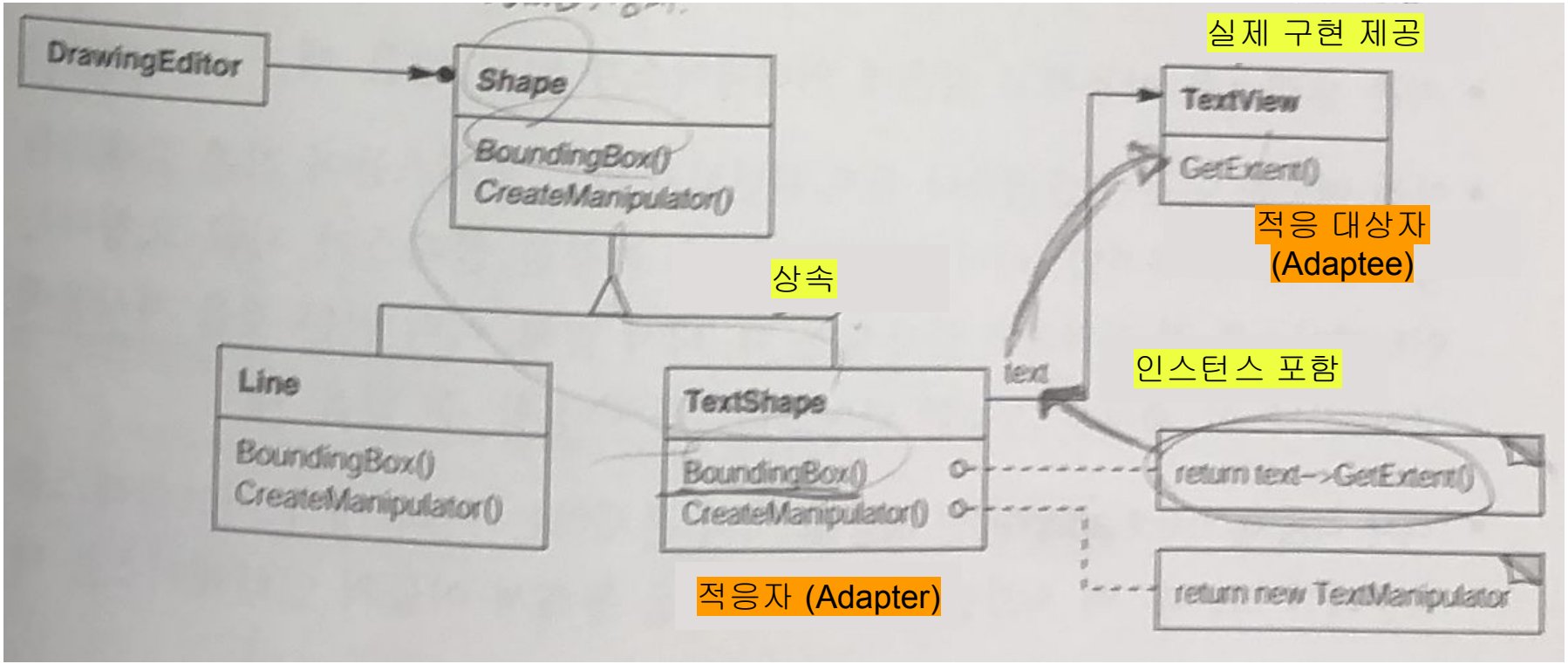
기존 클래스의 인터페이스를 수정할 수 없다면, Shape와 TextView 인터페이스 둘 다 맞도록 **우리가 개발한 TextShape 클래스를 조정해야 함**.

두가지 아이디어.

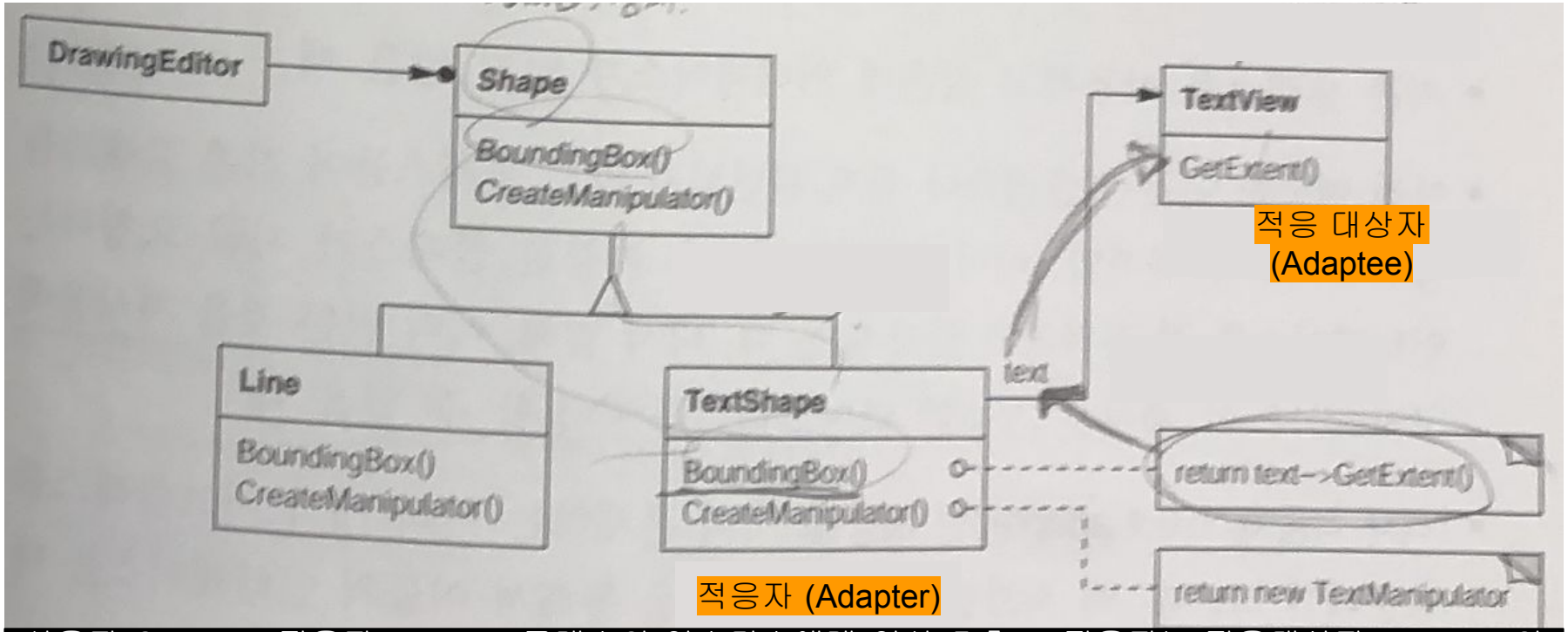
- Shape 인터페이스와 TextView의 구현을 모두 상속받기. → **클래스 버전**

  or..
- TextView의 인스턴스를 TextShape에 포함시키고, TextView 인터페이스를 사용하여 TextShape를 구현하기. → **객체 버전**

그림판 구조


DrawingEditor
Shape
BoundingBox()
CreateManipulator()
실제 구현 제공
TextView
GetExtent()
적응 대상자
(Adaptee)
상속
Line
BoundingBox()
CreateManipulator()
TextShape
BoundingBox()
CreateManipulator()
text
인스턴스 포함
return text->GetExtent()
return new TextManipulator
적응자 (Adapter)

그림판 구조

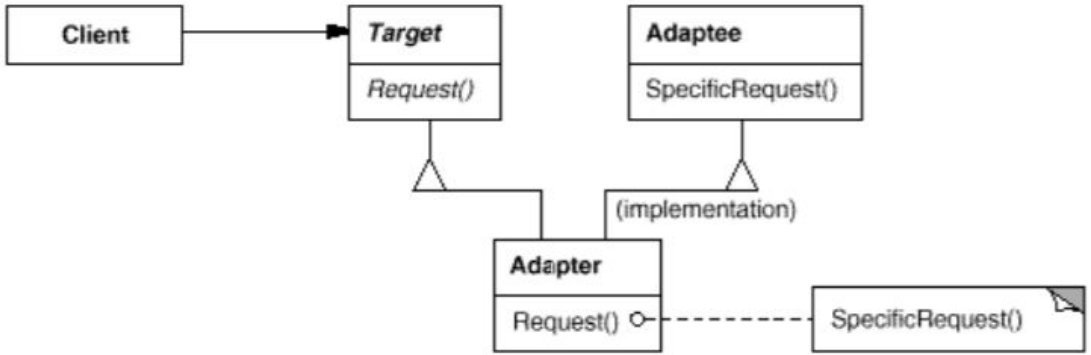


사용자(Client) → 적응자(Adapter) 클래스의 인스턴스에게 연산 호출 → 적응자는 적응대상자(Adaptee)의 연산 호출

구조 1 - 클래스 적응자

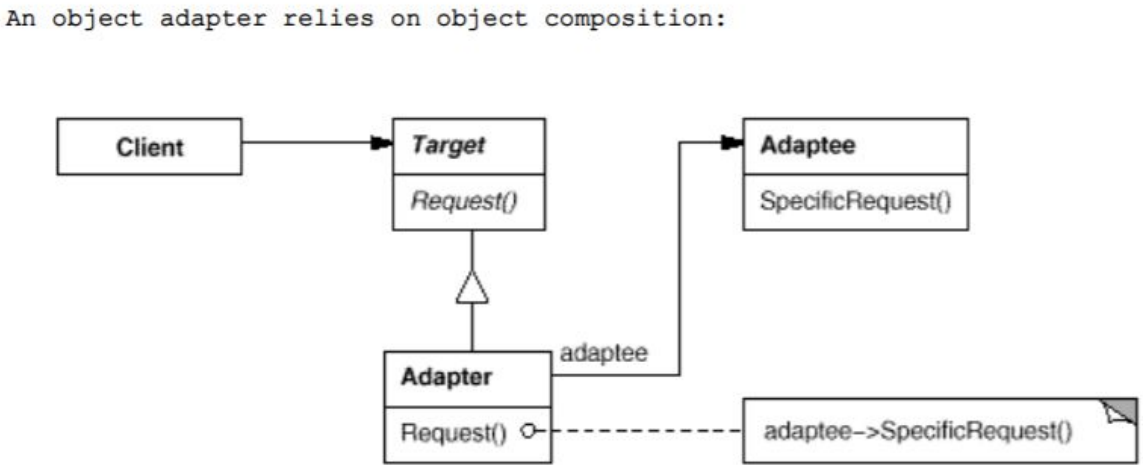
클래스 적응자는 다중 상속을 활용해서 한 인터페이스를 다른 인터페이스로 적응시킨다.

A class adapter uses multiple inheritance to adapt one interface to another:



구조 2 - 객체 적응자

객체 적응자는 객체 합성을 써서 적응시킨다.



결과 - 클래스 적응자

- Adapter(적응자)는 명시적으로 Adaptee(적응대상자)를 상속받고 있을 뿐, **Adaptee의 서브클래스들을 상속받는 것은 아니므로**, Adaptee의 서브클래스에 정의된 기능들을 사용할 수 없다.

- Adapter 클래스는 Adaptee 클래스를 상속하기 때문에 **Adaptee에 정의된 행동을 재정의할 수도 있다.** (그러나, 오남용은 적응자 패턴을 사용하는 의미가 없어지는..)

- 양방향 적응자를 통한 투명성 제공이 가능. **Target과 Adaptee의 인터페이스 모두를 제공할 수 있다**.

결과 - 객체 적응자

- Adapter 클래스는 하나만 존재해도 수많은 Adaptee 클래스들(**서브클래스 포함**)과 동작할 수 있다.

- Adaptee 클래스의 행동을 재정의하기가 매우 어렵다.

One more thing…

e.g.

A사는 ~~마진을 위해~~ 새 버전의 어른폰에 구 버전 어른폰에서 개발했던 카메라 모듈을 활용하고자 한다. 그런데, 새 버전의 어른폰 개발진은 새 카메라 모듈을 넣을줄 알고, 기존 카메라 모듈과는 다른 인터페이스를 갖도록 만들어버렸다!

기존의 카메라 모듈은 건드리지 말고, 새 어른폰에 기존의 카메라 모듈을 넣어야 한다.

→ 새 어른폰의 인터페이스와 일치하는 인터페이스를 만들어 기존의 카메라 모듈을 **감싸는 작업**을 진행한다.
→ **적응자 패턴 == 래퍼(Wrapper)**

결론

- 기존에 존재하는 다른 클래스를 내가 원하는 인터페이스에 맞추기 위해 사용하는 패턴.

- TextShape라는 클래스의 구현을 위해 Shape의 인터페이스와 TextView의 구현을 **모두 상속**받든지 → **클래스 버전**

- 아니면, **TextView의 인스턴스를 TextShape에 포함**시키고, **TextView 인터페이스를 사용**하여 TextShape를 구현한다. → **객체 버전**

- 이 때 TextShape를 **적응자(Adapter)**라고 한다.

The End

 # ref

- ➔ https://scvgoe.github.io/2018-12-24-GoF의-디자인-패턴-(Summary)-5.-적응자(Adapter)/
- ➔ https://yukariko.github.io/designpattern/2016/08/28/adapter.html
- ➔ https://m.blog.naver.com/lhs860226/140100483
- ➔ https://ehpub.co.kr/6-적응자-패턴adapter-pattern/