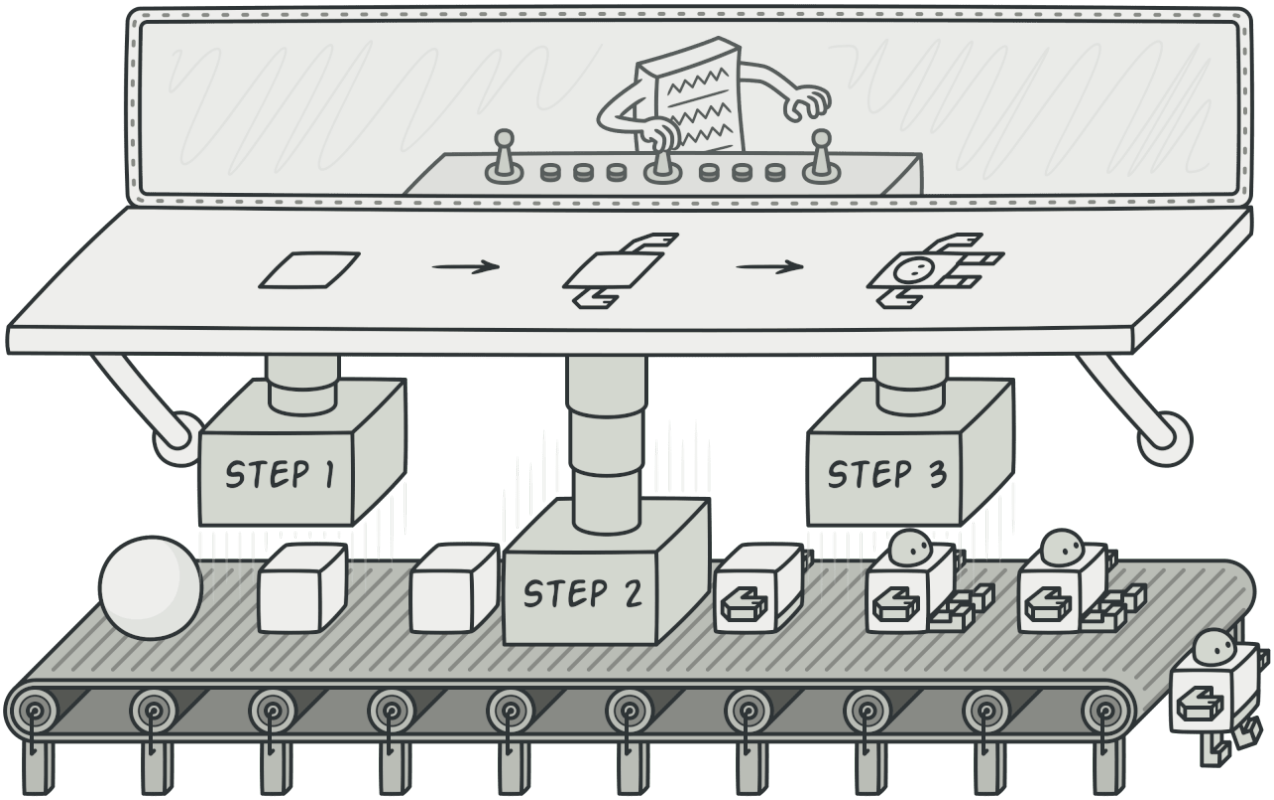
출처

- https://refactoring.guru/design-patterns/builder (builder패턴 설명)
- https://projectlombok.org/features/Builder (lombok)

Builder란?

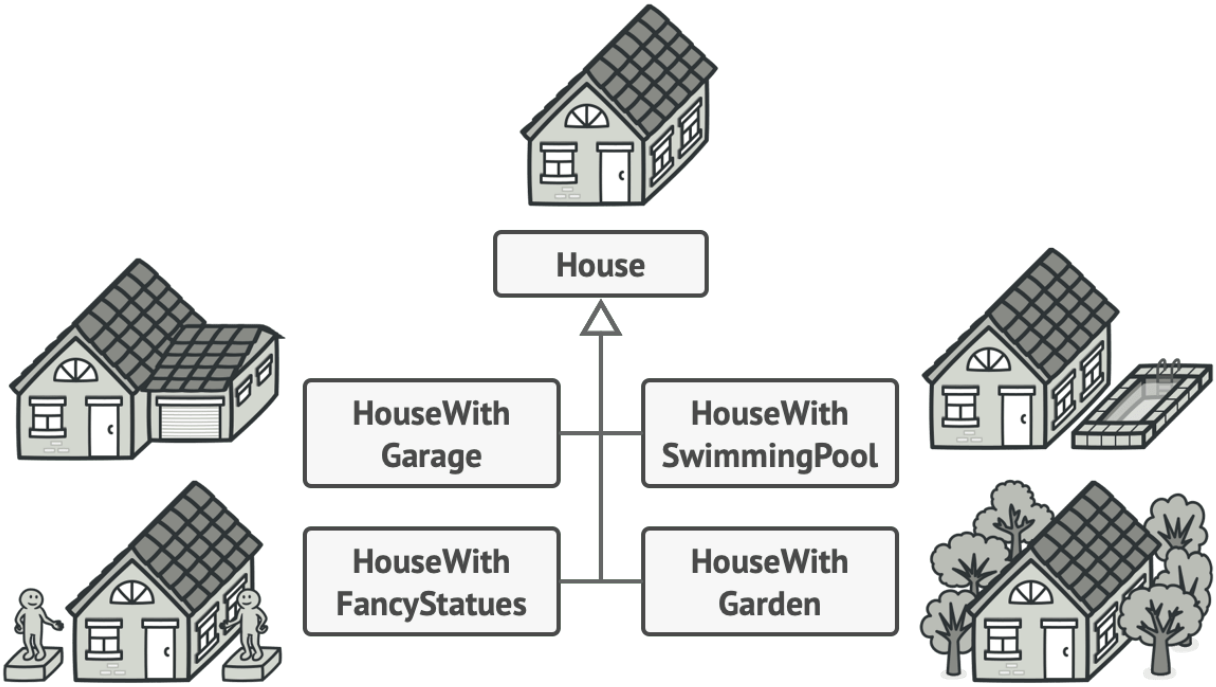
- 생성 디자인 패턴
- 복잡한 objects를 하나씩 하나씩(step-by-step)생성할 수 있게 해주는 디자인 패턴입니다.
- 동일한 생성 코드로 다른 타입의 object를 만들 수 있도록 합니다.



상황

- 많은 field와 nested objects를 하나하나 생성해야하는 **복잡한 object**를 만들어야 합니다.

권장하지 않는 방법1 - class 상속을 통해 구현



방법 설명

1. **기본 클래스 생성**

이 방법은 House라는 기본이되는 클래스를 만든다고 합시다. 이 클래스는 지붕 및 창문 외벽 등을 가지고 있습니다. 이때, 구체적인 필드 값을 가지도록 만들어줍니다. 다시 말해서 오두막집 클래스, 벽돌집 클래스를 만들어주는 겁니다. 예를 들어 오두막집 클래스의 외벽은 나무일 것이고 벽돌집 클래스의 외벽은 벽돌이 되겠지요.

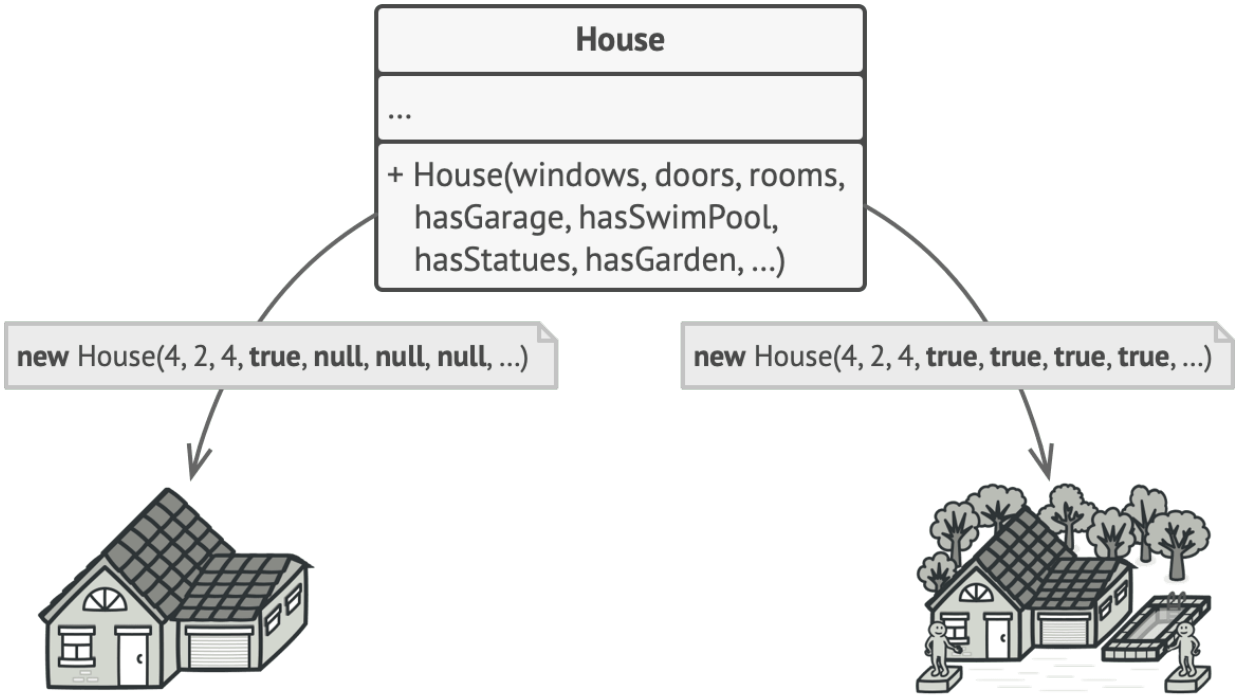
2. **확장시 - 서브 클래스 생성**

오두막집에 창고를 둔 집을 만들고 싶다면? 오두막집을 상속받고, 창고를 추가해주면 됩니다.

문제점

1. sub클래스가 무수히 많아집니다
    - 매 새로운 파라미터마다 서브클래스를 만들어주어야하기 때문입니다.
2. 계층이 깊어질 수 있습니다.
    - 상속은 계층적인 형태를 띕니다. 만약에 부속건물 확장이 아닌, 스타일(ex: modern)을 추가하여 확장을 한다면 한단계 깊게 만들어야겠죠.

권장하지 않는 방법2 - giant 생성자



방법 설명

가능한 모든 파라미터를 받는 생성자 작성

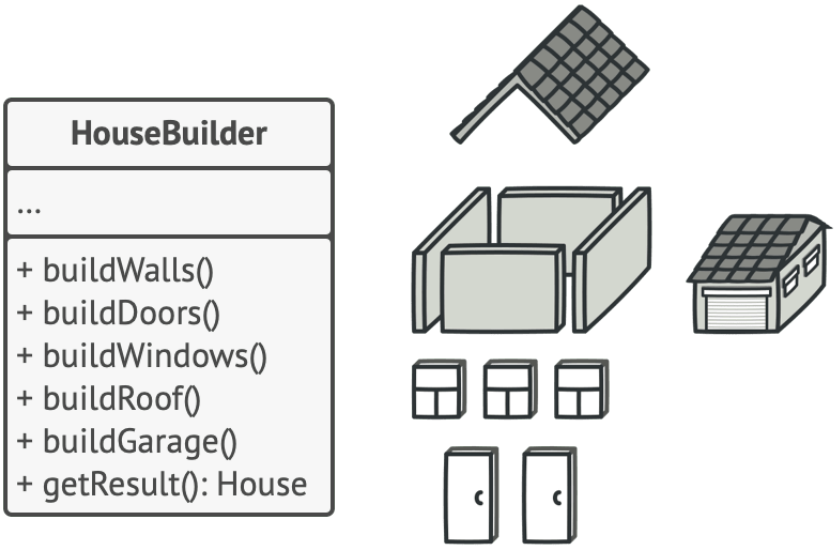
문제점

1. 필요없는 파라미터를 고려하여 argument를 넣어야 합니다.
    - 위 그림의 왼쪽에서 보시듯, 창고만 달린 일반 집을 생성함에도 다른 옵션까지 고려하여 null로 넣어주어야 합니다.
2. 각 인자의 의미를 파악하기 어렵습니다.
    - 파라미터가 많아질 경우 몇 번째 인자가 어떤 의미를 가지는지 파악하기 어렵기 때문입니다.

해결책 - Builder 패턴

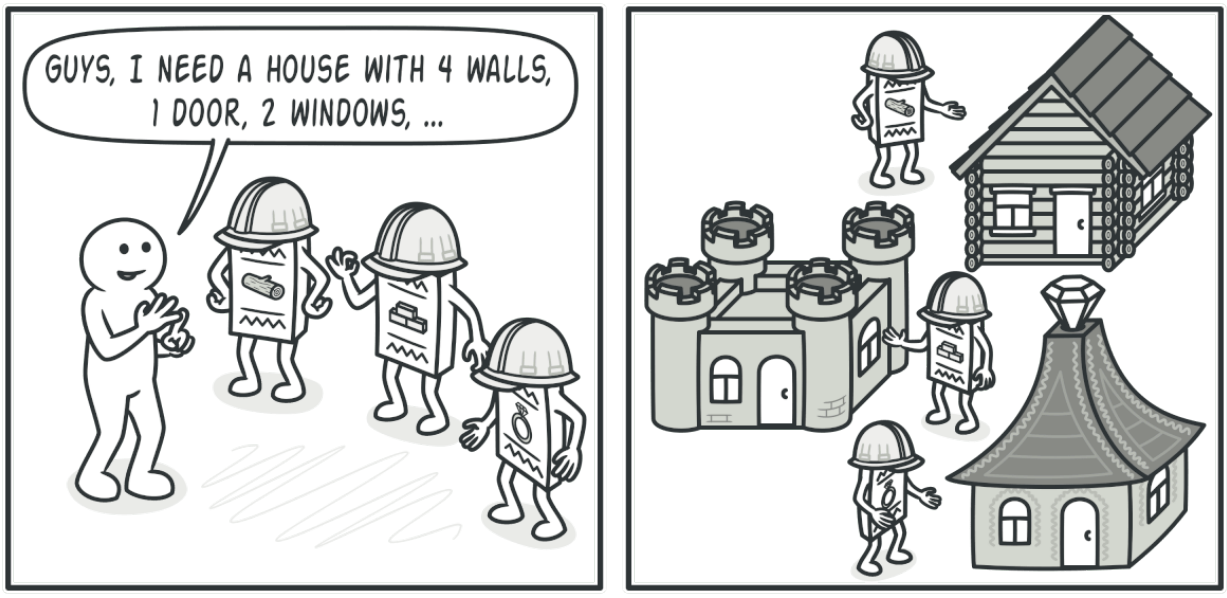
Builder 패턴은 object의 생성코드를 생성하려는 클래스의 밖으로 꺼냅니다. 그리고 *builders*라는 개개의 오브젝트에게 그 생성 책임을 맡깁니다. 이 builder는,

1. 생성해야할 필드를 하나씩 정할 수 있도록 해줍니다.
2. 생성이 필요한 필드만 지정하여 필드를 정할 수 있습니다.



[그림의 설명] *The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.* (여기서 빌더가 객체를 생성하는 중엔 다른 객체들이 접근하지 못한다는 뜻은 해당 패턴을 어떻게 구현하는지 보여드린 뒤 설명드리겠습니다.)

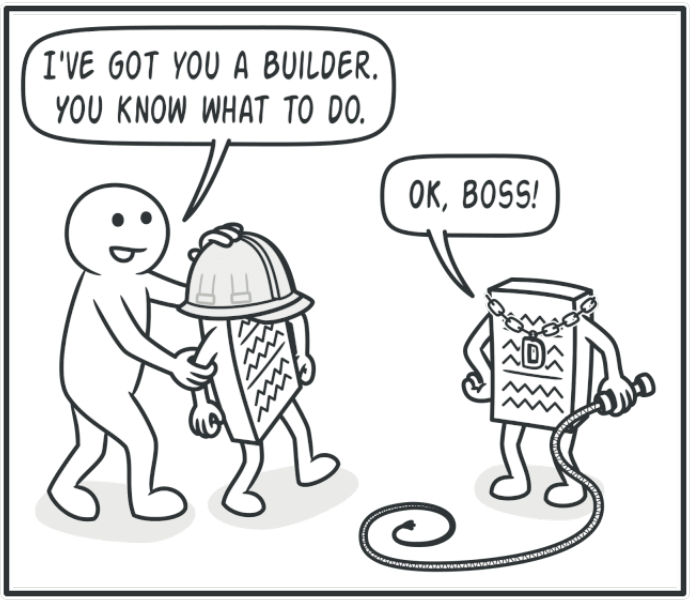
3. 그리고 객체를 만드는 동일한 행동으로 서로 다른 객체도 만들 수 있습니다.



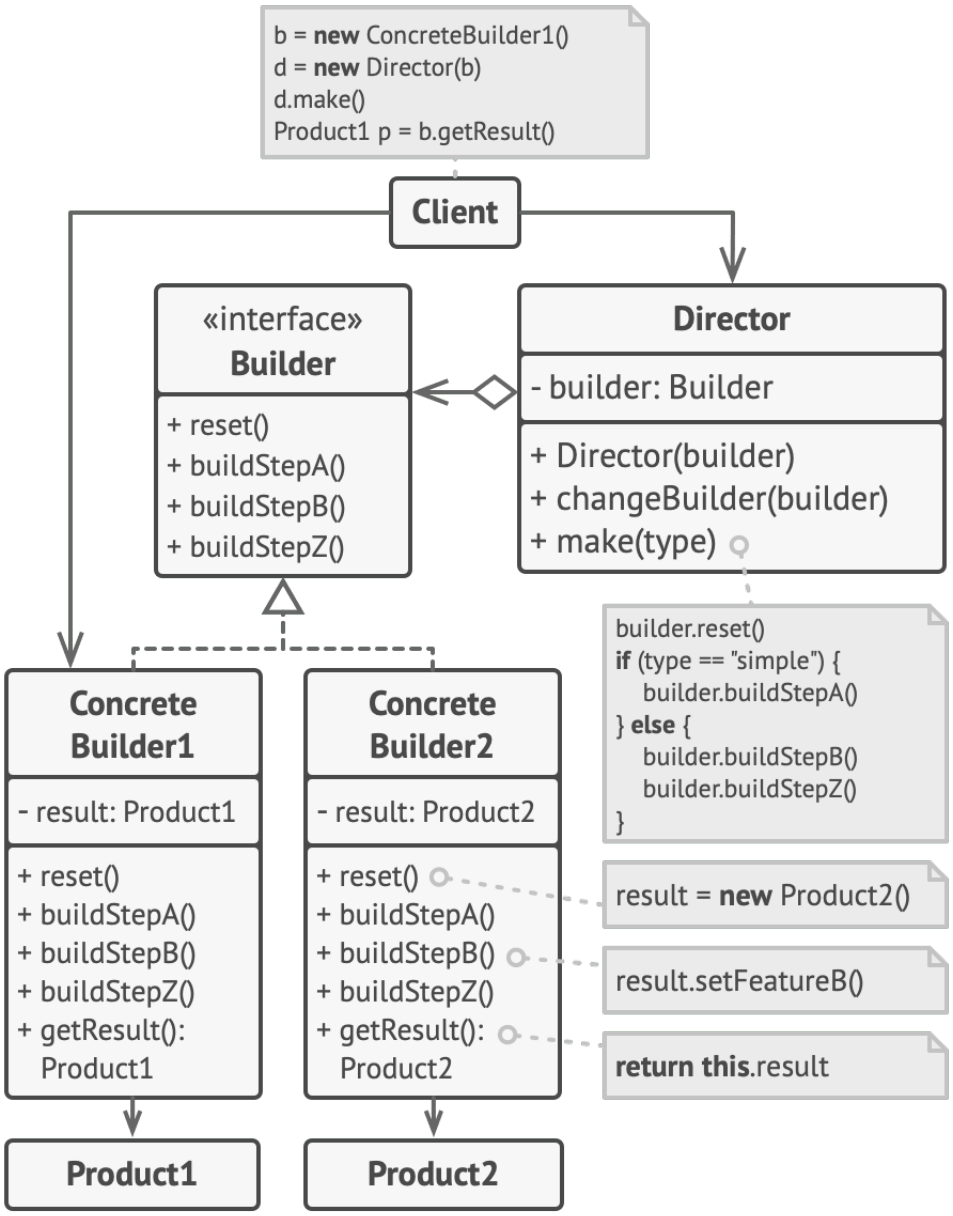
이 그림은, 통나무집, 벽돌집, 보석집을 지으려고 하려고 합니다. 모두다 다르나, 외벽을 짓고, 지붕을 올리는 등 동일한 행동 패턴을 가지고 있기 때문에 동 일한 행동패턴을 지닌 빌더로 다른 스타일의 집을 만들 수 있다는 그림입니다.

Director

- builder가 특정한 절차를 따라서 만들 수 있도록 도와주는 class입니다.
- 이 클래스의 용도는 자주 생성할 객체가 있다면 director에서 builder를 통해 생성할 수 있도록하는 것입니다. (재사용성)
- 따라서 빌더 패턴에서 이 클래스를 만드는 것은 선택사항입니다.
- 추가로, client에게 객체가 어떻게 생성되는지 숨길 수 있습니다.



구조



Builder Interface

- 이 인터페이스는 builder의 필드별 생성 메서드(step)를 지정해줍니다.
- 객체 초기화를 위해 reset()이라는 메서드를 작성합니다.

Concrete Builders

- 서로 다른 객체를 생성합니다. 다시 말해서, 객체를 생성하기 위한 행동(Step)이 동일하다면 Builder를 구현하면 됩니다.
- 그래서 생성된 객체가 타입이 불일치 할 수 있습니다.

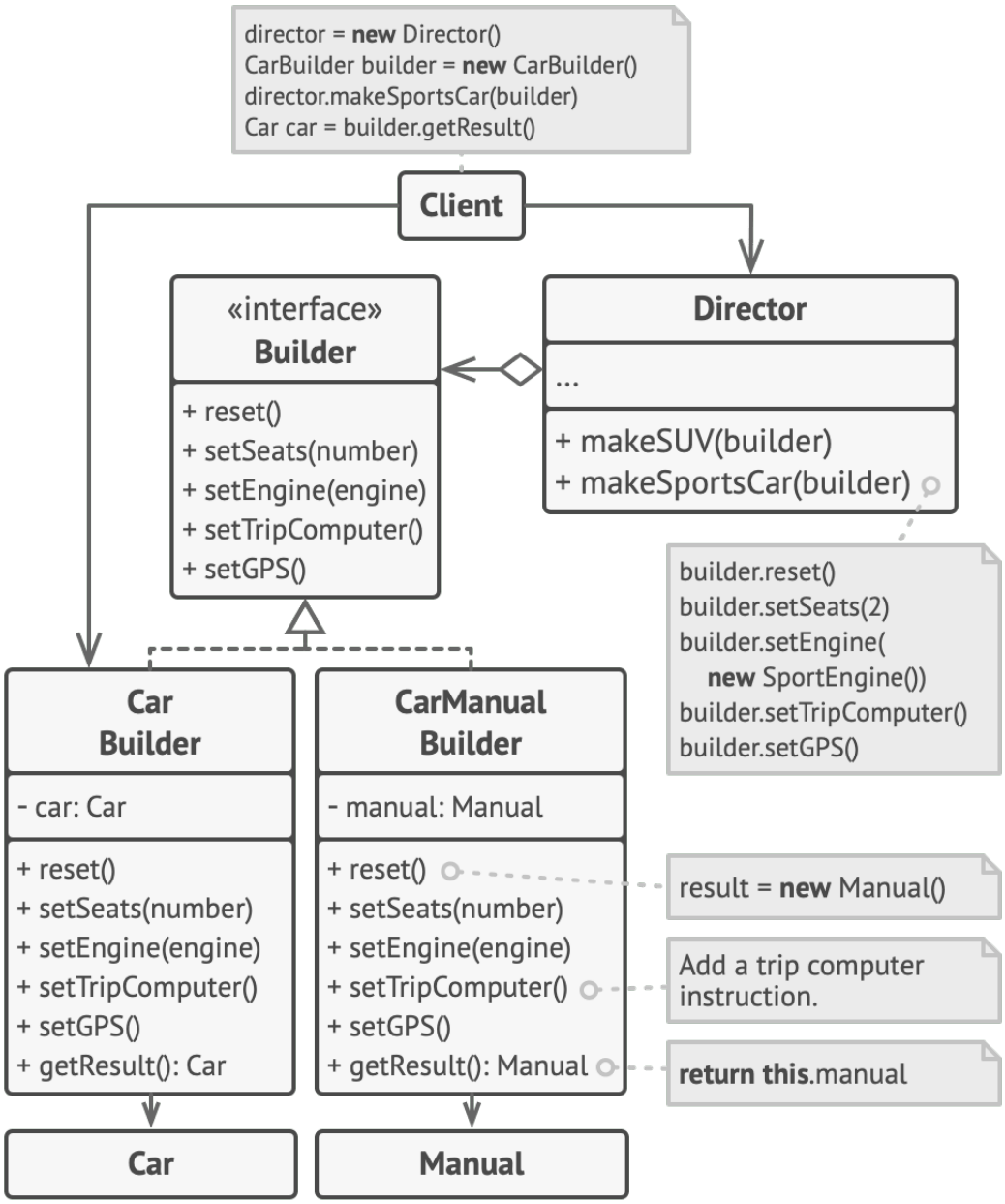
Products

- concrete Builder에 의해 생성된 객체입니다

Director

- 자주 사용되는 builder를 구현합니다.

또 다른 예



여기서 주목해야할 부분은 두가지입니다.

1. Car Builder와 CarManual Builder가 만드는 interface는 다릅니다.

이전 하우스 예제에서는 벽돌집, 오두막집, 보석집을 집이라는 인터페이스로 추상화시킬 수 있었지만, Car와 Manual은 그럴 수 없습니다. 서로 다르기 때문이지요.

따라서 getResult()를 구현하도록 하였습니다. 타입이 달라질 수 있기 때문이지요.

이렇기 때문에, Director에서 반환된 값은 Car나 Manual이 될 수 없습니다. 이를 반환하도록 한다면, 구체적인 클래스에 의존하게 됩니다.(DIP 위반) ***따라서 반환값이 builder입니다.***

(혹시나 행동 패턴이 같다고 LSP를 따른 것이 아니냐할 수 있지만, Builder가 필드를 정하기 위한 행동 패턴이 동일한 것이지, Car나 Manual의 행동 패턴이 동일한 것이 아닙니다.)

2. reset()을 통해 builder내부에서 객체를 생성하고, 다 만들었을 때 getResult()를 통해 객체를 반환하도록 합니다.

   위에서 해결책 중 사진에 대한 설명을 이제 말씀드릴 수 있겠습니다. (builder가 생성 중엔 왜 다른 object가 접근할 수 없는지)

   builder가 자신의 내부에(필드 값) 객체를 생성하도록 하고, 이 객체의 접근은 private로 선언함으로써 오로지 builder만이 해당 객체에 접근하여 값을 수정하도록 하고 있습니다. 그렇기 때문에 사진 설명처럼 말할 수 있는 것이지요.

어떤 상황에 적용할까?

1. 생성자의 파라미터가 많아질 때

장,단점

장점

1. 필드를 하나씩 하나씩 생성할 수 있습니다
2. 동일한 생성 코드를 재사용할 수 있습니다.
3. 복잡한 객체 생성 로직을 비즈니스 로직과 분리시켜줍니다.

단점

빌더 패턴은 새로운 클래스들을 생성해야 하므로 전체적으로 코드의 복잡성이 증가합니다.

다른 패턴과 연관관계

1. Factory Method가 복잡해지면 Builder패턴으로 바꿀 수 있습니다.
   - 객체를 생성하는 패턴이 좀 더 복잡해지기 때문입니다.

2. Builder는 복잡한 object를 한단계 한단계 생성하는 것에 초점을 두었고, Abstract Factory는 연관된 objects를 생성하는데 촛점을 두었습니다. 따라서 Abstract패턴은 단번에 product를 반환하고, Builder는 product를 보내주기 전에 한단계 한단계 생성하도록 합니다.