GoF의 디자인 패턴

그 중에서 생성패턴 중 에 하나인 팩토리 메소드

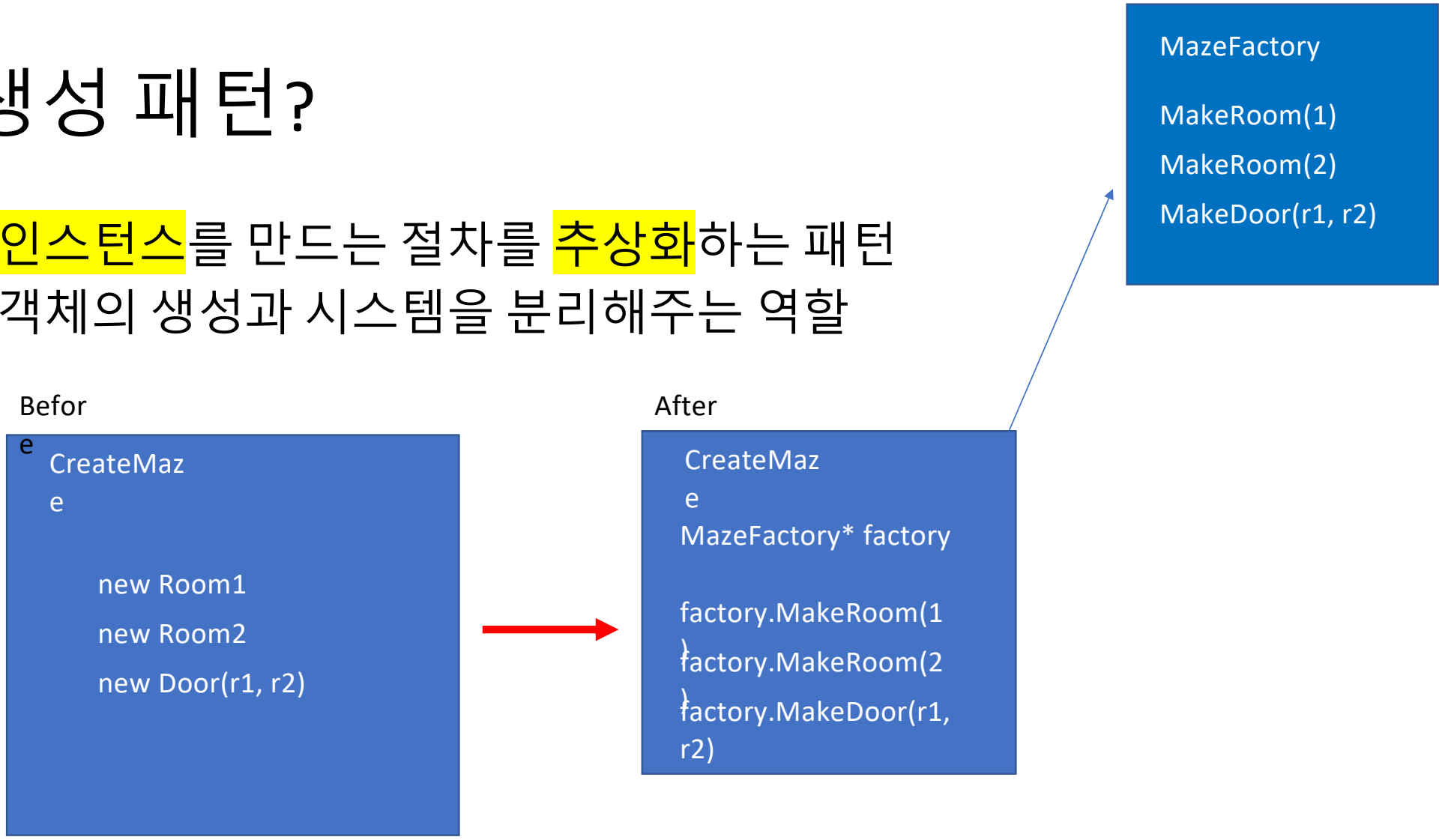
by 잭슨

오늘 스터디할 내용

- 생성 패턴이란?
- 팩토리 메서드 패턴란?
- 왜 써야하나?
- 팩토리 메서드
- 예제 코드
- 관련 패턴 소개

생성 패턴?

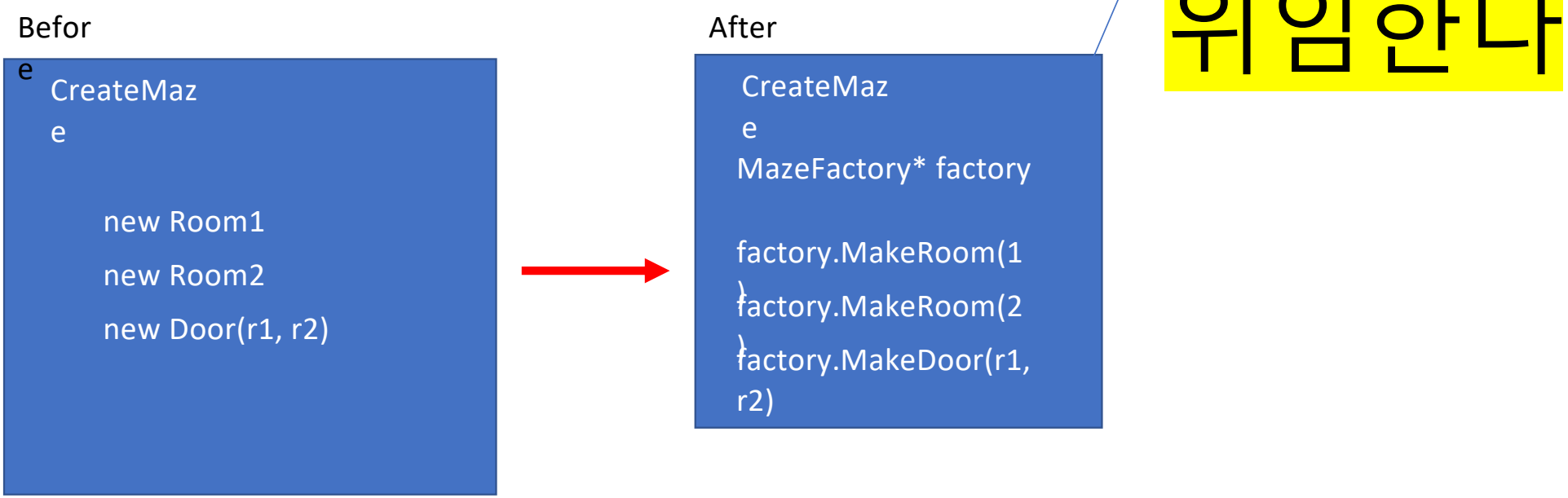
- 인스턴스를 만드는 절차를 추상화하는 패턴
- 객체의 생성과 시스템을 분리해주는 역할



객체 생성 패턴

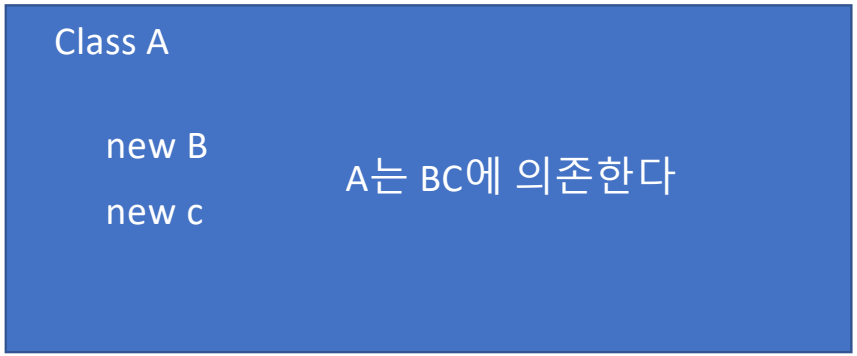
- 인스턴스화 작업을 다른 객체에게 떠넘길 수 있다.

직접하지 않고, 다른 객체에게 전달한다.



객체 생성 패턴

- 객체 생성에 대한 책임을 다른 객체에게 위임한다..
  why?



- 생성에 의한 의존성이 생겨
  객체간의 강결합(Tightly **Coupled**)상태
  = 재활용 x , 유연성 x

팩토리 메서드란 무엇인가?

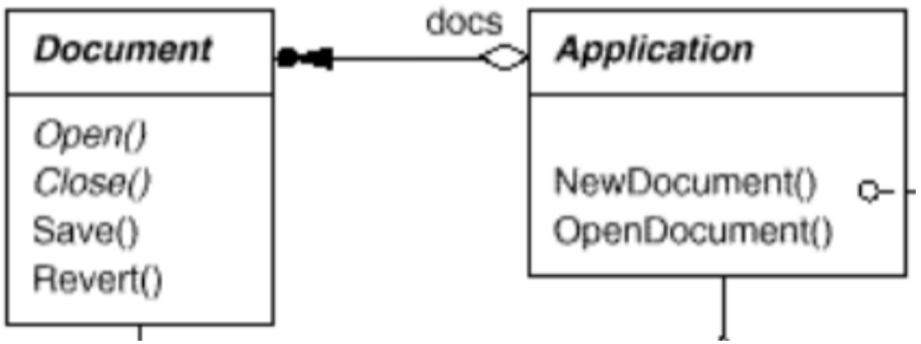
- 다른 이름 : 가상생성자 (Virtual Constructor)
- 팩토리 (factory) + 메서드 (method)
- 요약:

객체를 생성하기 위해 인터페이스를 정의하지만,
어떤 클래스의 인스턴스를 생성할지에 대한 결정은
서브클래스에 의해 정해진다. **(객체 생성을 지연)**

팩토리 메서드 왜 사용?

사용자에게 다양한 종류의 문서를 표현할수 있는 시스템 프레임워크을 구현

Application , Document



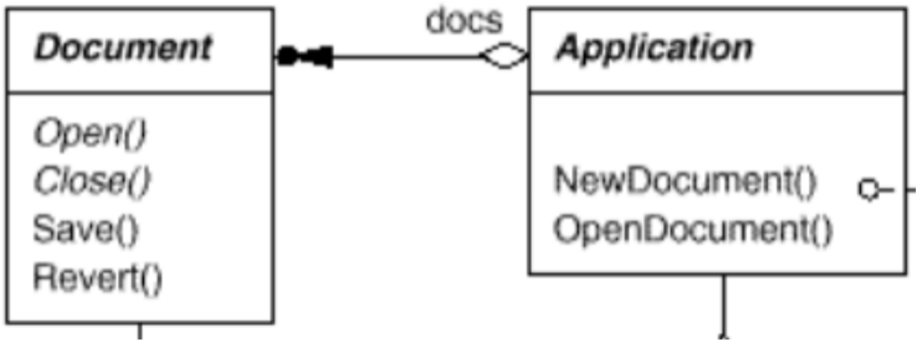
Document 객체를 관리하는 책임을 가지고있고, 필요에 따라 Document 를 생성하기도 한다.

팩토리 메서드 왜 사용? 문제!

사용자에게 다양한 종류의 문서를 표현할수 있는 시스템 프레임워크을 구현

Application , Document

추가 요구 사항 : 시스템에 따라 Document의 종류가 달라진다



Application은 Document 인스턴스를 생성해야되는지만 알고, **어떤 종류의 Document를 생성해야되는지는 모름**

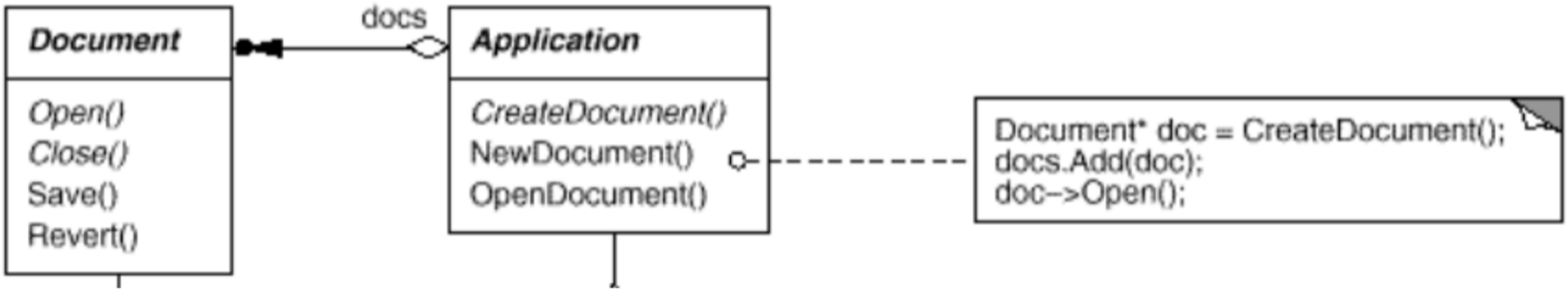
Document 인스턴스 생성이 불가

팩토리 메서드 왜 사용?

사용자에게 다양한 종류의 문서를 표현할수 있는 시스템 프레임워크을 구현

Application , Document

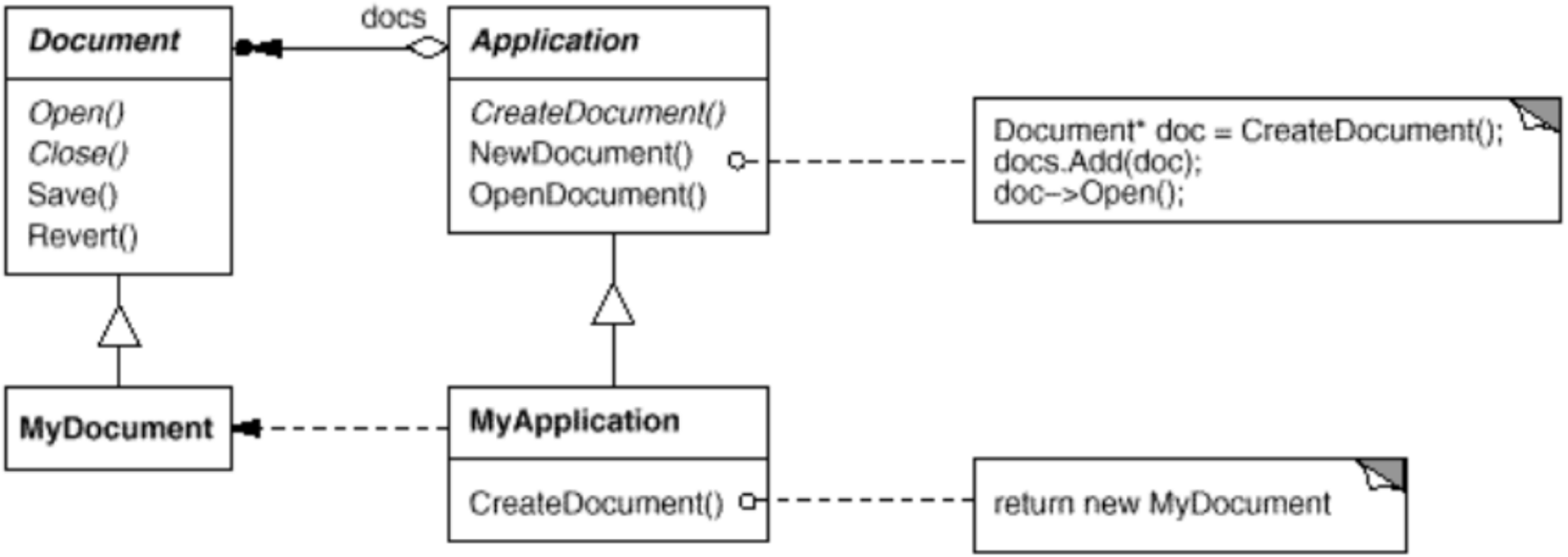
추가 요구 사항 : 시스템에 따라 Document의 종류가 달라진다



Document의 어느 것을 생성해야 하는지에 대한 정보를 **추상화하여 캡슐화**하고, 그것을 프레임워크에서 떼어낸다.

팩토리 메서드 왜 사용?

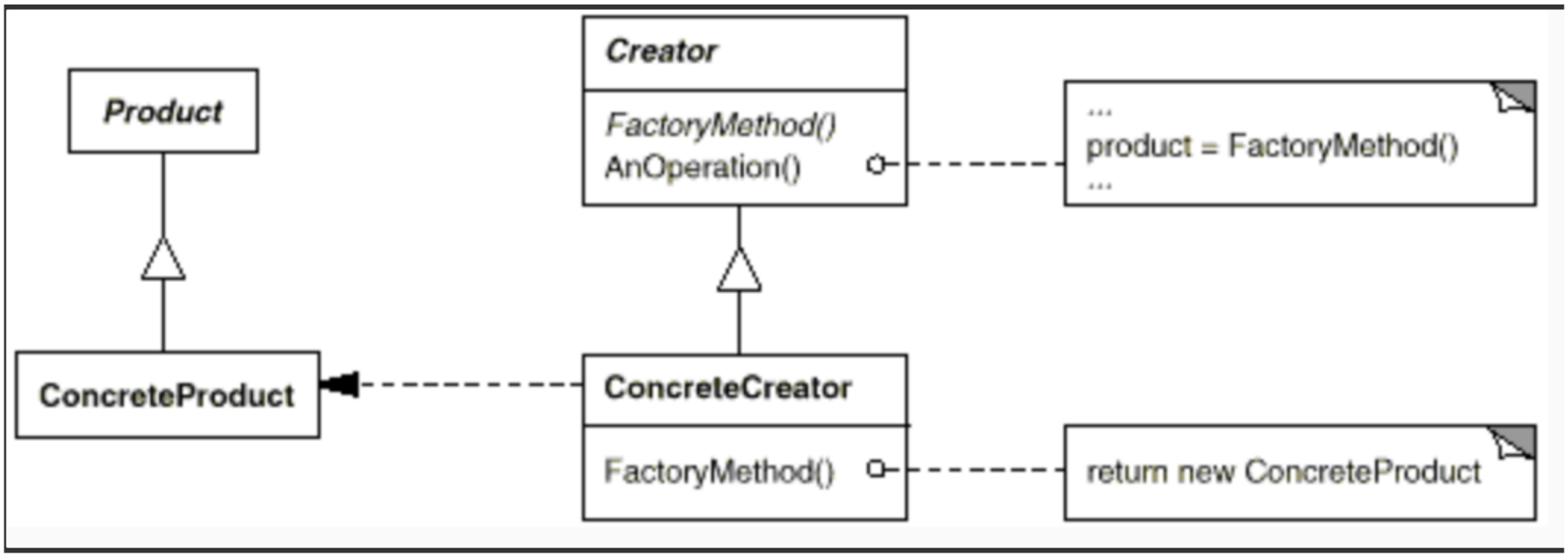
application 클래스의 서브클래스는 추상화된 CreateDocument 메소드를 override해서 상황에 맞는 Document의 서브클래스를 반환한다.



팩토리 메서드 활용성

- 어떤 클래스가 자신이 생성해야 하는 **객체의 클래스를 예측할 수 없을 때**
- 생성할 객체를 기술하는 **책임을 자신의 서브클래스가 지정**했으면 할 때 (객체생성 책임 위임)
- 객체 생성의 책임을 몇 개의 보조 서브클래스 가운데 하나에게 위임하고, 어떤 서브클래스가 위임자인지에 대한 **정보를 국소화**시키고 싶을 때 (생성할 클래스에 대한 캡슐화?)

팩토리 메서드 구조



Creator는 자신의 서브 클래스를 통해 실제 필요한 팩토리 메서드를 재정의해서, 적절한 ConcreateProduct 인스턴스를 반환할수 있게함

팩토리 메서드

- 시스템에서는 Product 클래스에 정의된 인터페이스와만 동작하기 때문에, **사용자가 필요한 어떠한 ConcreateProduct가 오더라도 동작**할수있게됨

  BUT 잠재적인 단점>
- *ConcreateProduct 객체 하나만 만드려고 할때도, Creator클래스를 상속해서 서브클래스를 만들어야 할지도 모른다.*

1. 서브 클래스에 대한 **hook-메서드**를 제공한다.
   팩토리 메서드에 대해서, abstract로 만들지 않고, 기본 기능을 하는 메서드로 만든다.
2. 병렬적으로 creator 계열의 클래스들이 증가할수록 병렬적으로 product 클래스 계통도 늘어난다.

팩토리 메서드 고려사항 1

- 구현 방법은 크게 2가지
  1. Creator 클래스를 추상 클래스로 정의하고, 정의한 팩토리 메서드에 대한 구현은 제공하지 않는 경우
  2. Creator가 ConCreator 클래스이고, 팩토리 메서드에 대한 기본 구현을 제공하는 경우
- 팩토리 메서드를 매개변수화 할 수 있다.
  - 팩터리 메서드에 매개변수를 추가하여, 생성될 인스턴스에 대한 정보를 받는다

```
class Creator{
public:
    virtual Product* Create(productId);
};
Product* Creator::Create (ProductId id) {
        if(id==mine) return new MyProduct;
        if(id==your) return new YourProduct;
}
```

팩토리 메서드 고려사항 2

- 언어마다 구현방법이 다를수 있다.

- 템플릿을 사용하여 서브 클래싱을 피한다.
  - 팩토리 메서드를 쓰면 생길 수 있는 잠재적인 문제점 중 하나는 그냥 Product 클래스 하나를 추가하려 할 때마다 서브클래싱을 해야 한다는 점입니다.
    c++의 template (java의 generic)을 활용하면 이를 피할 수 있다

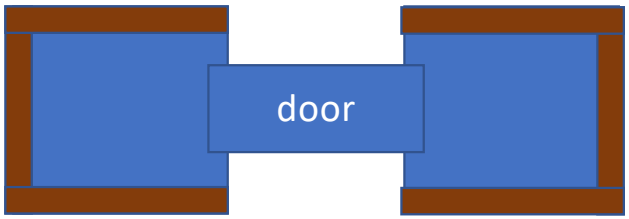
- 명명규칙에 따르는것도 중요하다
  - MacApp같은 경우 Class DoMakeProduct()...

컴퓨터 게임에 넣을 미로를 만들어보자 1

- 미로생성은 어떻게 할까?

CreateMaze

// 유연한 설계 = 직접 생성 ㄴㄴ
// 방 사이에 문이 있는 두개의 방으로 구성된 미




```
Maze * MazeGame :: CreateMaze () {
Maze* aMaze = new Maze;
Room* r1 = new Room(1);
Room* r2 = new Room(2);
Door* theDoor = new Door(r1, r2);

aMaze->AddRoom(r1);
aMaze->AddRoom(r2);

r1->SetSide(North, new Wall);
r1->SetSide(East, theDoor);
r1->SetSide(South, new Wall);
r1->SetSide(West, new Wall);

r2->SetSide(North, new Wall);
r2->SetSide(East, new Wall);
r2->SetSide(South, new Wall);
r2->SetSide(West, theDoor);
```

컴퓨터 게임에 넣을 미로를 만들어보자 - 팩터리 메서드2

팩터리 메서드들을 정의한다.

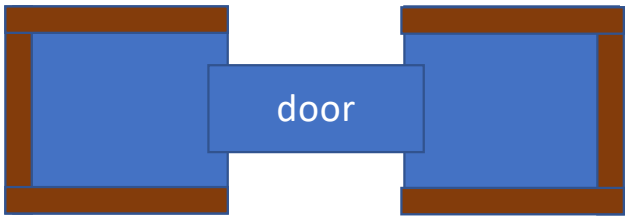
```
class MazeGame{
public:
        Maze* CreateMaze () ;

 virtual Maze* MakeMaze() const
   { return new Maze; }
 virtual Wall* MakeWall() const
   { return new Wall; }
 virtual Room* MakeRoom(int n) const
   { return new Room(n); }
 virtual Door* MakeDoor(Room* r1, Room* r2)
   { return new Door(r1, r2); }
```

컴퓨터 게임에 넣을 미로를 만들어보자 3

- 미로생성은 어떻게 할까?

CreateMaze

// 유연한 설계 = 직접 생성 ㄴㄴ
// 방 사이에 문이 있는 두개의 방으로 구성된 미




```
Maze * MazeGame :: CreateMaze () {
Maze* aMaze = MakeMaze();
Room* r1 = MakeRoom(1);
Room* r2 = MakeRoom(2);
Door* theDoor = MakeDoor(r1, r2);

aMaze->AddRoom(r1);
aMaze->AddRoom(r2);

r1->SetSide(North, MakeWall);
r1->SetSide(East, theDoor);
r1->SetSide(South, MakeWall);
r1->SetSide(West, MakeWall);

r2->SetSide(North, MakeWall);
r2->SetSide(East, MakeWall);
r2->SetSide(South, MakeWall);
r2->SetSide(West, theDoor);
```

컴퓨터 게임에 넣을 미로를 만들어보자 - 팩터리 메서드4

변형된 다른 게임도 쉽게 구현이 가능하다.

```
class BombedMazeGame: MazeGame{
public:
        BombedMazeGame() ;

 virtual Wall* MakeWall() const
  { return new BombedWall; }
 virtual Room* MakeRoom(int n) const
  { return new RoomWithBomb(n); }
```

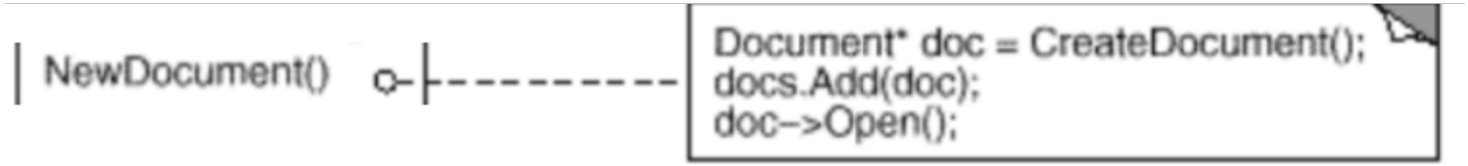
```
class EnchatedMazeGame: MazeGame{
public:
        EnchatedMazeGame() ;

 virtual Wall* MakeRoom(int n) const
  { return new EnchatedWall(n); }
 virtual Door* MakeDoor(Room* r1, Room* r2) const
  { return new DoorNeedingSpell(r1, r2); }
```

관련 패턴

- 추상 팩토리 패턴은 팩터리 메서드를 이용해서 구현할때가 많다

  이미 테디님이 발표하신곳에 **추상 팩토리내에는 팩터리 메서드로 구현**되어있다.

- 팩터리메소드는 템플릿 메서드 패턴에서도 사용될때가 많다.



Allen Holub은 "실용주의 디자인 패턴"

Factory Method 패턴은 기반 클래스에 알려지지 않은 구체 클래스를 생성하는 Template Method라 할 수 있다.

Factory Method의 반환 타입은 생성되어 반환되는 객체가 구현하고 있는 인터페이스이다.

Factory Method는 또한 기반 클래스 코드에 구체 클래스의 이름을 감추는 방법이기도 하다
(**Factory Method는 부적절한 이름이다**. :
사람들은 객체를 생성하는 모든 메소드를 자연스레 팩토리 메소드라 부르는 경향이 있는데, 이러한 **생성 메소드가 모두 Factory Method 패턴을 사용하는 것은 아니다**).[4]

내용 정리

객체 생성 처리를 서브 클래스로 분리 해 처리하도록 캡슐화하는 패턴

- 객체를 생성하기 위해 인터페이스를 정의하지만,
  어떤 클래스의 인스턴스를 생성할지에 대한 결정은 서브클래스에 의해 정해진다

(객체 생성의 책임을 서브클래스에게 위임)

-> 결합도가 낮아짐-> 확장가능성

(재사용보단 서브클래스의 확장으로 관리해야할 클래스가 늘어날수 있다는 단점)