GoF의 디자인 패턴

그 중에서 생성패턴 중 에 하나인 추상팩토리

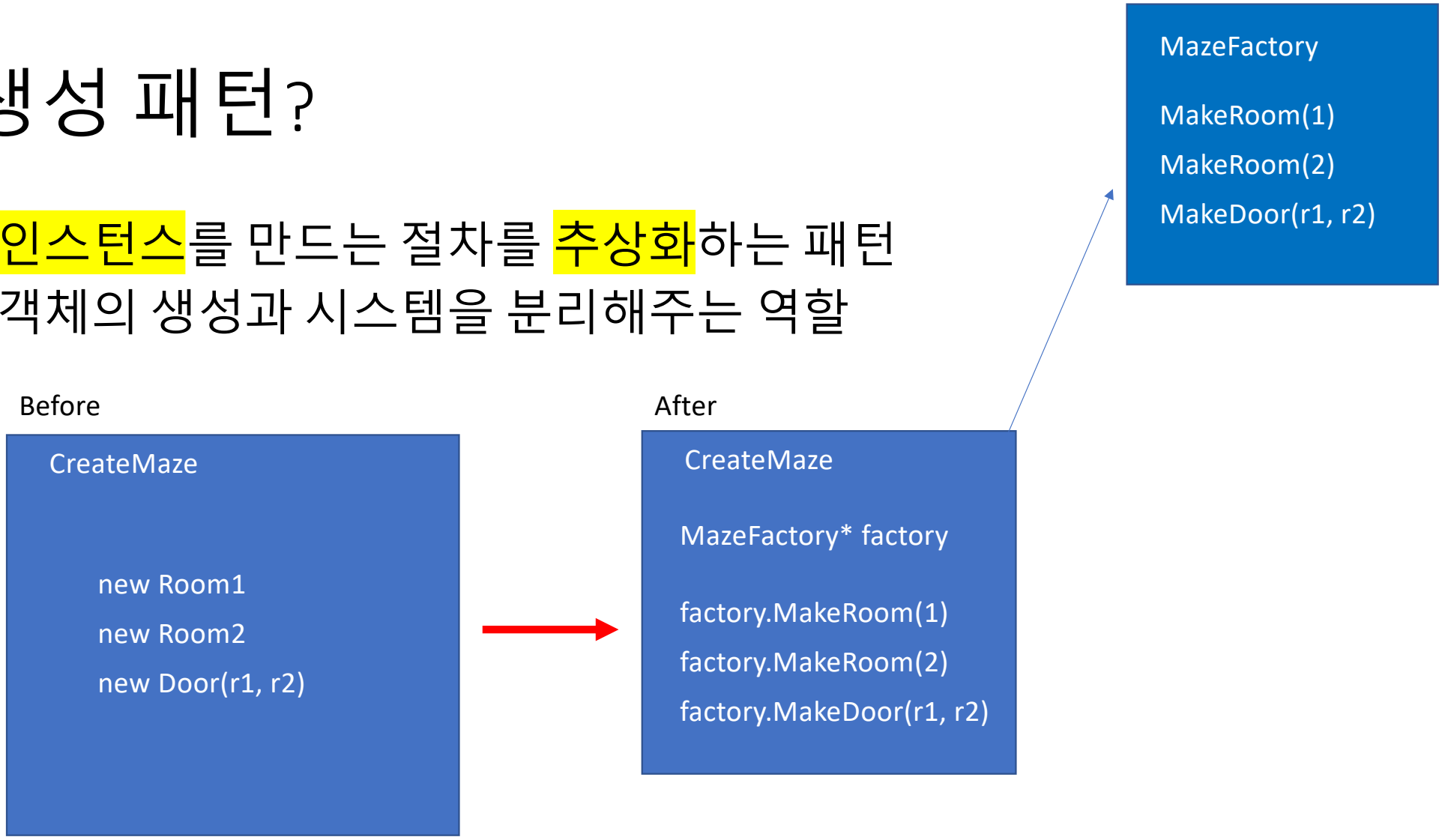
by 테디

오늘 스터디할 내용

- 생성 패턴?
- 추상팩토리 란 무엇인가?
- 추상 팩토리 pros and cons
- 추상 팩토리 구현
- 내용 정리

생성 패턴?

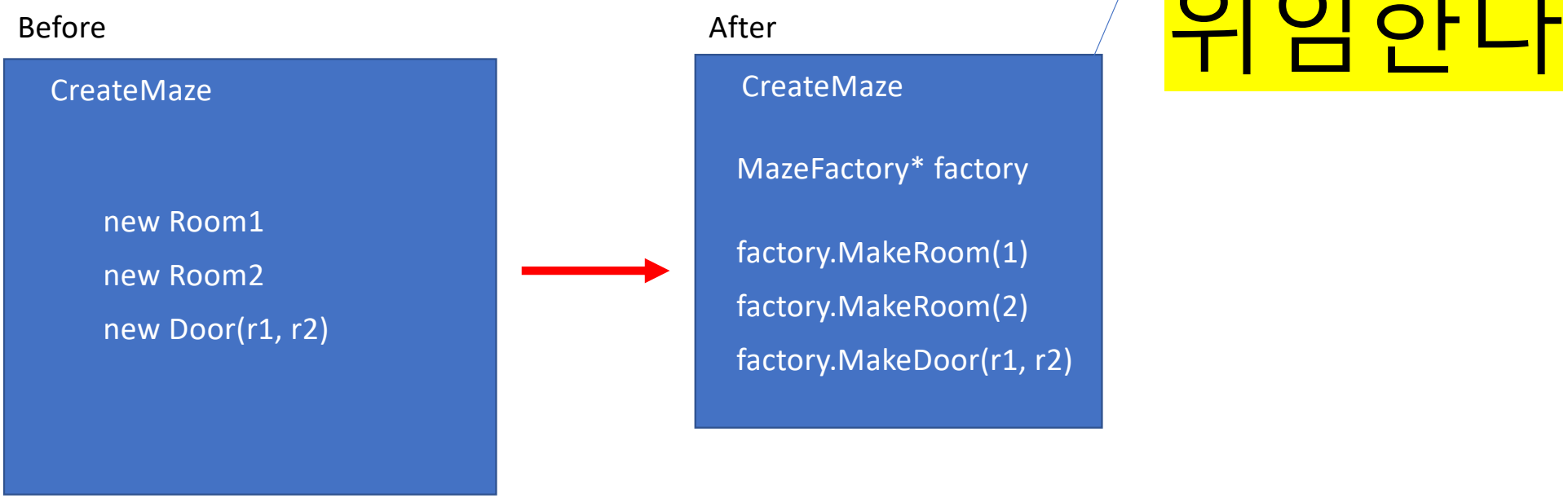
- 인스턴스를 만드는 절차를 추상화하는 패턴
- 객체의 생성과 시스템을 분리해주는 역할



객체 생성 패턴

- 인스턴스화 작업을 다른 객체에게 떠넘길 수 있다.

직접하지 않고, 다른 객체에게 전달한다.



추상팩토리란 무엇인가?

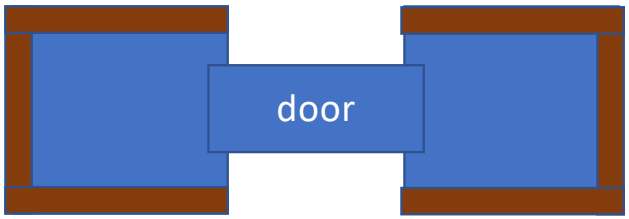
- 추상 (abstract) + 팩토리 (factory)

인스턴스를 만드는 절차를 추상화 하였다 (객체 생성 패턴)

컴퓨터 게임에 넣을 미로를 만들어보자 (1/4)

- 미로생성은 어떻게 할까?

```
CreateMaze

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




```
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 (2/4)

- 미로생성은 어떻게 할까?

```
CreateMaze

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

// 모든 방향의 면(side)들을 벽(wall)으로 초기화
```

반복되는 코드를 줄여서, 가독성(Readability)을 높였다.

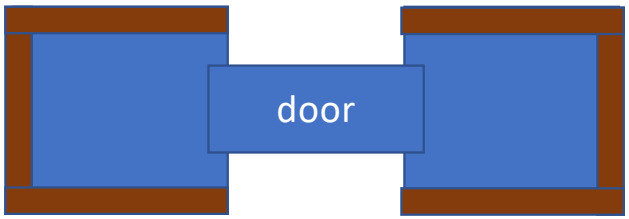
하지만, 코드의 유연성이 떨어진다.

```
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 (3/4)

- 미로생성은 어떻게 할까?

```
CreateMaze

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

// 모든 방향의 면(side)들을 벽(wall)으로 초기화
```



이렇게 직접 생성 ㄴㄴ

```
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 (4/4)

- MazeFactory 클래스

  추상 팩토리 패턴의 예

```
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); ...
```

```
class MazeFacgtory {
public:
  MazeFactory();

  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) const
    { return new Door(r1, r2); }
};
```

방, 벽, 문을 생성하기 위해
생성 방법을 알고 있는 객체를 매개변수로 넘겨받으면

생성할 객체의 유형을 달리할 수 있다.

컴퓨터 게임에 넣을 미로를 만들어보자 (4/4)

- MazeFactory 클래스

  추상 팩토리 패턴의 예

  객체생성을 직접하지 않게 한다.

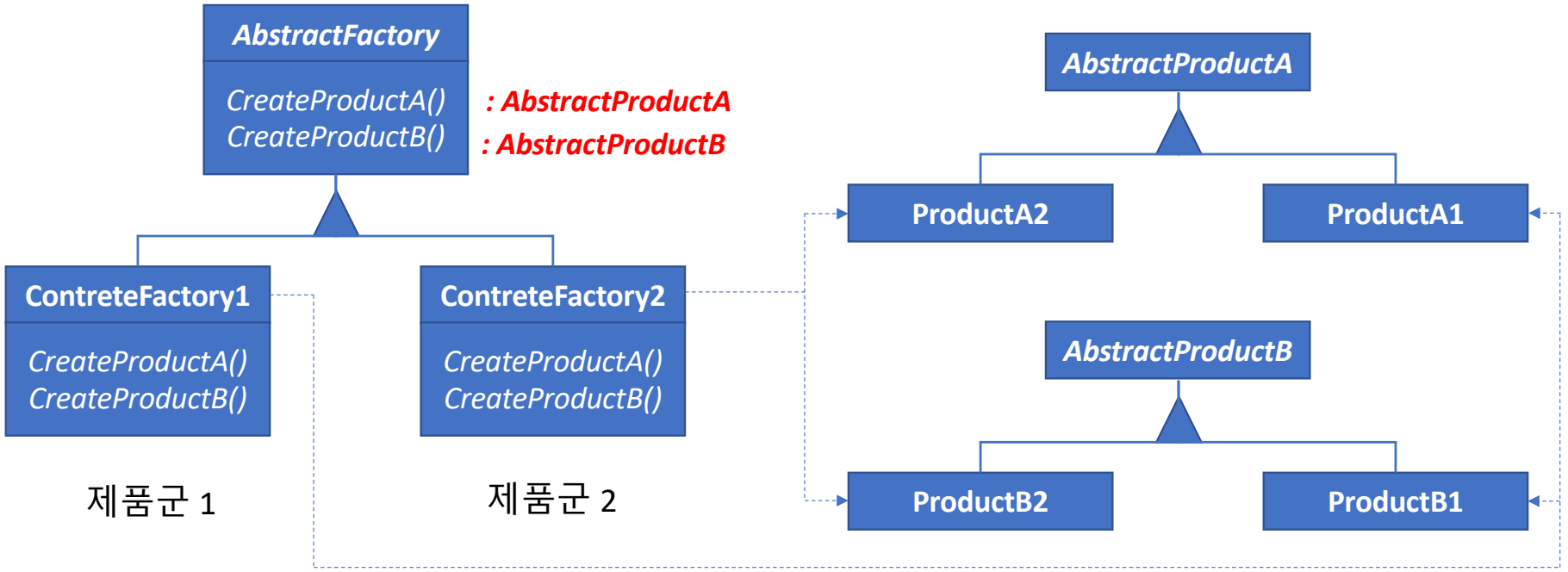
```
Maze * MazeGame :: CreateMaze () {
Maze* aMaze = new Maze;
Room* r1 = new Room(1);
Room* r2 = new Room(2);
Door* theDoor = new Door(r1, r2);

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); ...
```

코드가 요렇게 변합니다

```
Maze * MazeGame :: CreateMaze (MazeFactory& factory) {
Maze* aMaze = factory.MakeMaze();
Room* r1 = factory.MakeRoom(1);
Room* r2 = factory.MakeRoom(2);
Door* aDoor = factory.MakeDoor(r1, r2);

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

r1->SetSide(North, new Wall);
r1->SetSide(East, aDoor);
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, aDoor); ...
```

본격적으로 추상 팩토리

- 상세화된 서브클래스를 정의하지 않고도
- 서로 관련성이 있거나 독립적인 여러 객체의 군을 생성하기 위한 인터페이스를 제공



추상 팩토리 pros and cons

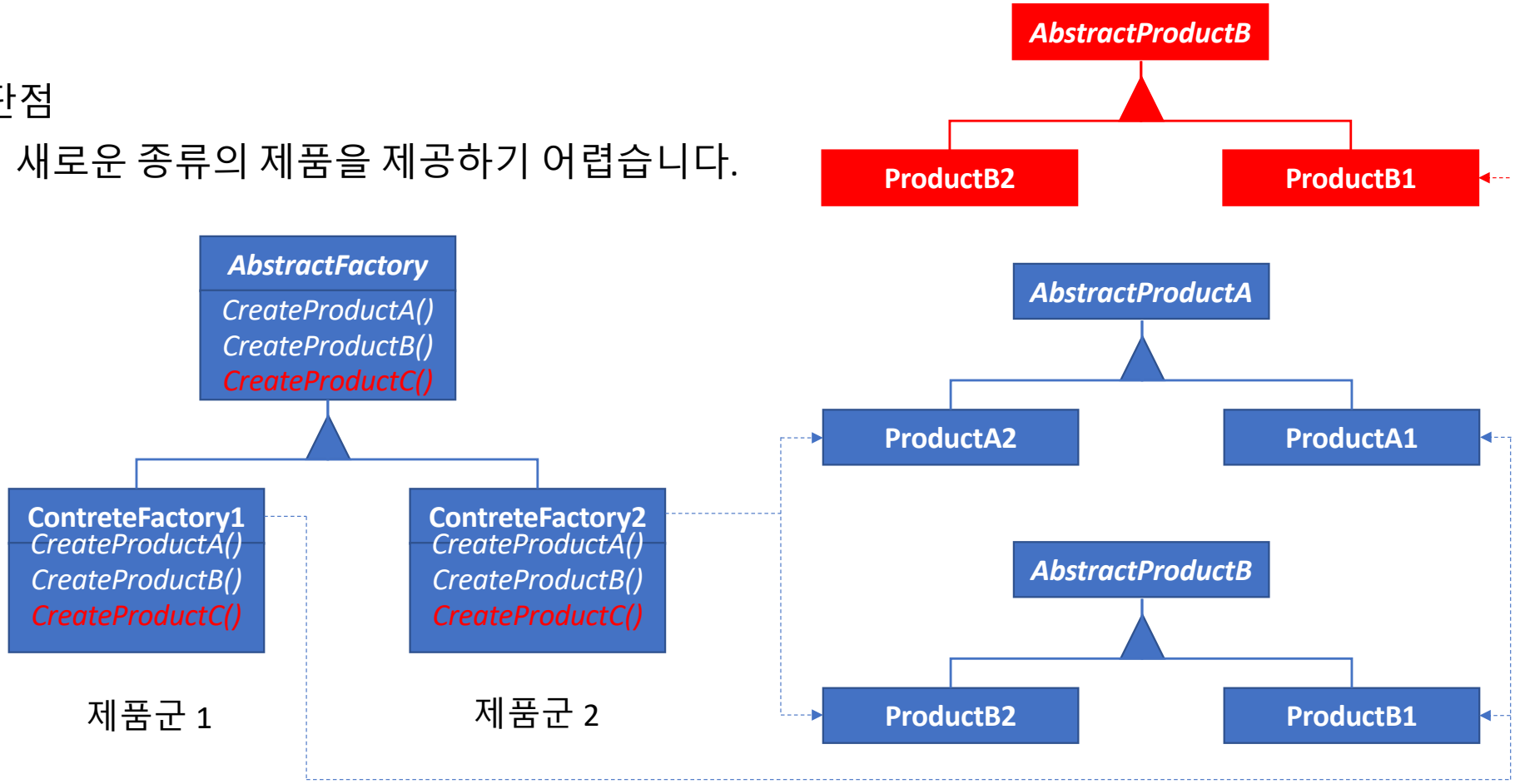
장점

- 구체적인 클래스를 분리합니다.
- 제품군을 쉽게 대체할 수 있도록 합니다.
- 제품 사이의 일관성을 증진시킵니다.

추상 팩토리 pros and cons

단점

- 새로운 종류의 제품을 제공하기 어렵습니다.

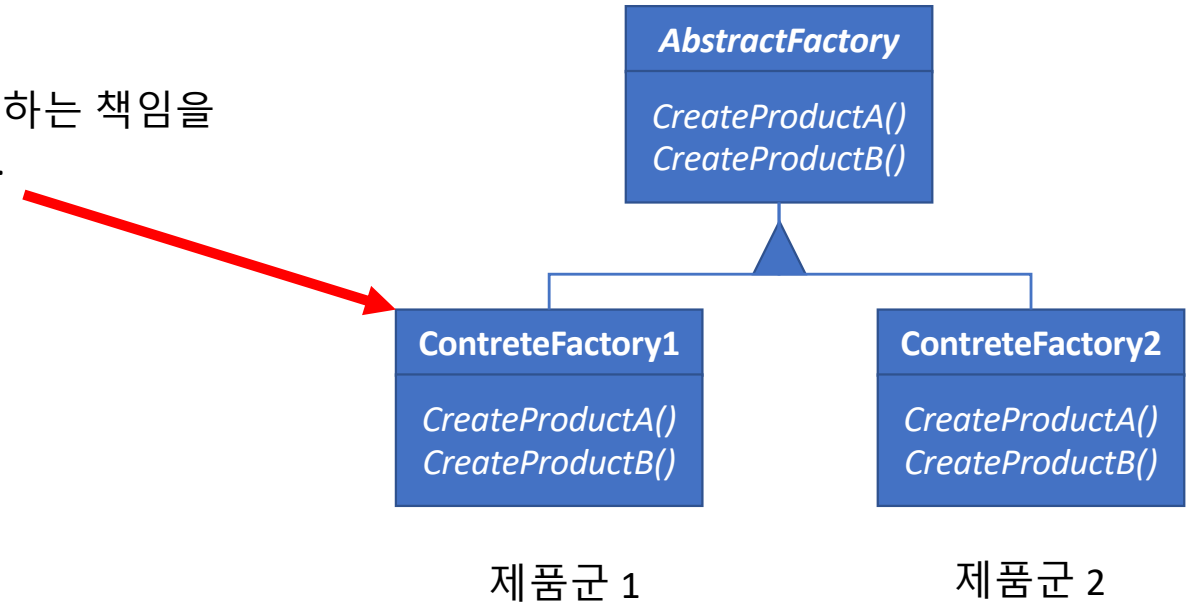


추상 팩토리 구현

- 팩토리를 단일체로 정의합니다.

// 한 제품군에 대해 하나의 ConcreteFactory 만 있으면 됨

AbstractFactory는 필요한 제품 객체를 생성하는 책임을 ConcreteFactory 서브클래스에 위임합니다.



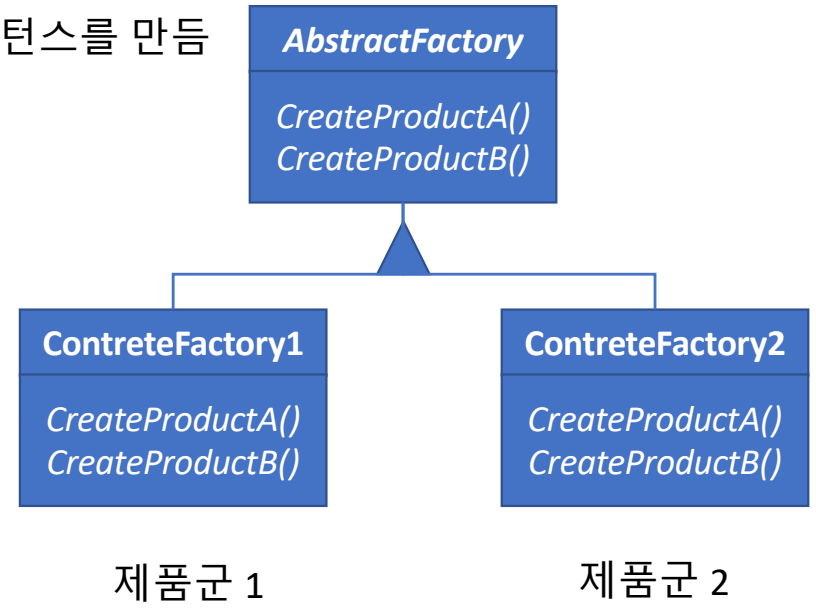
추상 팩토리 구현

- 제품을 생성합니다.

AbstractFactory는 단지 제품을 생성하기 위한 인터페이스를 선언하는 것
그것을 생성하는 책임은 Product의 서브클래스인 ConcreteProduct에 있습니다

팩토리 메서드를 재정의(overriding) 하는 식으로 각 제품의 인스턴스를 만듬

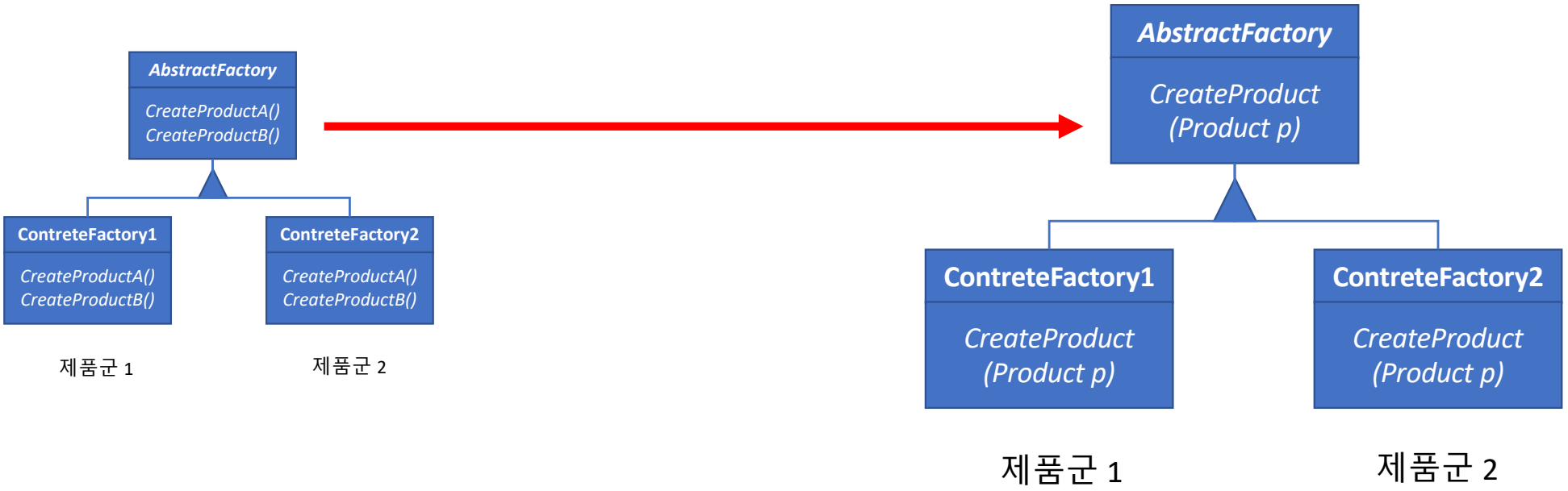
가능하다면 구체 팩토리는 프로토타입 패턴을 이용해서 구현
제품군별로 새로운 구체 팩토리를 생성할 필요를 없애줌



추상 팩토리 구현

- 확장 가능한 팩토리들을 정의합니다.

// 생성할 객체를 매개변수로 만들어 연산에 넘기면 좀 더 유연하게 제품을 생성할 수 있다



내용 정리

- 추상 (abstract) + 팩토리 (factory) 는 객체 생성 패턴이다.
- 시스템에서 객체의 생성을 분리시킨다.

참고문헌

- 에릭 감마 외 3명, Gof의 디자인패턴, 프로텍미디어 2015, p125 ~ 143