과제 - 김태우

사용 패턴

Template Method / Factory Method / Singleton

▼ Template Method Pattern

Template Method Pattern - 패턴 설명

> 상위 클래스에서 처리의 뼈대를 결정하고 하위 클래스에서 구체적 내용을 결정하는 패턴이다.
>
> 상위 클래스 - 구체적인 세부 구현을 호출 및 알고리즘의 흐름을 관리
>
> 하위 클래스 - 구체적 단계, 세부 구현
>
> 코드 재사용성, 확장성을 높인다. 비슷한 행동을 하는 클래스를 쉽게 생성할 수 있도록 돕는다.

패턴 기본 UML, 클래스

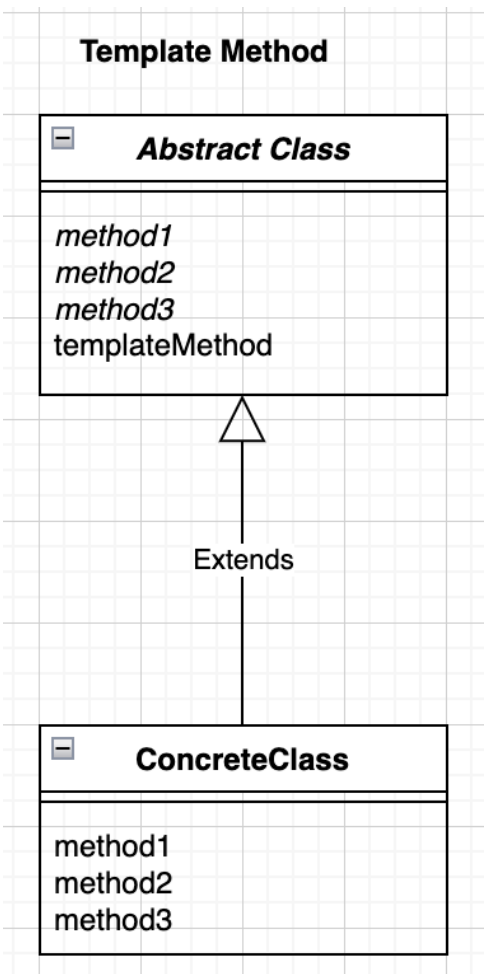
Abstract Class

- 템플릿 메소드 구현
- 템플릿 메소드에서 사용할 추상 메소드 선언

  → 추상 메소드들은 ConcreteClass에서 구현

ConcreteClass

- 정의된 추상 메소드를 구체적으로 구현



패턴 예시

- 주제 : 라면 끓이기
- 배경

> 다양한 라면이 있다.
>
> 신라면, 진라면, 삼양라면, 안성탕면 → 이들은 모두 클래스
>
> 각 라면 클래스마다, 조리를 하는 과정이 있음 → 끓이기, 면 넣기, 스프 넣기, 시간 대기
>
> **Before**
>
> 거의 비슷한 조리 과정을 가지고 있는데, 각 라면 클래스마다 개별적으로 정의한다면 라면의 종류가 늘어날 때 마다 새롭게 중복되는 메서드를 정의해야한다

> **After**
>
> 공통 - 물 끓임, 면 넣기, 스프 넣기, 대기 + 스프나 첨가물(special)
>
> +공통적인 메서드는 사용하되, 개별적으로 존재하는 과정에 대한 메서드를 추가해서 사용한다.

클래스 구상 - before / after로 구분

- before

라면1 - 일반 클래스

- 끓이기 / 면 넣기 / 스프 / 대기

라면2 - 일반 클래스

- 끓이기 / 면 넣기 / 스프 / 조미유 / 대기

라면3 - 일반 클래스

- 끓이기 / 면 넣기 / 스프 / 계란블럭 / 대기

- after

라면 - 추상클래스

- 물 끓임 / 면 넣음 / 스프 넣기 / 대기
- 로직 → 물 - 면 - 스프 - 대기

개별 라면 - 라면을 상속

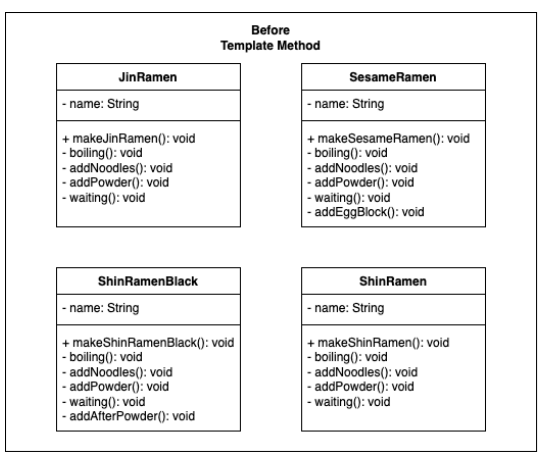
- 각 스타일에 맞는 물 끓임, 면 넣음, 스프, 대기
- 특별한 과정이 있는 경우 별도 설정

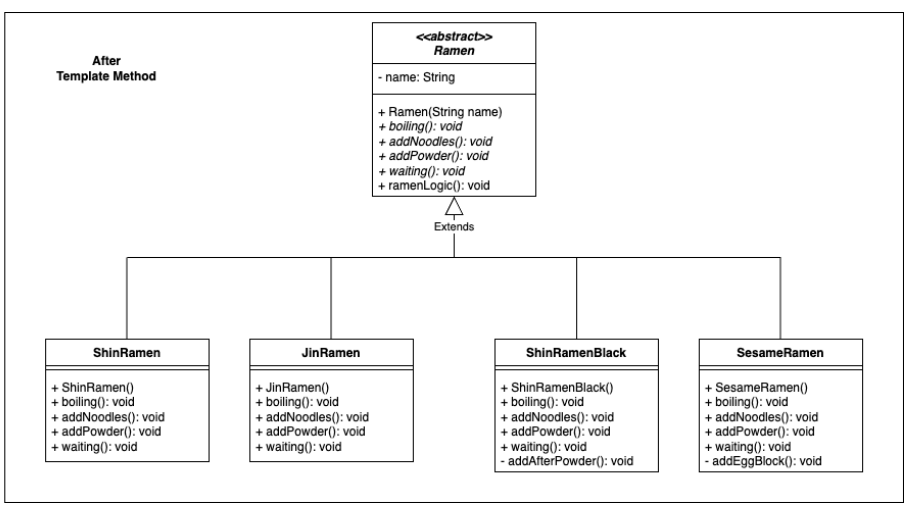
클래스 세부 정의

```
ShinRamen
- name: String
+ makeShinRamen()
- boiling(): void
- addNoodles(): void
- addPowder(): void
- waiting(): void

SesameRamen
- name: String
+ makeSesameRamen()
- boiling(): void
- addNoodles(): void
- addPowder(): void
- waiting(): void
- addEggBlock(): void

JinRamen
- name: String
+ makeJinRamen()
- boiling(): void
- addNoodles(): void
- addPowder(): void
- waiting(): void

ShinRamenBlack
- name: String
+ makeShinRamenBlack()
- boiling(): void
- addNoodles(): void
- addPowder(): void
- waiting(): void
- addAfterPowder(): void
```

```
<<Abstract>>
Ramen
- name: String
+ Ramen(String name)
+ boiling(): void
+ addNoodles(): void
+ addPowder(): void
+ waiting(): void
+ ramenLogic(): void


ShinRamen, ShinRamenBlack,
JinRamen, SesameRamen
+ Ramen(String name)
+ boiling(): void
+ addNoodles(): void
+ addPowder(): void
+ waiting(): void
+ ramenLogic(): void
- special(): void
```

UML





UML 설명

- Before → 각 라면별로 별도의 조리 방법을 가지고 있기에 별도의 클래스로 구분되어있다.
- After

  Ramen이라는 추상 클래스가 있고, 해당 추상 클래스에서 각 세부 과정에 대한 추상 메서드가 있다.

  private으로 라면의 이름을 정하기 위한 변수가 있고, 공통 조리 과정에 대한 메서드 ramenLogic()이 있다.

  각 세부적인 라면 클래스는 Ramen 추상 클래스를 상속받는다. 정의된 세부 조리 과정에 대한 추상 메서드들을 재정의 하도록 한다. 추가적으로 해당 라면의 특별한 조리과정이 있는 경우 클래스 내 별도 생성하여 사용한다.

▼ Factory Method Pattern

Factory Method Pattern - 패턴 설명

> Template Method - 상위클래스 - 처리의 뼈대 / 하위 클래스 - 구체적인 처리의 살
>
> ⇒ 이 Template Method 패턴을 인스턴스 생성에 적용한 것이다.
>
> 인스턴스 생성 방법 - 상위 클래스 / 구체적인 살은 모두 하위 클래스에서 붙인다.

패턴 기본 UML, 클래스

<프레임워크 - 추상적 뼈대>

Creator

- Product 역을 생성하는 추상 클래스
- 구체적 내용은 하위 ConcreteCreator가 결정

Product

- 생성되는 인스턴스가 가져야 할 인터페이스(API)를 결정하는 추상 클래스
- 구체적 내용은 하위 ConcreteProduct에서 결정

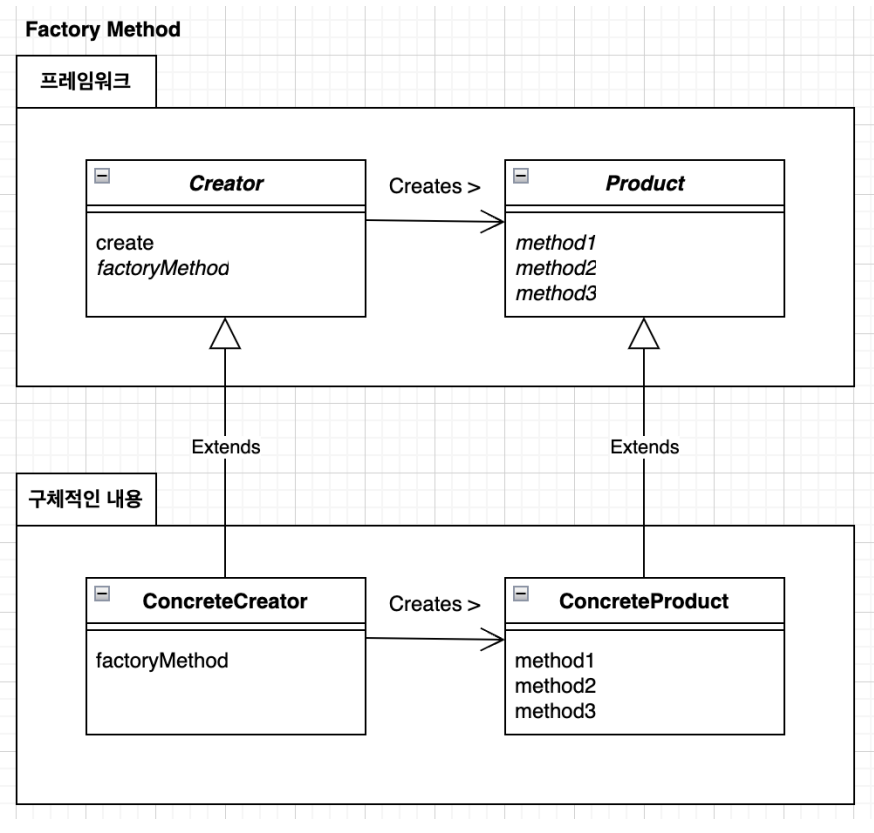
<구체적 구현>

ConcreteCreator

- 구체적인 제품을 만들 클래스 결정

ConcreteProduct

- 구체적인 제품을 결정



패턴 예시

- 주제 : 라면 공장
- 배경

> 라면을 생산하는 공장이 있고, 각 공장은 특정한 라면을 생산한다.
>
> 진라면을 생산하는 공장, 신라면을 생성하는 공장과 같이 존재한다.
>
> **Before**
>
> 새로운 라면을 생산하기 위해선 새로운 공장을 만들어야한다.
>
> 각 라면별로 생산 공장이 필요하다.
>
> **After**
>
> 공장과 상품에 대한 추상 클래스를 만든다.

> 각 공종 생산 과정을 두고 새로운 라면을 생산할 공장을 해당 공장을 상속받아서 정의해준다.
>
> product - 상품, concreteProduct - 라면
>
> factory - 공장, concreteFactory - 라면 생산 공장

클래스 구상 - before / after로 구분

- before

라면1 - 일반 클래스

- 포장
- 배송

라면2 - 일반 클래스

- 포장
- 배송

라면1 공장 - 일반 클래스

- 라면1 생산

라면2 공장 - 일반 클래스

- 라면2 생산

- after

<프레임워크>

Factory - 추상 클래스

- 상품 생산

Product - 추상 클래스

- 상품 포장
- 상품 배송

<구체적 내용>

Ramen1Factory - 일반 클래스

- 라면1을 생산

Ramen2Factory - 일반 클래스

- 라면2을 생산

Ramen1 - 일반 클래스

- 라면1 포장
- 라면1 배송

Ramen2 - 일반 클래스

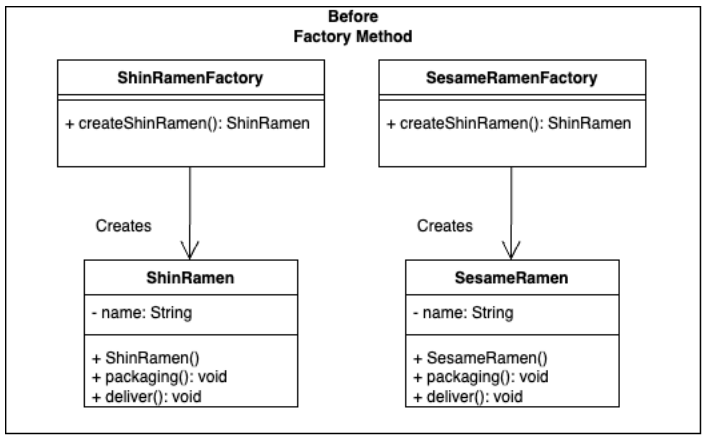
- 라면2 포장
- 라면2 배송

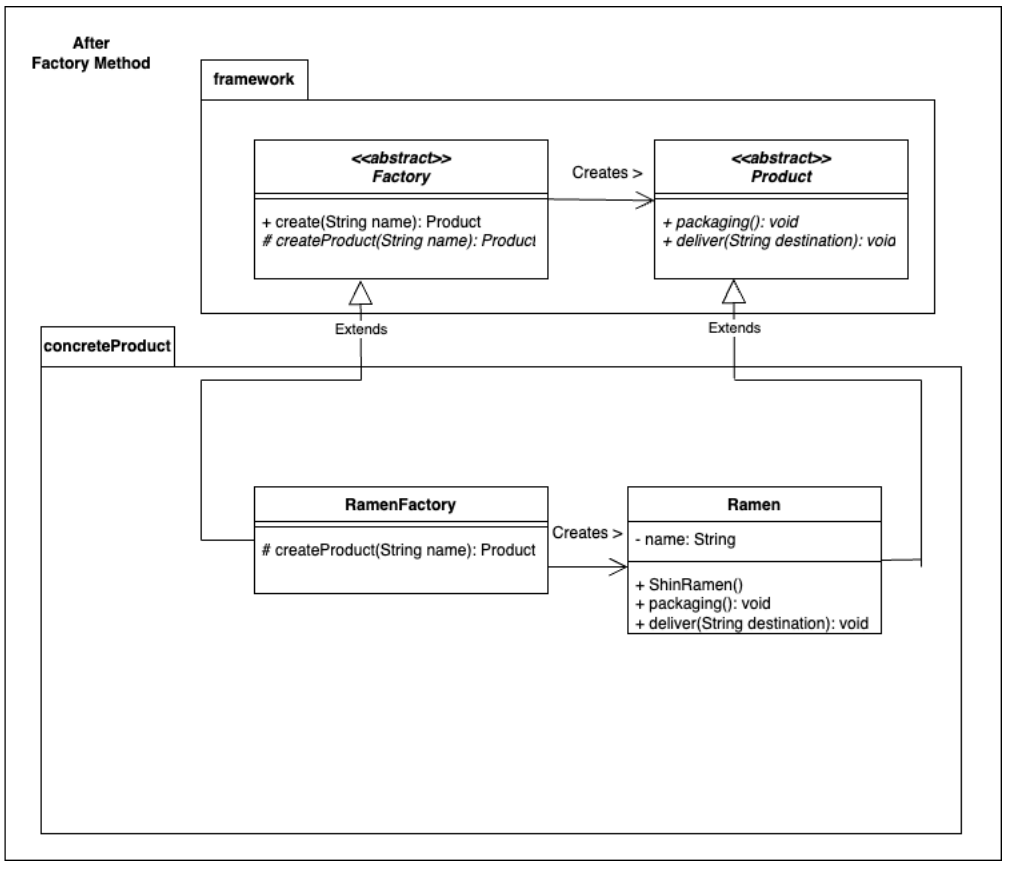
클래스 정의

```
ShinRamen
- name: String
+ packaging()
+ deliver()

ShinRamenFactory
+ createShinRamen()

SesameRamen
- name: String
+ packaging()
+ deliver()

SesameRamenFactory
+ createSesameRamen()
```

```
<프레임워크>
<<abstract>>
Product
+ packaging()
+ deliver(Product product)

<<abstract>>
Factory
+ create()
# createProduct()

<구체적 내용>
Ramen
- name: String
+ Ramen()
+ packaging()
+ deliver(Product product)

RamenFactory
# createProduct()
```

UML





UML 설명

- Before → 특정 라면을 생성하는 공장이 별도로 존재하고, 각 공장에서는 각 라면 클래스를 create 한다. 별도 상속 관계가 없다.
- After

  크게 추상 클래스를 가지는 framework 패키지와 구체적으로 구현하는 클래스를 가지는 concreteProduct 패키지로 구분이된다.

  - framework 패키지에는 무언가를 생산하는 Factory, 생성될 상품인 Product 추상클래스가 있다.
    - Factory 클래스의 경우 Product를 생산하는 create 메서드를 가진다. 해당 메서드는 추상 메서드로 구현된 createProduct를 호출하여 사용하는 메서드로 final로 지정하여 Factory 클래스를 상속받는 모든 클래스는 해당 함수를 통해 생성을 하도록 지정하였다.
    - Product 클래스의 경우 생성되는 상품의 기능에 대한 메서드와 배송을 위한 메서드가 추상 메서드로 정의되어 있다.
  - concreteProduct 패키지에는 Factory와 Product를 각각 상속받는 구체적으로 구현한 RamenFactory, Ramen이 있다.
    - RamenFactory의 경우 Factory 내부의 추상 메서드를 override 하였고, Ramen을 생성한다.
    - Ramen의 경우 Product 내부의 추상 메서드를 override 하였다.

▼ Singleton Pattern

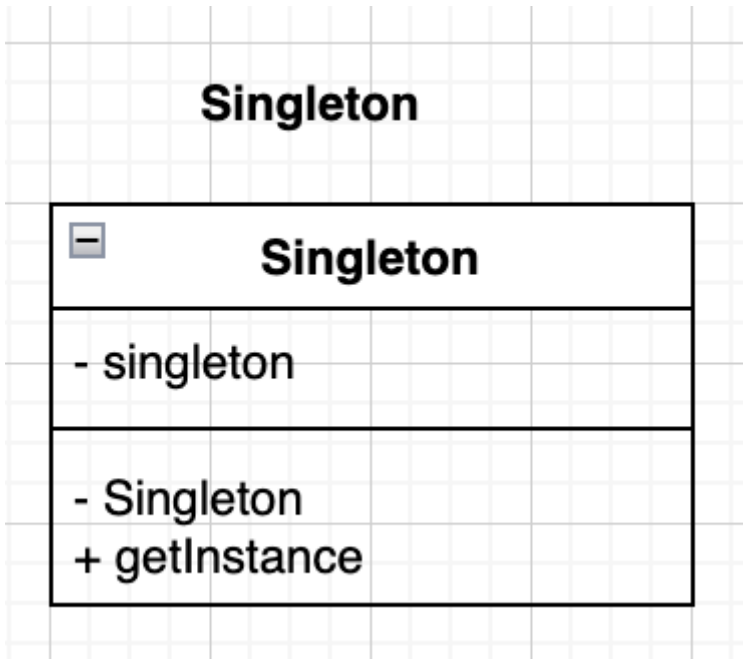
Singleton Pattern - 패턴 설명

> 인스턴스가 하나만 존재하는 것을 보증하는 패턴
>
> 인스턴스 수를 제한하여 전제 조건을 늘린다. → 해당 전제 조건 하에서 프로그래밍이 가능
>
> 클래스의 생성자는 private이다. → 외부에서 생성자 호출을 금지하기 위해서 → 유일한 인스턴스를 얻기 위해 getInstance를 이용한다.

패턴 기본 UML, 클래스

Singleton

- 유일한 인스턴스를 얻기 위한 static 메소드를 가진다.
- 항상 같은 인스턴스를 반환



패턴 예시

- 주제 : 여러개의 라면 공장과 공장의 생산 기록
- 배경

> 여러 개의 라면을 생산하는 공장이 있다. 이들의 생산 기록을 관리하고 싶다.
>
> 공장의 경우 Factory Pattern으로 생성 및 사용되고 있지만, 각 공장별로 생산 기록의 관리가 별도로 되어있다. 이를 하나의 로그로 관리해서 각 공장이름과 생산 제품, 시간을 가지는 것이 목표이다.
>
> **Before**
>
> 여러 개의 공장이 있고, 각 공장은 다양한 라면을 생산한다. 이들은 생산될 때 생산에 관한 정보를 로그에 기록한다. 하지만 개별 공장으로 로그가 관리 되기에 각 라면별로 생산 정보를 추적하기에 불편하다.
>
> **After**
>
> 여러 개의 공장의 기록을 하나의 로그로 관리할 수 있다. 각 공장의 생산기록은 하나의 생산 기록에 추가되고, 이를 언제든지 볼 수 있다.

클래스 구상 - before / after로 구분

- before

factory method pattern 단독

<framework>

공장 - 추상 클래스

- 제품 생성
- 생성 기록

제품 - 추상 클래스

- 생성 확인

<concrete>

제품 공장 - 일반 클래스

- 구체적 제품 생성
- 구체적 제품 생성 기록

구체적 제품 - 일반 클래스

- 생성 확인

- after

factory method pattern + singleton pattern

공장 기록 객체 - singleton

- 기록 객체 반환
- 생성 기록

<framework>

공장 - 추상 클래스

- 제품 생성
- 생성 기록(기록 객체를 통해)

제품 - 추상 클래스

- 생성 확인

<concrete>

제품 공장 - 일반 클래스

- 구체적 제품 생성
- 구체적 제품 생성 기록(기록 객체 통해)

구체적 제품

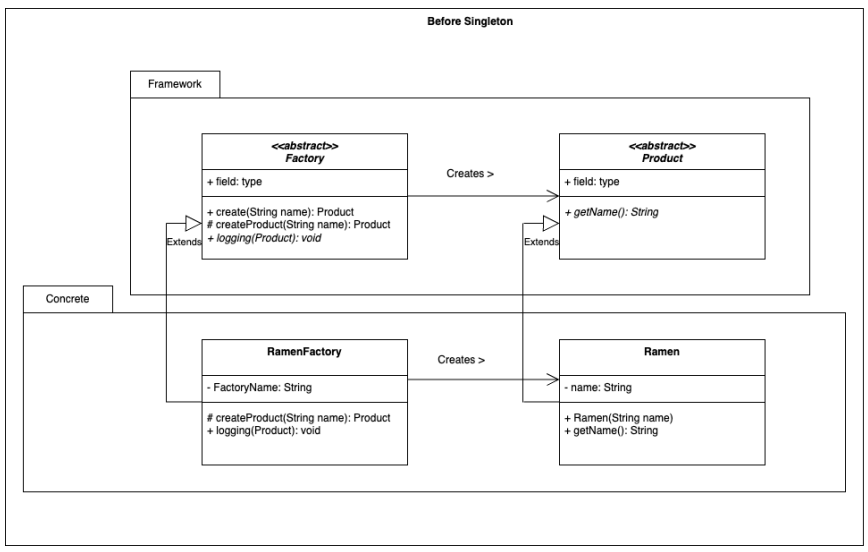
- 생성 확인

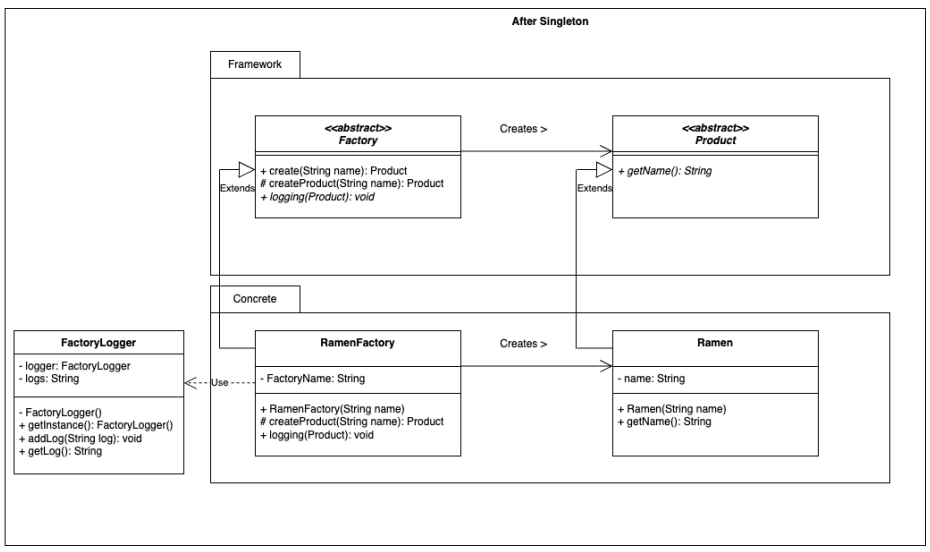
클래스 정의

```
<framework>
<<abstract>>
Factory
+ create(String name)
# createProduct(String name)
+ logging(Product product)

<<abstract>>
Product
+ getName()
+ checkProduct()

<concrete>

RamenFactory
- FactoryName: String
# createProduct(String name)
+ logging(Product product)

Ramen
- name: String
+ getName(): String
+ checkProduct()
```

```
<Singleton>
FactoryLogger
- logger: FactoryLogger
- logs: String
+ getInstance(): FactoryLogger
+ addLog()
+ getLog()

<framework>
<<abstract>>
Factory
+ create()
# createProduct()
+ logging()

<<abstract>>
Product
+ getName(): String

<concrete>

RamenFactory
- FactoryName: String
# createProduct(String name)
+ logging(Product product)

Ramen
- name: String
+ getName(): String
```

UML





UML 설명

- Before → factory method 패턴을 적용한 라면 공장이 있다. 하지만 각 공장의 생산 기록은 각 공장별로 관리하고 있다. Factory를 상속받는 RamenFactory, Product를 상속받는 Ramen이 있다. 그리고 Factory는 Product를 생성한다. 마찬가지로 RamenFactory는 Ramen을 생성하는 구조이다.
- After

  기본적으로 Factory Method 패턴이 적용된 Ramen과 이를 생산하는 RamenFactory가 있다.

  Factory를 상속받은 RamenFactory는 여러개의 공장을 만들 수 있다.

  여러개의 공장의 생산기록은 singleton 패턴이 적용된 FactoryLogger로 관리가 된다.

  Private으로 생성자를 정의하고, FactoryLogger객체를 하나만 존재할 수 있게 static으로 설정하였다. 또한 모든 공장의 로그 기록을 하나로 관리하기 위해 logs를 static으로 설정하였다.

  각 공장들은 FactoryLogger의 객체를 getInstance로 받아서 로그를 추가할 수 있고, 추가된 로그는 getLog를 통해 확인이 가능하다.

  Concrete 패키지의 RamenFactory는 FactoryLogger를 사용하는 관계가 표시되어있다.