GoF 의 디자인 패턴

정리 : 이영한 (hotyoung@windowslove.net)

목 차

1. 생성 패턴 (CREATIONAL PATTERNS) 5

1.1. ABSTRACT FACTORY 5
1.1.1. 구조 5
1.1.2. 참여 객체 5
1.2. BUILDER 6
1.2.1. 구조 6
1.2.2. 참여 객체 6
1.3. FACTORY METHOD 8
1.3.1. 구조 8
1.3.2. 참여 객체 8
1.4. PROTOTYPE 9
1.4.1. 구조 9
1.4.2. 참여 객체 9
1.5. SINGLETON 10
1.5.1. 구조 10
1.5.2. 참여 객체 10

2. 구조 패턴 (STRUCTURAL PATTERNS) 11

2.1. ADAPTER 11
2.1.1. 구조 11
2.1.2. 참여 객체 12
2.2. BRIDGE 13
2.2.1. 구조 13
2.2.2. 참여 객체 13
2.3. COMPOSITE 14
2.3.1. 구조 14
2.3.2. 참여 객체 14
2.4. DECORATOR 15
2.4.1. 구조 15
2.4.2. 참여 객체 15

2.5. FAÇADE **16**
2.5.1. 구조 16
2.5.2. 참여 객체 16
2.6. FLYWEIGHT **17**
2.6.1. 구조 17
2.6.2. 참여 객체 17
2.7. PROXY **19**
2.7.1. 구조 19
2.7.2. 참여 객체 19
2.7.3. 프록시 종류 19

3. 행위 패턴 (BEHAVIORAL PATTERNS) **21**

3.1. CHAIN OF RESPONSIBILITY **21**
3.1.1. 구조 21
3.1.2. 참여객체 21
3.2. COMMAND **22**
3.2.1. 구조 22
3.2.2. 참여객체 22
3.3. INTERPRETER **23**
3.3.1. 구조 23
3.3.2. 참여객체 23
3.4. ITERATOR **25**
3.4.1. 구조 25
3.4.2. 참여객체 25
3.5. MEDIATOR **26**
3.5.1. 구조 26
3.5.2. 참여객체 26
3.6. MEMENTO **27**
3.6.1. 구조 27
3.6.2. 참여객체 27
3.7. OBSERVER **28**
3.7.1. 구조 28
3.7.2. 참여객체 28
3.8. STATE **29**
3.8.1. 구조 29

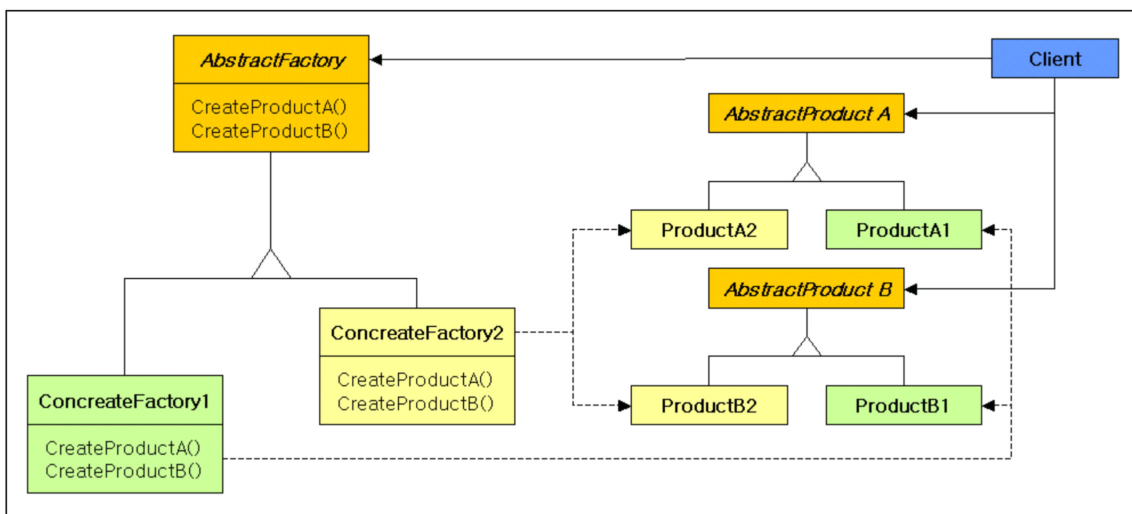
3.8.2. 참여객체 29
3.9. STRATEGY **30**
3.9.1. 구조 30
3.9.2. 참여객체 30
3.10. TEMPLATE METHOD **31**
3.10.1. 구조 31
3.10.2. 참여객체 31
3.11. VISITOR **32**
3.11.1. 구조 32
3.11.2. 참여객체 32

1. 생성 패턴 (Creational Patterns)

1.1. Abstract Factory

구체적인 클래스를 지정하지 않고 관련성을 갖는 객체들의 집합을 생성하거나 서로 독립적인 객체들의 집합을 생성할 수 있는 인터페이스를 제공한다.

1.1.1. 구조



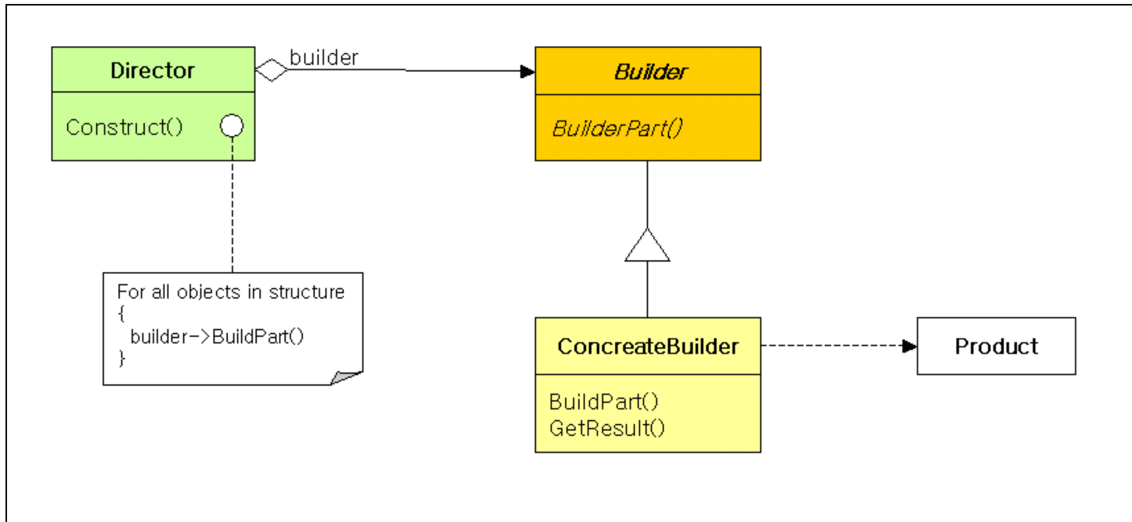
1.1.2. 참여 객체

- AbstractFactory: 개념적 제품에 대한 객체를 생성하는 오퍼레이션으로 인터페이스를 정의한다.
- ConcreateFactory: 구체적인 제품에 대한 객체를 생성하는 오퍼레이션을 구현한다.
- AbstractProduct: 개념적 제품 객체에 대한 인터페이스를 정의한다.
- ConcreateProduct: 구체적으로 팩토리가 생성할 객체를 정의하고, AbstractProduct 가 정의하고 있는 인터페이스를 구현한다.
- Client: AbstractFactory 와 AbstractProduct 클래스에 선언된 인터페이스를 사용한다.

1.2. Builder

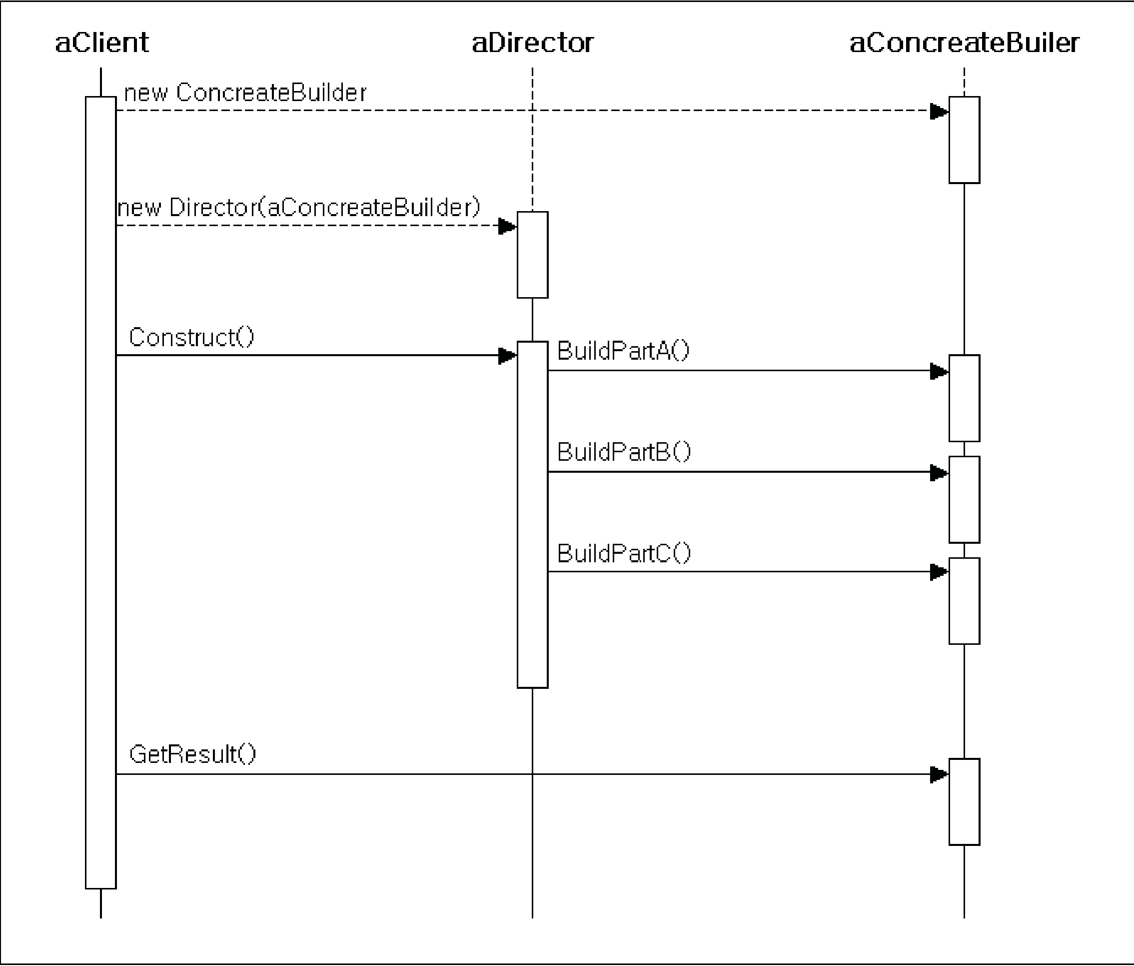
복합 객체의 생성 과정과 표현 방법을 분리함으로써 동일한 생성 공정이 서로 다른 표현을 만들 수 있게 한다.

1.2.1. 구조



1.2.2. 참여 객체

- **Builder**: Product 객체의 일부 요소들을 생성하기 위한 추상 인터페이스를 정의한다.
- **ConcreateBuilder**: Builder 클래스에 정의된 인터페이스를 구현하며, 제품의 부품들을 모아 빌더를 합성한다. 생성한 요소의 표현을 정의하고 관리한다. 또한 제품을 검색하는데 필요한 인터페이스를 제공한다.
- **Director**: Builder 인터페이스를 사용하는 객체를 합성한다.
- **Product**: 구축할 복합 객체를 표현한다. ConcreateBuilder 는 제품의 내부 표현을 구축하고 어떻게 모아 하나로 만드는지의 과정을 정의한다.

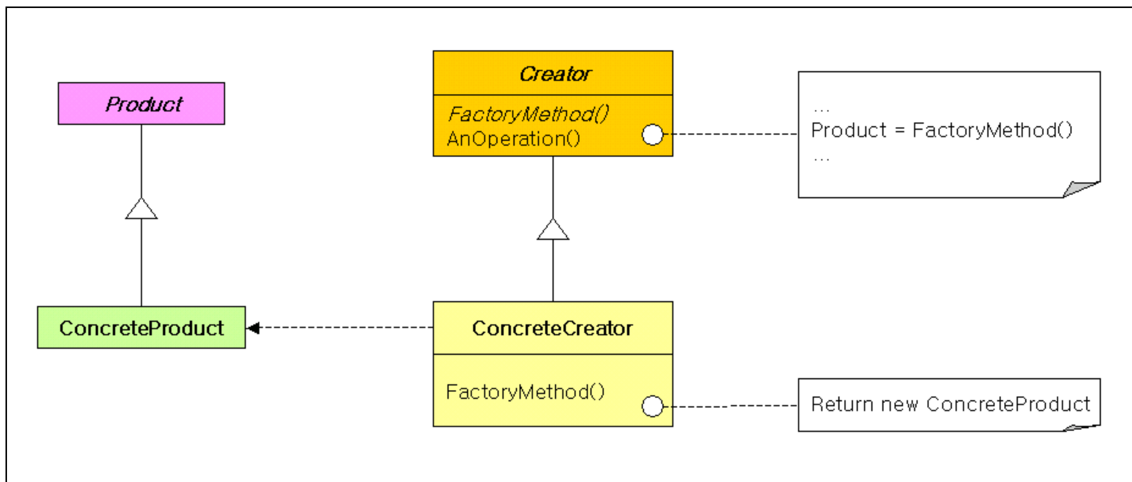


aClient
aDirector
aConcreateBuiler
new ConcreateBuilder
new Director(aConcreateBuilder)
Construct()
BuildPartA()
BuildPartB()
BuildPartC()
GetResult()

1.3. Factory Method

객체를 생성하는 인터페이스를 정의하지만, 인스턴스를 만들 클래스의 결정은 서브클래스가 한다. Factory Method 패턴에서는 클래스의 인스턴스를 만드는 시점을 서비클래스로 미룬다.

1.3.1. 구조



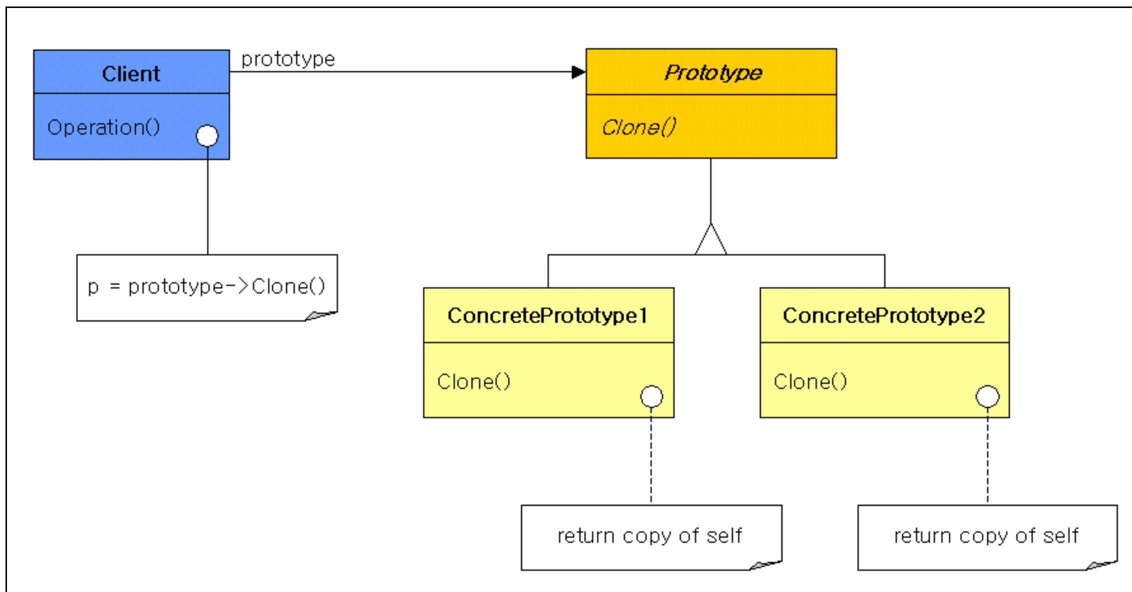
1.3.2. 참여 객체

- **Product**: 팩토리 메소드가 생성하는 객체의 인터페이스를 정의한다.
- **ConcreteProduct**: Product 클래스에 정의된 인터페이스를 실제로 구현한다.
- **Creator**: Product 타입의 객체를 반환하는 팩토리 메소드를 선언한다. Creator 클래스는 팩토리 메소드를 기본적으로 구현하는데, 이 구현에서는 ConcreteProduct 객체를 반환한다. 또한 Product 객체의 생성을 위해 팩토리 메소드를 호출한다.
- **ConcreteCreator**: ConcreteProduct 의 인스턴스를 반환하기 위해 팩토리 메소드를 재정의한다.

1.4. Prototype

프로토타입의 인스턴스를 이용해서 생성할 객체의 종류를 명세하고 이 프로토타입을 복사해서 새로운 객체를 생성한다.

1.4.1. 구조



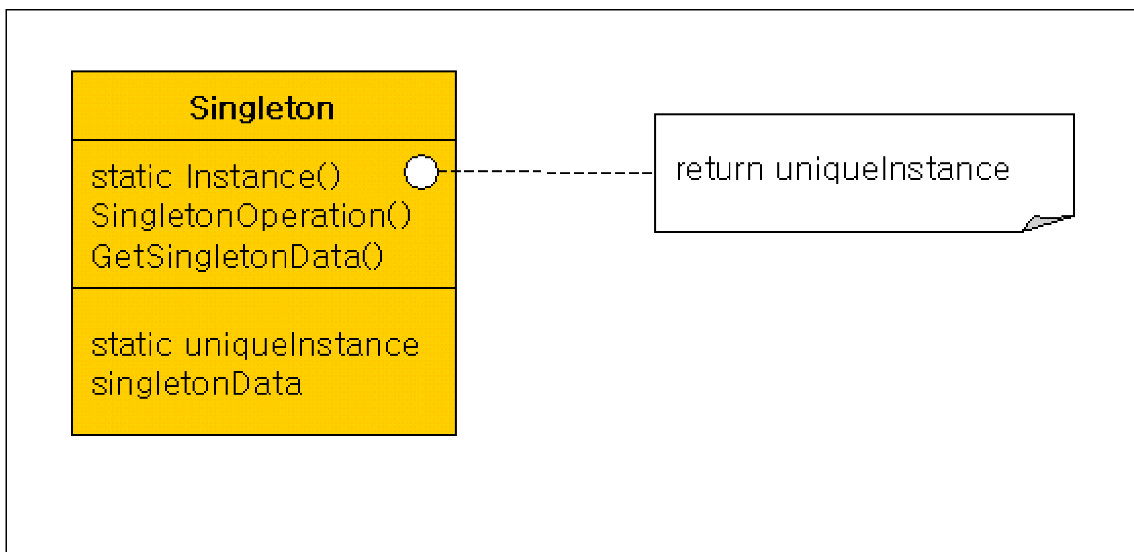
1.4.2. 참여 객체

- Prototype: 자신을 복제하는데 필요한 인터페이스를 정의한다.
- ConcretePrototype: 자신을 복제하는 오퍼레이션을 구현한다.
- Client: prototype 에 복제를 요청함으로써 새로운 객체를 생성한다.

1.5. Singleton

클래스의 인스턴스는 오직 하나임을 보장하며 이 인스턴스에 접근할 수 있는 방법을 제공한다.

1.5.1. 구조



1.5.2. 참여 객체

- Singleton: Instance() 오퍼레이션을 정의하여, 유일한 인스턴스로의 접근이 가능하도록 한다. Instance() 오페레이션은 클래스 오퍼레이션이다. 클래스 오퍼레이션이란 C++에서는 static 으로 정의되는 멤버함수로 클래스에서 만드는 모든 인스턴스에 걸쳐서 공유하는 함수이다. 이로써 유일한 인스턴스의 생성에 대한 책임을 지게 된다.

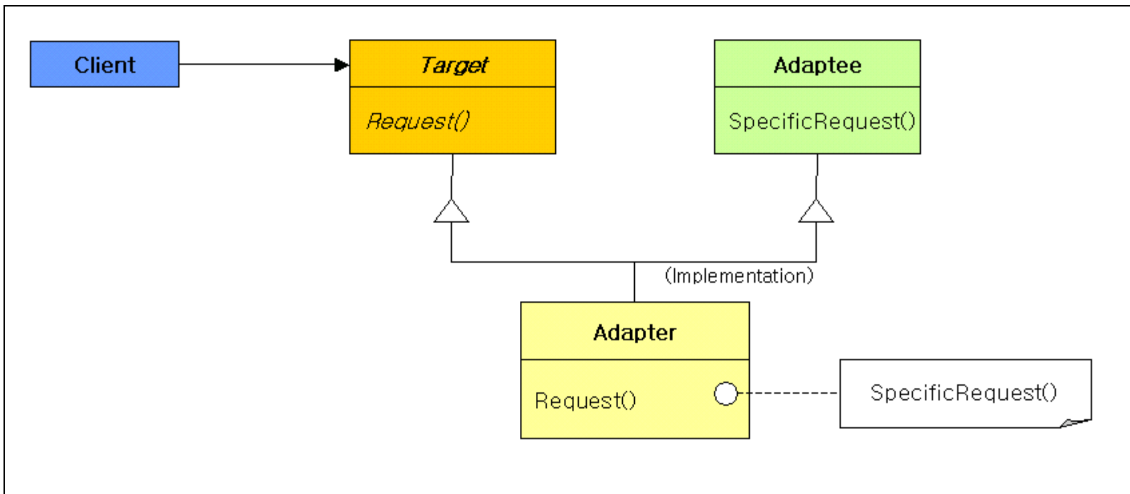
2. 구조 패턴 (Structural Patterns)

2.1. Adapter

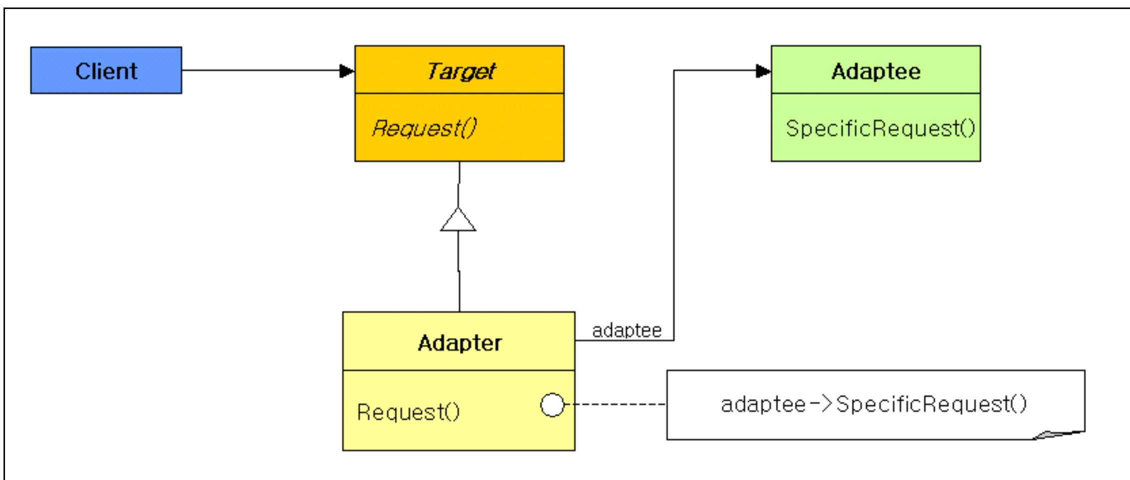
클래스의 인터페이스를 클라이언트가 기대하는 다른 인터페이스로 변환한다. Adapter 패턴은 호환성이 없는 인터페이스 때문에 함께 사용할 수 없는 클래스를 개보하여 함께 작동하도록 해 준다.

2.1.1. 구조

(1) 다중상속을 활용해서 설계한 Adapter 패턴



(2) 객체합성에 의한 Adapter 패턴



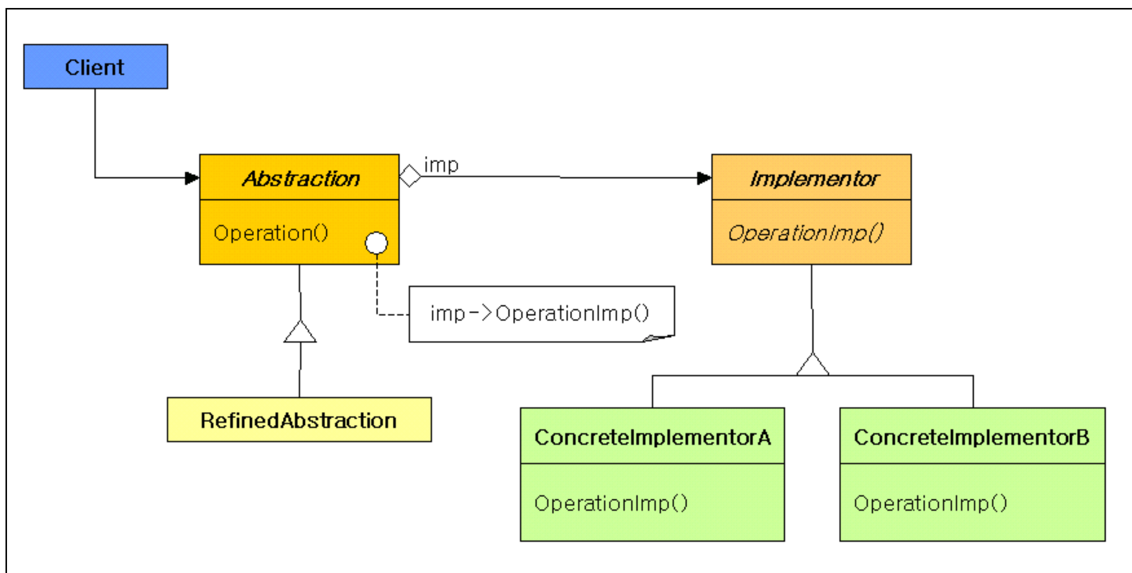
2.1.2. 참여 객체

- **Target**: 클라이언트가 사용할 도메인에 종속적인 인터페이스를 정의하고 있는 클래스이다.
- **Client**: Target 인터페이스를 만족하는 객체와 동작할 대상이다.
- **Adaptee**: 인터페이스 개조가 필요한 기존의 인터페이스를 정의하고 있는 클래스이다.
- **Adapter**: Target 인터페이스에 Adaptee 의 인터페이스를 맞춰주는 클래스

2.2. Bridge

추상화와 구현을 분리하여 각각을 독립적으로 변형할 수 있게 한다. 구현과 추상화 개념을 분리하려는 것이다. 이로써 구현 자체도 하나의 추상화 개념으로 다양한 변형이 가능해지고, 구현과 독립적으로 인터페이스도 다양함을 가질 수 있게 된다.

2.2.1. 구조



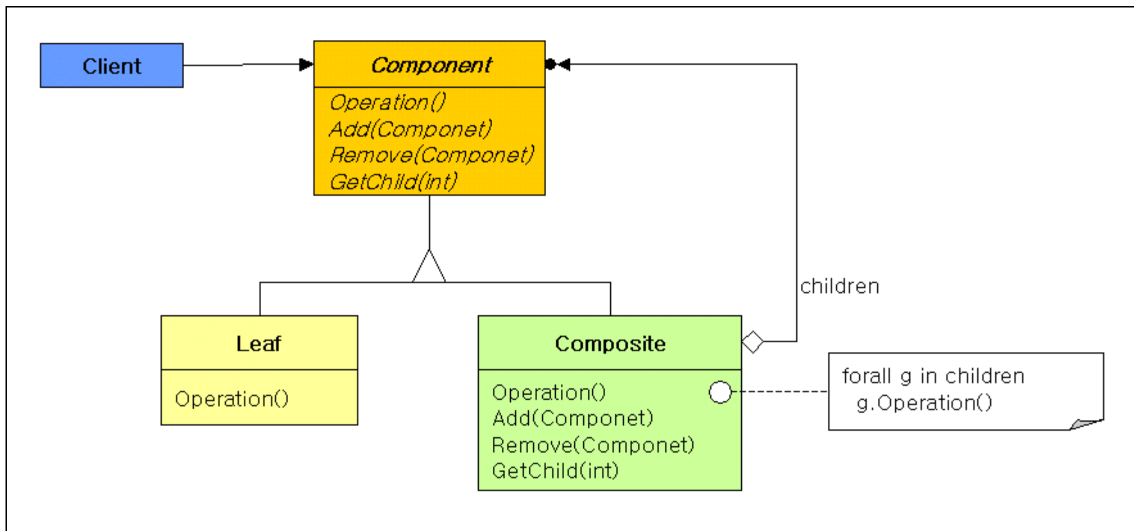
2.2.2. 참여 객체

- Abstraction: 추상화 개념에 대한 인터페이스를 제공하고 객체 구현자(Implementor)에 대한 참조자를 관리한다.
- RefinedAbstraction: 추상화 개념에 정의된 인터페이스를 확장한다.
- Implementor: 구현 클래스에 대한 인터페이스를 제공한다. 다시 말해서 실질적인 구현을 제공한 서브클래스들에 있어서 공통적인 오퍼레이션의 시그니처만을 정의하고 있다. 이 인터페이스는 Abstraction 클래스에 정의된 인터페이스에 정확하게 대응할 필요가 없다. 즉, 두 인터페이스는 서로 다른 형태일 수 있다. 일반적으로 Implementor 인터페이스는 기본적인 구현 오퍼레이션을 수행하고 Abstraction 은 보다 추상화된 서비스 관점의 인터페이스를 제공한다. 그러므로 서비스 관점의 인터페이스를 구현하기 위해서 Implementor 에 정의된 여러 개의 오퍼레이션이 필요할 수도 있는 것이다.
- ConcreteImplementor: Implementor 인터페이스를 구현하는 것으로 실제적인 구현 내용을 담고 있다. 구현 방식이 달라지면 지속적으로 만들어지는 클래스이다.

2.3. Composite

부분-전체 계층을 나타내기 위해 복합 객체를 트리 구조로 만든다. Composite 패턴은 클라이언트가 개별적 객체와 복합 객체 모두를 동일하게 다루도록 한다.

2.3.1. 구조



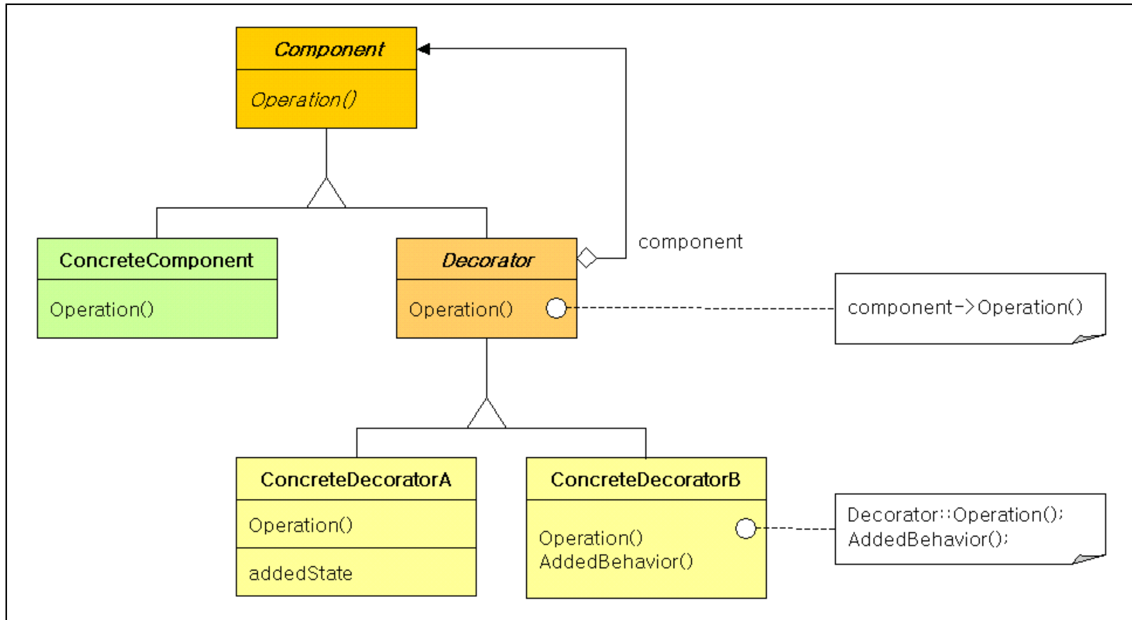
2.3.2. 참여 객체

- Component: 집합 관계에 정의될 모든 객체에 대한 인터페이스를 정의한다. 모든 클래스에 해당하는 인터페이스에 대해서는 공통의 행위를 구현한다. 전체 클래스에 속한 요소들을 관리하는데 필요한 인터페이스를 정의한다. 순환 구조에서 요소들을 포함하는 전체 클래스로의 접근에 필요한 인터페이스를 정의한다. 필요하다면 공통의 행위는 구현할 수 있다.
- Leaf: 집합 관계에서의 다른 객체를 포함할 수는 없고 포함되기만 하는 객체로 객체에 가장 기본이 되는 행위를 정의한다.
- Composite: 포함된 요소들을 갖는 복합 객체에 대한 행위를 정의한다. 자신이 합성하고 있는 요소들을 저장하고 있으면서, 각 합성 요소를 다루는데 관련된 오퍼레이션을 구현한다.
- Client: Component 인터페이스를 통해 집합 관계에 있는 객체들을 관리한다.

2.4. Decorator

객체에 동적으로 책임을 추가할 수 있게 한다. Decorator 패턴은 기능의 유연한 확장을 위해 상속 대신 사용할 수 있는 방법이다.

2.4.1. 구조



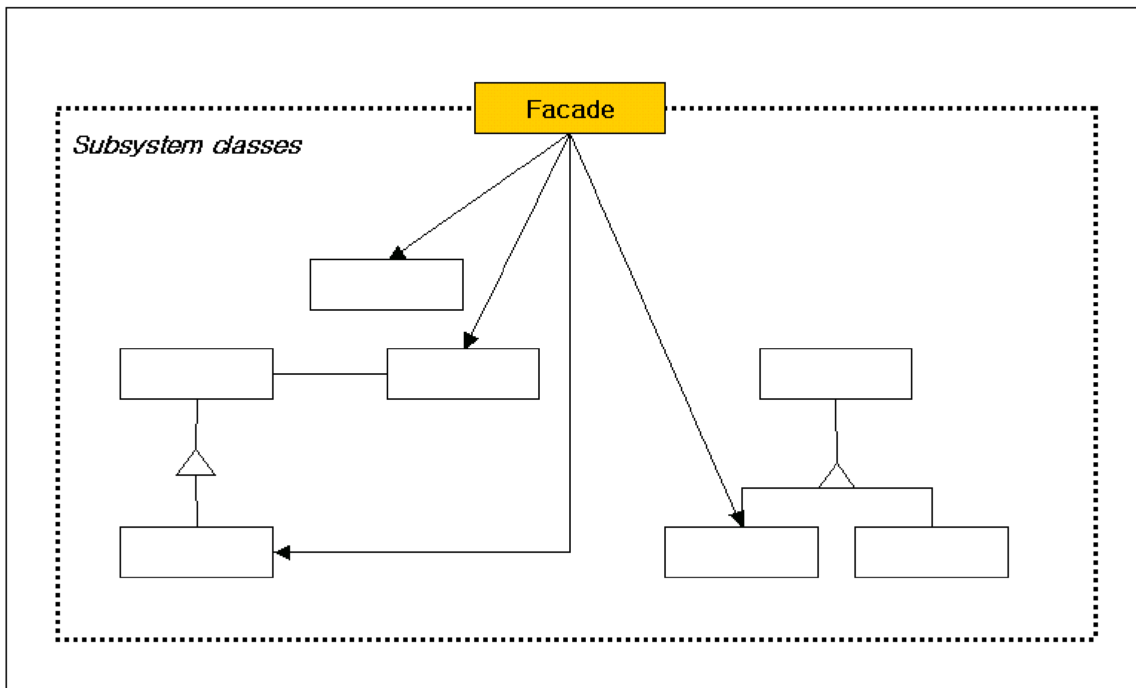
2.4.2. 참여 객체

- **Component**: 동적으로 추가할 서비스를 가질 가능성이 있는 객체들에 대한 인터페이스
- **ConcreteComponent**: 추가적인 서비스가 실제로 정의되어야 할 필요가 있는 객체
- **Decorator**: Component 객체에 대한 참조자를 관리하면서 Component 에 정의된 인터페이스를 만족하도록 인터페이스를 정의
- **ConcreteDecorator**: Component 에 새롭게 추가할 서비스를 실제로 구현하는 클래스이고 Decorator 에 정의된 기본 오퍼레이션을 만족하면서, 추가적인 행위를 addBehavior 로 또는 addedState 로 처리한다.

2.5. Façade

서브시스템에 있는 인터페이스 집합에 대해서 하나의 통합된 인터페이스를 제공한다. Façade 패턴은 서브시스템을 좀 더 사용하기 편하게 하기 위해서 높은 수준의 인터페이스를 정의한다.

2.5.1. 구조



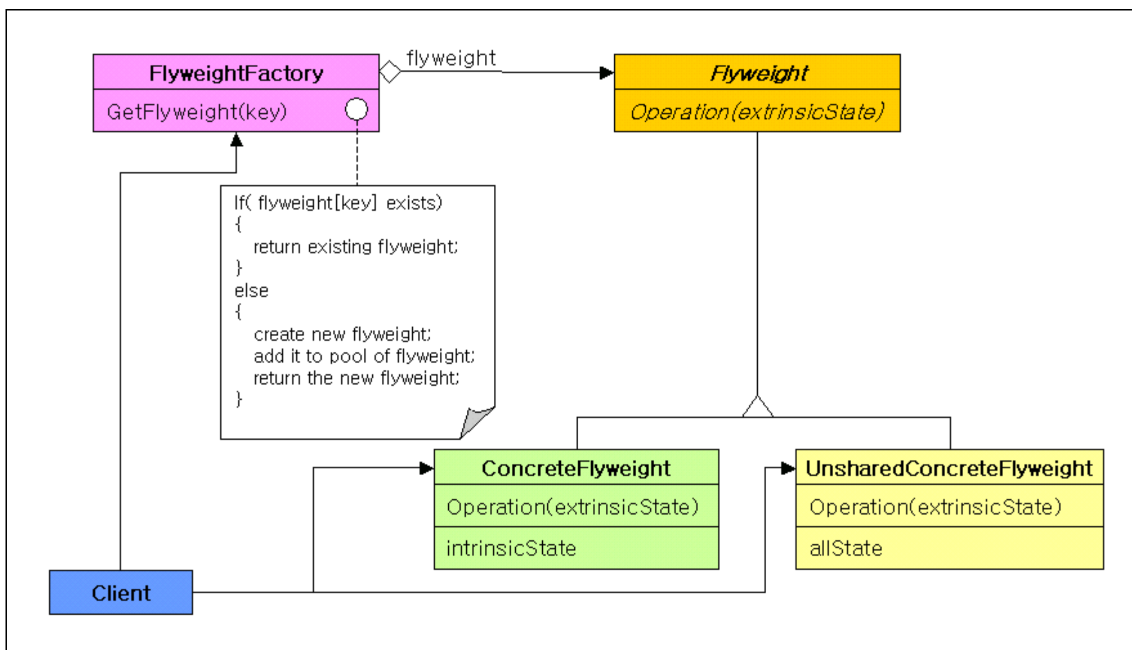
2.5.2. 참여 객체

- Facade: 단순하고 일관된 통합 인터페이스를 제공하며, 서브시스템을 구성하는 어떤 클래스가 어떤 요청을 처리해야 하는지를 알고 있으며, 클라이언트의 요청을 해당하는 서브시스템 객체에 전달한다.
- Subsystem Classes: 서브시스템의 기능성을 구현하고, Façade 객체에 의해 할당된 작업을 실제로 처리하지만 Façade 에 대한 아무런 정보를 갖고 있지 않다.

2.6. Flyweight

작은 크기의 객체들이 여러 개 있는 경우, 객체를 효과적으로 사용하는 방법으로 객체를 공유하게 한다.

2.6.1. 구조



2.6.2. 참여 객체

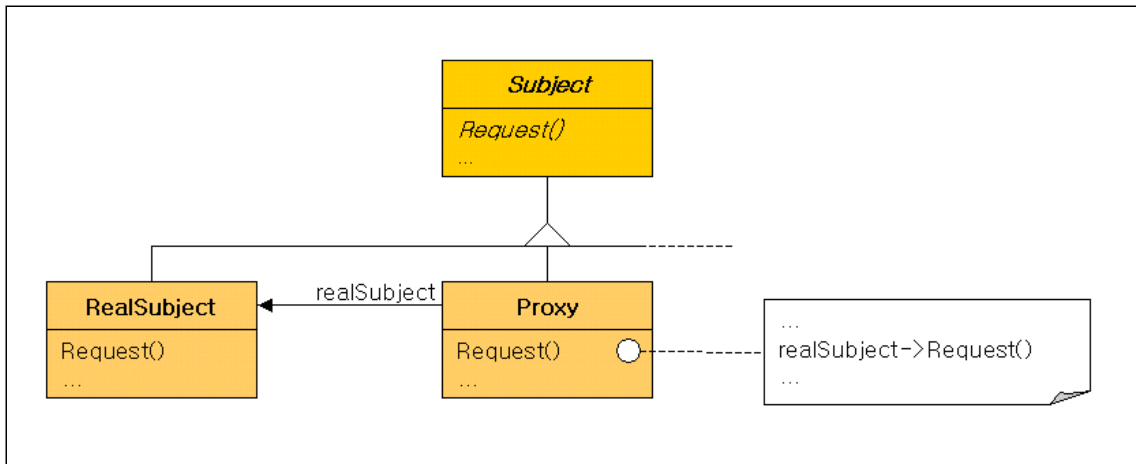
- **Flyweight**: Flyweight 가 받아들일 수 있고, 부가적 상태에서 동작해야 하는 인터페이스를 선언하고 있다.
- **ConcreteFlyweight**: Flyweight 인터페이스를 구현하고 내부적으로 갖고 있어야 하는 본질적 상태에 대한 저장소를 정의하고 있다. ConcreteFlyweight 객체는 공유할 수 있는 것이어야 한다. 그러므로 관리하는 어떤 상태라도 본질적인 것이어야 한다.
- **UnsharedConcreteFlyweight**: 모든 Flyweight 서브클래스들이 공유될 필요는 없다. Flyweight 인터페이스는 공유를 가능하게 하지만 그것을 강요해서는 안 된다. UnsharedConcreteFlyweight 객체가 ConcreteFlyweight 객체를 자신의 자식으로 갖는 것은 흔한 일이다.
- **FlyweightFactory**: Flyweight 객체를 생성하고 관리한다. Flyweight 가 적절히 공유되도록 보장해야 한다. 클라이언트가 Flyweight 를 요청하면 FlyweightFactory 객체는 이미 존재하는 인스턴스를 제공하거나, 만약 존재하지 않는다면 생성해야 한다.

- **Client:** Flyweight 에 대한 참조자를 관리하고 Flyweight 의 부가적 상태를 저장한다.

2.7. Proxy

다른 객체로의 접근을 통제하기 위해서 다른 객체의 대리자 또는 다른 객체로의 정보 보유자를 제공한다.

2.7.1. 구조



2.7.2. 참여 객체

- Proxy: 실제로 참조할 대상에 대한 참조자를 관리한다. Subject 와 동일한 인터페이스를 제공하여 실제 대상을 대체할 수 있어야 한다. 실제 대상에 대한 접근을 제어하고 실제 대상의 생성과 삭제를 책임진다.
- Subject: RealSubject 와 Proxy 에 공통적인 인터페이스를 정의하고 있어, RealSubject 가 요청되는 곳에 Proxy 를 사용할 수 있게 한다.
- RealSubject: 프록시가 대표하는 실제 객체이다.

2.7.3. 프록시 종류

- **원격지 프록시**: 서로 다른 주소 공간에 존재하는 객체에 대한 지역적 표현으로 사용된다. 요청 메시지와 Argument 를 인코딩하여 이를 다른 주소 고간에 있는 실제 대상에게 전달한다.
- **가상 프록시**: 요청이 있을 때만 필요한 객체를 생성한다. 실제 대상에 대한 추가적 정보를 보유하고 있어 실제 접근을 지연할 수 있도록 해야 한다.
- **보호용 프록시**: 원래 객체에 대한 실제 접근을 제어한다. 이는 객체별로 접근 제어 권한이 다를 때 유용하게 사용할 수 있다. 요청한 대상이 실제 요청을 할 수 있는 권한을

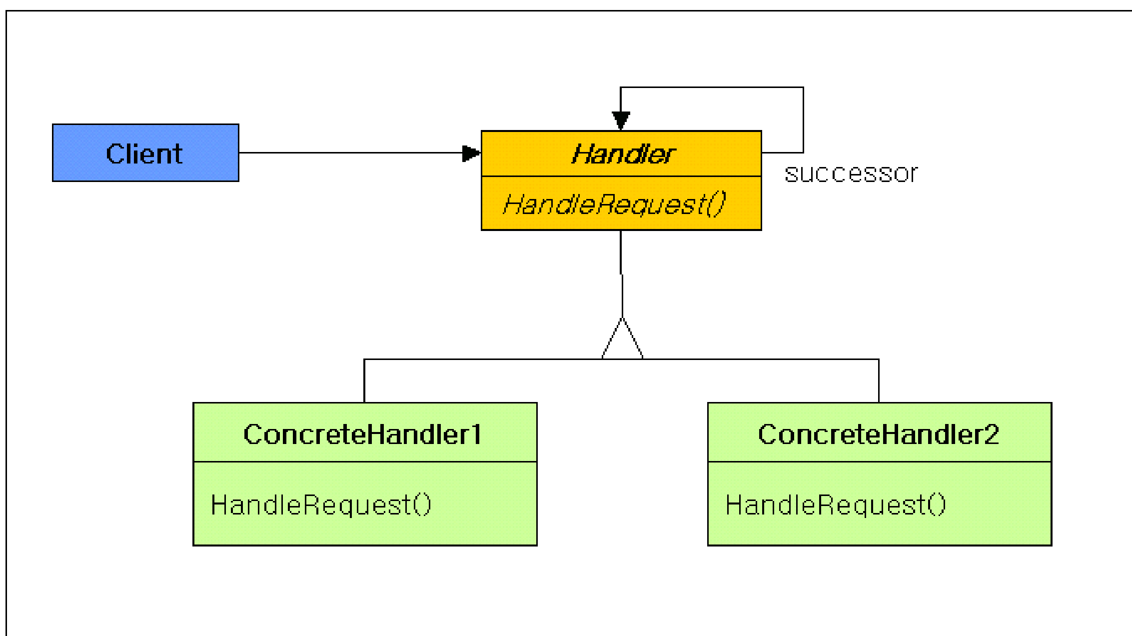
갖고 있는지를 확인한다.

3. 행위 패턴 (Behavioral Patterns)

3.1. Chain of Responsibility

요청을 처리할 수 있는 기회를 하나 이상의 객체에게 부여함으로써 요청하는 객체와 처리하는 객체 사이의 결합도를 없애려는 것이다. 요청을 해결할 객체를 만날 때까지 객체 고리를 따라서 요청을 전달한다.

3.1.1. 구조



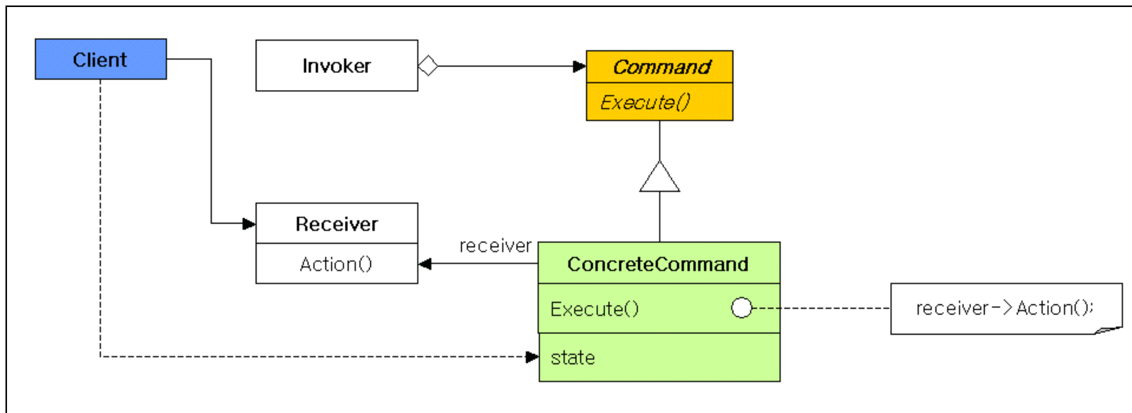
3.1.2. 참여객체

- Handler: 요청을 처리하는 인터페이스를 정의하고, 다음번 처리자와의 연결을 구현한다. 즉, 연결 고리에 연결된 다음 객체에게 다시 메시지를 보낸다.
- ConcreteHandler: 책임져야 할 행위가 있다면 스스로 요청을 처리한다. 다음번 처리자에 접근할 수 있다. 즉, 자신이 처리할 행위가 있으면 처리하고, 그렇지 않으면 다음번 처리자에게 다시 처리를 요청한다.
- Client: ConcreteHandler 객체에게 필요한 요청을 보낸다.

3.2. Command

요청을 객체로 갭슐화함으로써 서로 다른 요청으로 클라이언트를 파라미터화하고, 요청을 저장하거나 기록을 남겨서 오퍼레이션의 취소도 가능하게 한다.

3.2.1. 구조



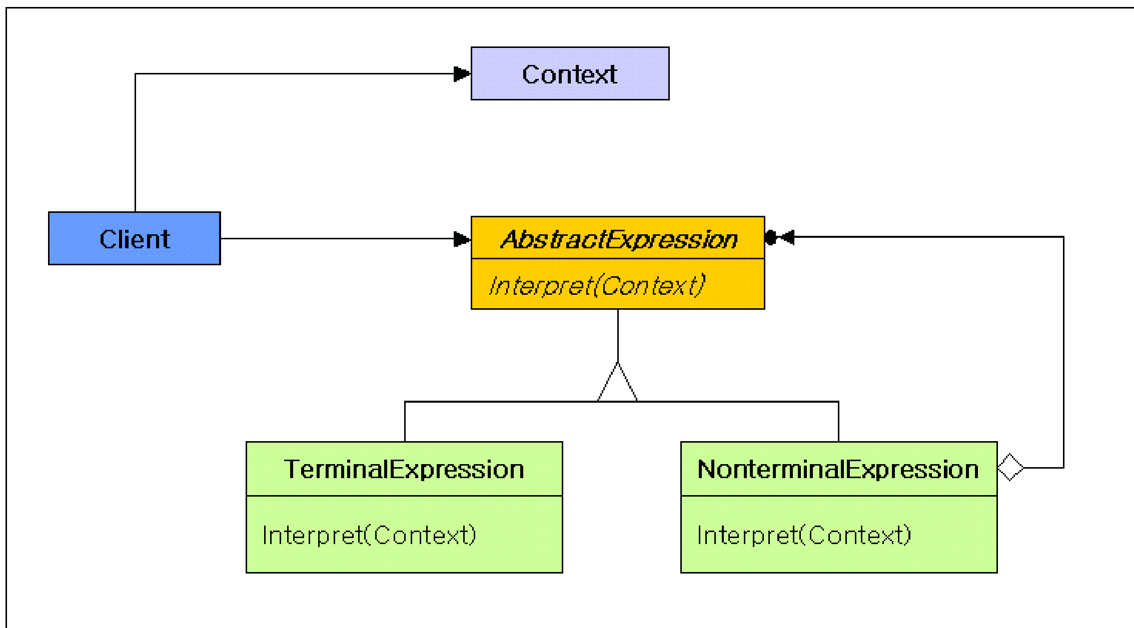
3.2.2. 참여객체

- Command: 오퍼레이션 수행에 필요한 인터페이스 선언.
- ConcreteCommand: Receiver 객체와 액션 간의 연결성을 정의한다. 또한 처리 객체에 정의된 오퍼레이션을 호출하도록 Execute 를 구현한다.
- Client: ConcreteCommand 객체를 생성하고 처리 객체로 정의한다.
- Invoker: 명령어에게 처리를 수행할 것을 요청한다.
- Receiver: 요청에 관련된 오퍼레이션 수행방법을 알고 있다.

3.3. Interpreter

언어에 따라서 문법에 대한 표현을 정의한다. 또 언어의 문장을 해석하기 위해 정의한 표현에 기반하여 분석기를 정의한다.

3.3.1. 구조



3.3.2. 참여객체

- AbstractExpression: 추상 구문 트리에 속한 모든 노드에 해당하는 클래스들이 공통으로 가져야 할 Interpret() 오퍼레이션을 추상 오퍼레이션으로 정의한다.
- TerminalExpression: 문법에 정의한 터미널 기호와 관련된 해석 방법을 구현한다. 문장을 구성하는 모든 터미널 기호에 대해서 해당 클래스를 만들어야 한다.
- NonterminalExpression: 문법의 오른편에 나타나는 모든 기호에 대해서 클래스를 정의해야 한다. 문법에

  R :: = R1 R2 ... Rn

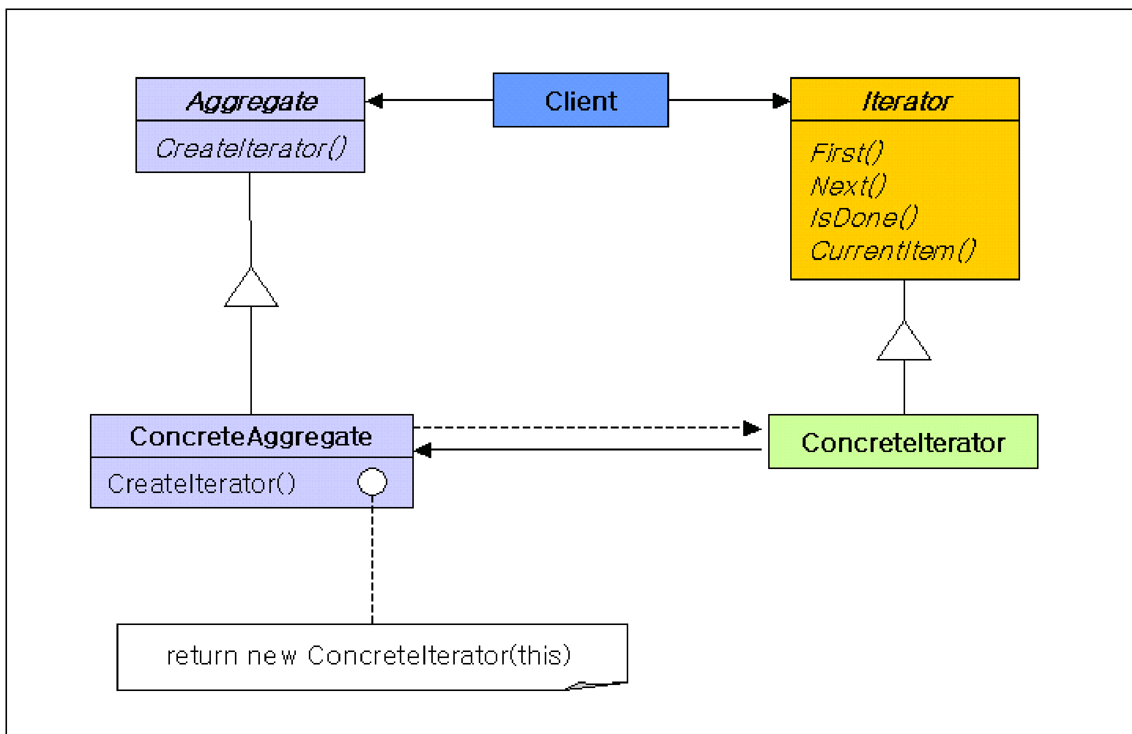
  을 정의하고 있다면 R 에 대해서 NonterminalExpression 에 해당하는 클래스를 정의해야 한다. 또한 규칙의 오른쪽에 나타난 R1 에서 Rn 에 이르기까지의 모든 기호에 대응하는 인스턴스 변수들을 정의해야 한다. 또한 터미널 기호가 아닌 모든 기호들에 대해서 Interpret() 오퍼레이션을 구현해야 한다. 이 Interpret() 오퍼레이션은 R1 에서 Rn 에 이르기까지의 각 인스턴스 변수를 재귀적으로 호출하는 것이 일반적이다.

- **Context:** 번역기에 대한 포괄적인 정보를 포함한다.
- **Client:** 언어로 정의한 특정 문장을 나타내는 추상 구문 트리이다. 이 추상 구문 트리는 NonterminalExpression 과 TerminalExpression 클래스의 인스턴스로 구성된다. 이 인스턴스의 Interpret() 오퍼레이션을 호출한다.

3.4. Iterator

내부 표현 방법을 노출하지 않고 복합 객체의 원소를 순차적으로 접근할 수 있는 방법을 제공한다.

3.4.1. 구조



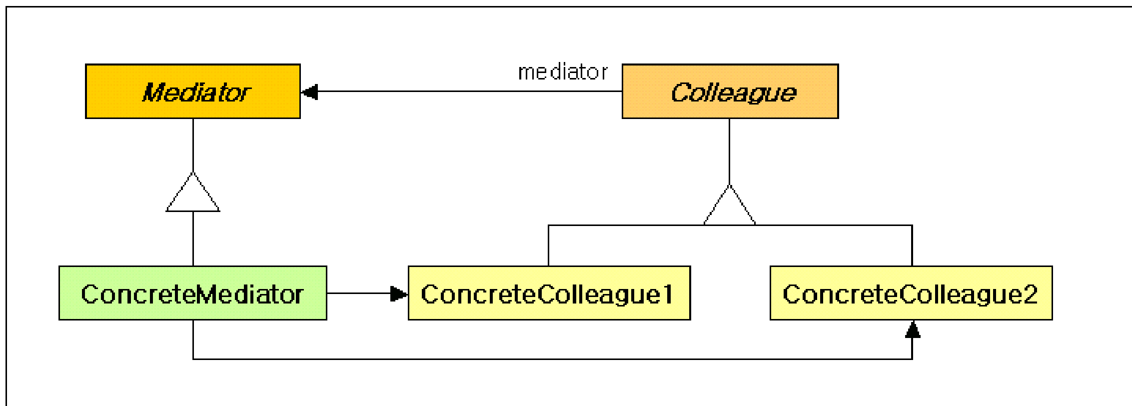
3.4.2. 참여객체

- Iterator: 요소를 접근하고 순회하는데 필요한 인터페이스 제공.
- ConcreteIterator: Iterator 에 정의된 인터페이스를 구현하는 클래스로서 순회 과정 중에 집합 객체 내의 현재 위치를 기억한다.
- Aggregate: Iterator 객체를 생성하는 인터페이스를 정의
- ConcreteAggregate: 해당하는 ConcreteIterator 의 인스턴스를 반환하도록 Iterator 생성 인터페이스를 구현한다.

3.5. Mediator

객체들 간의 상호작용을 객체로 캡슐화한다. Mediator 패턴은 객체들 간의 참조 관계를 객체에서 분리함으로써 상호작용만을 독립적으로 다양하게 확대할 수 있다.

3.5.1. 구조



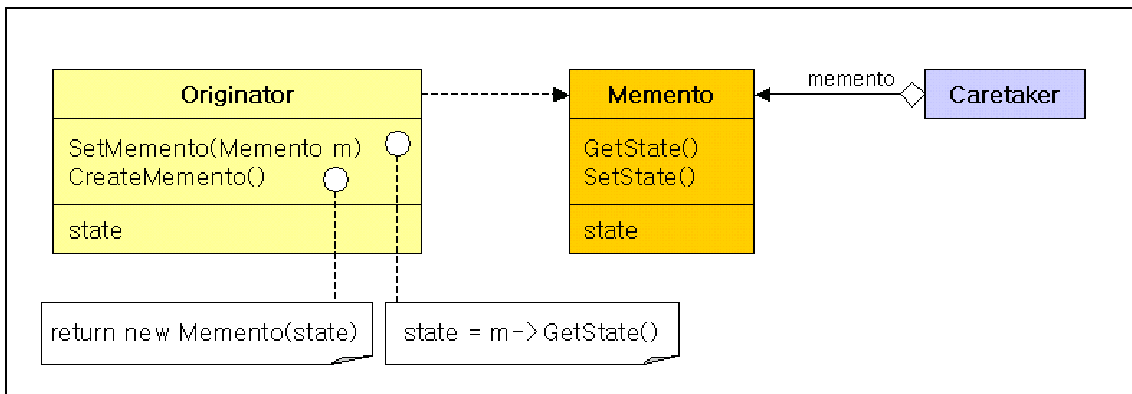
3.5.2. 참여객체

- **Mediator**: Colleague 객체와 교류하는 데 필요한 인터페이스를 정의한다.
- **ConcreteMediator**: Colleague 객체와 조화를 이룸으로써 이루어지는 협력 행위를 구현하고 자신의 colleague 가 무엇인지를 알고 이를 관리한다.
- **Colleague**: Mediator 객체가 누구인지를 안다. 다른 객체와 연결성을 필요하면 Mediator 를 통해 이루어지도록 한다.

3.6. Memento

캡슐화를 위배하지 않고 객체 내부 상태를 객체화하여, 나중에 객체가 이 상태로 복구 가능하게 한다.

3.6.1. 구조



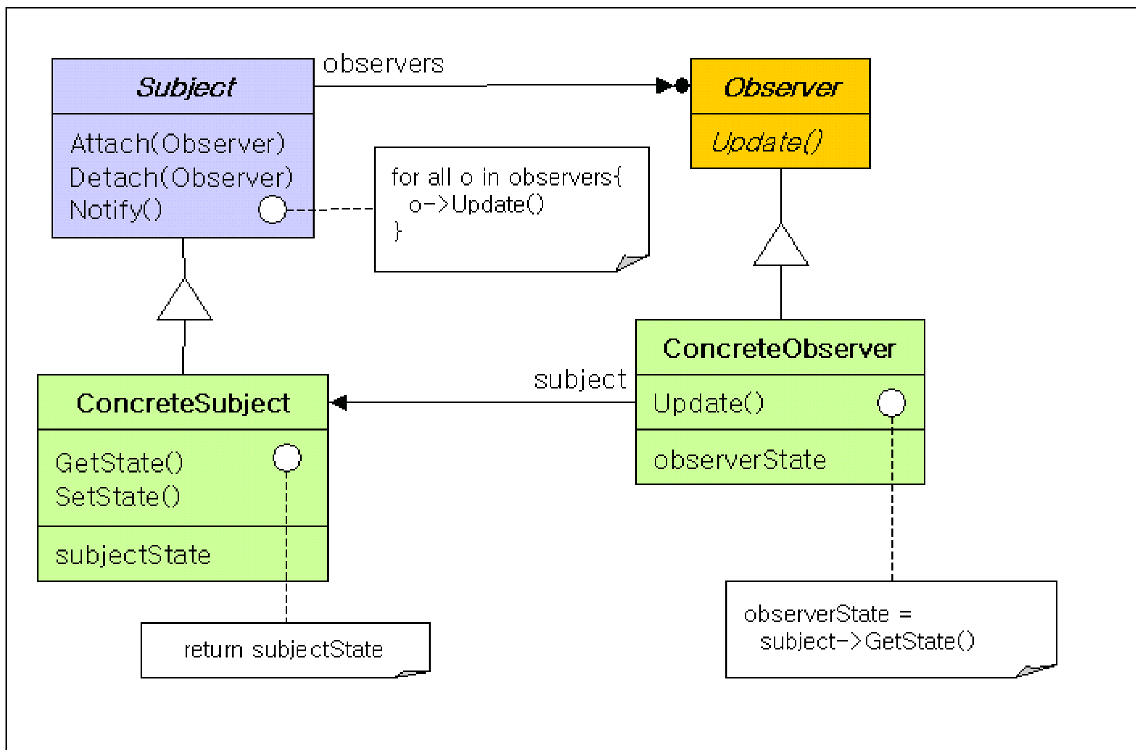
3.6.2. 참여객체

- **Memento**: Originator 객체의 내부 상태를 저장한다. 메멘토는 originator 객체의 내부 상태를 필요한 만큼 저장해 둔다. originator 객체를 제외한 다른 객체는 Memento 클래스에 접근할 수 없다. 그래서 Memento 클래스는 두 개의 인터페이스를 갖는다. 관리 책임을 갖는 객체인 CareTaker 클래스는 Memento 에 정의된 모든 인터페이스를 다 보지 못하고 단지 Memnto 를 다른 객체에 전달한다. 이에 비해 Originator 클래스는 자신의 상태를 이전 상태로 복구하기 위해 필요한 모든 자료에 접근하는데 필요한 Memento 의 다양한 인터페이스를 사용할 수 있다. 즉, 메멘도를 생성하는 Originator 클래스만이 메멘토의 내부상태에 접근할 수 있는 권한을 갖는다.
- **Originator**: 메멘토를 생성하여 현재 객체의 상태를 저장하고 내부 상태를 복구한다.
- **Caretaker**: 메멘토의 보관을 책임지기는 하지만, 메멘토의 내용을 확인하거나 처리하지는 않는다.

3.7. Observer

객체 사이에 일 대 다의 종속성을 정의하고 한 객체의 상태가 변하면 종속된 다른 객체에 통보가 가고 자동으로 수정이 일어나게 한다.

3.7.1. 구조



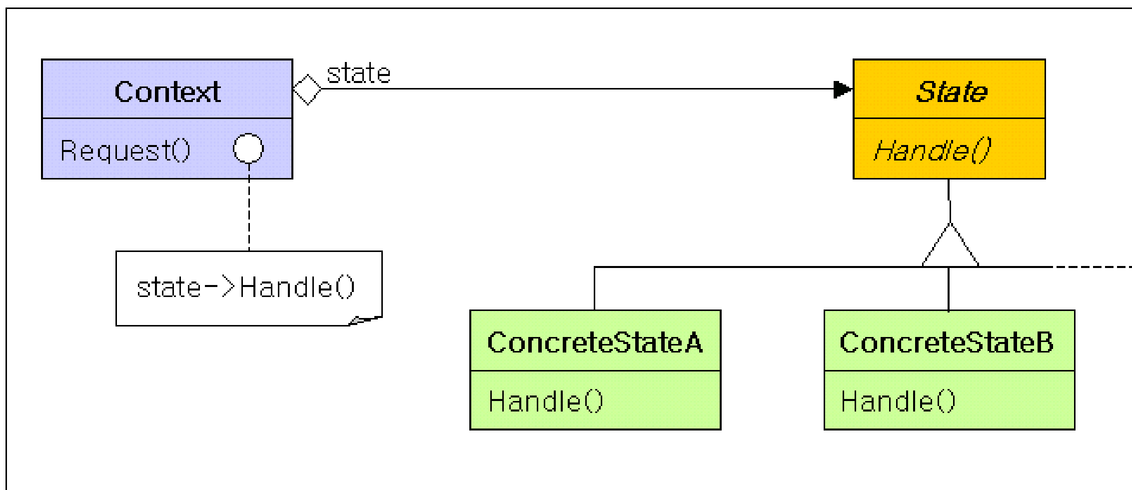
3.7.2. 참여객체

- **Subject**: 관찰자들을 알고 있다. 다수의 Observer 객체는 대상을 관찰한다. Observer 객체를 대상과 연결하거나 무관한 것으로 만드는 데 필요한 인터페이스를 갖는다.
- **Observer**: 대상에 생긴 변화에 관심 있는 객체를 변경하는데 필요한 인터페이스를 갖고 있다. 이로써 Subject 의 변경에 따라 변화되어야 하는 객체들의 일관성을 유지한다.
- **ConcreteObserver**: ConcreteSubject 객체에 대한 참조자를 관리한다. 대상과 일관성을 유질해야 하는 상태를 저장하고 있다. 대상과 일관성을 유지하기 위해 관찰자를 수정해야 하므로 이에 필요한 인터페이스를 구현한다.

3.8. State

객체의 내부 상태에 따라 행위를 변경할 수 있게 한다. 이렇게 하면 객체는 마치 클래스를 바꾸는 것처럼 보인다.

3.8.1. 구조



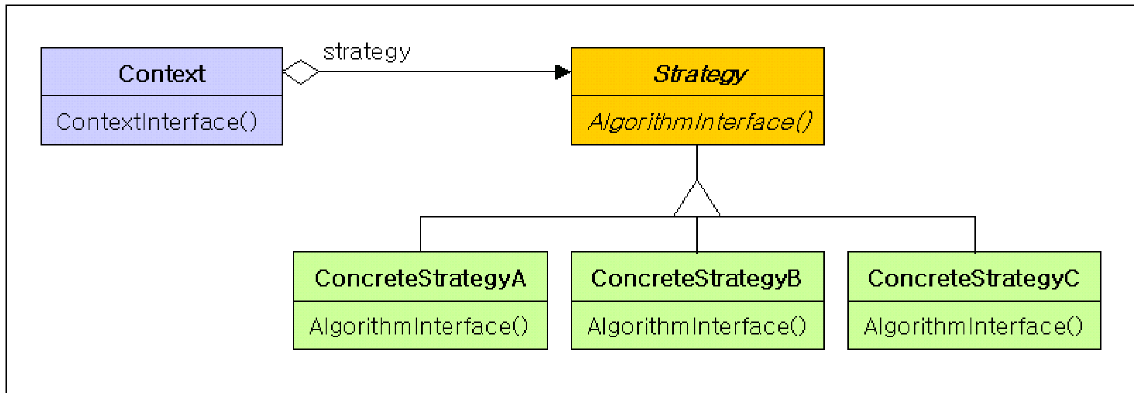
3.8.2. 참여객체

- Context: 클라이언트가 관심을 갖고 있는 인터페이스를 정의한다. ConcreteState 서브클래스의 인스턴스를 관리하고 있는데, ConcreteState 의 서브클래스들이 객체의 현재 상태를 정의하고 있다.
- State: Context 가 갖는 각 상태별로 필요한 행위를 캡슐화하여 인터페이스로 정의한다.
- ConcreteState Subclass: 각 서브클래스들은 Context 의 상태에 따라 처리되어야 할 실제 행위를 구현하고 있다.

3.9. Strategy

알고리즘군이 존재할 경우 각각의 알고리즘을 별도의 클래스로 캡슐화하고 이들을 상호 교환 가능한 것으로 정의한다. Strategy 패턴은 클라이언트에 영향을 주지 않고 독립적으로 알고리즘을 다양하게 변경할 수 있게 한다.

3.9.1. 구조



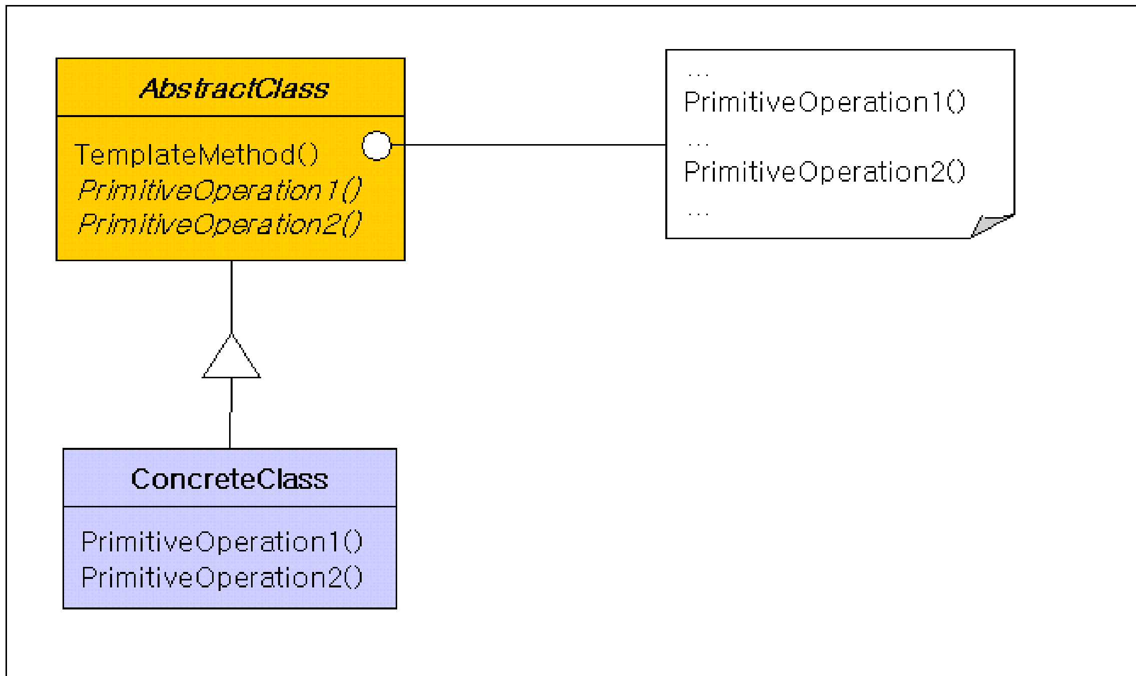
3.9.2. 참여객체

- **Strategy**: 제공하는 모든 알고리즘에 대한 공통의 오퍼레이션들을 인터페이스로 정의한다. Context 클래스는 ConcreteStrategy 클래스에 정의한 인터페이스를 통해서 실제 알고리즘을 사용한다.
- **ConcreteStrategy**: Strategy 인터페이스를 실제 알고리즘으로 구현한다.
- **Context**: ConcreteStrategy 객체가 무엇인지 구체화한다. 즉, Strategy 객체에 대한 참조자를 관리하고, 실제로는 Strategy 서브클래스의 인스턴스를 갖고 있음으로써 구체화한다. 또한 Strategy 가 자료에 접근해가는데 필요한 인터페이스를 정의한다.

3.10. Template Method

오퍼레이션에는 알고리즘의 처리 과정만을 정의하고 각 단계에서 수행할 구체적 처리는 서브클래스에 정의한다. Template Method 패턴은 알고리즘의 처리 과정은 변경하지 않고 알고리즘 각 단계의 처리를 서브클래스에서 재정의할 수 있게 한다.

3.10.1. 구조



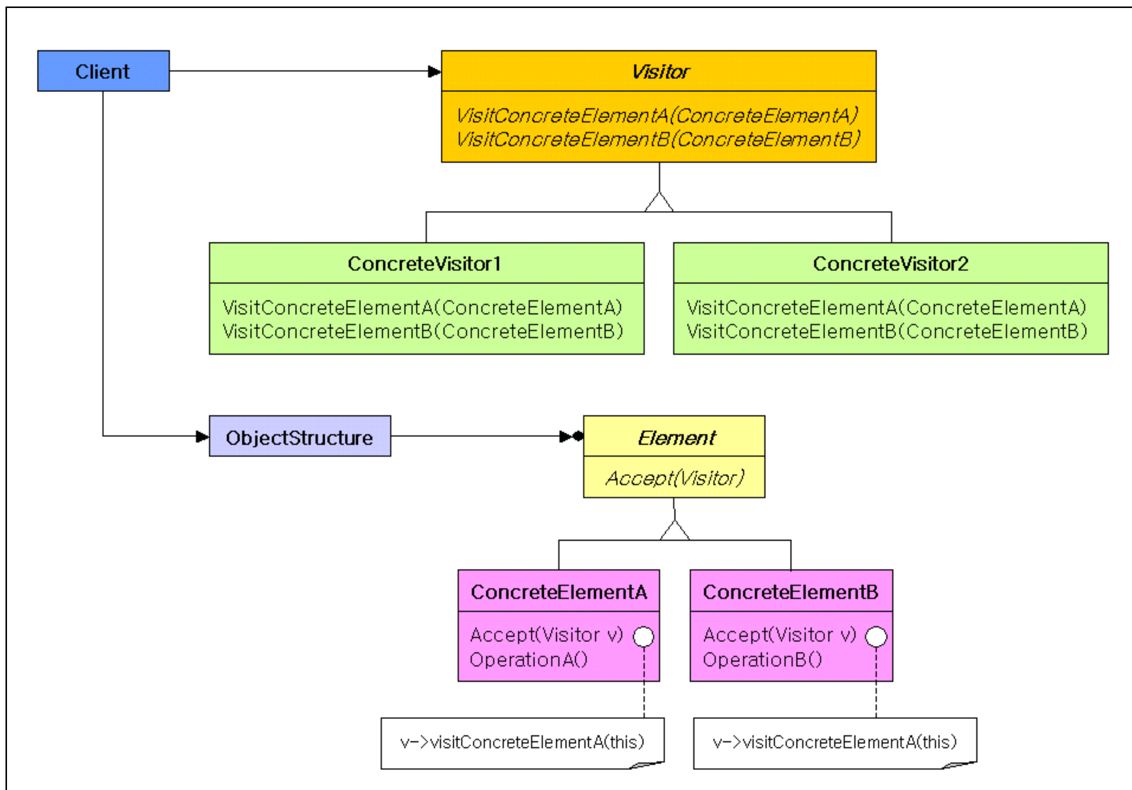
3.10.2. 참여객체

- AbstractClass: 서브클래스들이 반드시 구현해야 하는 알고리즘 처리 단계 내의 기본 오퍼레이션이 무엇인지를 정의한다. 서브클래스에서 이들 오퍼레이션들을 구현한다. 이 클래스에서 템플릿 메소드를 구현하는데, 그 방법은 알고리즘의 기본 골격 구조를 정의하여 템플릿 메소드가 다른 객체에 의해 정의된 오퍼레이션뿐만 아니라 인터페이스로 정의된 기본 오퍼레이션을 재정의한 서브클래스에게 메시지가 전달되어 기본 골격 구조를 준수하되 다른 결과가 나타나게 된다.
- ConcreteClass: 서브클래스마다 기본 오퍼레이션을 다르게 구현한다.

3.11. Visitor

객체 구조의 요소들에 수행할 오퍼레이션을 표현한 패턴이다. Visitor 패턴은 오퍼레이션이 처리할 요소의 클래스를 변경하지 않고도 새로운 오퍼레이션을 정의할 수 있게 한다.

3.11.1. 구조



3.11.2. 참여객체

- **Visitor**: 각 ConcreteElement 클래스에 대한 Visit 오퍼레이션을 선언한다. 오퍼레이션의 이름과 인터페이스 형태는 Visit() 요청을 보내는 방문자에게 보내는 클래스를 명시한다. 이로써 방문자는 방문할 요소의 실제 서브클래스를 결정한다. 그리고 나서 방문자는 Element 가 제공하는 인터페이스를 통해 직접 Element 객체에 접근할 수 있다.
- **ConcreteVisitor**: Visitor 클래스에 정의된 오퍼레이션을 구현한다. 각 오퍼레이션은 객체에 해당하는 클래스에 정의된 알고리즘을 구현한다. ConcreteVisitor 클래스는 내부 상태를 저장하고 있으며, 알고리즘이 운영될 수 있는 상황 정보를 제공한다. ConcreteVisitor 클래스가 저장하고 있는 내부 상태는 구조의 순회 과정에서 도출되기도 한다.

- **Element**: 아규먼트로 Visitor 클래스를 받아들이는 Accept() 오퍼레이션을 정의한다.
- **ConcreteElement**: 아규먼트로 Visitor 객체를 받아들이는 Accept() 오퍼레이션을 구현한다.
- **ObjectStructure**: 요소들을 나열하고 방문자로 하여금 이들 요소에 접근하게 하는 인터페이스를 제공한다. ObjectStructure 는 Composite 패턴의 복합 객체일 수도 있고, 리스트나 집합과 같은 컬렉션일 수도 있다.