The Pursuit of Professional Society

INFOREVER

2016년 17회 정보시스템감리사 대비

정보시스템감리사 특강
(디자인 패턴)

2016. 03. 19

김 성 근
정보시스템감리사
heryoun12@nate.com

Table of Contents

가. 객체지향 설계 5대 원리

구분	개념
인터페이스 분리 (ISP : Interface Segregation)	- 객체 기능, I/F를 구체적으로 분리(입/출력 분할 사례 등) - 지나친 일반화를 경계해야 하며, ISP 어기면 비대한 클래스 발생
개방 폐쇄 (OCP : Open and Close)	- 변화 부분은 확장 가능하되, 변경은 어려운 구조 (불변하는 부분만 클라이언트에 제공), Open (클래스 수직 관계), Close (클래스 수평 관계) - 인터페이스 변경은 어려우나 구현은 열려 있음 - 예: 상속과 어덥터 클래스를 통한 클라이언트 클래스 접근
단일 책임 (SRP : Single Response Principal)	- 객체는 둘 이상의 책임을 갖지 않는 형태를 가져야 함 - 두 개 이상의 책임을 가지면, 온전한 책임을 다할 수 있도록 분리 - 예) 쇼핑 (쇼핑,고객등록) → 쇼핑, 고객으로 분리
리스코프 치환 (LSP : Liskov Substitution)	- 자식타입들은 부모 타입들이 사용되는 곳에 대체될 수 있어야 함 - 자식클래스는 부모의 책임을 넘지 말고, 자식 기능의 제약 필요
의존관계 역전 (DIP : Dependency Inversion)	- 참조의 대상은 파생클래스가 아닌 추상클래스여야 함 - 객체의 참조는 부모 클래스의 인터페이스에 의존

1. 디자인 패턴 요약

나. 디자인패턴의 유형

구분		Creational Pattern (생성패턴)	Structural Pattern (구조패턴)	Behavioral Pattern (행위패턴)
의미		객체의 생성방식을 결정하는 패턴	객체를 조직화하는데 유용한 패턴	객체의 행위를 Organize, Manage, Combine하는 데 사용되는 패턴
범위	클래스	Factory Method	Adapter(Class)	Interpreter, Template Method
	객체	Abstract Factory, Builder, Prototype, Singleton	Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy	Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor Chain of Responsibility

다. 디자인패턴 Keyword 요약

구분	패턴	Keyword
생성 패턴	Singleton	- 객체 생성을 n개 이하로 제한, Static 변수
	Builder	- 부분 부분 생성 후 조합하여 전체 생성, BuildPart() 함수
	Factory Method	- 상속 구조, 객체 생성을 자식 클래스에 위임
	Abstract Factory	- 제품군(Product Family) 생성
	Prototype	- 복제, clone() 함수
구조 패턴	Adaptor	- 돼지코 패턴, 재활용하고자 하는 클래스가 인터페이스가 다른 경우 사용
	Bridge	- 구현과 추상을 독립적으로 확장
	Composite	- 부분 전체 Tree 구조, 폴더 구조, 메뉴 구조
	Decorator	- 동적으로 객체에 기능을 추가
	Facade	- Complex System에 대한 인터페이스 제공
	Flyweight	- 경량급 객체, 공유(Share)
	Proxy	- 대리인, 중계자, 객체에 직접적인 접근 제한

다. 디자인패턴 Keyword 요약

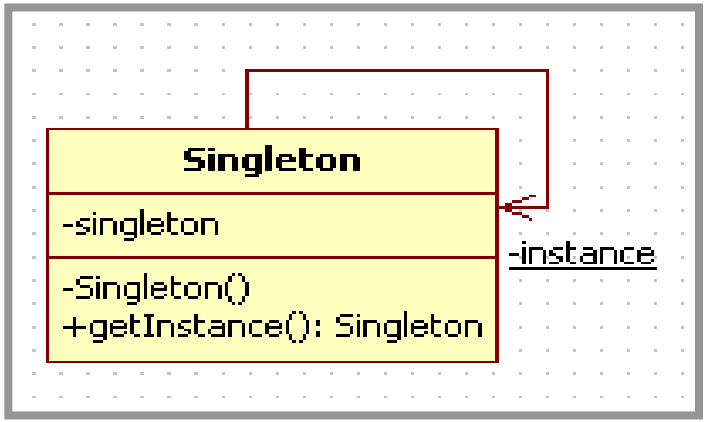
구분	패턴	Keyword
행위 패턴	Iterator	- 순회 접근, next() 함수
	Interpreter	- 문법 정의, Composite Pattern과 Class Diagram 동일
	Template Method	- 상속 구조, 알고리즘 재정의(Overriding)
	Strategy	- 알고리즘, 상황에 따라 문제해결 방법을 동적으로 적용
	Visitor	- 데이터 구조와 처리 분리, visit() 함수
	Chain of Responsibility	- 객체 연결(Chain), 객체 자신의 책임에 맞는 문제만 해결
	Mediator	- 인터페이스 단순화, M:N 인터페이스를 1:M, 1:N으로 해결
	Observer	- Publish & Subscribe 패턴, 상태 변화 시 알림, notify() 함수
	Memento	- 이전 상태로 복원
	State	- 상태에 따라 객체의 행위를 동적으로 변경
	Command	- 명령, Undo & Redo

1. Singleton : 클래스의 인스턴스를 오직 하나만 만들어 사용하는 패턴

의도 (Intent)

Ensure a class only has one instance, and provide a global point of access to it.
클래스는 오직 하나의 인스턴스를 가짐을 보장하고, 해당 인스턴스에 접근할 수 있는 전역 접점을 제공함

구조와 참여객체



▪**Singleton :**
유일한 인스턴스를 얻기 위한 static 메소드를 가지고 있으며, 이 메소드는 언제나 동일한 인스턴스를 반환함.

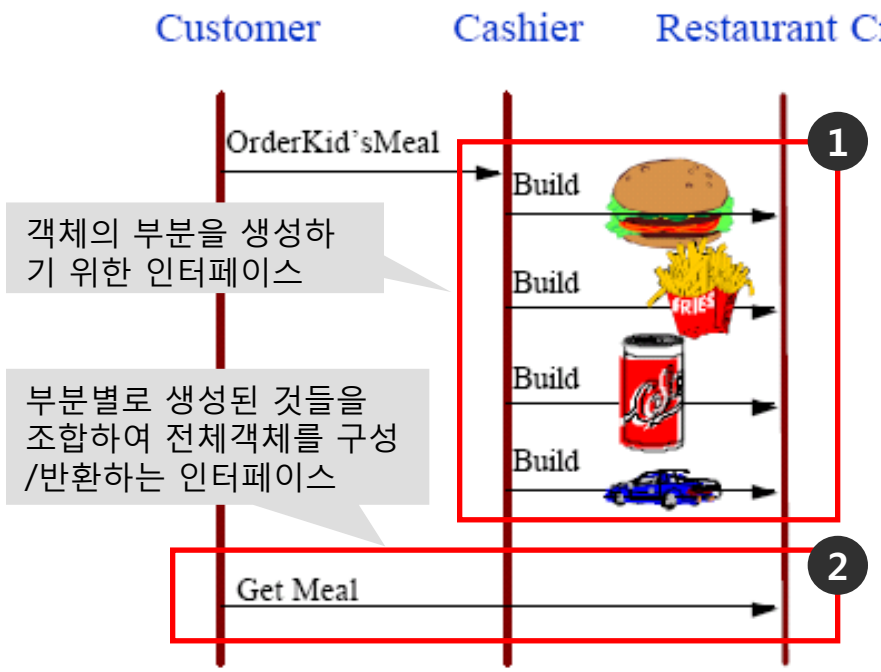
✓생성자 : private - 클래스 외부에서 생성자 호출을 금지하기 위해서

적용과 활용

복수 개의 인스턴스가 존재하면 인스턴스들이 서로 영향을 미치고, 뜻하지 않은 버그가 발생할 가능성이 있는 경우 활용

2. Builder : 객체를 구성하는 부분 부분을 먼저 생성하고, 이를 조합함으로써 전체 객체를 생성하는 패턴

패스트푸드 점 예시



1. Customer는 Cashier에게 Kid's Meal을 주문함
2. Cashier는 주방직원에게 Kid's Meal의 구성품인
햄버거
포테이토
콜라
장난감을 만들도록 함.
3. 각각 만들어진 햄버거, 포테이토, 콜라, 장난감을 쟁반에 담으면 Kid'sMeal 완성

Q : breakfast Meal 주문한다면??

Cashier	construct()	**Director , Façade 역할**
Restaurant Crew	buildBread, buildSidedish, buildBeverage, buildGift	**Builder**
KidsMeal	햄버거, 포테이토, 콜라, 장난감, getMeal	**ConcreteBuilder**
BreakfastBuilder	크로와상, 커피, 30%쿠폰	

2. Builder (계속)

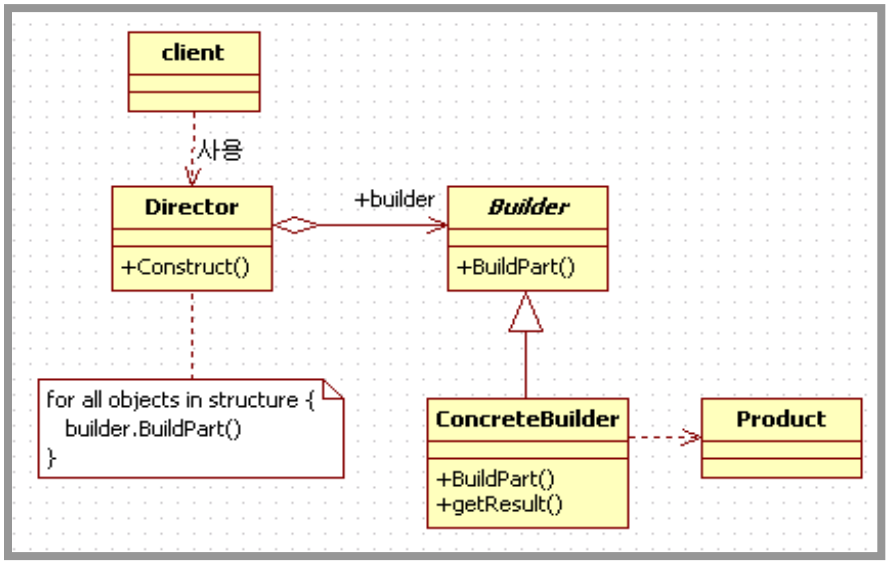
의도 (Intent)

Separate the construction of complex object from its representation so that the same construction process can create different representations.

복잡한 객체를 생성할 때 표현과 분리시켜서, 동일한 생성과정을 거치지만 서로 다른 표현을 생성해낼 수 있음.

✓**Builder 라는 인터페이스를 통해서 구축 프로세스 재활용**

구조와 참여객체



▪Builder :
추상인터페이스를 정의하여 Product 객체의 부분들을 생성하도록 해줌.

▪ConcreteBuilder : Builder 인터페이스를 구현함으로써 프로덕트의 부분을 조합하여 구축함.

▪Product :
구축과정을 거쳐서 복잡하게 표현된 객체, ConcretBuilder는 Product의 내부 표현을 만들고, 이것들을 조립함으로써 정의함

▪Director :
- Builder 인터페이스를 사용하여 임의의 객체 생성

Material from: - "Design Pattern",

2. Builder (계속)

적용과 활용

✓복잡한 객체를 생성하는데 있어 그 객체를 구성하는 부분 부분들을 생성한 후 그것을 조합해도 객체의 생성이 가능할 때, 즉 객체의 부분 부분을 생성하는 것과 그들을 조합해서 전체 객체를 생성하는 것이 서로 독립적으로 이루어질 수 있는 경우
✓서로 다른 표현 방식을 가지는 객체를 동일한 방식으로 생성하고 싶을 때.
(번역소프트웨어 에서 번역 대상언어가 달라도 객체를 동일한 방식으로 생성하고 싶을 때)

장/단점

✓Builder 클래스는 Director 클래스에게 객체를 생성할 수 있는 인터페이스를 제공한다. 대신 생성되는 객체의 내부 구조나 표현 방식. 조합 방법 등은 Builder 클래스 내부로 숨긴다. 이런 형태이기 때문에 Builder 패턴은 생성되는 객체의 내부구조를 변경시킬 필요가 있을 경우 기존 소스 코드에는 영향을 주지 않고 새로운 하위 클래스를 추가 정의하면 원하는 작업을 수행할 수 있는 장점이 있다.
✓객체를 생성하는 부분과 그것을 실제 표현하고 구성하는 부분을 분리시켜 줌으로서 서로간의 독립성을 기할 수 있게 해준다.
✓객체 생성을 부분 부분 을 생성한 후 최종 결과를 얻어가는 방식이므로 객체 생성 과정을 좀더 세밀하게 제어 할 수 있다.
✓새로운 종류의 객체 생성을 추가하기는 쉬우나 객체를 구성하는 각 부분을 새롭게 추가하기는 어렵다.

Material from: - "Design Pattern",

2. Builder (계속)

관련패턴

Abstract Factory 패턴 vs. Builder 패턴

builder는 복잡한 객체의 단계별 생성에 중점을 두고 있는 패턴,
abstract factory는 제품의 유사군들이 존재하는 경우 유연한 설계에 중점을 두는 패턴.

빌더패턴은 그룹으로 묶이는 객체들이 정해지지 않은 상태에서 객체들의 생성과 반환작업을 처리한다.
다시 말해, 추상 팩토리 패턴은 하나의 팩토리 객체를 선택하는 순간, 생성되고 반환되는 객체들이 이미 정해져 있지만,
빌더 패턴의 경우에는 팩토리 객체를 사용하는 순간에도 객체들의 그룹이 정해져 있지 않다

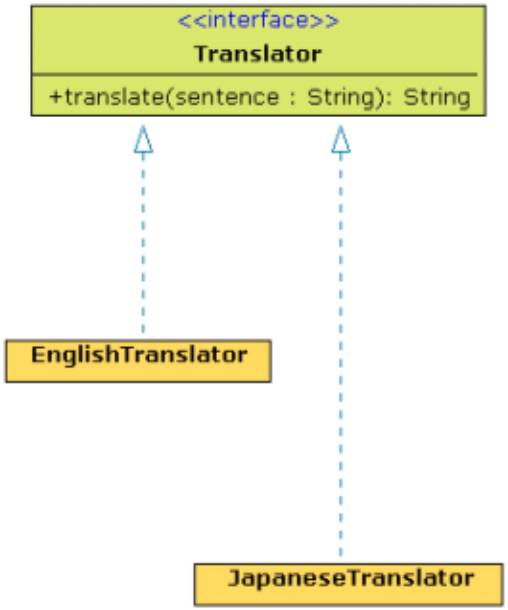
Template Method 패턴 vs. Builder 패턴

Template Method 패턴 : 상위클래스가 하위 클래스를 제어
Builder 패턴 : Director라는 역할자가 Builder의 역할을 제어

Material from: - "Design Pattern",

3. Factory Method :
객체 생성에 대한 인터페이스를 정의한 후 실제 객체는 인터페이스를 구현한 서브클래스에서 정의하는 패턴

번역기 시스템 예시




```
public class TranslatorClient {
    …
    private Translator tr_;
    if(language == ENGLISH) {
        tr_ = new EnglishTranslator();
    } else if (language == JAPANESE) {
        tr_ = new JapaneseTranslator();
    }
    …
    translatedSentence = tr_.translate(inputSentence);
    …
}
```

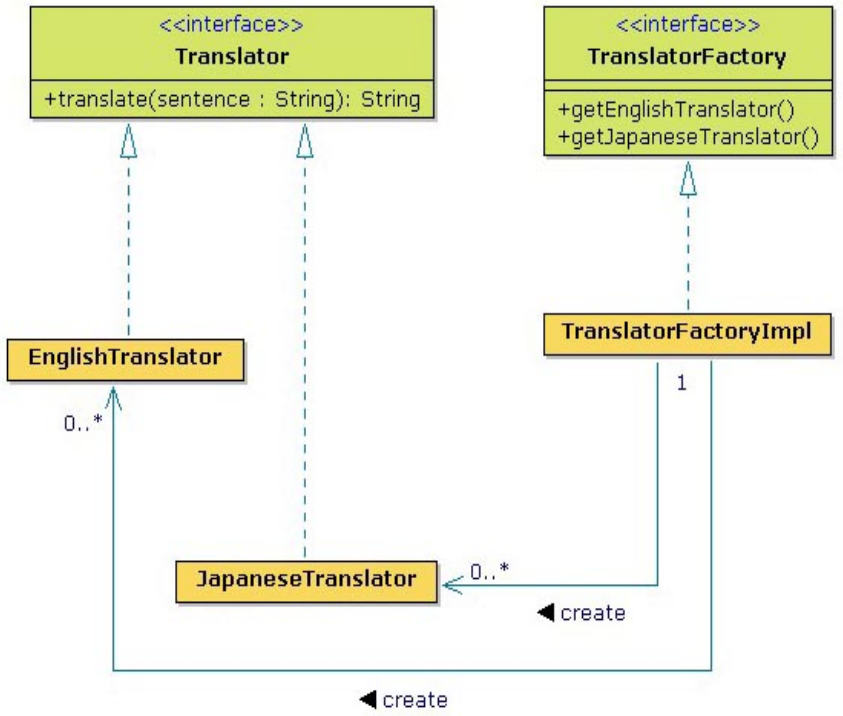
▪사용되는 번역기에 상관없이 같은 메소드를 호출한다. 따라서, 이 경우 해당 번역기의 소스가 수정되더라도 클라이언트의 메소드 호출부분에 대한 소스는 고칠 곳이 없다.

▪**필요한 번역기의 객체를 생성하기 위해서 그 해당 번역기의 타입을 알아야 하는 문제점이 남아 있다**. 즉, 다른 타입의 번역기를 사용하게 되면 이 부분을 바꾸어 주어야 하는 것이다.

Material from: - "Design Pattern",

3. Factory Method (계속)

직접 해당 객체를 생성할 때 발생하는 이러한 의존성 문제를 해결하기 위해서 **객체 생성을 다른 객체에 의뢰**하는 디자인을 생각하게 됨




```
private Translator tr_;

if(language == ENGLISH) {
    tr_ = new EnglishTranslator();
} else if (language == JAPANESE) {
    tr_ = new JapaneseTranslator();
}
```

```
private TranslatorFactory tf_;
private Translator tr_;

if(language == ENGLISH) {
    tr_ = tf_.getEnglishTranslator();
} else if (language == JAPANESE) {
    tr_ = tf_.getJapaneseTranslator();
}
```

- *객체 생성을 전담하는 TranslatorFactoryImpl 객체를 하나 만들어 클라이언트가 Translator를 통해서 하던 번역기 객체 생성 일을 맡김.*
- *객체 생성 부분에 남아 있던 의존성 부분까지도 제거 했음*

*즉, 클라이언트는 **생산될 객체의 정확한 명세 없이도** 다룰 수 있게 해줌*

- *번역기 객체 타입 -> 생성책임을 맡고 있는 객체의 인터페이스로 변경*

Material from: - "Design Pattern",

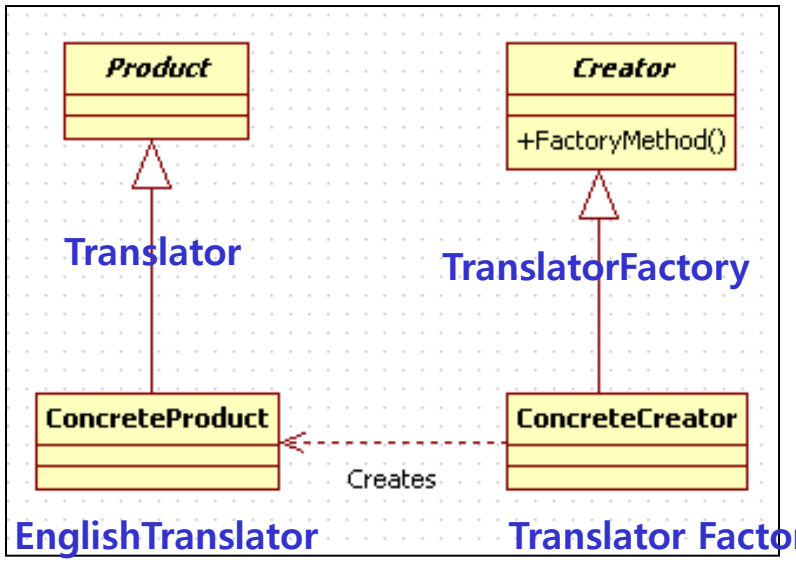
3. Factory Method (계속)

의도 (Intent)

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

객체 생성에 대한 인터페이스를 정의한다. 그러나, 생성할 객체에 대한 클래스의 선택은 이 인터페이스를 구현(Implementation)한 클래스가 결정하도록 한다. Factory Method는 객체 생성에 대한 책임을 인터페이스를 구현한 클래스에 전가한 것이다.

구조와 참여객체



Product :
팩토리 메소드가 생성하는 오브젝트의 인터페이스를 정의함

Creator :
- Product 타입의 객체를 반환하는 팩토리 메소드를 선언

ConcreteCreator :
ConcreteProduct를 반환하기 위해 팩토리 메소드를 재정의함

ConcreteProduct :
Product 클래스에 정의된 인터페이스를 실제로 구현함

Material from: - "Design Pattern",

3. Factory Method (계속)

적용 / 활용

✓어떤 클래스의 객체를 생성해야 할지 미리 알지 못할 경우
✓하위 클래스가 객체를 생성하기를 원할 경우, 하위 객체에게 객체 생성을 위임하려 할 경우
✓동일한 프로그램 로직을 가지면서 내부적으로 생성할 객체만 다를 경우

장점 / 단점

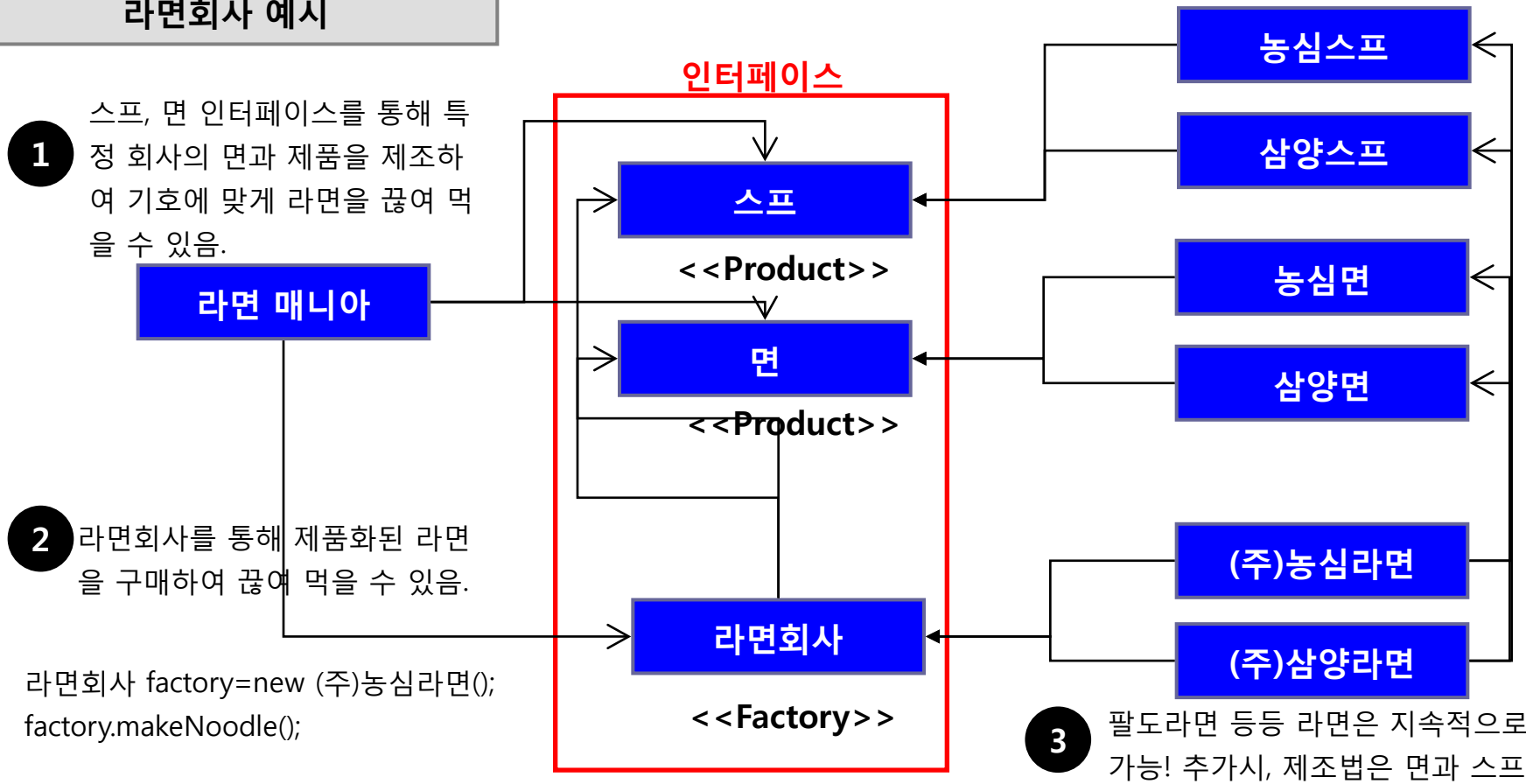
✓객체 생성 흐름을 일괄적으로 관리 가능
✓객체 생성 과정을 제어 가능
✓제품이 많아지면, 즉 ConcreteCreator 객체가 많아지면, 관리해야 할 객체가 많아짐

Material from: - "Design Pattern",

4. Abstract Factory

- Factory Method 패턴을 확장
- 제품군을 구성하는 객체들을 전담 생성하는 클래스를 두고, 상속을 이용하여 수정과 확장에 용이하게 만든 패턴

라면회사 예시



1 스프, 면 인터페이스를 통해 특정 회사의 면과 제품을 제조하여 기호에 맞게 라면을 끓여 먹을 수 있음.

2 라면회사를 통해 제품화된 라면을 구매하여 끓여 먹을 수 있음.

라면회사 factory=new (주)농심라면();
factory.makeNoodle();

3 팔도라면 등등 라면은 지속적으로 추가 가능! 추가시, 제조법은 면과 스프로 구성되어야 함.

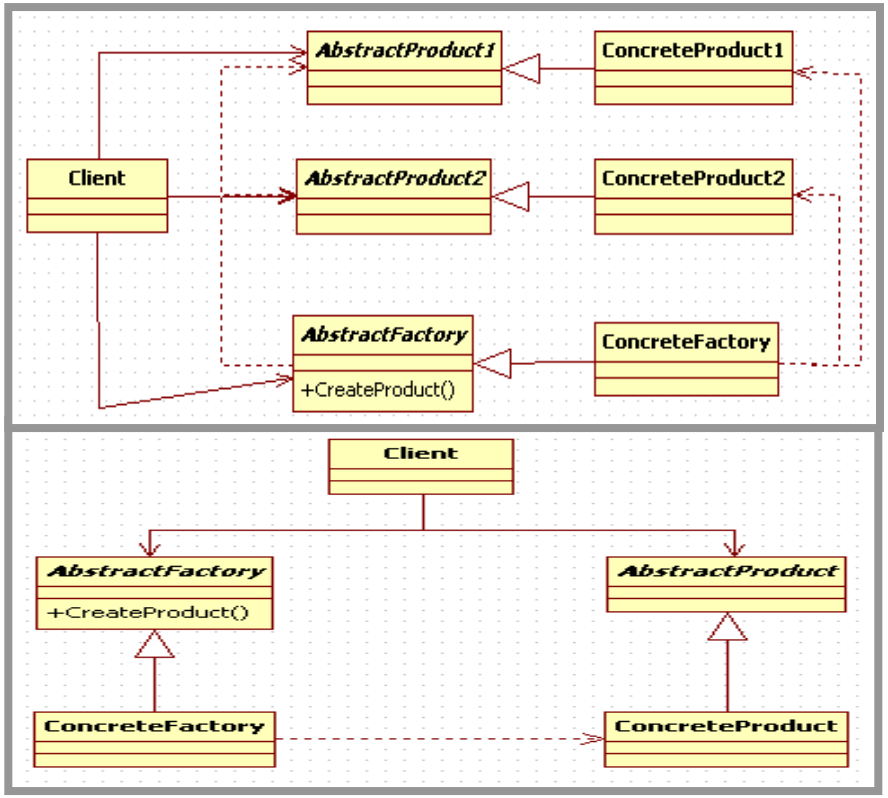
Material from: - "Design Pattern",

4. Abstract Factory (계속)

의도 (Intent)

Provide an interface for creating families of related or dependent objects without specifying their concrete classes
서로 밀접하게 관련된 객체들 또는 별로 연관되지 않는 객체들의 패밀리(Family)를 생성할 때 그들의 클래스가 무엇인지 구체적으로 알 필요 없이 해당하는 객체를 생성할 수 있는 인터페이스를 제공한다.

구조와 참여객체



▪**AbstractFactory :**
객체를 생성하는 인터페이스를 집단적으로 선언한다. 즉, 클라이언트가 사용할 객체를 생성하는 FactoryMethod들을 선언한다. 이 메소드는 Product를 반환한다. 이때 FactoryMethod 메소드는 생성해야 할 객체의 종류별로 여러 개가 존재할 수 있다.

▪**AbstractProduct :**
클라이언트가 사용할 객체의 타입, ConcreteFactory에서 생성된 실제 객체는 본 클래스의 하위 클래스의 ConcreteProduct 객체이다.

▪**ConcreteFactory :**
AbstractFactory를 상속받아 FactoryMethod를 오버라이딩하여 Product클래스를 상속받는 SpecificProduct객체를 생성하는 코드를 구현한다.

▪**ConcreteProduct :**
ConcreteFacoty에 의해 실제로 생성되는 객체이다. AbstractProduct 클래스를 상속받는다

Material from: - "Design Pattern",

4. Abstract Factory (계속)

적용과 활용

✓컴포넌트들이 어떻게 생성, 조합, 표현되어지는가에 무관하게 독립적인 시스템을 만들고자 할 경우
✓하나 이상의 제품군들 중 하나를 선택해서 시스템을 설정해야 하고 , 한번 구성된 제품을 다른 것으로 대체하고자 하는 경우
✓관련된 객체군을 함께 사용해서 시스템을 설계하였고, 이 제품이 갖는 제약사항을 강제하고자 할 경우
✓제품의 라이브러리를 제공하는데 있어서 구현내용은 숨긴 채 인터페이스만을 노출시키고자 할 경우

장/단점

Factory 추가는 간단하지만, 부품 추가는 어려움.
왜냐하면 AbstractFactory 인터페이스는 생산되어질 Product의 집합을 고정해놓기 때문이다.
새로운 Product를 지원하는 데는 factory의 인터페이스를 확장할 필요가 있음.
따라서, 이미 만들어진 Factory에 대한 연속적인 수정 작업이 발생하게 됨.
(AbstractFactory 클래스와 모든 서브 클래스들을 바꾸어야 함.)

Material from: - "Design Pattern",

4. Abstract Factory (계속)

관련패턴

Factory Method 패턴 vs. Abstract Factory 패턴

• **Factory Method 패턴 :**

- 대행 함수를 통한 객체 생성 문제
- 객체를 생성하는 작업이 복잡하고 어렵거나 특정한 절차를 따라야 할 경우 이를 대행하는 함수를 둠
- ex) 윈도우OS에서 더블클릭 시 : 문서이름의 확장자에 따라 적절한 응용프로그램 실행

• **Abstract Factory 패턴 :**

- 제품군별 객체 생성 문제
- 제품군을 구성하는 객체를 전담 생성하는 클래스를 두되, 새로운 제품군 생성을 추가하는 것이 쉽도록 클래스 상속을 도입하고, 구체적인 제품군별 Factory 클래스 상위에 Abstract Base Class를 정의한 형태의 설계 구조
- ex) 운영체제 : 파일구조, 메모리관리... -> 1.윈도우 / 2유닉스 / 3리눅스

Material from: - "Design Pattern",

5. Prototype

객체를 생성하는 방법

1 *클래스가 있는 경우*

```
Something sth=new Something();
```

객체선언 — 인스턴스화 → 클래스에 기반한 객체 생성

- Something 이라는 클래스를 미리 선언해 두고, 객체를 생성할 때마다 컴파일러에게 생성하려는 객체의 정보를 줌
- 클래스라는 거푸집을 만들어 같은 타입의 객체를 무한히 찍어 낼 수 있음.

2 *클래스가 없는 경우*

```
AnyOther ao=new AnyOther();
Something sth=ao.clone();
```

복제 → Prototype에 기반한 객체 생성

- 정의된 클래스가 없으므로 컴파일러에게 생성될 객체에 대한 정보를 줄 수 없는 상황이므로 이미 생성되어 있는 객체를 이용함.
- 즉, 복제를 통해서 자신과 똑같은 혹은 일부 기능을 가지는 객체를 만들어 냄
- 전제조건 : Prototype으로 사용되는 객체가 자신을 복제할 수 있어야 함.

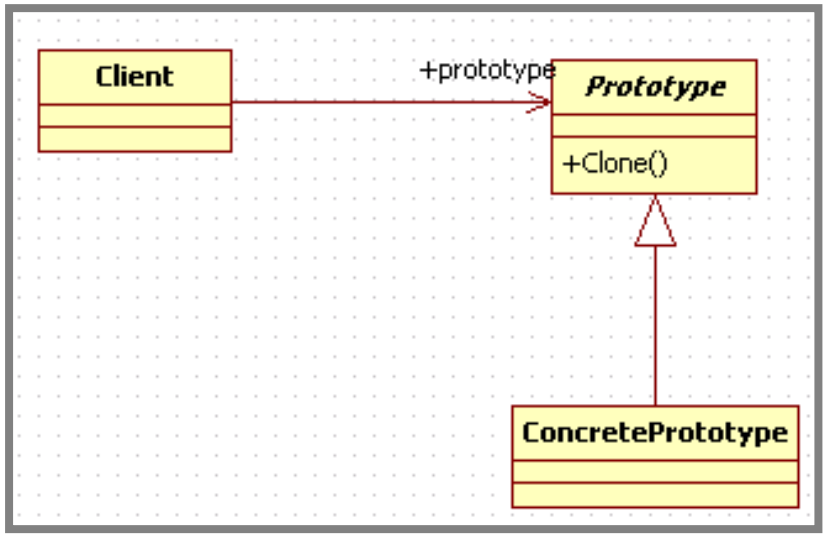
Material from: - "Design Pattern",

5. Prototype (계속)

의도 (Intent)

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype cloning이 가능한 객체(프로토타입 객체)를 복사 함으로써 새로운 객체를 생성함

구조와 참여객체



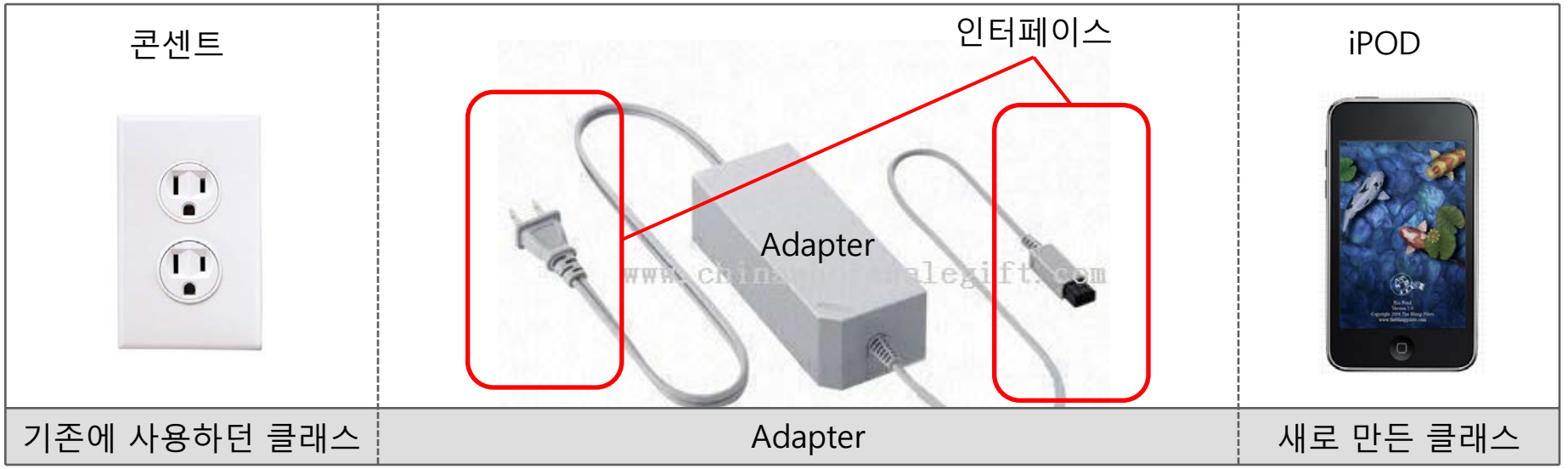
- Prototype : 복제하기 위한 인터페이스를 정의
- ConcretePrototype : 복제를 위한 구체적인 오퍼레이션을 구현
- Client :

인스턴스를 복사하는 메소드를 이용하여 새로운 인스턴스를 만듬

Material from: - "Design Pattern",

iPOD과 110v 콘센트

1. Adapter



-우리가 일사 생활에서 10v를 사용하는 전자 제품에 220볼트의 전원을 연결할 때, 중간에 어뎁터라는 것을 사용해서 전류를 변화 시켜주는 것과 같이 Adapter패턴은 한 클래스의 인터페이스를 다른 인터페이스로 변환해 주는 패턴

-Adapter패턴을 사용하면 인터페이스 호환성 문제 때문에 같이 사용할 수 없는 클래스를 연결해 줌

의도 (Intent)

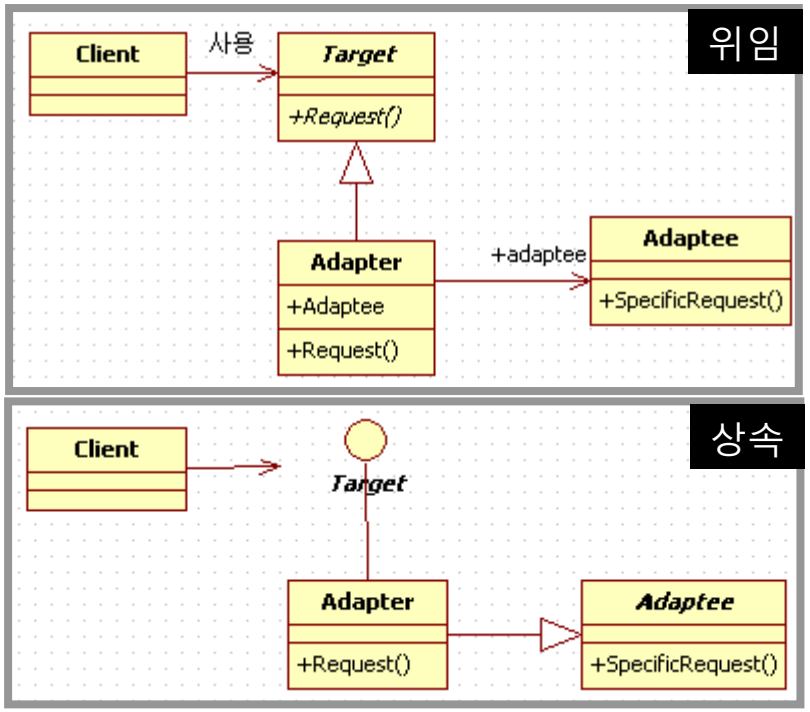
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

클래스의 인터페이스를 클라이언트가 기대하는 형태의 인터페이스로 변환함. 어댑터 패턴은 서로 일치하지 않는 인터페이스를 갖는 클래스들을 함께 동작시킴

Material from: - "Design Pattern",

1. Adapter (계속)

구조와 참여객체



▪Target :
클라이언트가 사용할 도메인에 종속적인 인터페이스를 정의

▪Client :
Target 인터페이스에 만족하는 객체와 협력하여 일함

▪Adaptee :
인터페이스 개조가 필요한 기존의 인터페이스를 정의

▪Adapter :
Target 인터페이스에 Adaptee 인터페이스를 맞춰 주는 클래스

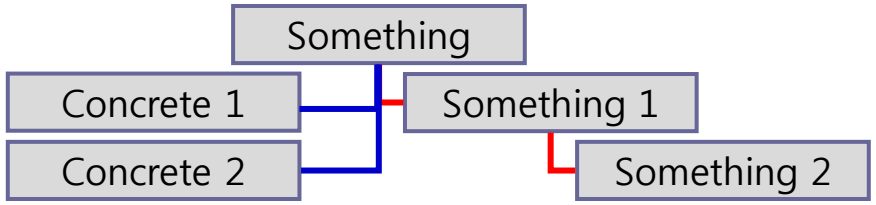
적용과 활용

✓기존 클래스를 재사용하려고 하나 그 인터페이스가 원하는 것과 동일하지 않을 경우
✓서로 관계가 적고 호환되는 인터페이스가 별로 없는 클래스들을 활용하여 새로운 클래스를 생성하려고 할 경우

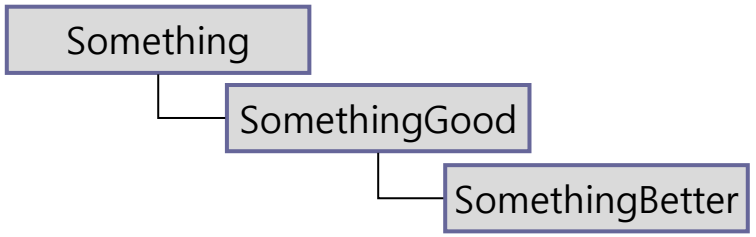
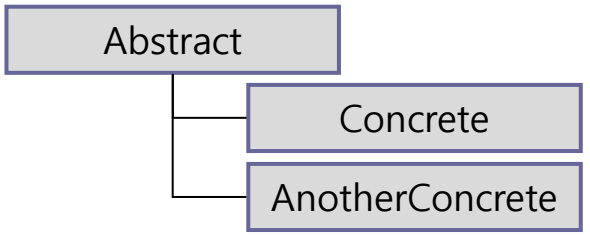
Material from: - "Design Pattern",

2. Bridge

확장과 구현의 혼재



✓클래스 계층이 하나라면 기능 클래스 계층과 구현 클래스 계층이 하나의 계층구조 안에 혼재되므로 클래스 계층을 복잡하게 하여 예측하기 어렵게 만듦.
✓따라서, 기능클래스계층과 구현클래스계층을 두 개의 독립된 클래스 계층으로 분리하고, 두 클래스 사이에 Bridge 역할을 하는 매개를 활용하여 둘 간의 연결고리를 만듦.

기능클래스계층		구현클래스계층
	**Bridge** 기능 클래스계층과 구현클래스 계층을 매개하는 역할	
자식(하위,확장) 클래스로 계층을 깊게 하면서 기능을 추가		상위,추상 클래스에서 인터페이스 규정하고 하위,구상클래스에서 인터페이스를 구현

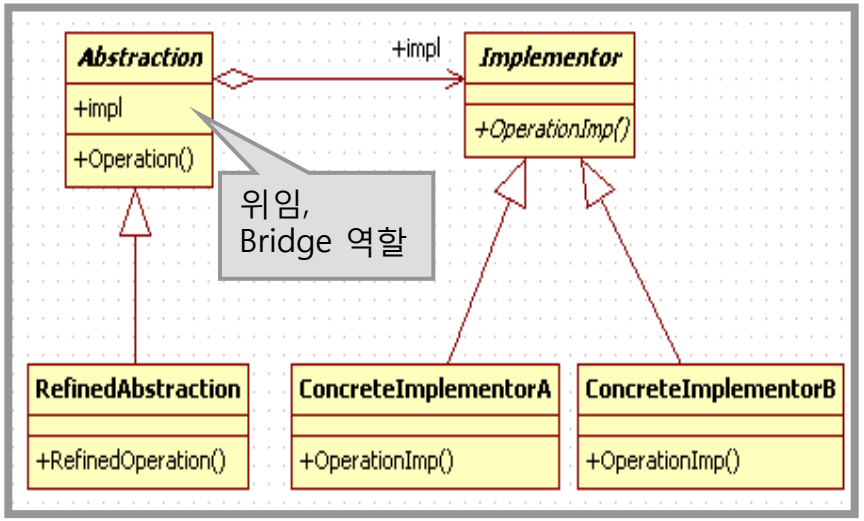
Material from: - "Design Pattern",

2. Bridge (계속)

의도 (Intent)

Decouple an abstraction from its implementation so that the two can vary independently
구현과 추상화 개념을 분리함. 이렇게 함으로써 구현 자체도 하나의 추상화 개념으로 다양한 변형이 가능해지고, 구현과 독립적으로 인터페이스도 다양함을 가질 수 있음.

구조와 참여객체



▪Abstraction :
- 추상화 개념에 대한 인터페이스 제공
- 구현자(Implementor)에 대한 참조자 관리

▪RefinedAbstraction :
- Abstraction에 정의된 인터페이스 확장
- Abstraction에 대해 기능 추가

▪Implementor :
- 구현 클래스에 대한 인터페이스 제공
- Abstraction의 인터페이스와 정확하게 일치할 필요는 없음

▪ConcreteImplementor :
Implementor 인터페이스를 구현하여 실제적인 구현내용을 담고 있음.

✓Implementor 인터페이스는 기본적인 오퍼레이션을 제공하고,
✓Abstraction 은 이러한 기본적인 것을 바탕으로 한 고수준의 오퍼레이션을 정의함

Material from: - "Design Pattern",

2. Bridge (계속)

적용과 활용

✓인터페이스와 구현 방식이 완전 결합되는 것을 피하고자 할 경우
✓인터페이스와 구현 방식이 각각 서로 다른 형태의 하위 클래스 구조를 가지면서 확장되기를 원할 경우
✓인터페이스의 구현 방식이 변경되더라도 그 인터페이스를 사용하는 client 소스코드는 다시 컴파일하지 않아야 할 경우
✓인터페이스의 구현 방식을 client에게 완전히 숨기고 싶을 경우
✓어떤 클래스의 상속 구조가 여러 개의 분류 기준에 의해 정의되어 복잡하고 새로운 하위 클래스 정의가 힘들어 각 분류 기준마다 독립된 클래스 상속 구조를 정의하고 싶을 경우
✓하나의 구현 객체를 여러 개의 인터페이스 객체가 공유하게 만들면서도 client는 이를 알지 못하게 하고 싶을 경우

장 / 단점

✓인터페이스와 구현을 분리시켜 줌
- 디자인 측면 : 계층화, 구조화 가능
- 사용자 측면 : 인터페이스와 어떤 객체로 이루어지는 지만 알고 구체적 구현 내용은 알 필요 없음
✓실행 시간에 구현 객체를 바꾸거나 설정할 수 있게 해줌
✓인터페이스와 구현이 분리됨으로써 구현 내용이 변경되더라도 인터페이스 클래스와 client는 다시 컴파일 할 필요 없음. 이런 특징은 특히 서로 다른 버전의 클래스 라이브러리에 대해 호환성을 보장해야 할 경우 유용
✓인터페이스 클래스와 구현 클래스가 별도의 상속 구조를 가지므로 서로 독립적인 확장이 가능

Material from: - "Design Pattern",

2. Bridge (계속)

관련패턴

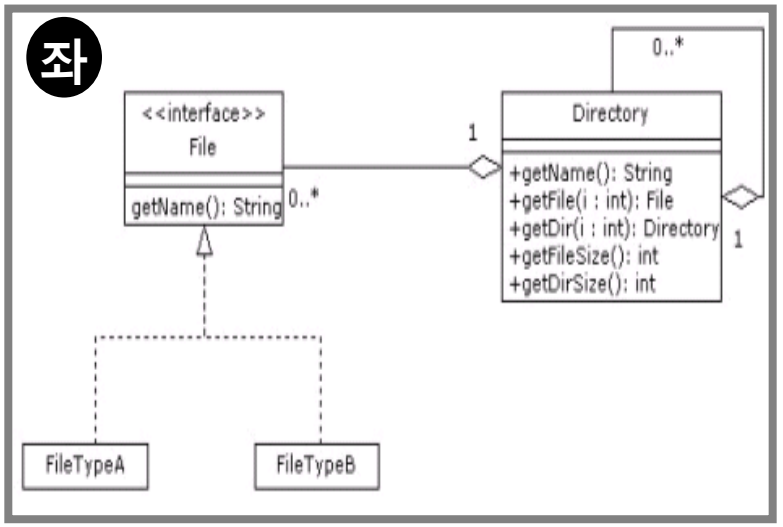
Adapter 패턴 vs. Bridge 패턴

Adapter는 존재하고 있는 객체에 대한 인터페이스를 바꾸기 위한 의도를 갖지만,
Bridge는 인터페이스를 해당 구현체(implementation)와 분리하기 위한 목적을 띈다.

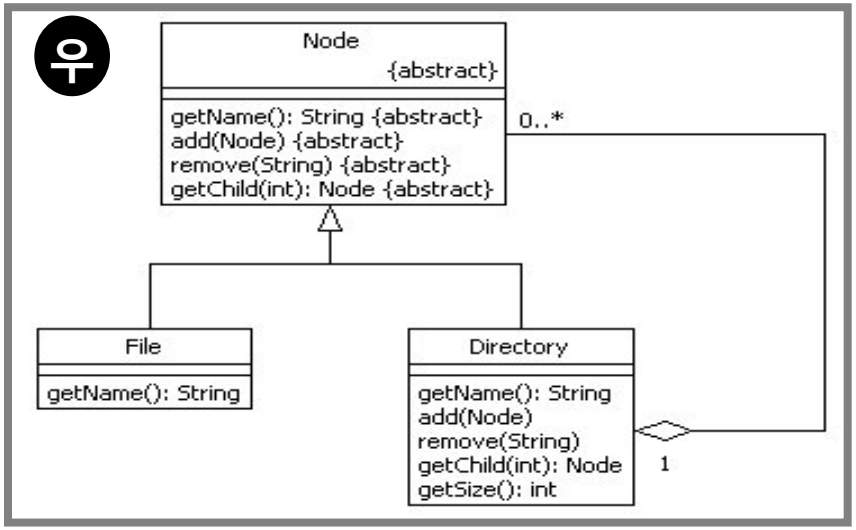
Material from: - "Design Pattern",

3. Composite : 그릇과 내용물을 동일시하고 재귀적 구조를 형성하여 객체간 Tree 구조를 처리하는 데 유용한 패턴

Tree 구조 디자인



Composite 패턴 적용



▪Directory와 File 객체가 서로 다른 인터페이스를 가지도록 설계

▪각각의 객체가 서로 다른 인터페이스를 외부에 제공함으로써, 해당 파일 시스템을 사용하는 클라이언트가 각각의 객체 타입을 구분하여 해당 객체의 서비스를 이용하도록 되어 있다.

▪File이 tree 구조의 leaf역할을 하고 Directory는 또 다른 Directory나 File 객체를 포함하는 composite의 기능을 가지는 internal node의 역할을 하고 있기 때문에, Composite 패턴을 여기서의 예에 적용

▪두 객체가 동일한 인터페이스를 외부에 제공함으로써 클라이언트가 해당 객체의 타입을 고려하지 않고, 즉, 시스템이 투명성(transparency)을 제공하여, 서비스를 일관성(uniformity) 있게 이용가능

Material from: - "Design Pattern",

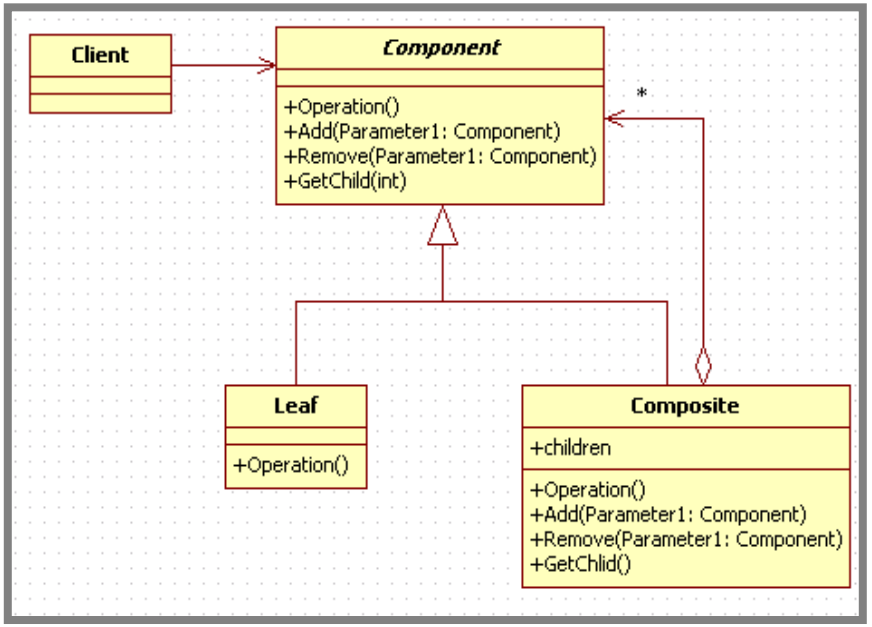
3. Composite (계속)

의도 (Intent)

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

Composite 패턴은 객체간의 존재하는 부분-전체의 계층 구조를 tree 형태로 나타내주며, 클라이언트가 각각의 객체와 객체를 포함하는 객체(composite)를 일관되게 다룰 수 있게 해 준다.

구조와 참여객체



▪**Component :**

Leaf 역할과 Composite 역할을 동일시하도록 Leaf와 Composite역할의 공통적인 상위 클래스로 정의

▪**Composite :**

- Component 인터페이스에서 정의된 자식과 관련된 동작을 구현함.
- Leaf 역할과 Compsite 역할을 모두 담을 수 있음.

▪**Leaf :**

- Component 인터페이스에서 정의한 Primitive Operation을 위한 행위 정의
- 자식 노드를 가질 수 없음

▪**Client :**

Composite 패턴 사용자

Material from: - "Design Pattern",

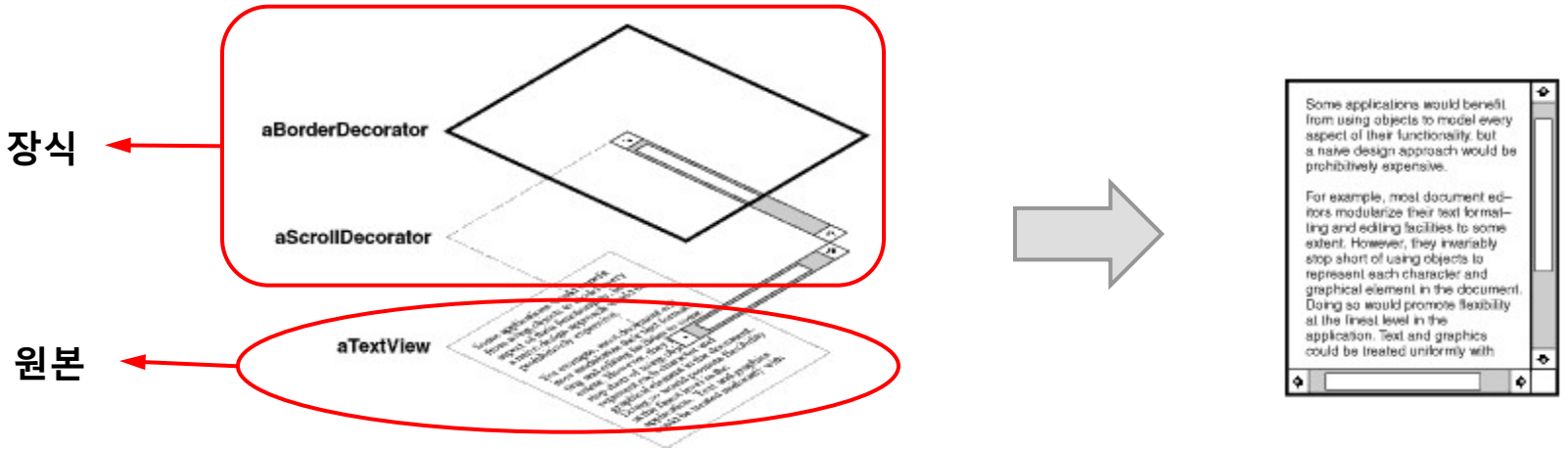
4. Decorator

- **Also Know as Wrapper Pattern**
- **장식도 내용도 동일한 인터페이스를 가지도록 동적으로 기능을 추가할 수 있는 패턴**
- **API 투과적 : 장식을 통해 내용물을 감싸도 인터페이스는 전혀 감출 수 없음.**

개념정립

✓어떤 클래스에 대해서 테두리(border)를 부여하고자 하는 경우, 상속을 통해 테두리를 갖는 하위 클래스를 만들 수 있음. 그러나, 이는 정적인 방법으로 테두리가 생성되어, 테두리가 생기는 시점이나 방법을 통제할 수가 없음

✓이보다 유연한 방법은 테두리를 갖는 객체가 테두리를 갖고자 하는 객체를 포함하는 것이다. 이렇게 부가적인 기능을 원하는 객체를 포함하는 객체를 Decorator라고 부름.



위 그림은 aTextView라고 하는 객체에 스크롤과 테두리를 추가하고자 하는 경우, 상속을 통해 클래스를 확장하지 않고, 이를 위한 Decorator를 생성하여 해결하는 모습을 나타낸다.

Material from: - "Design Pattern",

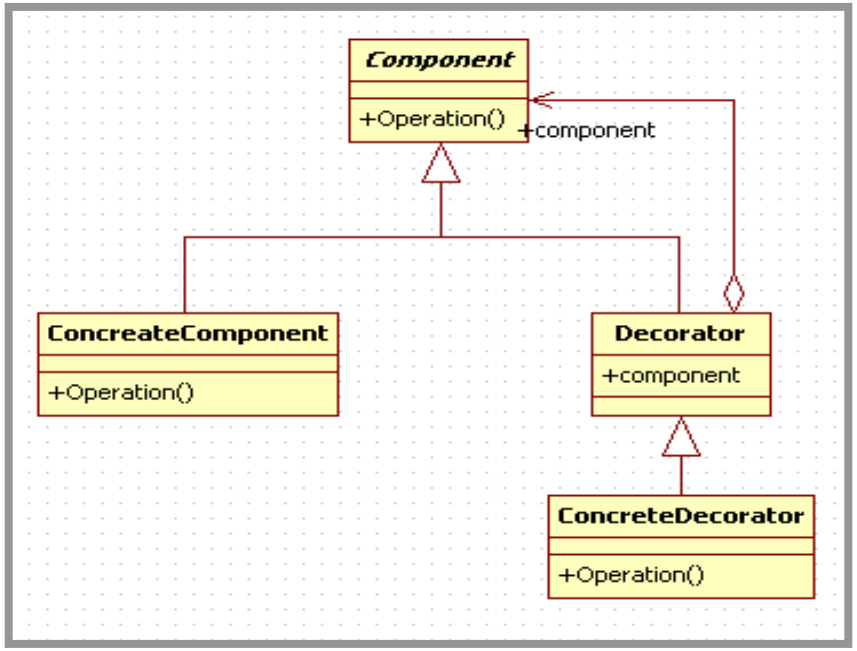
4. Decorator (계속)

의도 (Intent)

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

객체에 동적으로 새로운 서비스를 추가할 수 있게 함. 데코레이터 패턴은 기능의 확장을 위해 서브클래스를 생성하는 것보다 융통성 있는 방법을 제공함.

구조와 참여객체



▪Component :
동적으로 추가할 서비스를 가질 가능성 있는 객체들에 대한 인터페이스 정의

▪ConcreteComponent :
Component에서 정의한 인터페이스를 구체적으로 구현하는 객체

▪Decorator :
Component객체에 대한 참조자를 관리하면서 Component에 정의된 인터페이스에 순응하는 인터페이스를 정의

▪ConcreteDecorator :
- Component에 추가할 서비스를 실제로 구현하는 클래스
- Decorator에 정의된 기본 오퍼레이션을 만족하면서 추가적인 행위를 처리

Material from: - "Design Pattern",

4. Decorator (계속)

장/단점

✓정적인 상속보다 더 유연함. 데코레이터는 실시간 중에 기능을 붙이고 뗄 수 있다.
✓쓰지 않는 기능을 위해 비용을 지불하지 않음. – 데코레이더를 붙이지 않은 클래스는 단순한 유닛 클래스.
✓확장이 쉽다.

특징

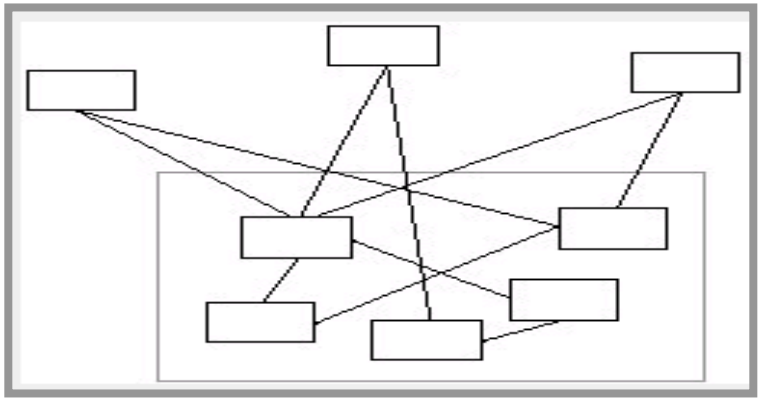
✓데코레이터의 슈퍼클래스(부가적 기능의 슈퍼클래스)는 자신이 장식하고 있는 객체(중심 되는 기능의 클래스)의 슈퍼클래스와 같다.
✓한 객체를 여러 개의 데코레이터로 감쌀 수 있다.
✓데코레이터는 자신이 감싸고 있는 객체와 같은 슈퍼클래스를 가지고 있기 때문에 원래 객체가 들어갈 자리에 데코레이터 객체를 집어넣어도 상관 없다.
✓데코레이터는 자신이 장식하고 있는 객체에게 어떤 행동을 위임하는 것 외에 원하는 추가적인 작업을 수행할 수 있다.
✓객체는 언제든지 감쌀 수 있기 때문에 실행 중에 필요한 데코레이터를 마음대로 적용할 수 있다.
- 데코레이터 패턴은 기본적인 데이터에 첨가할 데이터가 다양하고 일정하지 않을 때 효율적이다.

Material from: - "Design Pattern",

5. Facade

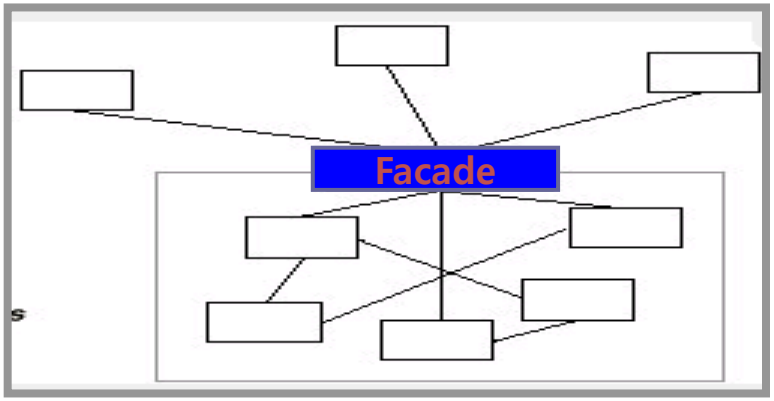
- **facade = 건물의 정면**
- **복잡하게 얽혀 있는 것을 정리해서 높은 레벨의 인터페이스를 시스템 외부에 제공하는 패턴**

개념정립



많은 클래스,
사용법이 복잡한 것

단순화된 하나의
인터페이스 도입



단순화된 하나의 인터페이스,
즉 Facade 도입을 통해
연결성, 종속성 최소화

기대효과

✓서브시스템의 구성 요소를 보호할 수 있음
✓서브시스템과 클라이언트 코드간 결합도를 줄일 수 있음
✓서브시스템 클래스를 직접 사용하는 것을 막지는 않으므로, Facade를 이용할지 직접 서브시스템 클래스를 접근할지는 클라이언트의 결정에 달려 있음

Material from: - "Design Pattern",

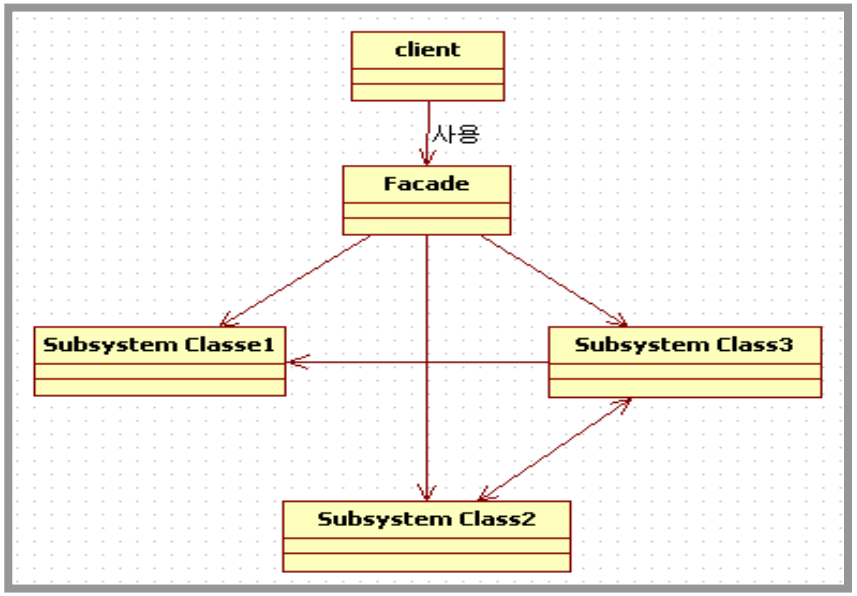
5. Facade (계속)

의도 (Intent)

Provide a unified interface to a set of interfaces i a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use

서브시스템을 합성하는 다수의 객체들의 인터페이스 집합에 대해 일관된 하나의 인터페이스를 제공함. 퍼사드는 서브시스템을 사용하기 쉽게 하기 위한 포괄적인 개념의 인터페이스를 정의함

구조와 참여객체



▪Facade :
단순하고 일관된 통합 인터페이스를 제공하며, 서브시스템을 구성하는 어떤 클래스가 어떤 요청을 처리해야 하는지 알고 있으며 클라이언트 요청을 해당하는 서브시스템 객체에 전달

▪Subsystem Classes :
- 서브시스템의 기능성을 구현
- Facade 객체에 의해 할당된 작업을 처리하지만, Facade에 대한 아무런 정보도 갖고 있지 않음

Material from: - "Design Pattern",

5. Facade (계속)

활용 / 적용

복잡한 서브시스템에 대한 단순한 인터페이스 제공이 필요할 때
클라이언트와 구현 클래스간 너무 많은 종속성이 존재할 때 Facade 사용으로 결합도를 낮출 수 있음
서비스시스템들의 계층화를 이루고자 할 때 Facade가 각 서브시스템의 계층별 접점을 제공함

관련패턴

✓Abstract Factory : 서브시스템에 독립적인 방법으로 서브시스템 객체를 생성하는 인터페이스를 제공하기 위해 Facade와 Abstract Factory를 함께 사용할 수 있음
✓Singleton : 대개 Facade 객체는 오직 하나만 필요하므로 Facade 객체는 종종 Singleton임

Mediator 패턴과 비교

공통점	기존이 클래스들의 기능을 추상화한다는 점에서 두 패턴은 유사함.
차이점	✓Mediator : - [목적] 여러 객체들 사이의 협력 관계를 추상화하여 기능성이 집중화를 막음 - Mediator에게 기능이 집중되어 있음 - colleague 클래스들은 자기들끼리 직접 통신하지 않고 Mediator만을 인식, 의사소통함. ✓Facade : - [목적] 서브시스템 인터페이스 자체를 추상화하여 사용을 용이하게 함 - 새로운 기능을 추가할 수 없음. - 서브시스템의 클래스들은 Facade를 알지 못함.

Material from: - "Design Pattern",

6. Flyweight

- **Flyweight = 플라이트급, 체중 50kg 이하**
- **경량급 다수의 객체들을 대상으로 이들의 공통 정보를 공유하여 메모리 사용을 효율화하는 패턴**

다수의 게임 캐릭터 구현

게임을 예로 생각해보자.

- 주인공 캐릭터에 비해 적 캐릭터는 동일한 것이 다수 나오는 경우가 대부분
- 수많은 적 캐릭터를 일일이 하나씩 객체로 생성한다고 가정한다면, 상당한 리소스가 요구됨을 직감할 수 있음.
- 보다 효율적인 디자인은 없을까?

1	[대전제] 적 캐릭터 인스턴스를 가능한 대로 공유시켜서 쓸데없이 new하지 않도록 하자. 즉, 인스턴스가 필요할 때마다 항상 new 하는 것이 아니라 이미 만들어져 있는 인스턴스를 사용할 수 있으면 그것을 공유해서 이용함.	Flyweight의 기본철학
2	공유할 만한 것? 생김이 동일하니 그래픽적 요소 공유가능	intrinsic information
3	공유하지 못할 것? 맵상에서 캐릭터의 위치, 에너지 게이지는 서로 다름.	extrinsic information

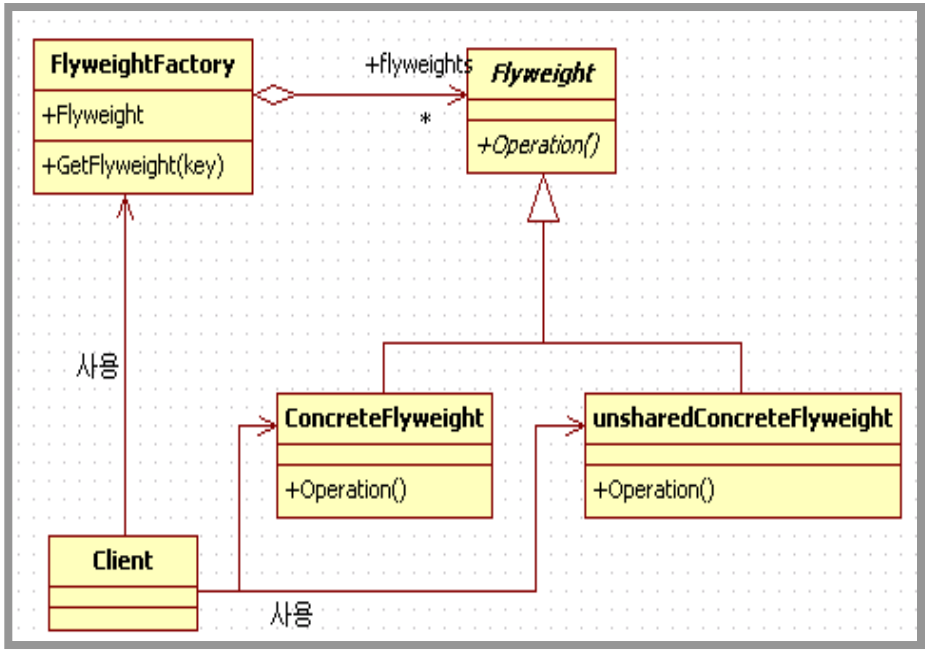
Material from: - "Design Pattern",

6. Flyweight (계속)

의도 (Intent)

Use sharing to support large numbers of fine-grained objects efficiently
수많은 미세한 객체들을 효과적으로 지원하여 공유하도록 함.

구조와 참여객체



▪Flyweight :
Flyweight가 받을 수 있고 extrinsic 상태에서 동작하는 인터페이스를 선언
▪ConcreteFlyweight :
- Flyweight 인터페이스를 구현하고 intrinsic 상태를 위한 저장소를 추가함
- ConcreteFlyweight 객체는 공유할 수 있어야 함
- instrinsic 상태만 저장해야 함. 즉, ConcreteFlyweight 객체의 context에 무관하게 정보가 저장되어야 함.
▪UnsharedConcreteFlyweight :
- Flyweight 서브클래스들이 모두 공유되어져야 하는 것은 아님.
- Flyweight 인터페이스가 공유가능하다 하더라도 강제적으로 공유하지는 않음.
- UnsharedConcreteFlyweight 객체가 자신의 자식으로 ConcreteFlyweight을 가지는 것은 흔한 일임.
▪FlyweightFactory :
- Flyweight 객체를 만들고 관리함
- flyweight가 올바르게 공유되도록 보증함, 주로 singleton 패턴을 사용
▪Client : -flyweight에 대한 참조자 관리
- flyweight의 extrinsic 상태를 계산하고 저장함

Material from: - "Design Pattern",

6. Flyweight (계속)

적용 / 활용

✓어플리케이션이 대량의 객체를 사용해야 할 경우
✓객체의 수가 너무 많아서 저장 비용이 너무 높아질 경우
✓대부분의 객체 상태를 부가적인 것으로 만들 수 있을 경우
✓어플리케이션이 객체 식별자에 비중을 두지 않는 경우
– flyweight 패턴은 서로 공유될 수 있음을 의미하는데, 식별자가 있다는 것은 서로 다른 객체로 구별해야 한다는 의미이므로 flyweight 패턴 적용이 어려움

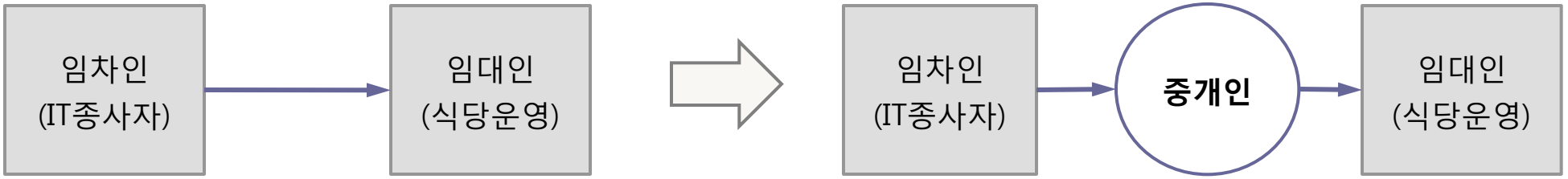
장 / 단점

✓Extrinsic State를 찾거나 계산하는데 일반적인 경우보다 실행 시간 비용이 더 들 수 있다.
✓저장 공간이 절약되는 정도는 다음의 요소에 의해 결정됨
- 객체를 공유함으로써 감소되는 객체의 총 개수
- 객체당 Intrinsic State의 양
- Extrinsic State가 계산되는 것이냐, 저장되는 것이냐에 따라 필요한 기억 공간

Material from: - "Design Pattern",

7. proxy = 대리인 : 당사자의 일 처리를 대신 하는 사람.
그러나 대리에 지나지 않으므로 할 수 있는 일에는 한계가 존재하므로,
대리인이 할 수 있는 범위를 넘는 일이 발생시, 당사자에게 와서 상담함.

부동산 계약 예시



임차인, 임대인 모두 본연의 직업이 있으나, 부동산 계약이라는 일을 하기 위해 임차인과 임대인 직접 서로 의사소통해야하므로 자신의 일을 잠시 멈추고 부동산 계약일에 전념해야 함

임차인과 임대인 사이에 부동산 중개업자를 두면, 임차인/임대인 모두 자신의 원래 업무를 수행하면서, 부동산 계약관련일을 대행시킬 수 있음.
→ 임차인/ 임대인 성능 향상
→ 부동산 관련한 법적 사항에 대한 위험 부담 감소
→ 중개수수료 지불필요

의도 (Intent)

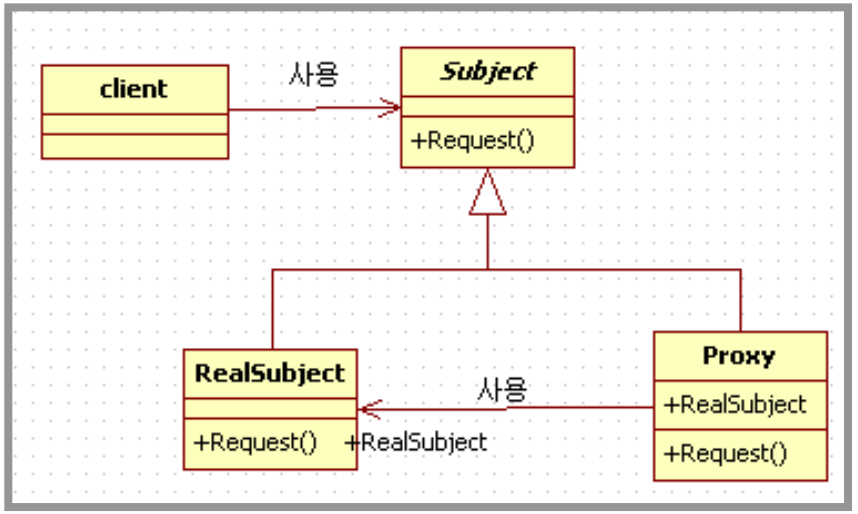
Provide a surrogate or placeholder for another object to control access to it

다른 객체에 접근하기 위해 중개자 또는 변수를 제공함.

Material from: - "Design Pattern",

7. Proxy (계속)

구조와 참여객체



▪Subject :
- RealSubject와 Proxy를 동일시하도록 공통적인 인터페이스를 정의
- RealSubject가 요청되는 곳에 proxy를 사용할 수 있게 함

▪RealSubject :
-proxy가 대표하는 실체 객체

▪Proxy :
- 실제로 참조할 대상에 대한 참조자 관리
- Subject와 동일한 인터페이스를 제공하여 실제 대상을 대체할 수 있어야 함
- 실제 대상에 대한 접근제어, 생성, 삭제를 책임짐

적용 / 활용

✓Remote Proxy : RealSubject가 네트워크 상대 쪽에 있음에도 불구하고 마치 자신의 옆에 있는 것처럼 메소드를 호출
ex) RMI(Remote Call Invocation)
✓Virtual Proxy : 정말로 인스턴스가 필요한 시점에 생성/초기화
✓Protection Proxy : RealSubject 기능에 대해서 액세스 제한을 설정 . 정해진 사용자만 메소드 호출을 허가
✓Smart Reference : 객체에 대한 단순한 접근 이외의 부가적인 작업을 수행할 필요가 있을 때
ex1) 실제 객체에 대한 참조수를 저장하고 있다가 더 이상 참조되지 않을 경우 해당 객체를 자동으로 제거
ex2) 맨 처음 참조되는 시점에 영구적인 객체로서 메모리에 로딩
ex3) 객체에 접근하기 전에 lock이 걸려 있는지 확인하여 다른 객체가 변경하지 못하도록 함

Material from: - "Design Pattern",

7. Proxy (계속)

관련패턴

Adapter 패턴 vs. Proxy 패턴 :

Adapter 패턴의 경우 기존 객체가 가지고 있는 인터페이스와 다른 인터페이스를 client에게 제공하는 것이 목적인데 반해

Proxy 패턴은 기존 객체가 제공하는 것과 동일한 인터페이스를 제공하되 부가적인 기능이나 역할을 수행하게 만드는 것이 주목적

Decorator 패턴 vs. Proxy 패턴 :

Decorator 패턴의 동적으로 객체에게 역할이나 기능을 추가하는 것인데 반해,

Proxy 패턴은 Proxy 객체를 이용해 기존 객체에 대한 접근제어나 성능향상 등의 부수적인 작업을 수행하는 것이 주목적

Material from: - "Design Pattern",

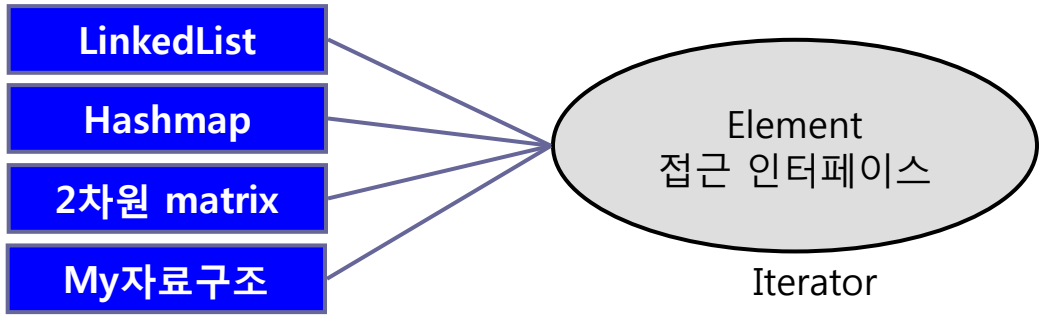
1. Iterator : 자료 구조를 캡슐화하고, 해당 내용물을 traverse 할 수 있도록 지원하는 패턴

내용물 접근

[과제] 아래 자료구조안에 element의 내용을 프린트하시오.

- **LinkedList** → 순차리스트 구조이므로, 차례대로 하나씩 가져옴
- **Hashmap** → (Key,Value) 쌍으로 이루어진 구조이므로, 키를 먼저 가져와서 해당 value를 가져옴
- **2차원 matrix** → N x N 구조이므로, for문을 2중으로 사용하여 value를 가져옴
- **My자료구조** → *어떻게 해당 자료구조의 element를 순차적으로 접근하면 좋을까?*

→문제점 : 자료구조를 알아야만 자료구조 안에 element에 접근할 수 있는 제약사항이 생김.
→해결안 : for문의 변수i의 기능을 추상화, 구현과 사용을 분리



1.어떤 자료구조든 자신의 element에 접근할 수 있는 인터페이스 정의
2. 각 자료구조는 자신이 갖고 있는 element를 접근할 수 있는 인터페이스를 구현함
3. Client는 자료구조에 대한 사전정보 없이도 구성 element에 접근이 가능하게 됨

Material from: - "Design Pattern",

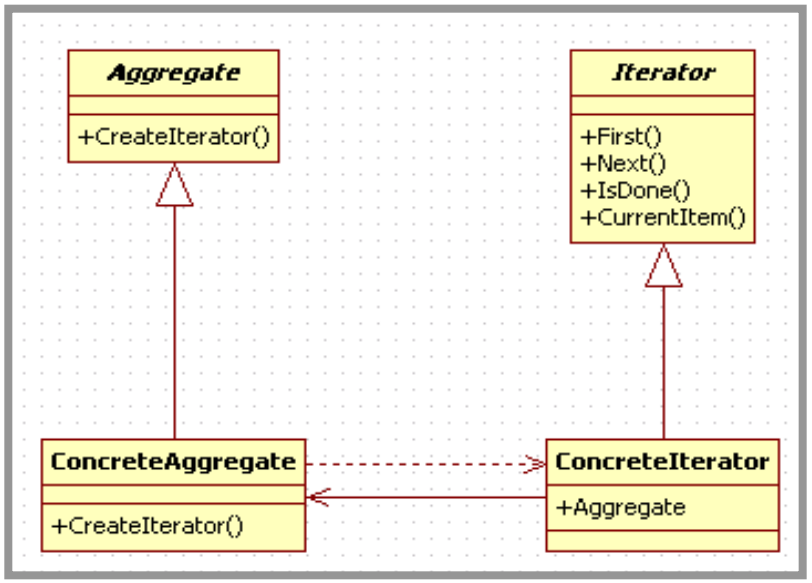
1. Iterator (계속)

의도 (Intent)

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

복합 객체 요소들의 내부 표현 방식을 공개하지 않고도 순차적으로 접근할 수 있는 방법을 제공함.

구조와 참여객체



▪Iterator :

요소를 접근하고 순회하는데 필요한 인터페이스 정의

▪ConcreteIterator :

Iterator에 정의된 인터페이스를 구현하는 클래스로서 순회 과정 중에 집합 객체 내의 현 위치를 기억함

▪Aggregate :

Iterator 객체를 생성하는 인터페이스를 정의

▪ConcreteAggregate :

해당 ConcreteIterator의 인스턴스를 반환하도록 Iterator 생성 인터페이스를 구현

Material from: - "Design Pattern",

1. Iterator (계속)

적용 / 활용

✓집합객체의 **내부 표현 방식을 노출하지 않은 채** 내용에 접근하고자 할 경우

✓집합객체의 **다양한 순회방식**을 지원해야 할 경우

✓서로 다른 구조를 가진 집합객체에 대해서 **통일된 순회 인터페이스**를 제공하고자 할 경우

관련패턴

✓**Composite :** Iterator 패턴은 종종 Composite 패턴처럼 재귀적 구조를 적용됨

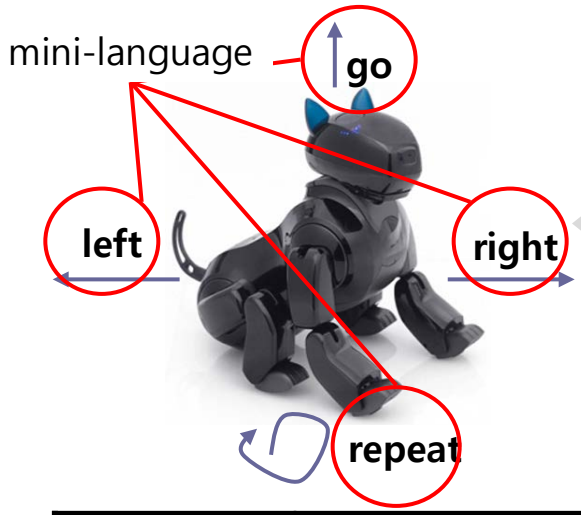
✓**Factory Method :** 다양한 Iterator를 사용하고 싶은 경우, 적당한 Iterator의 서브클래스를 얻기 위해 Factory Method 패턴을 사용함

✓**Memento :** Iterator는 iteration 상태를 포착하고 내부적으로 저장하기 위해 memento를 사용할 수 있음

Material from: - "Design Pattern",

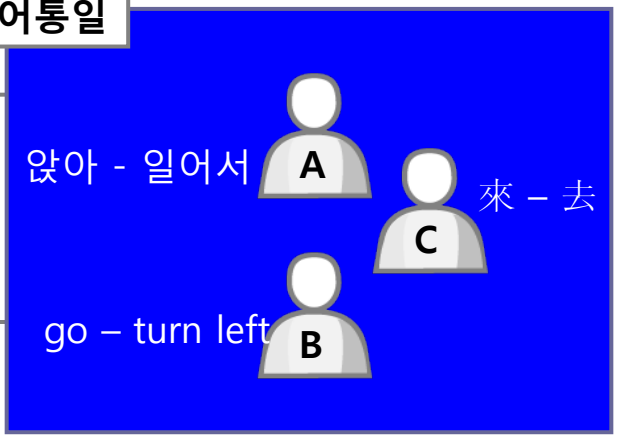
2. Interpreter

로봇에게 명령하기



§program go left end
§program 2 go right left end
§program left repeat end
§program repeat 4 go right end end

언어통일



문제상황	✓다양한 언어 존재 ✓한 사람이라도 상황에 따라 명령어들이 달라짐 ✓여러명의 사람이 서로 다른 명령어를 내림
해결방안	✓프로그램이 해결하려고 하는 문제를 간단한 '**미니언어**'로 표현 – 터미널표현, 터미널이 아닌 표현, **문법정의** ✓구체적인 문제를 미니언어로 쓰여진 '**미니 프로그램**'으로 표현 ✓미니 프로그램을 통역하는 역할을 하는 **통역프로그램**을 만듦 ✓통역 프로그램은 미니언어를 이해하고 미니 프로그램을 해석 및 실행 ✓해결해야 할 문제에 변화가 생겼을 때, **프로그램을 고치지 않고 미니 프로그램을 고쳐서** 해결

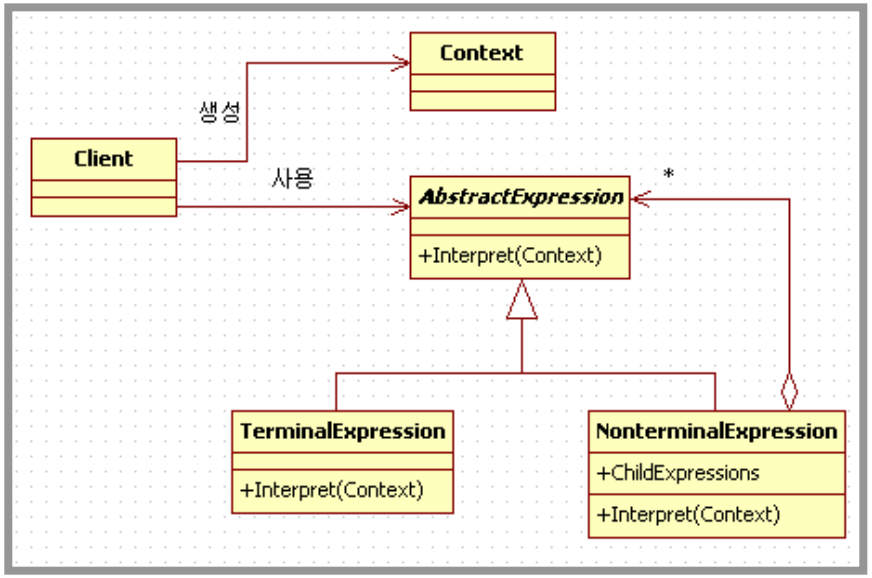
Material from: - "Design Pattern",

2. Interpreter (계속)

의도 (Intent)

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language
어떤 언어는 문법에 대한 표현을 정의하면서 그 언어로 기술된 문장을 해석하는 기법을 표현하기 위해서 인터프리터도 함께 정의되는 경우가 있다.

구조와 참여객체



▪AbstractExpression :
추상 구문 트리에 속한 모든 노드에 해당하는 클래스들이 공통적으로 가져야 할 Interpret() 오퍼레이션을 추상 오퍼레이션으로 정의

▪TerminalExpression :
- 문법에서 정의한 터미널 기호와 관련된 해석방법을 구현
- 문장을 구성하는 모든 터미널 기호에 대해 해당 클래스를 만들어야 함

▪NonterminalExpression :
- [문법] R ::= R1 R2 ..... Rn
- R에 대해 터미널기호가 아닌 클래스 정의
- R1 ~ Rn 의 모든 기호에 대응하는 인스턴스 변수 정의
- 터미널기호가 아닌 기호들에 대해 interpret() 오퍼레이션 구현함. interpret() 오퍼레이션은 R1~Rn의 interpret() 오퍼레이션을 재귀적으로 호출하는 것이 일반적임

▪Context :
인터프리터가 구문해석을 실행하기 위한 정보제공

Material from: - "Design Pattern",

2. Interpreter (계속)

적용 / 활용

✓정의할 문법이 간단한 경우
- 문법이 복잡한 경우는 문법을 정의하는 클래스 계층도가 복잡해지고 관리가 어렵게 됨. 이런 경우는 인터프리터 패턴을 사용하기 보다는 파서 생성기와 같은 도구를 이용하는 편이 더 나은 방법

✓효율성은 별로 고려 사항이 아닌 경우
- 효과적인 방법은 parse tree를 직접 해석하도록 구현하는 방식이 아닌 parse tree를 다른 형태로 변형시키는 것, 예를 들어 정규식 표현의 경우 일반적으로 유한 상태 기계 개념으로 변형됨

✓프로그램이 다양한 종류의 결과를 만들어내야 할 경우

관련패턴

✓Composite : NonterminalExpression 역할은 재귀적인 구조를 갖는 경우가 많기 때문에 Composite 패턴을 사용해서 표현되는 경우가 많음

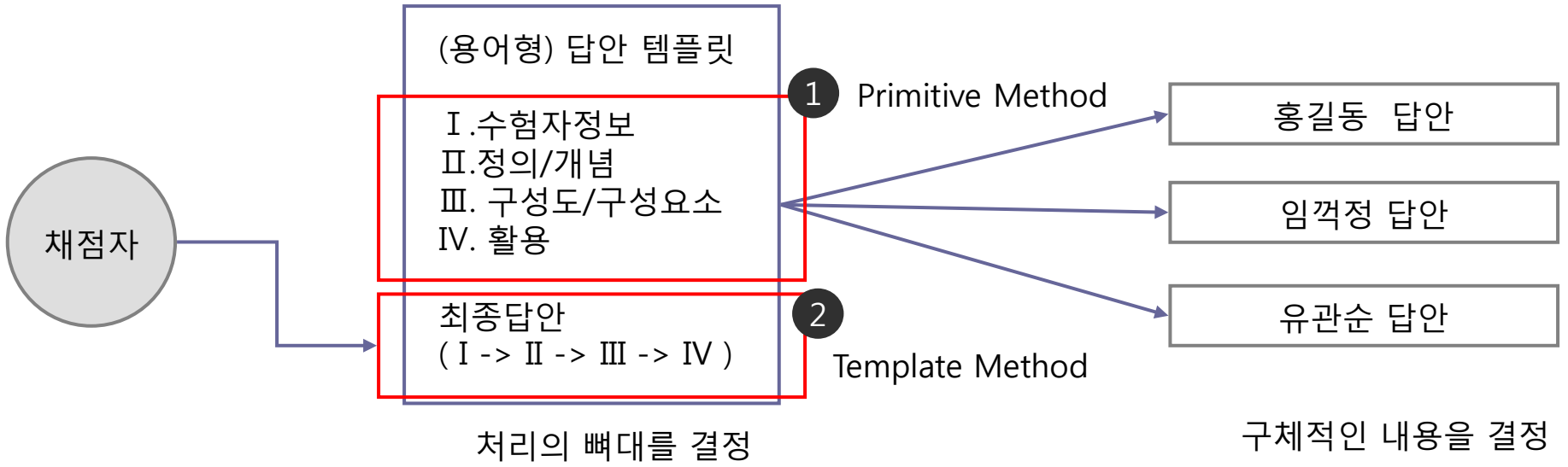
✓Flyweight : 하나의 구문 트리 내에 여러 개의 터미널 기호를 공유하기 위해서는 Flyweight 패턴을 적용할 수 있음

✓Iterator : 인터프리터는 Iterator를 이용하여 자신의 구조를 순회함

Material from: - "Design Pattern",

3. Template Method

기술사 답안지 작성



의도 (Intent)

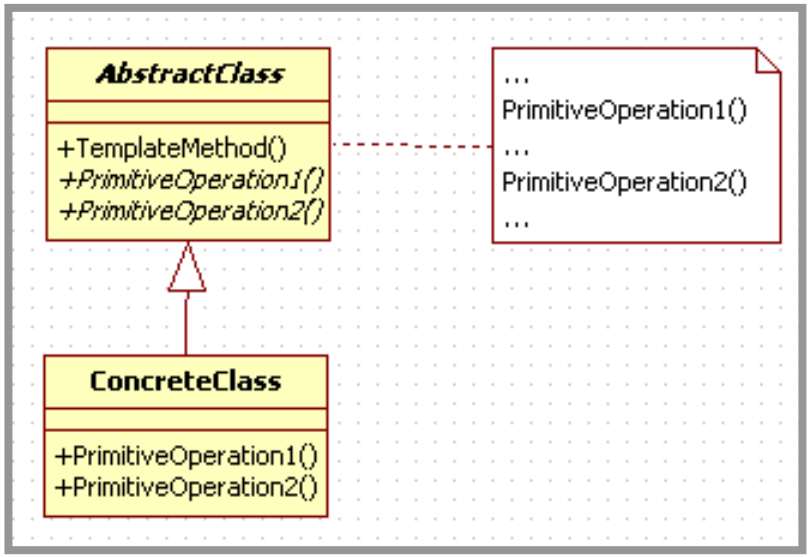
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm¡¯s structure

오퍼레이션에 알고리즘의 기본 골격 구조를 정의하고 구체적인 단계는 서브클래스에 위임한다. 템플릿 메소드는 전체적인 알고리즘 구조를 변경하지 않고 서브클래스에서 세부 처리 단계들을 재정의할 수 있게 해준다.

Material from: - "Design Pattern",

3. Template Method (계속)

구조와 참여객체



▪AbstractClass :

- 서브클래스들이 반드시 구현해야 하는 알고리즘 처리 단계 내의 기본 오퍼레이션 정의

-알고리즘의 기본 골격구조를 정의하여 템플릿 메소드 구현

- 템플릿 메소드에는 다른 객체에 의해 정의된 오퍼레이션 뿐만 아니라 인터페이스로 정의된 기본 오퍼레이션을 호출하도록 구현

▪ConcreteClass :

- 서브클래스마다 기본 오퍼레이션을 해당 클래스의 성격에 맞게 서로 다르게 구현

적용과 활용

✓알고리즘이 변하지 않는 부분을 한 번 정의하고 다양해질 수 있는 부분을 서브 클래스에서 정의할 수 있도록 구현하고자 할 때

✓Refactoring : 기존 코드에서 공통행위와 고유행위를 구별한 후, 고유행위로 구성된 템플릿 메소드를 정의함으로써 리팩토링을 수행할 수 있음.

✓서브클래스의 확장 제어 : 템플릿 메소드가 어떤 특정 시점에 훅 오퍼레이션을 호출하도록 정의하면, 해당 시점에 확장이 이루짐

Material from: - "Design Pattern",

3. Template Method (계속)

장/단점

✓로직의 공통화 : 전반 로직에 버그 발생시 템플릿 메소드만 수정
✓상위,하위클래스간 긴밀한 연계관계를 이해해야 함.
✓하위클래스를 상위클래스와 동일시 함.

```
AbstractClass ac=new ConcreteClass();
ac.commonMethod();
```

관련패턴

✓**Factory Method :** Template Method 패턴을 인스턴스 생성에 응용한 전형적인 예가 Factory Methods 패턴임
✓**Strategy :** Template Method는 상속을 이용하여 임의의 한 알고리즘의 부분을 다양화하지만,
Strategy 패턴은 위임을 사용하여 알고리즘 전체를 다양화시킨다.

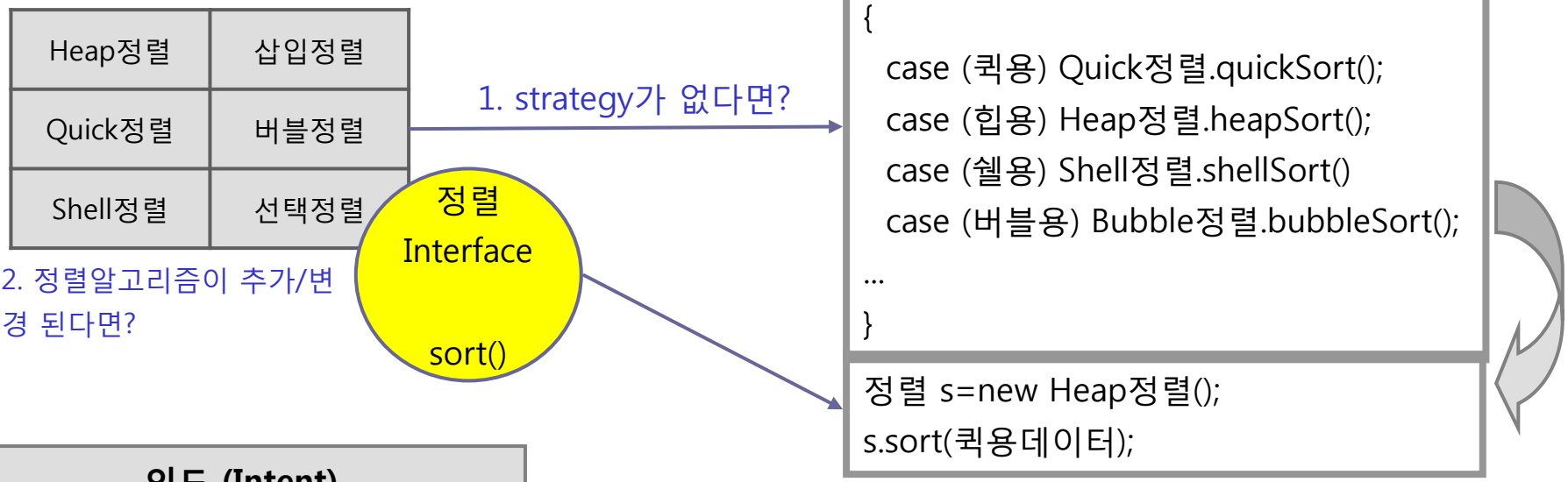
※ 어느 레벨에서 처리를 어떻게 분배할지,
어떤 처리를 상위 클래스에 두고 어떤 처리를 하위 클래스에 둘 것인지 정해진 메뉴얼은 없음.
이것은 프로그램 설계자의 몫.

Material from: - "Design Pattern",

4. Strategy

- **strategy = 전략, 작전, 방책 = Algorithm**
- **알고리즘을 객체화하여 같은 문제에 다양한 알고리즘을 적용할 수 있도록 도와주는 패턴**

알고리즘의 동적 교체



의도 (Intent)

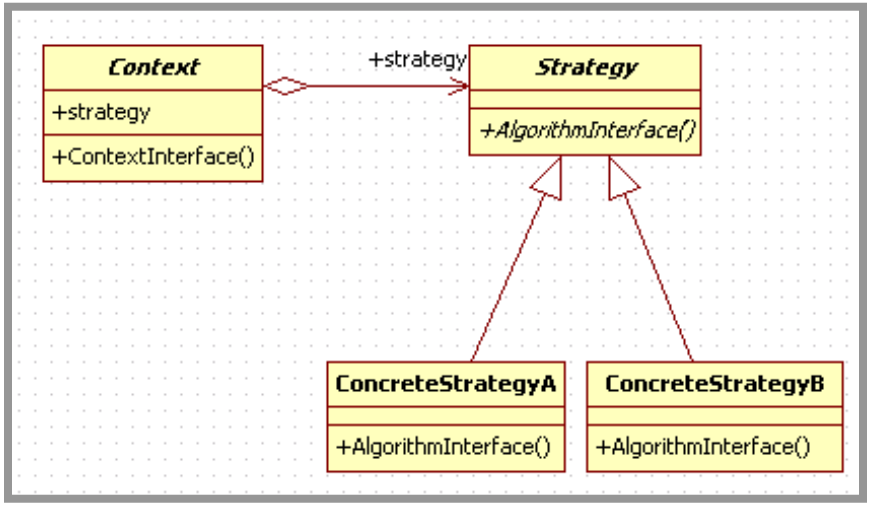
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it

다양한 알고리즘이 존재하면 이들 각각을 클래스로 캡슐화하여 대체가 가능하도록 함. 스트레터지 패턴을 사용하면 알고리즘을 사용하는 클라이언트와 독립적으로 다양한 알고리즘으로 변경 가능함.

Material from: - "Design Pattern",

4. Strategy (계속)

구조와 참여객체



▪**Strategy :**
제공하는 모든 알고리즘에 대한 공통의 오퍼레이션들을 인터페이스로 정의

▪**ConcreteStrategy :**
Strategy 인터페이스를 실제 알고리즘으로 구현

▪**Context :**
- ConcreteStrategy 클래스에 정의한 인터페이스를 통해서 실제 알고리즘을 사용
- Strategy가 데이터에 접근하도록 인터페이스를 정의
- ConcreteStrategy 객체를 설정
- Strategy 객체에 대한 참조자를 유지관리

적용과 활용

✓행위들이 조금씩 다를 뿐 개념적으로 관련된 많은 클래스들이 존재하는 경우
- 개념에 해당하는 클래스는 하나만 정의, 서로 다른 행위들은 별도의 클래스로 만듦

✓알고리즘의 변형이 필요한 경우

✓사용자가 모르고 있는 데이터를 사용해야 하는 알고리즘이 있을 경우

✓많은 행위를 정의하기 위해 클래스 안에 복잡한 다중 조건문을 사용해야 하는 경우

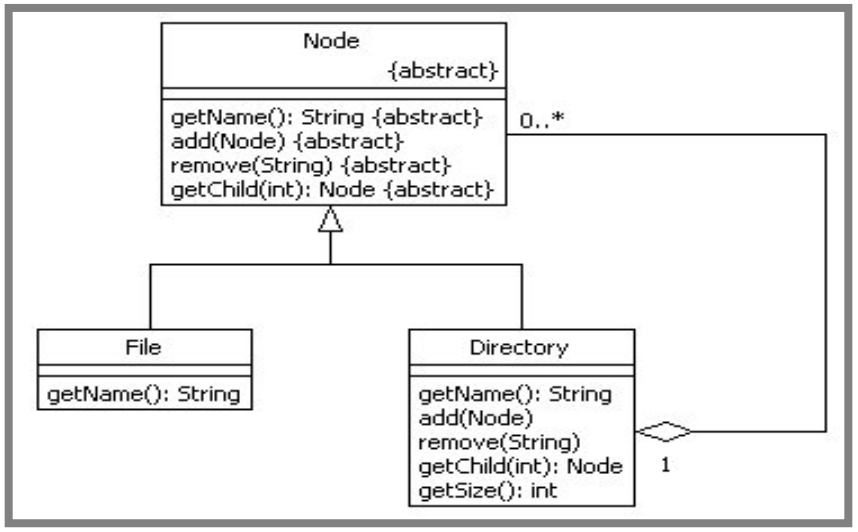
✓한쪽 알고리즘을 다른 쪽 알고리즘의 검산용으로도 사용 가능

Material from: - "Design Pattern",

5. Visitor

기능 확장에 대한 가정

Composite 패턴에서 제시한 예제를 예로 들자.



[요구1] getName() 메소드를 호출 시, 디렉토리 인경우는 "/"를 포함하여 리턴
[해결안1] File, Directory의 getName()을 다르게 구현함.
→ 소스 재컴파일 필요!

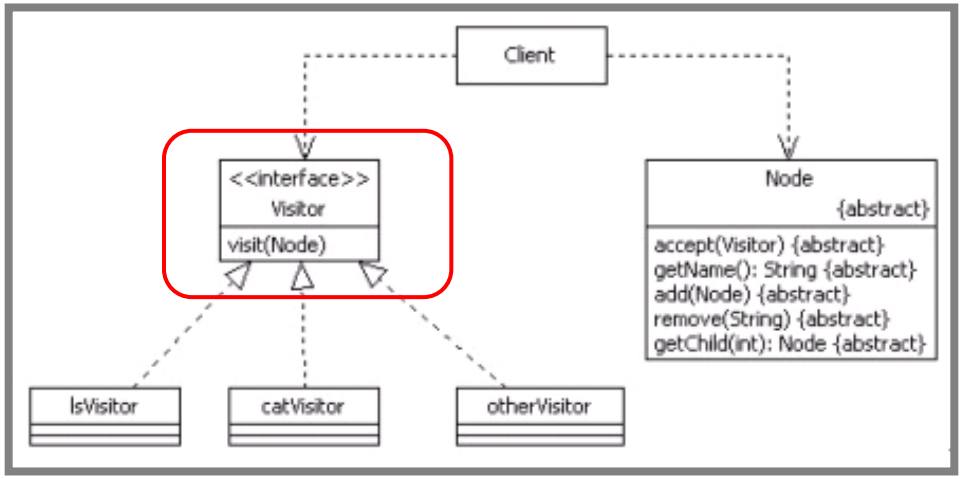
[요구2] Node 에 새로운 메소드 추가
[고민2] 새로운 메소드를 File, Direcotry 에 따로 추가할 것인가, Node 추상클래스에 추가할 것인가?
추가되는 기능을 위한 인터페이스 operation의 증가로 전체 파일 시스템의 encapsulation과 abstraction이 저하되고, 복잡해짐.

Node를 위한 인터페이스의 operation은 가장 기본적인 최소한의 것만을 유지하면서 node에 새로운 operation을 추가할 수 있는 방법 모색이 필요함.
→ Visitor 패턴이 그 해결책이 될 수 있음.

Material from: - "Design Pattern",

5. Visitor (계속)

Visitor 패턴이 적용된 예시



데이터 구조와 처리를 분리
Double Dispatch

Visitor.visit(Element);
Element.accept(Visitor);

✓이런 구조로 디자인하면, 클라이언트는 Node를 수정하지 않고도 Node에 대한 새로운 operation을 무한히 만들어 사용할 수 있게 된다.
✓기능을 확장할 때마다 Element 클래스의 메소드를 추가하는 수정작업을 하지 않고,
Element 클래스의 메소드를 가진 클래스를 추가하여 기능을 확장할 수 있음

의도 (Intent)

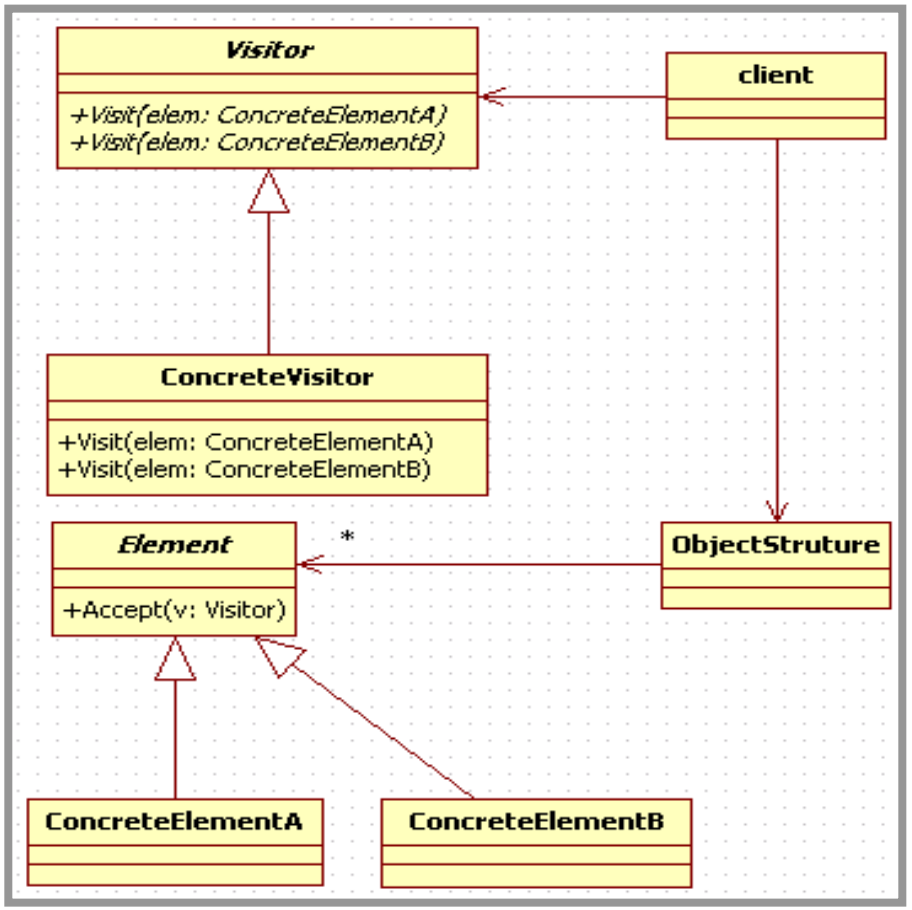
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor 패턴은 객체 구조체의 element에 연산을 가하는 새로운 operation을 해당 element의 클래스에 수정을 가하지 않은 상태로 추가할 수 있게 해 준다.

Material from: - "Design Pattern",

5. Visitor (계속)

구조와 참여객체



▪Visitor :
- ConcreteElement 클래스에 대한 Visit 오퍼레이션을 선언
- 오퍼레이션 이름과 signature는 어떤 visitor에게 Visit()요청을 보내야 할지를 결정해줌

▪ConcreteVisitor :
- Visitor에 의해 선언된 오퍼레이션을 구현
- ConcreteVisitor는 자신의 local 상태를 저장하고 있으며 알고리즘을 위한 context를 제공함

▪Element :
Argument로 Visitor 클래스를 받아들이는 accept() 오퍼레이션을 정의

▪ConcreteElement :
Argument로 Visitor 클래스를 받아들이는 accept() 오퍼레이션을 구현

▪ObjectStructure :
- 요소들을 나열
- 방문자들로 하여금 이들 요소에 접근하게 하는 인터페이스를 제공
- Composite 패턴의 복합객체, 리스트나 집합과 같은 컬렉션일 수 있음

Material from: - "Design Pattern",

5. Visitor (계속)

적용과 활용

✓다양한 인터페이스를 가진 객체 여러 개에 대해 그들의 자료형에 따라 각각의 작업을 수행하고자 할 경우
✓객체 여러 개에 대해 다양한 작업을 수행하고 싶은데, 각각의 작업들간에는 연관성이 적고 개발 객체마다 원하는 작업을 멤버함수로 추가 정의하는 것은 지저분해서 피하고 싶을 경우
✓작업 대상이 되는 클래스 구조는 확장될 가능성이 없는 대신 수행할 작업 항목은 계속해서 추가, 확장될 소지가 많을 경우

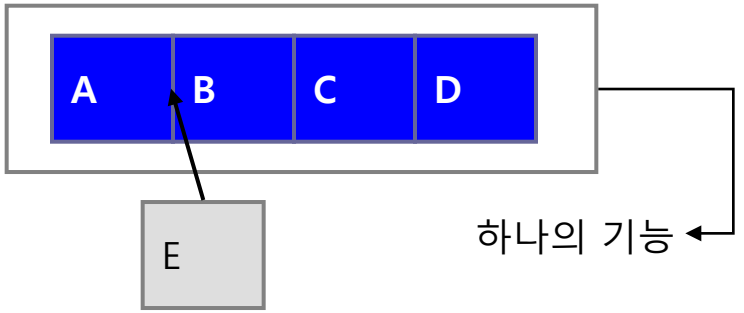
장 / 단점

✓객체 여러 개로 구성된 자료구조에 대해 다양한 작업을 쉽고, 편하게 추가 가능
✓작업 대상과 작업 항목을 분리시켜 줌
✓객체 여러 개에 대해 작업을 수행하면서 각 객체들의 특정 상태 값을 누적, 관리하게 만들 수 있음
✓작업 대상 추가, 확장이 어려움 (OCP : The Open-Closed Principle)
- ConcreteVisitor 추가는 간단, 그러나 ConcreteElement 역할 추가는 곤란
✓작업 항목 객체가 작업 수행 시 작업 대상 객체의 데이터를 참조하는 구조이기 때문에 Visitor 및 그 하위 클래스와 Element 및 그 하위 클래스간 결합도를 증가시킴
- 방문자는 데이터 구조에서 필요한 정보를 취득해서 동작하므로, 필요 정보를 얻을 수 있도록 해야 하지만,불필요한 정보까지 공개하면 미래의 데이터 구조를 개량하기 어려울 수 있음.

Material from: - "Design Pattern",

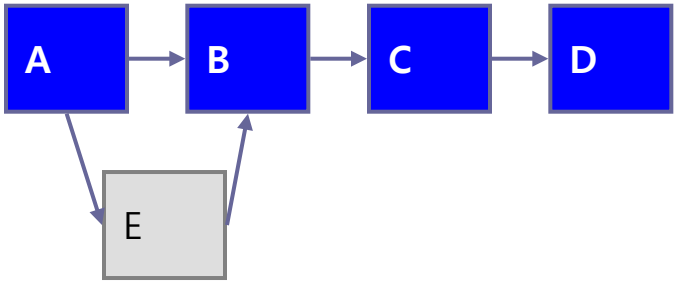
6. Chain of Responsibility

모듈화의 효과



각 모듈간 결합도가 높음

유연성, 독립성



각 모듈간 결합도가 낮음

자신의 고유영역에 해당하는 역할만 처리하고,
그 외의 것은 다른 역할자에게 넘김으로써,
고유 내용 처리에 집중할 수 있음.

의도 (Intent)

Avoid coupling the sender of request to its receiver by giving more than one object a chance to handle the request.

Chain the receiving objects and pass the request along the chain until an object handles it

요청을 보내는 객체와 그것을 받아 처리하는 객체간 결합도를 없애기 위한 패턴으로,

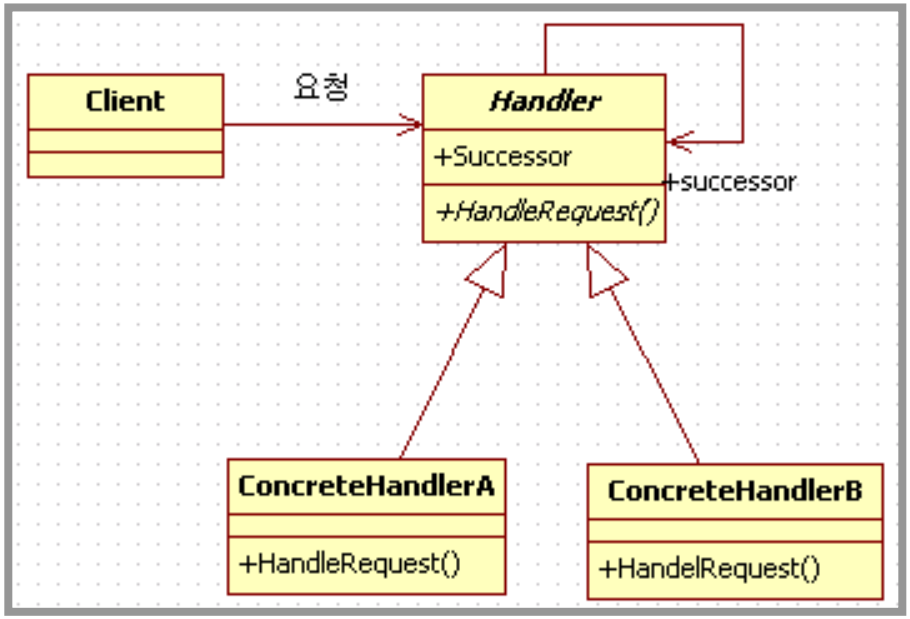
하나 이상의 객체에게 요청을 처리할 수 있는 기회를 줌으로써 결합도 제거한다.

요청을 받는 객체들을 연결하고, 객체가 전달받은 요청을 처리할 수 있을 때까지 그 연결고리를 따라서 요청을 전달한다.

Material from: - "Design Pattern",

6. Chain of Responsibility (계속)

구조와 참여객체



▪**Handler :**
- 요청을 처리하는 인터페이스를 정의

▪**ConcreteHandler :**
- 책임져야 할 요청처리를 구현
- 만일 ConcreteHandler가 요청처리가 가능하다면 처리하고,
  그렇지 않으면 연결고리에 연결된 다음 객체에게 요청을 포워딩 함

▪**Client :**
ConcreteHandler 객체에게 필요한 요청을 보냄

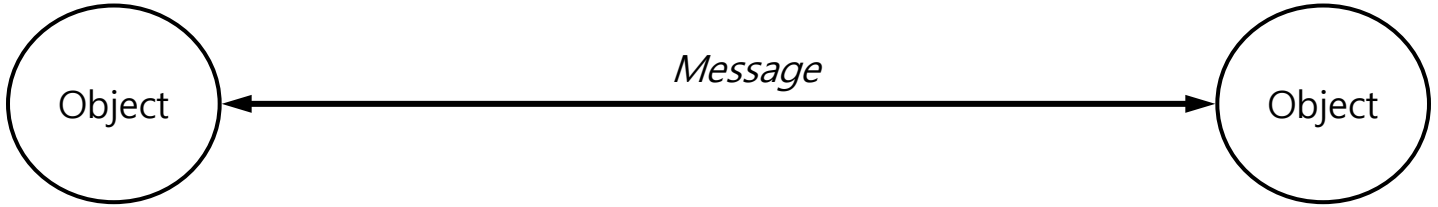
적용과 활용

✓하나 이상의 객체가 요청을 처리해야 하는 경우, 핸들러가 누가 선행자인지 모를 때, 다음 핸들러로 자동 결정
✓메세지를 받을 객체를 명시하지 않은 채 여러 객체 중 하나에게 처리를 요청하고 싶을 때
✓요청을 처리할 객체 집합을 동적으로 정의하고자 할 때

Material from: - "Design Pattern",

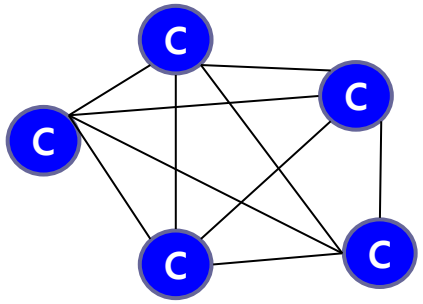
7. Mediator

객체지향의 의존성



- 객체와 객체는 인터페이스를 통해서 메세지를 주고 받으면서 특정한 관계(relationship)를 맺게 된다.
- 이런 관계는 객체간의 의존성을 지니게 하고, 이렇게 함으로써 전체 시스템의 모습을 갖추게 된다.
- 그러나, 대부분은 이런 객체 사이의 의존성을 최소화하는 것이 안정성을 증대시키고, 관리를 쉽게 한다.

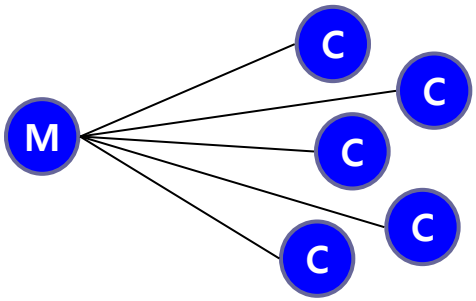
Before



- 오브젝트가 상호 관련되어 서로가 서로를 컨트롤 하는 상태
- colleague들이 많아져서 서로 통신해야 하는 경우는 그 통신라인이 기하급수적으로 증가함.

✓의존성 최소화
✓행위의 지역화(Localization)

After

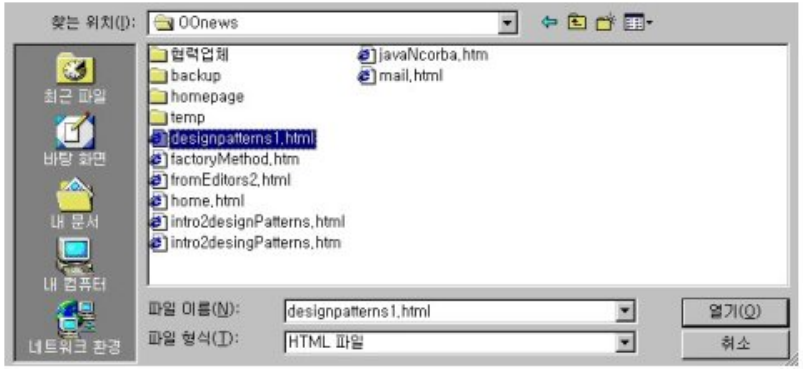


- 복잡하게 얽혀 있는 오브젝트간 상호 통신을 중지시키고, Mediator에게 정보를 집중,처리
- Colleague들은 Mediator하고만 통신, 컨트롤 로직은 Mediator 안에만 기술

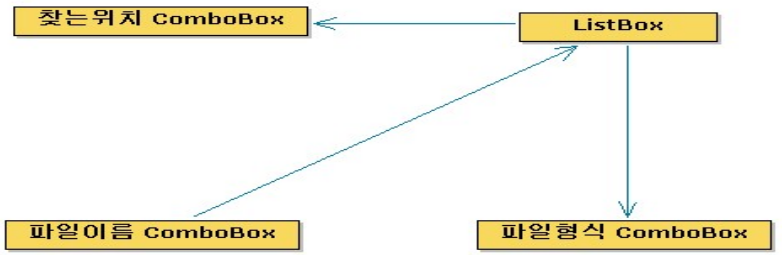
Material from: - "Design Pattern",

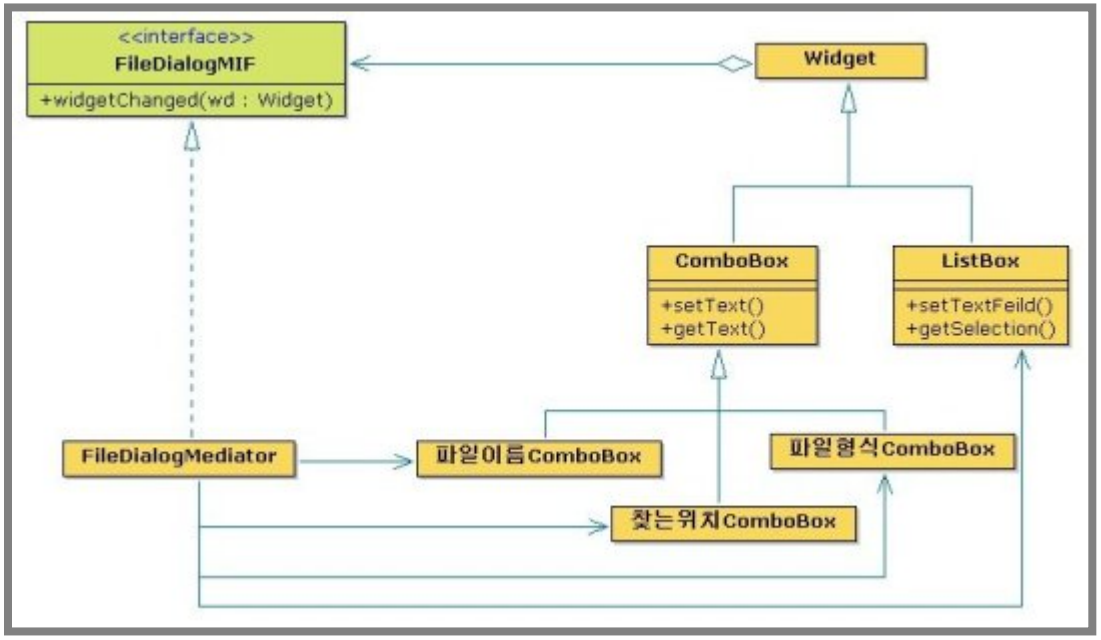
7. Mediator (계속)

다이얼로그 박스 예시



의존관계 살펴보기





Mediator 패턴을 적용
한 Design

Material from: - "Design Pattern",

7. Mediator (계속)

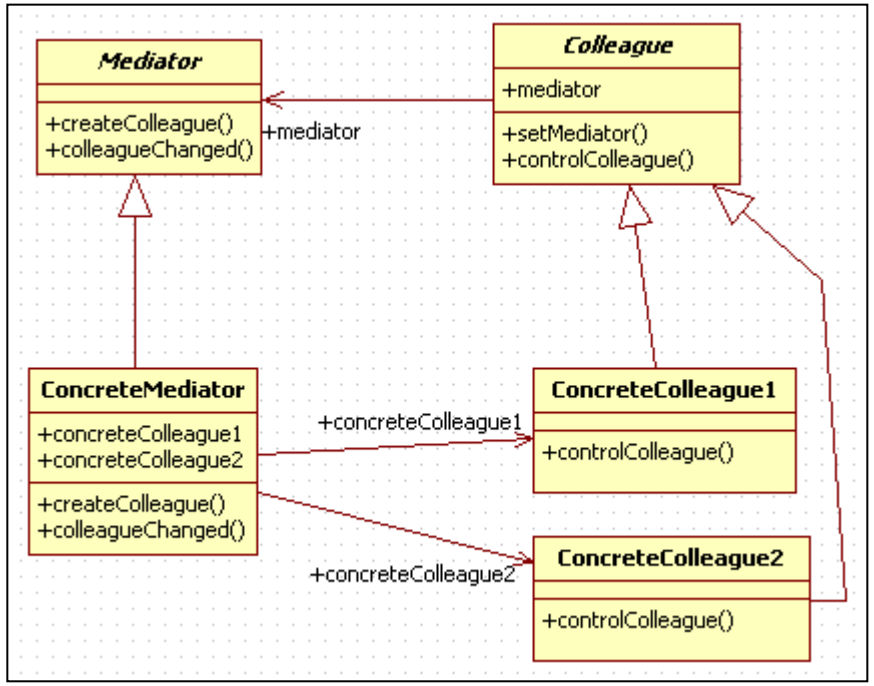
의도 (Intent)

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

객체들 간의 상호작용을 캡슐화하여 하나의 객체 안에 정의함. 메디에이터 패턴은 낮은 결합도를 촉진하는데, 객체간에 서로 노골적으로 참조하는 것을 못하게 함으로써 가능하다.

그리하여 각 객체간의 의존성을 줄여서 그들간의 다양한 상호 작용이 가능하게 해 준다.

구조와 참여객체



▪Mediator :
Colleague 역할과 통신하여 조정하기 위한 인터페이스를 정의

▪Colleague :
다른 Colleague와 통신할 일이 발생할 때마다 해당 Mediator와 통신함.

▪ConcreteMediator :
Colleague 객체들을 조정함으로써 상호협동적인 행동을 구체화하는 역할

✓ConcreteColleague역할은 재사용하기 쉽지만,
ConcreteMediator 는 재사용이 어려움
✓GUI 어플리케이션에 효과적인 패턴

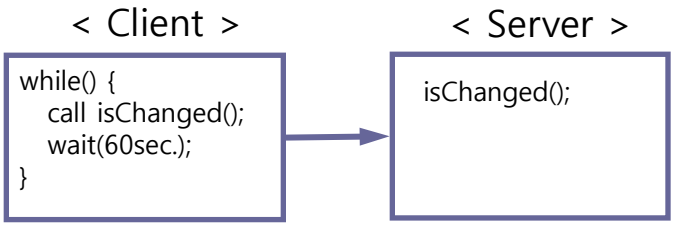
Material from: - "Design Pattern",

8. Observer

상태변화에 따른 처리방법

1 Polling

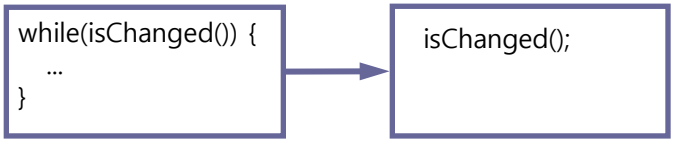
▪어떤 클라이언트가 (보통은 루프를 이용해서) 지속적으로 다른 객체의 상태 변화를 알아보려는 목적에서 그 객체가 제공하는 메소드를 지속적으로 호출하는 방법



2 Synchronous (동기식)

▪서버에서 제공하는 인터페이스를 클라이언트에서 동기적으로 호출하는 방법

▪이 경우에는 이 인터페이스를 호출한 클라이언트는 서버에서 상태 변화에 대한 공지를 해주기 전까지는 계속 해당 인터페이스의 호출에 묶여 있게 된다. 즉, 클라이언트는 서버에 상태변화가 감지되어 호출한 인터페이스가 리턴 되기 전까지는 다른 일을 할 수 없는 상태에 빠져 있게 된다.

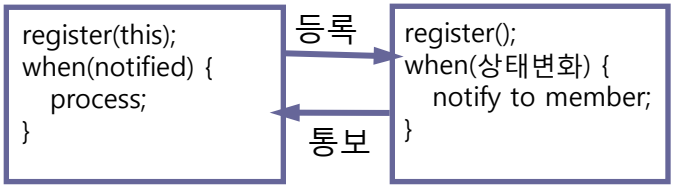


3 Asynchronous (비동기식)

▪ Callback : 서버에서 제공하는 인터페이스를 비동기적으로 호출함으로써 이루어지며, 이 방법은 callback 이라고 명칭함. 그 이유는 call을 했을 때, 지금 당장은 상태변화가 없으니 나중에 변화가 생기면, 나에게 알려달라고 나의 연락처를 남기는 과정이라고 해서 callback이라고 이름을 붙였다고 한다.

→ 서버에게 위임

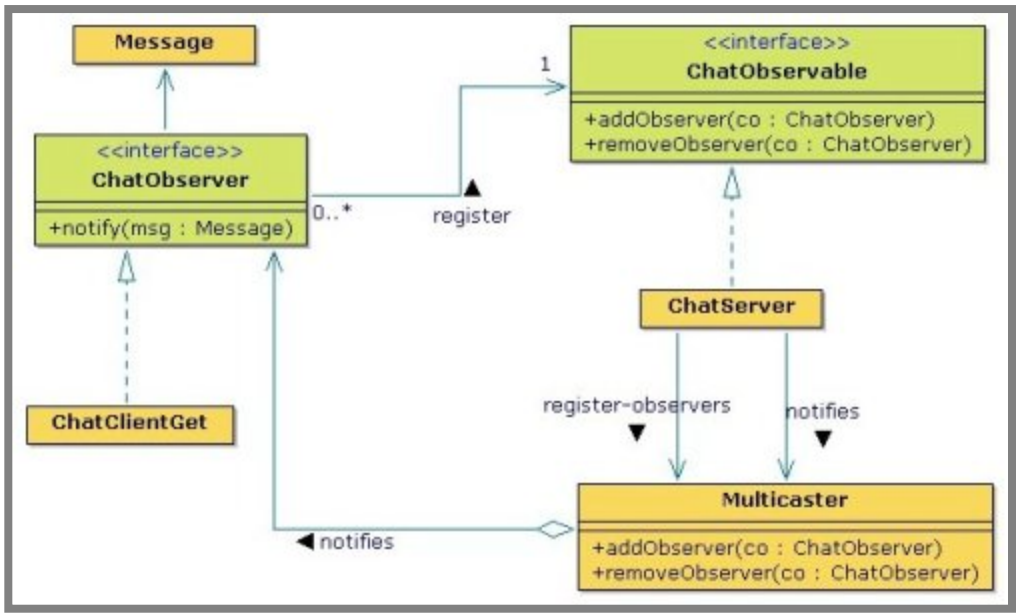
▪= Observer Pattern



Material from: - "Design Pattern",

8. Observer (계속)

채팅서비스 예시



- ChatObserver : 채팅에 참여하는 Clients
- Multicaster : ChatObservable 를 implement. 채팅에 참여자에 대한 리스트 관리, 메시지 도착 시 채팅참여자에게 전달
- publish-subscribe 패턴
- MVC 모델에서 M : Subject / V : Observer 의 역할에 대응

의도 (Intent)

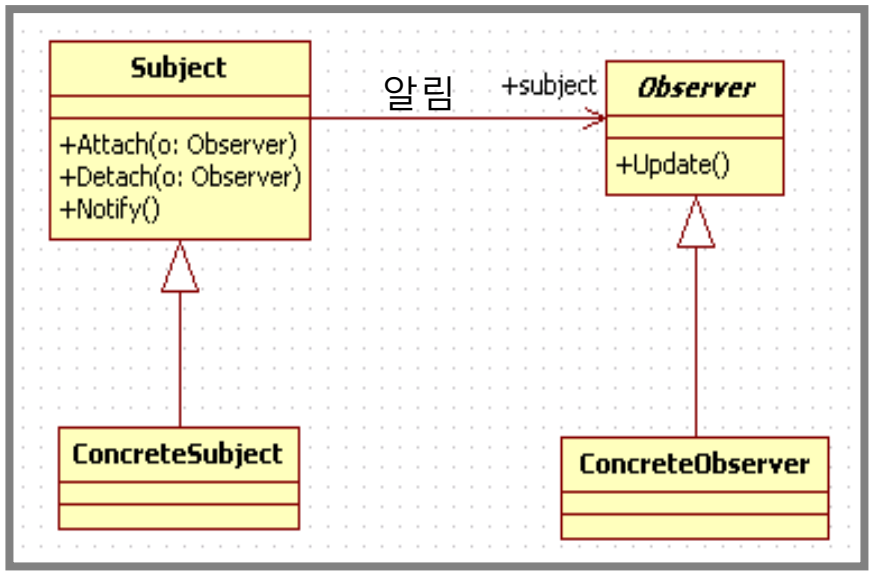
Define a one-to-many dependency between objects so that when one object changes states, all its dependents are notified and updated automatically.

객체들 사이에 일대다 의존성을 부여할 때 사용될 수 있는 패턴으로써 하나의 객체 상태가 바뀌면 그 객체에 의존하고 있는 모든 객체가 그 사실을 통보 받고 그에 해당하는 행위를 할 수 있게 한다.

Material from: - "Design Pattern",

8. Observer (계속)

구조와 참여객체



▪Subject :
관찰되는 대상, 관찰자인 Observer의 역할을 등록하는 메소드와 삭제하는 메소드를 가지고 있음

▪Observer :
Subject 역할자로부터 상태가 변경되었음을 통지 받는 역할

▪ConcreteSubject :
관찰되는 대상을 구체적으로 구현하는 역할

▪ConcreteObserver :
상태가 변경 통지를 받았을 때, 취해야 할 구체적인 변화를 구현하는 역할

관련패턴

Mediator 패턴 vs. Observer 패턴

✓Mediator 패턴 : 상태변화를 알리지만, colleague 역할의 조정이라는 목적으로 동작하고 있는 Mediator 패턴의 일부

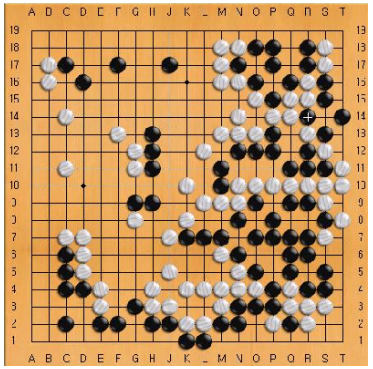
✓Observer 패턴 : Subject 역할의 상태변화를 Observer 역할에게 알리는 일, 알려서 동기화를 이루는 일에 주안점을 둠

Material from: - "Design Pattern",

9. Memento

- **Also Known as TOKEN**
- **undo , redo , history, snapshot**
- **어떤 시점의 인스턴스 상태를 확실하게 기록/저장해 두어 나중에 인스턴스를 그 시점의 상태로 되돌리도록 하는 패턴**

바둑게임의 '무르기' 기능



2차원 배열을 활용하여 배열에 저장되는 값이 몇 번째 수에 돌이 놓였는지를 의미하는 값이라고 가정할 경우,
다음의 몇 가지 **문제점** 발생
-2차원 배열은 현재 바둑판에 놓여진 돌들에 한해서만 정보를 가지고 있음. 즉 상대편에 잡혀서 들어내어진 돌에 대한 정보를 가지고 있지 않으므로 들어내어진 돌에 대해서는 무르기 불가능
-흰 돌이나 검은 돌의 사석 수나 패의 위치 등의 정보는 저장하지 않기 때문에 이전 상태로 되돌리기 불가능

원인	바둑판 객체가 무르기를 정상적으로 수행할 수 있을 만큼의 충분한 정보를 가지고 있지 않기 때문
해결	바둑판 객체가 가지고 있는 데이터 멤버를 하나의 클래스로 정의하고 이 클래스를 이용하여 바둑판 상태를 시점 별로 저장, 관리하는 리스트를 만들어 사용하도록 함

의도 (Intent)

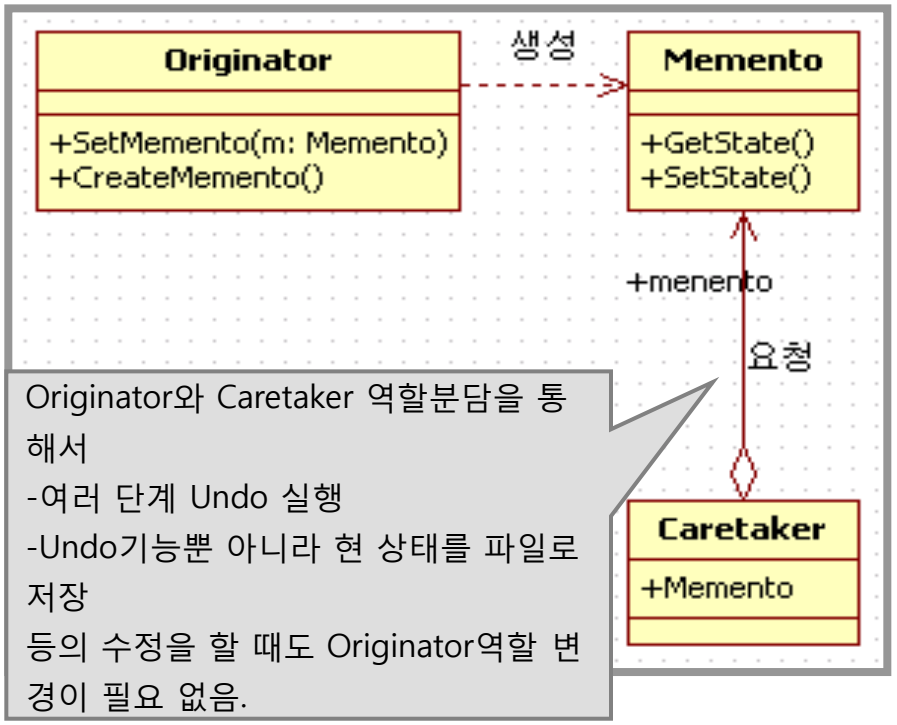
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later

캡슐화를 위배하지 않으면서, 객체의 내부 상태를 포착하여 표면화시킨다. 그러면 그 객체는 후에 지금의 상태로 복구될 수 있다.

Material from: - "Design Pattern",

9. Memento (계속)

구조와 참여객체



▪Memento :

- Originator의 내부상태를 필요한 만큼 저장함
- Originator를 위한 wide interface와 Caretaker를 위한 narrow interface를 정의

▪Originator :

- 메멘토를 생성하여 현재 객체의 상태를 저장하고 내부 상태를 복구함
- 오직 Originator 클래스만이 메멘토 내부 상태에 접근 권한을 가지고, 메멘토의 다양한 인터페이스를 사용

▪Caretaker :

- 메멘토의 보관을 책임지기는 하지만, 메멘토에 정의된 모든 인터페이스를 보지 못하고, 단지 메멘토를 다른 객체에 전달
- caretaker 입장에서는 메멘토는 블랙박스로서 단순히 만들어 준 메멘토를 통째로 저장하고 전달할 뿐임.

Wide Interface	메멘토의 내부상태를 모두 드러냄. Originator만 사용함.
Narrow Interface	한계가 있음, 내부 상태가 외부로 공개되는 것을 방지, Caretaker가 사용함.

→ 두 종류의 인터페이스를 구별해서 사용하면 오브젝트의 캡슐화가 파괴되는 것을 방지

Material from: - "Design Pattern",

9. Memento (계속)

적용 / 활용

✓각 시점에서 객체 상태를 저장한 후 나중에 이 상태로 복구해야 할 때
✓객체의 상태를 직접 접근하는 것이 해당 객체의 구현 상 너무 복잡하거나 정보 은닉을 깨드릴 경우

장/단점

✓Client가 Originator 객체의 내부 상태를 저장, 관리해야 할 경우 이를 직접 접근하는 것을 막아줌
✓Memento 패턴을 적용하지 않을 경우 Originator 객체는 자신의 상태 정보를 Client가 원하는 시점마다 관리하고 있어야 하지만 Memento 패턴을 적용할 경우 client가 스스로 자신이 원하는 시점의 Originator 객체 상태 정보를 Memento 객체를 이용해서 저장, 관리할 수 있으므로 Originator 객체의 내부 구현이 간단해진다.

✓Memento 객체의 생성은 Originator 객체의 내부 상태를 모두 복사해야 하는 비용을 포함하고 있다. 따라서 Memento 객체의 생성이 자주 일어나거나 Originator 객체의 내부 상태 정보가 많은 경우에는 부적절할 수도 있다.
✓Memento 객체의 저장이나 관리 또는 삭제는 client가 알아서 수행한다.

Material from: - "Design Pattern",

10. State : 상태를 객체화 하여 객체 스스로 상태 변화에 따라 다양한 행위를 할 수 있도록 지원하는 패턴

사무실 상태 표시

State 패턴을 사용하지 않았을 때

어떤 사무실의 상태를 나타낸다고 가정하자. 만약 비서도 없고 미리 준비된 알림판도 없다면 사무실을 비우거나 회의중임을 알릴 때마다 메시지(식사중, 외근중, 회의중 등)를 문에 써 붙여야 할 것이다.

State 패턴을 사용했을 때

오른쪽 그림은 왼쪽 그림에서와 같은 불편함 뿐만 아니라 불필요한 낭비도 해소할 수 있는 개선된 모습을 보여주며 이는 상태를 나타낼 수 있는 객체를 미리 생성해 두고 상태가 바뀔 때마다 원하는 객체로 바꿔주는 State Pattern을 잘 표현해 준다.

State패턴 사용하지 않았을 때 :

주간, 야간의 상태가 각 메소드 안의 if 문에 등장/ 그리고 각 메소드 안에서 현재 상태를 조사하고 있음.

State패턴을 사용하면 :

상태를 클래스로 표현하므로 그 안의 메소드에는 상태 검사를 위한 if문이 등장하지 않음.

Material from: - "Design Pattern",

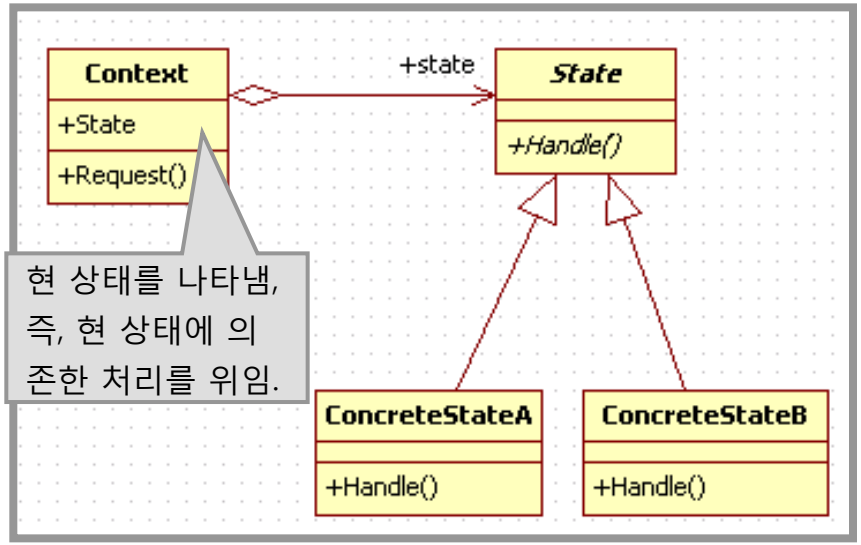
10. State (계속)

의도 (Intent)

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class

객체 자신이 객체의 내부상태 변화에 따라 행위를 변경하도록 함. 객체가 클래스를 바꾸는 것처럼 보일 수 있다.

구조와 참여객체



▪Context :

- ConcreteState 서브클래스의 인스턴스를 관리

▪State :

- Context가 갖는 각 상태별로 필요한 행위를 캡슐화하여 인터페이스로 정의

▪ConcreteState :

- 각 서브클래스들은 Context의 상태에 따라 처리되어야 할 실제 행위를 구현

상태전환 관리는 누가 해야 하는가?

- ConcreteState 하는 경우

장점) 다른 상태로 전환하는 것은 언제인가 하는 정보가 하나의 클래스 내에 정리되어 있음.

단점) ConcreteState 클래스간 서로 알아야 함. 즉...클래스 사이의 의존관계가 깊어짐

- Context가 하는 경우

장점)ConcreteState의 독립성이 높아져서 프로그램 전체의 예측이 좋아짐

단점)Context가 모든 ConcreteState를 알아야 함. 여기서 Mediator 패턴을 적용함도 고려해 볼만 함.

Material from: - "Design Pattern",

10. State (계속)

적용 / 활용

✓어떤 객체의 행위가 그 객체의 상태에 의존하고, 변경되는 상태에 따라 실행 시간(run-time)에 객체의 행위를 독립적으로 변경시켜야 할 경우
✓객체의 연산들이 내부 상태 값에 따라 여러 조건문으로 분기되어 처리되는 부분이 많을 경우
✓상태가 많을 때 State 패턴의 장점이 발휘됨.

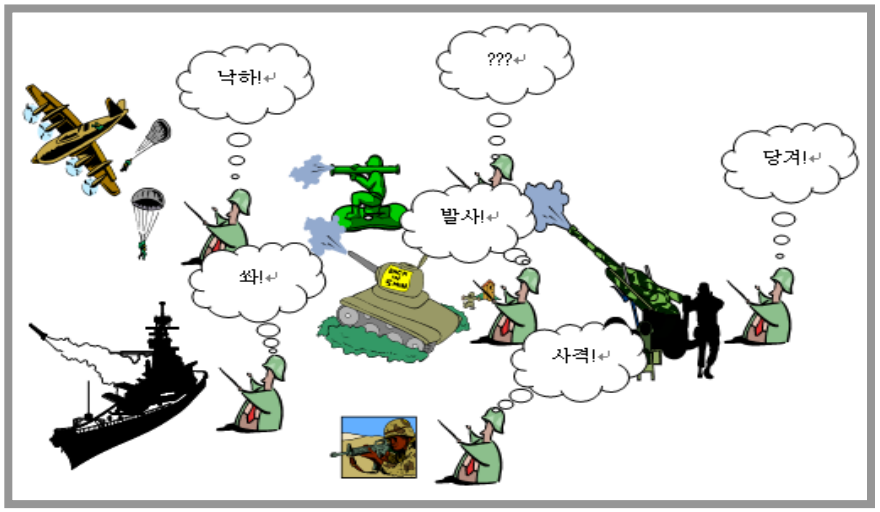
장 / 단점

✓수행할 행위를 결정하기 위해 객체 내부에서 상태 값을 비교하는 문장을 제거해줌
✓특정 상태와 관련된 행위들을 하나의 객체로 모아주는 역할을 함
✓객체의 상태 전환이 명백하게 드러나게 함
✓상태 정보가 일관성을 가지게 만들어 줌
✓상태를 나타내는 각 클래스들이 내부적으로 데이터 멤버를 가지지 않는다면 이들은 공유될 수 있음

Material from: - "Design Pattern",

11. Command : 클라이언트의 요청을 객체화하여 처리하는 패턴

지휘관의 전투지휘



전투가 일어나고 있는 전장을 생각해 보자. 지휘관이 전투 지휘를 위해 각각의 전투병과나 전투병에게 일일이 공격명령을 내린다면,
위 그림에서와 같이 소화병과에 사격명령을, 전차에 발사명령 등을 각각 통제한다면 전투의 성과를 효과적으로 얻을 수 있을까?

전투현장에서 각각의 전투대원은 지휘관의 일관된 명령에 대해 각자의 임무에 맞는 행동을 취하게 된다.
위 그림에서와 같이 지휘관의 사격개시 명령에 대해 소총수는 소화기 사격을, 포병을 포격을, 전차병은 또한 직사포 사격을 하게 되는 것이다.

✓이 패턴을 사용하면 오퍼레이션을 수행하는 객체와 오퍼레이션을 수행하는 방법을 구현하는 객체로 분리시켜 Command 자체도 클래스로 다른 객체와 같은 방식으로 조작 / 확장가능
✓결과적으로 사용자 인터페이스 작성 시 유통성을 부여할 수 있음.

Material from: - "Design Pattern",

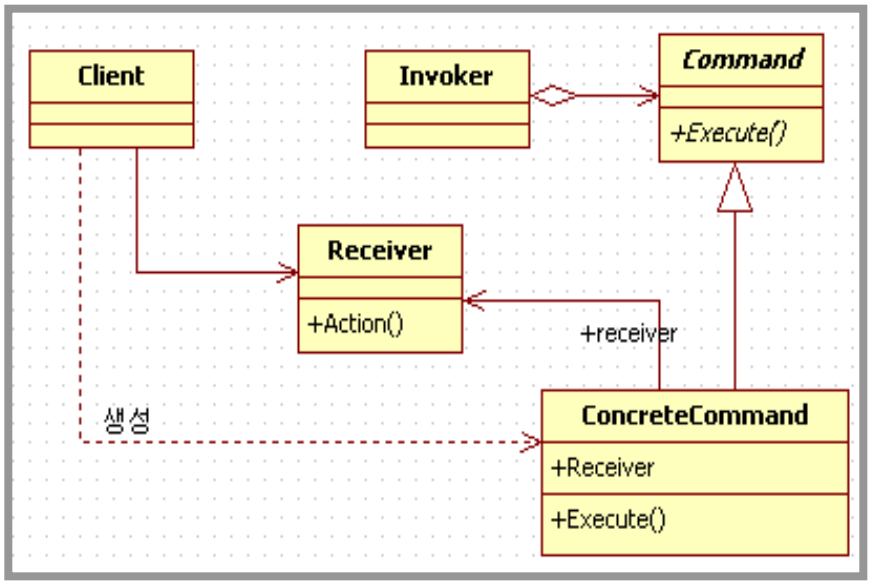
11. Command (계속)

의도 (Intent)

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

요청을 객체화한다. 그리하면 서로 다른 요청들을 가진 클라이언트를 매개변수화하고, 요청을 큐에 저장하거나 기록하고, 되돌리기 동작을 지원가능 하게 해준다.

구조와 참여객체



▪Command :
오퍼레이션 수행에 필요한 인터페이스 선언
▪ConcreteCommand :
- Receiver 객체와 action간 바인딩을 정의
- 처리객체에 정의된 오퍼레이션을 호출하도록 execute()를 구현
▪Client :
ConcreteCommand 객체를 생성하고 그것의 receiver를 세팅함
▪Invoker :
요청(request)을 처리할 명령어를 요청(ask)
▪Receiver :
요청을 처리하는데 관련된 오퍼레이션 수행방법을 알고 있음.

Material from: - "Design Pattern",

11. Command (계속)

적용 / 활용

✓근본적인 이점은 Invoker와 Receiver 사이의 연결이 느슨해졌다는 점에 있다. 더욱이 그 느슨한 연결이 Command 객체로 캡슐화 추상화되어 여러 가지 부가 효과들을 얻을 수 있음.
✓캡슐화된 Command를 logging 가능
✓캡슐화된 Command를 store, restore 가능
✓Oracle instance 처럼 command 를 caching 했다가 문제발생시 checkpoint 이후의 작업을 자동으로 복구할 수 있다.
✓캡슐화된 Command 여러 개를 묶어 MacroCommand 생성 가능
✓작업 수행을 요청한 시점과 실제 작업을 수행하는 시점을 달리하고 싶을 때 유용
- 커멘트 패턴은 직접 수행할 작업에 대한 함수를 불러주는 형태가 아니라 수행할 작업에 대한 객체를 먼저 생성하고 나중에 작업 수행을 요청하는 형태이므로, 수행할 작업을 큐에 쌓아두었다가 작업 수행이 필요한 시점에 작업을 수행하는 것이 가능

장/단점

✓작업 수행을 요청하는 객체와 실제 작업을 수행하는 객체를 분리시켜주므로 시스템의 결합도가 낮아질 수 있으며, 두 객체가 독립적으로 변경 가능함
✓Command 및 그 하위 클래스는 기존 클래스와 무관하게 확장이 가능하며, 확장된 클래스들은 client의 별다른 수정없이 사용 가능

Material from: - "Design Pattern",

Q & A