된다.

블록체인 기술은 Peer-to-peer 아키텍처를 기반으로 하고 있다. 보안을 해결하기 위하여 각 노드가 상대방의 노드를 비교하여 감시하고 일관성을 유지하게 된다.

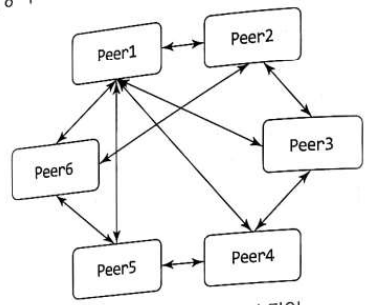


그림 7.14 Peer-to-Peer 스타일

■ 장점

- 전담하는 애플리케이션이나 서버가 없음.
- 규모 확장성과 신뢰성이 개선됨. 컴포넌트에 고장이 있더라도 전체 시스템은 가동됨.

■ 단점

- 보안 취약할 수 있음.
- 중앙에서 제어할 수 없음.
- 공유된 자원으로 성능이 떨어질 수 있음.

표 7.1 아키텍처 스타일의 비교

스타일	장점	단점
클라이언트 서버	클라이언트가 요청하는 서비스를 종합적으로 모델링하기 좋음.	서비스 요청이 서버에서 별도의 스레드로 처리됨. 프로세스 사이의 커뮤니케이션이 오버헤드가 될 수 있음. 클라이언트들이 다른 표현을 요구하기 때문.
계층형	하위층이 여러 상위층에 의하여 사용됨. 각 층을 정확히 정의하면 각 층이 표준화될 수 있음. 다른 층에 영향을 주지 않고 수정을 층 안에 국한시킬 수 있음.	광범위하게 적용하기 어려움(이웃 층과의 커뮤니케이션이 제한적). 특정한 층이 상황에 따라 통과될 수도 있음.
이벤트 기반	이벤트 생산자, 소비자 연결을 쉽게 하고 추가할 수 있음. 분산 응용에 효과적으로 적용 가능.	상태에 따른 복잡하고 정교한 제어가 필요함. 큰 규모에 적용하기 어려움.
MVC	동일 모델에 여러 가지 뷰를 런타임에 쉽게 연결하고 끊을 수 있음.	복잡도가 증가하고 이로 인하여 사용자 액션을 위하여 불필요한 수정이 증가될 수 있음.
파이프필터	병렬처리를 가능하게 함. 쉽게 필터 추가할 수 있고 시스템이 확장될 수 있음.	느린 필터에 의하여 시스템의 성능이 제한적. 필터 이동하면서 자료의 변환 오버헤드가 발생.
데이터 중심	새로운 응용을 쉽게 추가. 데이터 공간의 확장이 용이.	모든 부분이 영향을 받아 중심 데이터의 구조를 변경하는 것이 쉽지 않음.
Peer-to-peer	중앙집중을 피할 수 있다. 노드 하나의 고장으로 시스템이 멈추지 않는다. 자원과 컴퓨팅 파워의 확장성이 뛰어남.	서비스의 품질을 보장하기 어려움. 보안을 보장하기 어려움. 성능이 노드의 수에 좌우됨.

7.3 디자인 패턴

디자인 패턴은 아키텍처 설계 수준보다 낮은 수준의 설계 문제에 재사용 가능한 솔루션을 제공한다. 7.1절에서 언급한 것처럼 디자인은 여러 수준에서 이루어진다. 설계 문제와 목표는 설계 추상 수준마다 다르기 때문에 한 수준에 적합한 패턴이 다른 레벨의 문제에는 적합하지 않은 경우가 많다.

디자인 패턴과 아키텍처 스타일이 적용되는 문제 수준을 명확하게 보여주는 간단한 예를 들어보자. 요구 분석을 하였더니 '시스템은 자료를 갱신하기 위한 컨트롤과 함께 재고 자료를 표시해야 한다'라는 요구 사항이 있다고 가정하자.

요구 사항은 결국 자료를 이용하기 위하여 지시하고 결과를 표시하는 것이므로 아키텍처는 대화형 시스템이라는 결론을 내릴 수 있다. 대화형 응용 프로그램의 논리를 구성하는 데 적합한 아키텍처 스타일은 MVC 스타일이다.

MVC 스타일을 구현하기 위하여 뷰에 모델의 가장 최근 변경 사항이 반영되도록 하는 방법을 생각하다 보면 자연히 하위 디자인 수준의 문제에 접하게 된다. 즉 데이터를 가진 모델과 이용하는 뷰의 효과적인 설계 솔루션 모델을 찾게 되는데 이것이 디자인 패턴이 적용되는 수준이다.

디자인 패턴을 이해하면 주어진 문제에 대하여 두 가지 솔루션을 발견할 수 있다.

1. 옵서버 패턴에 따라 모델이 뷰의 상태 변경을 알리도록 할 수 있다.
2. 모델에 대한 변경이 컨트롤러에 의해 시작된다면 모델에 상태 변경이 있을 때 컨트롤러가 뷰에 알리도록 할 수 있다.

결론적으로 디자인 패턴이 적용되는 수준은 요구 사항을 고려하여 아키텍처를 확정한 후 아키텍처의 컴포넌트들을 구현하는 클래스의 역할과 동작이 결정된 후 발생하는 설계 이슈에 대하여 솔루션으로 적합한 디자인 패턴을 적용하고 이를 단계적으로 진화, 개선시킨다.

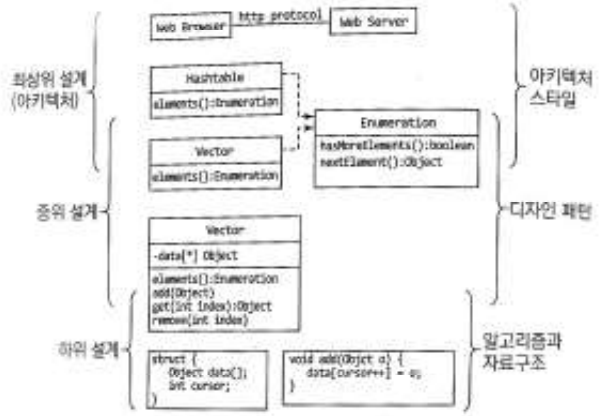


그림 7.15 디자인 패턴이 적용되는 설계 수준

7.3.1 디자인 패턴의 혜택

디자인 패턴을 알면 여러 가지 이로운 점이 많다.

1) **쉽게 재사용 가능** – 소프트웨어 개발에서 생산성을 높이는 가장 좋은 방법은 개발 검증된 것을 재사용하는 길이다. 소프트웨어 개발에 관련된 많은 이해 관계자들은 재사용의 필요성과 기회를 인정하지만 쉽게 재사용하기는 어려웠다. 디자인 패턴은 여러 응용 프로그램에서 디자인 솔루션을 재사용할 수 있는 방법을 제공한다.
2) **설계 작업이 쉬워짐** – 설계 이슈가 디자인 패턴에 존재하는 경우라면 높은 품질의 솔루션이 이미 제공된 것이다. 하지만 디자인 패턴을 적용하려면 특정 상황에 대하여 패턴을 특수화하는 방법을 알아야 한다.

3) **설계 관련 지식이 정리됨** – 소프트웨어에서 분석과 설계는 가장 까다로운 활동이며 가장 많은 경험을 필요로 한다. 디자인 패턴에 대한 학습은 개발자에게 제대로 된 설계를 할 수 있는 숙련된 엔지니어로 눈 뜰 수 있게 한다.
4) **디자인을 논의하기 위한 의사소통이 쉬워짐** – 설계를 위한 의사소통을 하려면 공통의 어휘가 필요하다. 또한 전달되는 단어는 아이디어에 대해 적절하게 추상적이어야 한다. 디자인 패턴은 설계에 대한 아이디어를 논의하기 위해 추상화 레벨에서 공통 어휘를 정의하고 정확하게 설명할 수 있는 기틀을 제공한다.

그림 7.16 설계 의사소통을 위한 숙어

5) **객체지향 설계 원리를 잘 따르게 됨** – 디자인 패턴은 캡슐화, 응집, 추상화와 같은 설계 원리와 객체지향 SOLID 원칙을 잘 지킨 모범 사례다. 따라서 디자인 패턴을 잘 이해하고 다양한 문맥 속에서 경험한다면 앞서 소개한 많은 디자인 원리를 지키게 된다.

7.3.2 디자인 패턴의 형식

소프트웨어 디자인 패턴을 설명하는 일관된 형식이 있다. 이것은 원래 건축의 패턴을 설명할 때 사용된 형식에서 유래되었다. 마치 요리책에서 메뉴별로 재료와 조리법을 설명하는 형식과 같이 디자인 패턴을 설명할 때 다음 형식을 사용한다.

- **패턴 이름** – 짧은 설명과 이름은 의사소통을 촉진하고 디자인 아이디어를 논의하기 위한 어휘를 제공한다.
- **소개** – 각 디자인 패턴에 대한 설명은 배경을 정의하고 패턴을 학습하는 동기를 제공하는 소개로 시작한다.
- **해결하는 문제** – 해당 디자인 패턴이 적용되는 문제가 무엇인지 알아야 한다. 디자인 패턴으로 해결되는 설계 이슈에 대해 설명한다.

- 솔루션 – 솔루션은 디자인 패턴의 본질이다. 디자인 패턴은 디자인 문제를 해결하기 위한 해법이므로 실행 코드보다 추상적인 설계 조각이다. 클래스 다이어그램이 포함되며 클래스 간 협업이 중요할 때는 상호작용 다이어그램이 포함된다.
- 예제 – 디자인 패턴은 추상적 해법이므로 적용하는 법을 습득하기가 쉽지 않다. 따라서 특정 예제에 적용된 것을 보면 해법을 더 이해하기 쉽게 된다. 예제에 구현 사례도 제공하여 원시코드에 반영된 패턴을 확인할 수도 있다.
- 관련 패턴 – 패턴은 서로 관련된 것이 있다. 한 패턴이 다른 패턴을 완전히 포함할 수 있고 패턴 그룹이 비슷한 문제를 해결할 수도 있다. 유사한 패턴이 있는 경우 구분하기 위한 설명이 추가 된다.

7.3.3 싱글톤 패턴

싱글톤 패턴은 비교적 간단한 디자인 패턴이다. 객체를 강제적으로 하나만 생성하려는 목적이며 관련된 클래스가 하나이기 때문이다.

클래스는 객체를 만들기 위한 템플릿이다. 어떤 클래스를 이용하는 클라이언트가 생성자를 접근할 수 있다면 그 클래스의 객체를 얼마든지 만들 수 있다. 하지만 클래스 생성에 대하여 통제, 예를 들면 단일 객체를 만들어야 할 필요가 있을 때가 있다.

■ 해결하는 문제

싱글톤 디자인 패턴은 한 클래스에 대해 생성된 객체 수를 제한한다. 싱글톤 디자인 패턴은 클래스의 객체에 대하여 단일 액세스 지점과 메커니즘을 제공하므로 결과적으로 해당 클래스에 대해 작성된 오브젝트 수를 제어할 수 있다.

결국 하나의 클래스 인스턴스만 원하고 모든 클라이언트가 동일한 인스턴스를 공유하려면 싱글톤 패턴을 적용한다. 예를 들어, 데이터베이스에 대한 인터페이스를 제공하는 클래스가 있다고 하자. 다중 데이터베이스에 연결하는 것은 비용이 많이 들고 제대로 지원되지 않을 수 있어 프로그램의 모든 부분이 동일한 데이터베이스 연결을 강제하는 경우가 있다.

■ 솔루션

우선 클래스 안에 자신을 정적 속성으로 갖게 하면 그 객체는 유일하게 된다. 다음은 클래스에 대하여 생성된 유일한 객체를 반환하는 정적 메서드를 사용하여 접근하게 한다. 메서드가 처음 호출될 때는 클래스의 인스턴스가 생성되어 리턴되지만 후속되는 호출에서는 만들어진 인스턴스가 리턴된다. 클래스의 생성자는 private으로 선언하여 객체를 접근하는 정적 메서드가 클래스의 인스턴스를 만들거나 유일하게 접근할 수 있게 하여야 한다.

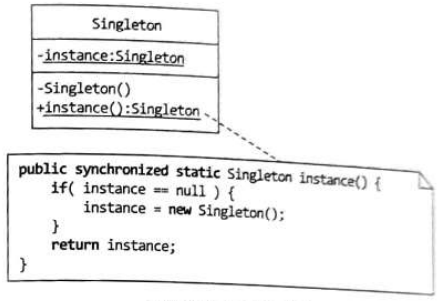


그림 7.17 싱글톤 패턴

■ 사례

```
public class DatabaseConnection{
  private DatabaseConnection() {
    // Establish a databaseconnection
    ...
  }
  private static DatabaseConnection instance = null;
  public synchronized static DatabaseConnection instance() {
    if( instance == null ) {
      instance = newDatabaseConnection( );
    }
    return instance;
  }
  // Database interface methods
  public synchronized String getAssignmentGrade(Stringstudent, String course,
String assignment) {
    ...
    return ...;
  }
}
```

■ 관련 패턴

싱글톤 패턴은 추상 팩토리 패턴과 함께 사용되어 최대 하나의 팩토리 클래스가 생성되도록 보장 할 수 있다. 상태 패턴에서도 상태가 전환될 때 상태 클래스가 다시 생성하지 않도록 싱글톤 디자인 패턴을 사용한다.

7.3.4 반복자 패턴

반복자(iterator) 패턴은 객체 그룹과 연관되어 있는 집합 클래스(컨테이너 클래스 또는 컬렉션 클래스라고도 함)와 함께 사용된다. 집합 클래스의 자료구조와 상관없이 집합에 소속된 요소들을 쉽게 접근하기 위하여 반복자를 두어 접근하고 검색하게 하려는 것이다.

예를 들어 Java에서 Vector는 불규칙한 크기의 객체 집합을 가진 클래스다. Vector라는 집합 데이터 구조를 접근하는 방법은 각 요소를 차례로 방문하여 처리하는 것이다. 집합 클래스가 이러한 내부요소를 하나씩 처리하기 위하여 getFirst(), getNext()와 같은 메소드를 제공하면 Vector를 이용하는 프로그램은 각 요소에 접근하기 위한 자세한 알고리즘을 몰라도 된다.

■ 해결하려는 문제

반복자 패턴은 집합 내부 요소의 저장 방법과 반복적 접근 알고리즘 구현에 관계없이 집합 요소에 순차적으로 액세스하는 방법을 제공한다. 다시 말해 반복자라는 것을 정의하여 벡터나 트리, 리스트 등 집합 구조와 함께 사용하면 클라이언트가 특정 집합의 유형과 유형별로 접근하고 총집계하는 방법을 신경 쓰지 않아도 된다. 또한 집계와 반복을 구현하는 방법은 클라이언트에 영향을 주지 않고 변경될 수 있다.

■ 솔루션

반복자 패턴을 구현하는 방법은 다음과 같다.

1. 반복 수행하는 집합 클래스에 대하여 접근하는 반복자를 Iterator 인터페이스로 정의한다.
2. 집합에 의해 구현될 Aggregate 인터페이스를 정의한다. 인터페이스에는 반복자를 반환하는 팩토리 메소드가 포함되어야 한다.
3. 추상 팩토리 메소드를 상속 구현하여 특정 집합의 반복자를 반환하는 집합 클래스를 작성한다.
4. 정의한 Iterator, Aggregate 인터페이스를 이용하는 클라이언트 코드를 작성한다.

[그림 7.18]은 UML 클래스 다이어그램으로 표현한 솔루션이다.

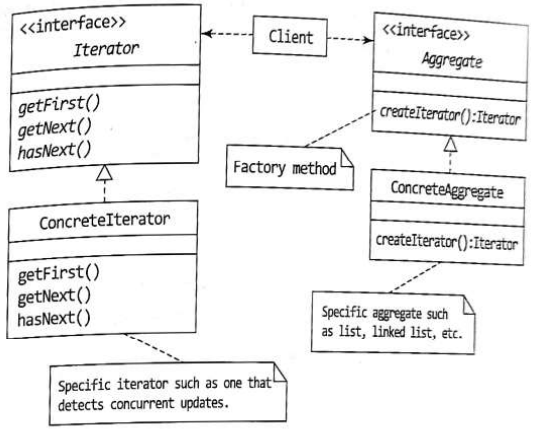


그림 7.18 반복자 패턴

■ 사례

Java 콜렉션 프레임워크에 구현된 대표적인 반복자 패턴이 있다. Collection 인터페이스가 추상적인 수준의 컬렉션을 나타낸 것이다. Collection 인터페이스를 구현한 두 가지 집합 클래스 ArrayList와 LinkedList는 배열과 연결 리스트를 보관한다.

반복자 인터페이스에는 집합에 속한 요소들을 접근하기 위한 hasNext(), next(), remove() 메소드가 정의되어 있고 구체적인 집합의 반복자에서 이를 구현한다. 그 이외에도 집합 요소들을 반복적으로 접근하는데 필요한 오퍼레이션을 추가할 수 있다.

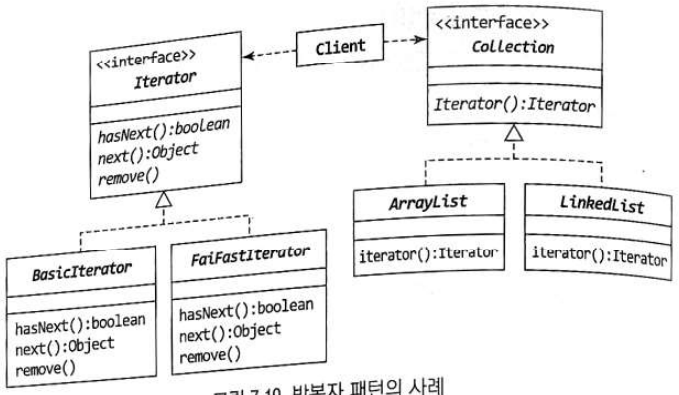


그림 7.19 반복자 패턴의 사례

7.3.5 어댑터 패턴

각종 디지털 전자제품을 사용할 때 전원 어댑터는 필수 휴대품이다. 220V 교류에서 5~12V 직류로 바꾸어 사용하는 제품은 220V의 높은 전압이 필요하지 않는 소형 전기 제품 혹은 직류의 특성을 이용해 정밀하게 작동시키는 전기제품을 사용하기 위해 교류 전압을 직류 전압으로 바꾸어 준다.

어댑터 패턴은 소프트웨어의 구성 요소에 대해 같은 역할을 수행한다. 어떤 클라이언트가 요구하는 인터페이스와는 다른 인터페이스를 가진 서비스가 있는 경우 어댑터 패턴을 사용하여 사용 가능한 서비스의 인터페이스를 클라이언트가 예상하는 인터페이스에 맞게 조정할 수 있다.

■ 해결하는 문제

어댑터 패턴은 클라이언트와 서비스가 호환되지 않는 인터페이스를 가지고 있지만 함께 작동하고 싶을 때 사용된다. 어댑터 패턴은 서비스가 제공하는 인터페이스를 클라이언트가 기대하는 인터페이스로 변환한다. 어댑터로 인하여 클라이언트와 서비스는 컴포넌트를 수정하지 않아도 함께 작동할 수 있다.

■ 솔루션

사용하고 싶은 서비스에 대하여 연관되어 있고 클라이언트가 기대하는 인터페이스를 구현한 어댑터 클래스를 작성한다. 어댑터 클래스는 클라이언트가 기대하는 인터페이스를 구현하므로 클라이언트는 어댑터 클래스와 상호작용할 수 있다.

어댑터 클래스에 대한 메소드 호출은 어댑터가 캡슐화한 서비스, 즉 사용하고 싶지만 인터페이스가 다른 서비스로 전달된다. 결국 클라이언트는 변환된 서비스에 느슨하게 연결되어 있다.

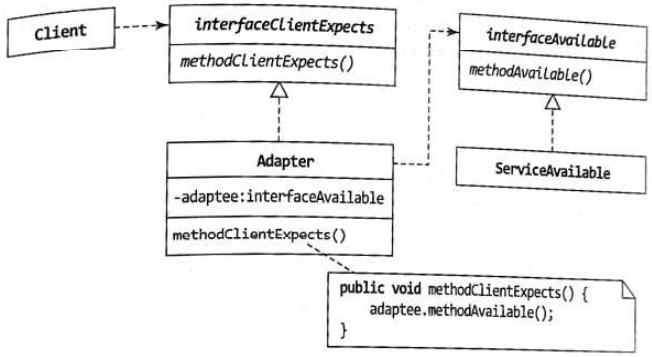


그림 7.20 어댑터 패턴

■ 사례

어댑터 패턴을 사용하여 Enumerator 타입의 인터페이스를 Iterator 인터페이스로 전환하여 사용할 수 있게 해보자.

메인 프로그램의 client() 메소드는 Enumeration 타입을 매개변수로 받는다. 한편 받고 싶은 서비스는 ArrayList 집합이다. ArrayList는 새로운 컬렉션 프레임워크의 일부이며 반복 인터페이스 Iterator만 지원한다. IterationAdapter 클래스는 Iterator가 갖는 인터페이스를 클라이언트가 기대하는 인터페이스 Enumeration에 적용하는 데 사용된다.

```
import java.util.*;
public class AdapterExample {
   public static void main(String[] args) {
      ArrayList container = newArrayList();
      container.add(new Integer(1));
      container.add(new Integer(2));
      Iterator iterator = container.iterator();
      IterationAdapter iterationAdapter = newIterationAdapter(iterator);
```

```
        client(iterationAdapter);
    }

    // This client only understands the older
    // interface Enumeration
    public static void client(Enumeration e) {
        while (e.hasMoreElements()) {
            System.out.println(e.nextElement());
        }
    }
}
class IterationAdapter implements Enumeration {
    private Iterator adaptee;
    public IterationAdapter(Iterator iterator) {
        adaptee = iterator;
    }
    public boolean hasMoreElements() {
        return adaptee.hasNext();
    }
    public Object nextElement() {
        return adaptee.next();
    }
}
```

■ 관련 패턴

어댑터 패턴은 이미 존재하는 인터페이스를 우회하여 정의한다는 측면에서 퍼싸드 패턴과 유사하다. 하지만 퍼싸드 패턴은 새로 인터페이스를 정의하는 것이 아니며 더 간단한 인터페이스 층을 제공하는 개념이다.

7.3.6 데코레이터 패턴

데코레이터 패턴은 집합 관계와 위임을 사용하여 기존 클래스의 동작을 가볍고 유연하게 확장한다.

데코레이터 패턴의 해법에 대한 가치를 이해하려면 클래스의 동작을 확장하기 위한 다른 방법들을 살펴보아야 한다. 클래스의 동작을 확장하는 가장 간단한 방법은 단순히 새로운 동작을 포함하도록 클래스를 수정하는 것입니다. 예를 들어, [그림 7.21]에서 기

존 클래스는 새로운 기능 A, B 및 C를 포함하도록 수정하였다. 이 방식의 가장 큰 문제점은 OCP 원리를 위배하고 있다.

OCP는 클래스가 확장을 위해 열려 있어야하지만 수정을 위해 폐쇄되어야 한다는 것이다. 이 방법을 사용하면 기능을 추가하거나 수정하려면 기존 클래스를 변경해야 하므로 기존 클래스가 닫히지 않는다.

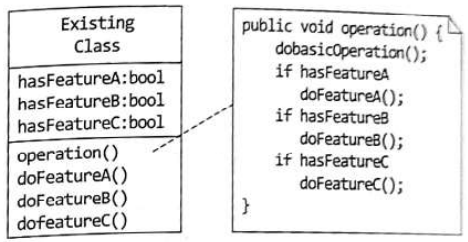


그림 7.21 수정에 의한 추가

클래스의 동작을 확장하는 방법으로 상속이 있다. 클래스 A가 클래스 B에서 상속되면 클래스 A는 클래스 B의 모든 기능을 가지며 클래스 B의 메소드를 대체하여 새 동작을 추가할 수 있다.

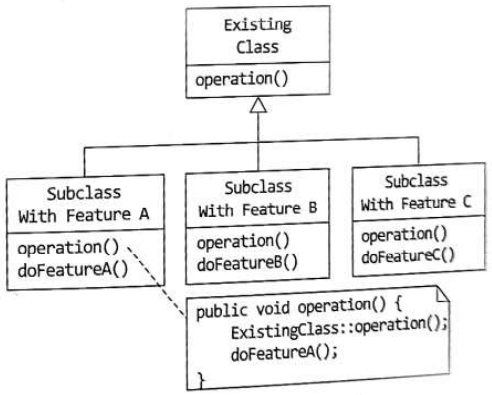


그림 7.22 상속에 의한 추가

서브클래스를 통하여 새로운 동작을 추가하는 것은 OCP 원칙을 따르는 것 같다. 기본 클래스에 코드를 변경하지 않고 새로운 서브클래스를 추가할 수 있기 때문이다. 하지만 기본 클래스 안에 보호된 인터페이스가 서브클래스에 노출되기 때문에 캡슐화를 약

화시켜 정보은닉에 취약성을 보이게 한다.

결국 상속을 통해 동작을 확장하면 기존 클래스가 변경에 취약하게 된다. 이에 비하여 위임은 클래스의 공용 인터페이스로만 종속성이 제한된다.

상속을 사용하여 클래스의 동작을 확장할 때의 또 다른 단점은 추가되는 기능과 확장되는 클래스 사이에 컴파일 타임 의존 관계가 생성된다는 것이다. 실행하는 동안 객체의 구성을 수시로 바꿀 수 없게 된다. 상속을 이용하여 런타임에 기능을 추가 및 제거한다는 것은 기능을 추가 또는 삭제한 서브 클래스를 새로 작성하거나 삭제하는 것을 의미한다.

상속을 통하여 클래스의 기능을 확장한다면 기능의 조합 수 만큼의 서브클래스가 필요하다. 예를 들어 [그림 7.23]에서 기능 A, B, C의 각 조합을 기본 클래스에 추가하려면 7개의 서로 다른 서브클래스가 필요하다. 상속만을 이용하면 실행 중 필요한 기능을 추가, 삭제하는 것은 불가능하다.

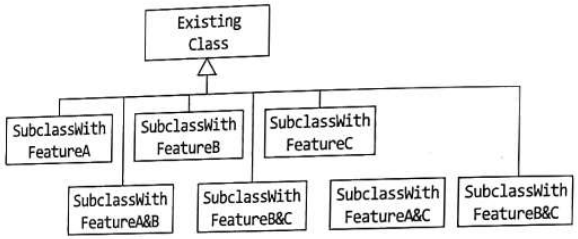


그림 7.23 상속을 이용한 기능추가 사례

객체의 구성(composition) 관계를 사용하면 참조하는 객체를 추가, 제거함으로써 런타임에 객체의 기능을 변경할 수 있다. 데코레이터 패턴이 이런 약간 복잡한 대안 방법을 제공하지만 위에서 언급한 문제들을 해결한다.

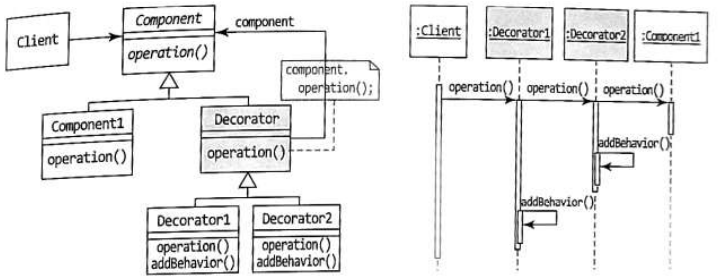
■ 해결하는 문제

데코레이터 패턴은 기본 클래스의 동작을 동적으로 추가하는 문제를 해결한다. 상속을 이용하면 컴파일 할 때 동작이 확정되므로 동적으로 확장할 수가 없다. 실행될 때 객체에 장식적인 책임을 추가할 수 있도록 유연하게 하기 위하여 상속 대신 구성 관계를 사용한다.

■ 솔루션

데코레이터 패턴의 두 가지 구성 요소는 component 클래스와 확장 기능이 담긴 데코레이터다. 상속된 구체적인 component1 클래스는 decorator가 가진 기능으로 확장되거나 장식될 기본 기능을 나타낸다. [그림 7.24]의 (a)에서 Decorator와 구성 관계에 있는 Component 요소를 생성자에 전달하여 추가할 수 있다. 이 때 Decorator 객체에는 Component1 또는 다른 Decorator 객체의 참조가 포함되어 확장된다. 이렇게 하면 긴 Decorator 체인을 사용하여 장식 요소를 얼마든지 래핑할 수 있다.

[그림 7.24(b)]는 런타임에 어떤 객체들이 어떻게 상호작용 하는지 보여준다. Decorator1 객체가 Component1 객체를 랩핑하고 Decorator2 객체가 랩핑되는 예를 보여준다.



(a) 정적 구조 (b) 동적인 객체 구성 및 상호작용

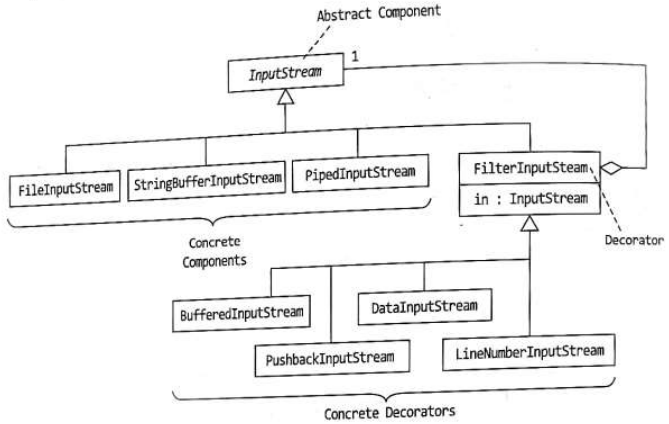
그림 7.24 데코레이터 패턴

[그림 7.24(b)]는 커뮤니케이션 다이어그램으로 기능이 추가되는 과정을 보인 것이다. Decorator1 객체에서 operation()을 호출할 때 발생하는 상호작용 순서를 보여준다. Decorator2는 operation()이 호출되면 메서드 안에서 addBehavior()가 수행되어 기능이 추가되며 체인의 다음 컴포넌트 Decorator1로 요청이 전달되어 addBehavior()로 추가된다.

데코레이터는 기본 클래스와 동일한 인터페이스, operation()을 구현한다. 따라서 클라이언트가 operation()을 호출할 때 기본 객체와 장식된 객체 모두를 필요한 모든 곳에서 투명하게 사용할 수 있다.

■ 사례

데코레이터 패턴은 Java 입력 클래스 라이브러리, java.io 패키지에서 많이 사용된다. [그림 7.25]는 데코레이터 패턴을 이용하여 스트림 입력에 장식 요소를 추가하는 과정을 보여준다. [그림 7.25]의 기본 구성 요소는 여러 가지 형태의 비트 입력 소스를 나타낸다. 데코레이터는 입력 스트림에 추가할 수 있는 기능을 나타낸다. 여러 개의 구체적인 컴포넌트, 여러 개의 구체적인 데코레이터가 있다. 기본 컴포넌트, 입력 스트림은 데코레이터가 제공하는 다양한 기능 조합으로 장식할 수 있다. 예를 들어, 아래 프로그램은 데이터를 버퍼링하고 푸싱, 즉 다시 화면에 뿌리는 기능으로 래핑된 파일 입력 스트림을 보여준다.




```
. . .
FileInputStream f =  new FileInputStream("input.txt");
BufferedInputStream b =  new BufferedInputStream(f);
PushbackInputStream p =  new PushbackInputStream(b);

int c = p.read();
while (c != ' ') {
   // Process c
   . . .
   c = p.read();
}
// Push back non-blank character read.
// It is expected by the next processing
// routine.
p.unread(c);
```

그림 7.25 Java 입출력에 사용된 데코레이터 패턴

■ 관련 패턴

데코레이터와 어댑터 패턴은 모두 위임을 사용하여 인터페이스를 구현한다는 점에서 비슷하다. 차이점은 어댑터 패턴은 요청과는 다른 인터페이스를 구현하고 데코레이터 패턴은 래핑하는 객체와 동일한 인터페이스를 구현한다는 것이다.

어댑터는 동일한 동작이지만 다른 인터페이스로 오브젝트를 래핑한다. 반면 데코레이터는 래핑 객체의 동작을 확장하면서 동일한 인터페이스를 유지한다. 즉 데코레이터 패턴의 초점은 확장에 있고 어댑터 패턴의 초점은 호환성이다.

7.3.7 팩토리 메소드 패턴

팩토리 메소드 패턴은 클래스의 새로운 객체를 생성할 때 사용된다. 클래스의 객체를 만드는 일반적인 방법은 다음과 같이 생성자를 호출하는 것이다.

```
SomeClass sc = new SomeClass ();
```

어떤 종류의 클래스에 대한 인스턴스가 필요한지 아는 경우는 위의 방법이 맞다. 하지만 때때로(특히 프레임워크에서) 어떤 일반적인 클래스의 유형이 필요한지는 알지만 구체적인 클래스 유형이 생성되는지는 모르거나 신경 쓰지 않고 싶을 때가 있다.

예를 들어, 자동차 견적 프로그램을 작성할 때 매장에서 실제 손님이 SUV의 견적을 요청할 때 어떤 종류의 SUV가 요청될지 상관하지 않고 싶다. 처음 프로그램을 만들어 가는 과정에서는 SUV의 구체적인 종류에 대한 인스턴스 생성보다 추상적인 SUV의 생성이 필요하다는 것만 신경 쓰고 싶을 때가 있다.

■ 해결하려는 문제

팩토리 메소드는 클라이언트에서 사용할 클래스의 객체를 생성하는 책임을 분리하여 객체 생성에 변화를 대비하려는 문제를 해결한다.

■ 솔루션

팩토리 메소드로 생성될 새로운 객체(product)를 위한 추상 인터페이스를 정의한다. 클라이언트는 이 인터페이스를 사용하여 객체를 참조할 것이다. 객체를 생성하기 위한 팩토리 메소드(createProduct)를 포함하는 추상 클래스(AbstractCreator)를 정의한

다. 클라이언트는 이 인터페이스를 사용하여 객체를 생성한다. 각 객체 유형을 생성하는 팩토리 메소드를 구현한 구체적 클래스(ConcreteCreator)를 작성한다.

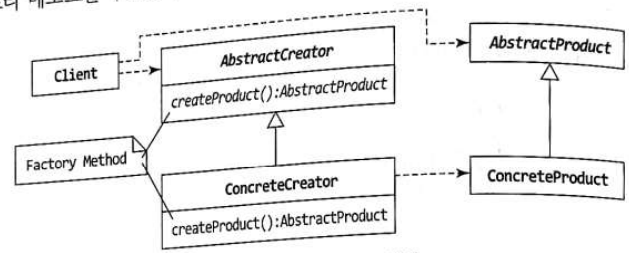
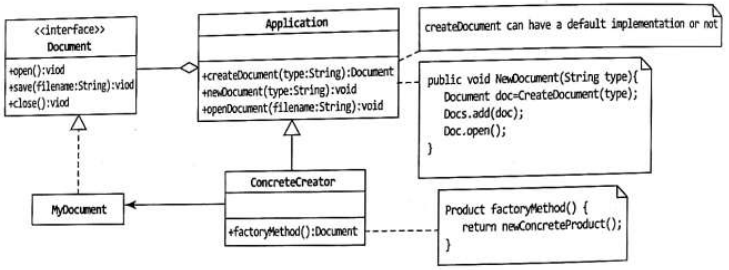


그림 7.26 팩토리 메소드 패턴

■ 사례

데스크탑 애플리케이션의 프레임워크를 생각해보자. 프레임워크에는 Document 열기, 작성, 저장과 같은 오퍼레이션의 정의가 포함되어 있다. 기본 클래스는 Application과 Document라는 추상 클래스이며, 클라이언트는 응용 프로그램을 정의하기 위해 하위 클래스를 만들어야 한다.




```
public Document CreateDocument(String type){
    if (type.isEqual("html"))
        return new HtmlDocument();
    if (type.isEqual("proprietary"))
        return new MyDocument();
    if (type.isEqual("pdf"))
        return new PdfDocument ();
}

public void NewDocument(String type){
    Document doc=CreateDocument(type);
    Docs.add(doc);
    Doc.open();
}
```

그림 7.27 팩토리 메소드 패턴 사례

예를 들어, 그림판 응용 프로그램을 생성하려면 DrawingApplication 및 DrawingDocument 클래스를 정의해야한다. Application 클래스에는 클라이언트의 요청, 예를 들면 사용자가 메뉴에서 열기 또는 저장 명령을 선택한 경우 Document를 관리하는 작업을 구동한다.

인스턴스로 생성할 Document 클래스(htmp, word, PDF 등)는 응용 프로그램에 따라 다르고 Application 클래스는 이를 미리 알지 못하므로 추상 클래스의 팩토리 메소드에게 의뢰한다.

팩토리 메소드 패턴은 인스턴스를 생성하는 클래스와 관련된 모든 정보를 한 곳에 넣고 프레임워크 외부에서 사용하여 문제를 해결한다.

7.3.8 추상 팩토리 패턴

추상 팩토리 패턴은 팩토리 메소드 패턴과 같이 클래스 외부로 객체를 만드는 책임을 이동시키는 방식으로 객체 생성에 융통성을 준다. 클라이언트는 언제 객체를 만들지 결정하고 어떤 클래스를 만들지는 클라이언트 외부의 다른 클래스에서 결정한다.

두 패턴의 차이점은 추상 팩토리 패턴의 경우 관련 객체의 패밀리를 작성하도록 설계되었다는 점이다. 객체의 패밀리란 부품 객체의 집합체를 의미한다. 예를 들어, CD와 CD 플레이어가 디지털 플레이어를 구성하며 다른 유형으로 레코드와 레코드플레이어가 아날로그 플레이어를 구성할 때 두 가지 유형의 패밀리가 있다.

각 플레이어는 다른 매체와 호환되지 않기 때문에 주어진 패밀리에 대한 부품 생성을 추상 팩토리에게 맡기려는 것이다.

■ 해결하려는 문제

추상 팩토리 패턴은 객체를 사용할 클라이언트에서 구체적인 객체 생성을 지정하는 책임을 분리하기 위하여 추상 인터페이스를 이용하여 관련 객체 패밀리를 작성하는 방법을 해결하려는 것이다.

■ 솔루션

팩토리 메소드 패턴의 솔루션과 유사하다. 먼저 사용하려는 모든 Product에 대하

여 추상 인터페이스를 정의한다. 관련 Product 그룹을 만들기 위한 추상 팩토리 클래스를 정의한다. 각 관련 Product 그룹에 대해 추상 팩토리 인터페이스를 구현한 ConcreteFactory 클래스를 작성하고 함께 사용되는 Product 패밀리를 작성한다. 클라이언트는 Product의 추상 인터페이스 및 추상 팩토리 인터페이스에 밀접하게 연결되며 특정 Product 및 특정 Product 팩토리에 느슨하게 연결된다. 특정 팩토리는 그들이 만든 Product와 밀접하게 연결될 것이다.

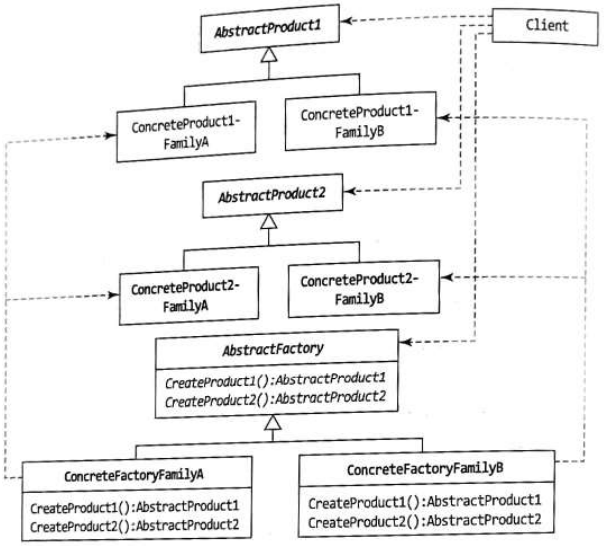


그림 7.28 추상 팩토리 패턴

■ 사례

위에서 언급한 음악 재생에 필요한 하드웨어 및 소프트웨어 요소를 생성하기 위한 추상 팩토리 패턴의 사례다. 음악을 재생하기 위한 하드웨어 및 소프트웨어 구성 요소가 관련되어 있기 때문에 추상 팩토리 패턴이 사용된다. 클라이언트는 음악을 재생하는 데 사용되는 하드웨어 및 소프트웨어 유형에는 관심이 없다. 클라이언트는 음악 재생에만 관심이 있다. 추상 클래스 플레이어, 미디어, AbstractMusicFactory는 음악 재생의 하드웨어, 소프트웨어 세부 사항으로부터 클라이언트를 보호한다.

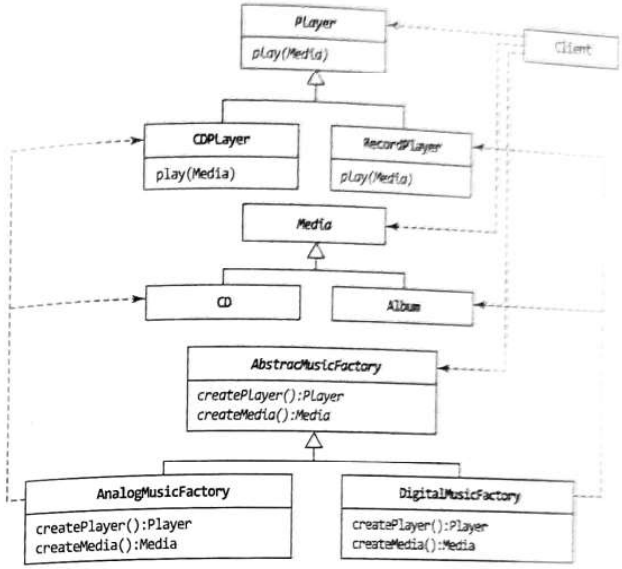


그림 7.29 추상 팩토리 패턴의 사례

7.3.9 상태 패턴

상태 패턴은 메서드의 교환에 초점을 둔 행위 패턴 중 하나다. 상태 패턴은 객체가 내부 상태에 따라 동작이 변경될 때 사용한다.

상태에 따라 객체의 동작을 변경해야하는 경우 객체에 상태 변수가 있고 if-else 조건 블록을 사용하여 상태에 따라 다른 동작을 수행하게 하는 방법을 생각할 수 있다. 하지만 이런 방법은 상태가 프로그램 구조에 잘 드러나지 않을 뿐만 아니라 상태의 변경이나 추가에 대하여 영향을 많이 받는 단점이 있다.

상태 패턴은 맥락과 상태를 별도로 구현하여 융통성을 달성하기 위한 체계적이고 느슨한 결합 방식을 제공하는 데 사용된다. 또한, 응용 프로그램의 상태에 따라 제어 흐름을 벡터화하여 동적으로 상태를 변경시킬 수 있는 구조로 만드는 장점이 있다.

■ 해결하는 문제

데스크탑 애플리케이션의 Document 클래스를 생각해 보자. Document는 초안(Draft), 검토(Moderation), 출판(Publish) 세 가지 상태 중 하나다. 문서의 출판 방법

은 각 상태에 따라 약간 다르게 작동한다.

- 초안 상태에서는 문서를 검토로 이동한다.
- 검토 상태에서는 현재 사용자가 관리자인 경우에만 문서를 출판할 수 있다.
- 게시에서는 아무 것도 하지 않는다.

```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == 'admin')
                    state = "published"
                break
            "published":
                // Do nothing.
                break
    // ...
```

그림 7.30 상태의 처리와 전환

조건문에 기반 한 상태 시스템의 가장 큰 약점은 Document 클래스에 점점 더 많은 상태와 동작을 추가하기 시작하면 나타난다. 현재 상태에 따라 적절한 동작을 선택하는 장황한 조건문이 될 수밖에 없다. 상태 전환 로직을 변경하려면 항상 상태 조건을 변경해야 하므로 이와 같은 코드를 유지하기가 매우 어렵다.

프로젝트가 진행됨에 따라 이 문제는 더 커진다. 설계 단계에서 가능한 모든 상태와 전환을 예측하는 것은 매우 어렵다. 상태 패턴은 제한된 상태 조건으로 구축된 상태 머신을 시간이 지남에 따라 원시코드의 구조에 영향을 적게 받는 코드가 되는 문제를 다룬다.

■ 솔루션

상태 패턴은 객체가 가질 수 있는 모든 상태에 대하여 새 클래스를 만들고 모든 상태별 동작을 상태 클래스 안으로 몰아둔다.

Context라는 원래 객체는 모든 동작을 조건문 내부에 구현하는 대신 현재 상태를 나타내는 상태 객체 중 하나에 대한 참조를 저장하고 모든 상태 관련 작업을 해당 객체에 위임한다.

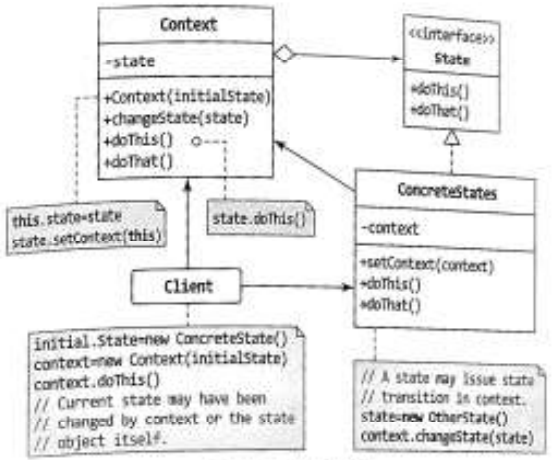


그림 7.31 상태 패턴

Context를 다른 상태로 전환하려면 활성 상태 오브젝트를 새 상태를 나타내는 다른 객체로 바꾸면 된다. 중요한 점은 모든 상태 클래스가 동일한 인터페이스를 따르고 Context도 같은 인터페이스를 통해 상태 객체와 작동하여야 한다는 것이다.

■ 사례

위에서 설명한 데스크탑 애플리케이션의 Document 클래스의 세 가지 상태(Draft, Moderation, Publish)를 인터페이스 State의 특정 객체로 구현한다. 각 상태 내부에

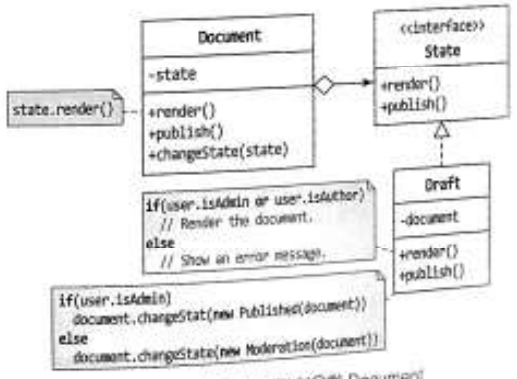


그림 7.32 상태 패턴을 이용한 Document

서의 동작은 특정 상태의 인터페이스를 구현한 메소드 안으로 분리되어 들어가 있다. 또한 상태 전환은 상태 내부 동작이 끝난 후 changeState() 메소드를 호출하여 다음 상태를 지정한다.

■ 관련 패턴

상태 패턴의 구조는 전략 패턴과 비슷하게 보일 수 있지만 한 가지 중요한 차이점이 있다. 상태 패턴에서 특정 상태는 서로를 인식하여 한 상태에서 다른 상태로의 전환을 구동시키는 반면 전략 패턴은 서로에 대해 거의 알지 못한다.

7.3.10 옵서버 패턴

6장에서 설명한 것처럼 잘 설계된 시스템은 협력하는 객체들이 느슨하게 결합된다. 필요하면 객체 사이에 의존될 수 있지만 의존은 가능한 추상적이거나 느슨하여야 한다. 객체 사이의 느슨한 결합은 객체가 독립적으로 확장, 진화할 수 있기 때문에 중요하다.

객체들은 한 객체를 변경하면 다른 객체를 변경해야하는 방식으로 의존되어 있다. 예를 들어, 스프레드시트의 의존 관계를 생각해 보자. 데이터는 셀에 저장되며 다양한 뷰가 있다.

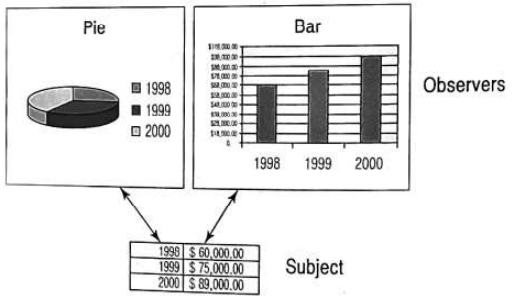


그림 7.33 옵서버 패턴이 적용될 사례

관심의 대상이 되는 데이터에 대한 변경 사항은 이 데이터에 관심 있는 모든 뷰에게 전달되어야 한다. 데이터의 변화가 관찰자에게 전파될 수 있도록 Subject는 옵서버와 연결되어야하지만, 밀접하게 연결될 필요는 없다. 위의 예에서 Subject는 원형 차트와 막대 차트에게 관찰되고 있음을 구체적으로 알 필요는 없으며 단지 상태가 변경 될 때마다 알려야 하는 두 개의 옵서버가 있다는 정도만 알면 된다. 이것이 느슨한 결합이며 옵서버 패턴은 Subject 객체가 옵서버 객체와 느슨하게 결합되는 방법을 제공한다.

■ 해결하는 문제

옵서버 디자인 패턴은 데이터를 보관하고 있는 Subject가 그 데이터를 이용하는 옵서버와 효과적으로 통신하면서도 어떻게 하면 느슨하게 결합할 수 있는가라는 문제를 다룬다. 옵서버가 계속 데이터의 변경을 체크하는 것이 아니라 데이터(Subject)가 변경될 때마다 관찰하는 옵서버들에게 통지하면 느슨한 결합이 된다. 결합이 느슨하기 때문에 옵서버는 Subject나 다른 옵서버에게 영향을 주지 않고 변경하거나 추가할 수 있다.

■ 솔루션

먼저 데이터를 가지고 있는 Subject를 정의하여야 한다. Subject 클래스는 자신의 데이터에 관심 있는 옵서버 목록을 유지하고 상태가 변경될 때 옵서버에게 알리는 등작을 갖도록 한다. 옵서버 클래스는 Subject가 변경 사항을 알리면 내용을 받아갈 수 있는 콜백 메소드(update)를 정의한다. Subject는 상태를 가진 특정 Subject로 상속될 수 있고 Subject에 옵서버로 등록할 특정 옵서버는 Observer 인터페이스를 구현한다.

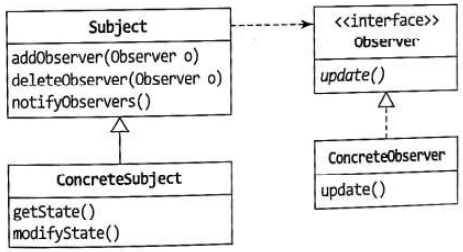


그림 7.34 옵서버 패턴

특정 Subject에는 옵서버가 통지를 받은 후 변경된 상태를 가져오기 위한 메소드로 getState()가 있어야 한다. 또한 특정 Observer는 옵서버 인터페이스에 정의된 update() 메소드를 구현한다.

옵서버 패턴에서 Subject와 옵서버가 동적으로 상호 작용하는 과정을 [그림 7.35]에

나타내었다. 첫 단계는 옵서버가 Subject에 자료를 구독하겠다고 등록하는 것이다. Subject를 업데이트하고 옵서버에게 알리는 다른 과정은 다음과 같다.

1. 한 옵서버가 Subject의 상태를 수정하는 경우 모든 옵서버에게 notifyObservers() 를 호출하여 즉시 통보된다.
2. 옵서버는 Subject의 상태에 따라 변경된 사항을 알리기 위해 update() 메소드를 통해 옵서버에게 업데이트를 전송한다.

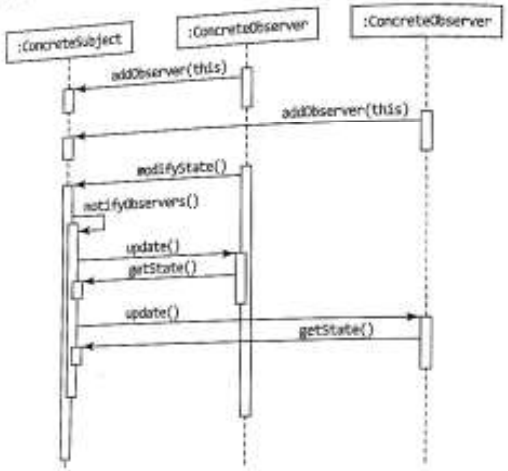


그림 7.35 옵서버 패턴의 동적 다이어그램

■ 사례

뉴스 대행사를 예로 들어 보자. 대행사는 뉴스를 수집하여 다른 구독자에게 알려준다. 새로운 기사가 발생되었을 때 가입자에게 뉴스에 대해 즉시 알릴 수 있는 프레임워크를 만들어야 한다. 가입자는 이메일, SMS 등 다양한 방법으로 뉴스를 수신할 수 있다. 새로운 통신 기술의 출현으로 새 유형의 가입자를 추가할 수 있도록 융통성이 있어야 한다.

우선 뉴스 대행사는 NewsPublisher라는 추상 클래스를 정의하고 Observable 인터페이스를 가질 수 있게 한다. 다음은 여러 유형의 Observable 객체를 생성하는데 차음에는 비즈니스 뉴스를 위한 구체적인 클래스, BusinessNewsPublisher을 구현한다.

NewsPublisher에서 구현되는 옵서버 로직은 구독자 목록을 유지하고 최신 뉴스를 알려주는 것이다. 가입자는 옵서버(SMSSubscriber, EmailSubscriber)로 표시된다. 구체적 옵서버 두 가지는 모두 Subscriber로부터 상속받는다. Subscriber는 Publisher에게 알려진 추상 클래스다. Publisher는 특정 옵서버에 대해서는 모르고 추상 클래스만 알고 있기 때문에 동적으로 추가될 수 있고 변경될 수 있다. 즉 옵서버 패턴을 적용하면 새로운 유형의 Subscriber를 쉽게 추가 할 수 있으며 새로운 유형의 Publisher, 예를 들면 날씨 뉴스, 스포츠 뉴스 등을 쉽게 추가 할 수 있다.

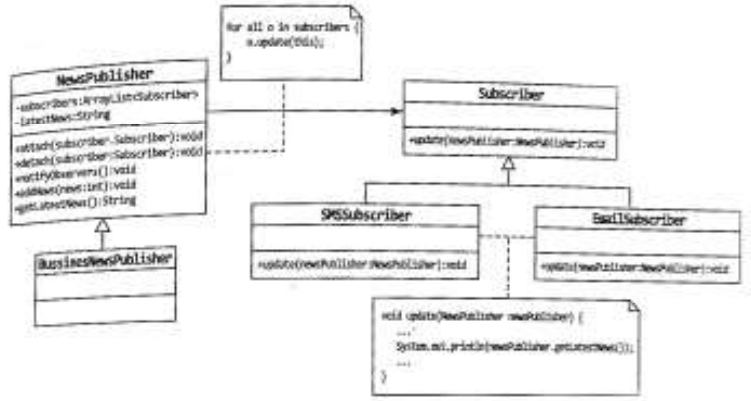


그림 7.36 옵서버 패턴 사례

7.4 아키텍처 평가

아키텍처 평가는 소프트웨어 아키텍처나 디자인 패턴의 속성, 강점 및 약점을 결정하는 방법이다. 아키텍처 평가는 개발자가 선택한 아키텍처가 기능적 및 비기능적 품질 요구 사항을 모두 충족시킬 수 있음을 보증하는 과정이다. 따라서 아키텍처 평가를 통해 시스템에 대하여 이해할 수 있고 문서화하며 기존 아키텍처의 문제를 감지할 수 있다.

소프트웨어 아키텍처 평가를 위해 몇 가지 방법과 기술이 제안되어 있다. 시나리오 기반 접근 방식(Software Architecture Analysis Method: SAAD) 이 가장 성숙되어 있고 속성 모델 기반 방법(Architecture Trade-off Analysis Method: ATAM)과 정량적 모델도 있다. 아키텍처 스타일의 속성과 디자인 패턴을 체계적으로 정당화하기 위