컴포넌트 재사용을 늘리기 위한 전략 패턴의 활용 방법

심준용*, 오정인*, 위성혁*, 김세환*
*LIG넥스원(주) 소프트웨어연구센터
e-mail:jyshim79@lignex1.com

Strategies for Component reuse using Strategy Design Pattern

Jun-Yong Shim*, Jung-in Oh*, Soung-Hyouk Wi*, Sae-Hwan Kim*
*Software R&D Center, LIG Nex1 Co., Ltd.

요 약

디자인 패턴은 소프트웨어 설계 시 반복적으로 발생하는 문제를 해결하기 위한 방법을 기술한다. 특히, 객체지향 기술을 기반으로 하는 컴포넌트 설계 시 디자인 패턴을 활용함으로써 특정 설계 문제에 대한 해결책을 재사용할 수 있다. 좋은 객체 지향 설계는 재사용성, 확장성 및 유지보수성을 제공하는 것이며, 디자인 패턴은 좋은 설계에 필요한 구성 요소들의 관계 구조를 제시한다. 본 논문은 객체지향 기반의 컴포넌트 프레임워크 설계 시 프레임워크의 확장성과 구현 컴포넌트의 재사용성을 늘리기 위한 방안으로 디자인 패턴의 활용법을 제시한다. 특히, 알고리즘 재사용의 구조를 제시하는 전략 패턴과 처리 절차 재사용의 구조를 제시하는 템플릿 메서드 패턴의 구조를 비교하고, 분산 통신 컴포넌트 설계 시 전략 패턴의 적용 사례를 보여준다.

1. 서론

소프트웨어를 개발하는데 있어서 개발 목표 및 일정 수립, 구현 언어, 소프트웨어 아키텍처에 대한 정의는 시스템의 요구사항을 만족시키기 위한 중요한 요소들이다. 특히, 소프트웨어 아키텍처는 요구사항 뿐만 아니라 시스템의 품질 속성을 만족시키기 위한 기반 구조가 되므로 좋은 설계를 가져야 한다. 좋은 설계는 개발 소프트웨어의 코드 또는 라이브러리에 대한 재사용성 및 확장성을 보장해야 하며, 무엇보다도 요구사항 변경에 대한 빠른 대처 능력을 갖춰야 한다[1]. 디자인 패턴은 소프트웨어 설계 시 반복적으로 발생하는 문제를 해결하기 위한 재사용 가능한 방법으로서 클래스 또는 객체 사이의 관계 및 상호작용 구조를 제시한다[2]. 즉, 디자인 패턴 적용을 통해 소프트웨어 설계 또는 개발 과정에서 발생하는 변경에 대한 문제를 최소의 비용으로 해결할 수 있다.

본 논문은 객체지향 기반의 프레임워크 설계 시 프레임워크의 확장성과 구현 컴포넌트의 재사용성을 늘리기 위한 디자인 패턴의 활용 방법을 제시한다. 특히, 기능 재사용 방법을 제시하는 전략 패턴과 수행 절차 재사용 방법을 제시하는 템플릿 메서드 패턴의 구조를 비교하고, 전략 패턴의 적용 사례를 보여준다. 구성은 다음과 같다. 2장은 프레임워크와 디자인 패턴의 개념 및 관계를 기술하고, 3장은 객체지향 기반의 프레임워크 설계를 위한 디자인 패턴의 활용 방법을 알아본다. 특히, 프레임워크 설계를 위한 전략 패턴과 템플릿 메서드 패턴 구조의 비교 및 적용 방법을 기술한다. 4장에서 프레임워크를 기반으로 구현된 분산 통신 컴포넌트의 설계를 위한 전략 패턴의 적용 사례를 보여준 후, 5장에서 결론을 맺는다.

2. 프레임워크와 디자인 패턴

2.1. 프레임워크

프레임워크는 소프트웨어의 특정한 클래스에 대하여 재사용할 수 있는 설계로 구성된 관련 클래스들의 집합이다. 설계를 추상적인 클래스로 분리하고 그들의 책임과 상호작용 관계를 정의함으로써 구조적인 가이드를 제공한다. 프레임워크를 기반으로 소프트웨어를 개발할 경우 높은 재사용성(reusability)과 확장성(extensibility)을 갖는다. 즉, 다수의 어플리케이션에서 반복적으로 사용될 수 있는 일반적인 컴포넌트를 정의함으로써 재사용성을 늘리고, 객체 지향의 다형성(polymorphism)을 통해 인터페이스를 확장함으로써 개발 어플리케이션의 목적에 맞게 최적화시킬 수 있다. 또한 프레임워크는 기능 처리에 대한 흐름을 제어함으로써(inversion of control) 외부 이벤트에 대해서 어플리케이션이 어떠한 메서드들을 수행해야 되는지를 결정할 수 있게 한다.

프레임워크 설계에는 크게 두 가지 접근 방식이 있다. 하나는 화이트박스 프레임워크 방식이고, 다른 하나는 블랙박스 프레임워크 방식이다[3]. 화이트박스 프레임워크는 주로 상속(inheritance)이나 동적 바인딩(dynamic binding)의 특징을 통해 구현되며, 내부 클래스의 결합도가 높다.

반면, 블랙박스 프레임워크는 객체 합성을 통해 구현되며, 컴포넌트로 구성된다.

2.2. 디자인 패턴

잘 알려진 GoF의 디자인 패턴은 패턴의 구성요소로서 패턴 이름, 문제, 해법 및 결과의 네 가지 요소를 정의하며, 그 종류를 객체 생성 절차를 추상화한 생성 패턴, 클래스와 객체의 구조를 다루는 구조 패턴 그리고 어떤 행위에 대한 책임을 어떤 객체에서 다루게 할 것인지를 정의하는 행동 패턴으로 분류하고 있다. 디자인 패턴의 적용은 개발자들이 특정 설계 문제에 대한 해결책을 재사용할 수 있도록 함으로써 적은 비용으로 좋은 설계를 할 수 있도록 도와준다. 디자인 패턴은 객체지향의 상속과 다형성의 특징을 갖기 때문에 프레임워크 또는 컴포넌트 설계시 변경에 대한 유연한 구조를 제공하는 장점을 갖는다.

2.3. 프레임워크와 디자인 패턴의 관계

유연한 설계는 프레임워크 설계의 핵심이다. 디자인 패턴은 재사용과 확장 가능한 설계의 핵심이 되는 다형성을 기반으로 클래스와 객체의 관계 구조를 기술하므로, 디자인 패턴을 이용한 프레임워크는 그렇지 않은 프레임워크보다 설계와 코드 재사용의 수준을 높일 수 있다. 패턴은 추상적인 디자인 및 구조에 대한 지식을 제공하므로, 패턴의 적용은 프레임워크의 구현과 함께 병행되어야 한다.

3. 디자인 패턴 활용 방안

앞에서 살펴본 화이트박스 방식과 블랙박스 방식의 프레임워크를 구현하기 위해서 적용할 수 있는 대표적인 디자인 패턴으로 행동 패턴에 기술된 템플릿 메서드와 전략 패턴이 존재한다. 각 패턴의 구조와 프레임워크의 활용 방안을 살펴보겠다.

3.1. 템플릿 메서드(Template Method) 패턴

템플릿 메서드 패턴은 객체의 연산에는 알고리즘의 구조만을 정의하고, 각 단계에서 수행할 구체적인 처리는 서브클래스가 재정의 할 수 있도록 하는 패턴이다. 구조는 그림 1과 같다.

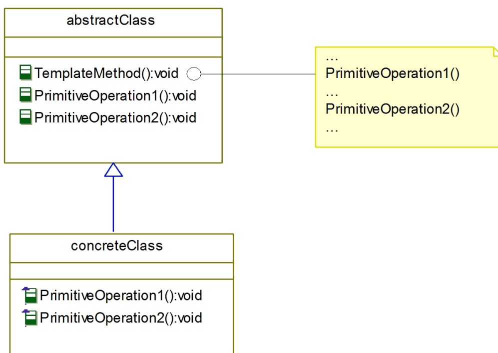

그림 1 템플릿 메서드 패턴 구조

AbstractClass는 ConcreteClass들이 상속받아 구현해야 하는 알고리즘 처리 절차를 기술한 기본 연산을 정의한다. ConcreteClass는 AbstractClass에서 정의한 기본 연산을 재정의(overriding)한다. 템플릿 메서드 패턴은 기본적으로 코드를 재사용하는 기술로써 프레임워크의 기능 제어 구조를 구현하는 수단이 된다. 이 때 서브클래스가 확장할 수 있는 기본 행동은 훅 연산(hook operation)을 통해 정의된다. 훅 연산은 프레임워크가 사용자 처리 부분을 가져오는 연산을 의미한다. 그림 2는 템플릿 메서드 패턴의 구현 예제이다.

```
// BaseClass
void  BaseClass::templateMethod()
{
     setFunction();
     primitiveOperation();
     resetFunction();
}
// ConcreteClass
void ConcreteClass::primitiveOperation()
{
     std::cout << "user requirement << std::endl;
}
// Main
BaseClass* testClass = new CoincreteClass;
testClass->templateMethod();
```

그림 2 템플릿 메서드 구현 예제

예제를 살펴보면, BaseClass는 어플리케이션의 기능 수행 절차를 templateMethod 함수로 정의하고, 개발자는 BaseClass를 상속받아 프레임워크에서 제공하는 추상화된 연산 즉, primitiveOperation 함수를 재정의 하면 된다. 주의할 점은 어떤 연산들이 추상화된 연산으로 설계되었는지를 이해해야 한다. 이 패턴은 코드 재사용 방식의 화이트박스 프레임워크 구현을 위한 핵심이다.

3.2. 전략(Strategy) 패턴

전략 패턴은 동일 계열의 알고리즘 군을 정의하고, 각각의 알고리즘을 캡슐화하며, 이들을 상호 교환이 가능하도록 하는 패턴이다. 즉, 전략이 바뀌더라도 사용자는 영향을 받지 않는다. 구조는 그림 3과 같다.

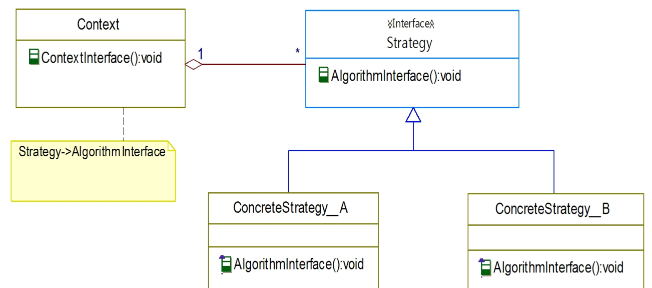

그림 3 전략 패턴 구조

Strategy는 제공하는 모든 알고리즘에 대한 공통의 연산들을 인터페이스로 정의하며, Context는 Strategy를 통해 실제 알고리즘을 사용하게 된다. ConcreteStrategy는

Context가 요구하는 AlgorithmInterface 함수를 구현한다. 그림 4는 전략 패턴의 구현 예제이다.

```
// Context 클래스의 Constructor
Context(Strategy*);

// Context 클래스의 Instance 생성
Context* contextA = new Context(new ConcreteStrategyA);
Context* contextB = new Context(new ConcreteStrategyB);

// Context 수행
contextA->ContextInterface();
```

그림 4 전략 패턴 구현 예제

ContextInterface 함수는 Context 객체가 실제 생성한 ConcreteStrategy 객체의 기능을 수행한다. 해당 패턴은 실행 시간에 알고리즘을 변경할 수 있는 구조를 제공하며, 블랙박스 방식의 프레임워크를 구현할 수 있는 핵심이 된다. 앞에서 다룬 패턴들 외에 생성 패턴에서는 팩토리 메서드(Factory Method) 패턴 및 단일체(Singleton) 패턴, 구조 패턴에는 복합체(Composite) 패턴, 행동 패턴에는 명령(Command) 패턴 및 감시자(Observer) 패턴 등이 프레임워크를 구현하는데 용이하게 사용된다.

일반적으로 프레임워크의 초기 버전은 설계 또는 구현이 용이한 화이트박스 방식의 프레임워크로 구현되지만, 어플리케이션 개발의 유연성을 높이기 위해서 최종 버전에서는 블랙박스 방식의 프레임워크로 구현된다.

4. 적용 사례

개발 중인 분산 시뮬레이션 프레임워크는 컴포넌트를 통합하여 어플리케이션을 개발하는 방식으로 사용자 요구사항의 추가 또는 삭제가 용이한 구조를 갖는다[4]. 프레임워크를 기반으로 구현된 컴포넌트 중 분산 객체 통신 컴포넌트는 TCP/UDP, RTI(Run Time Infrastructure) 및 DDS(Data Distribution Service)와 같은 통신 매체들의 재사용성을 높이고, 동일 군의 추가 확장이 용이하도록 전략 패턴을 적용했다. 컴포넌트의 구조는 그림 5와 같다.

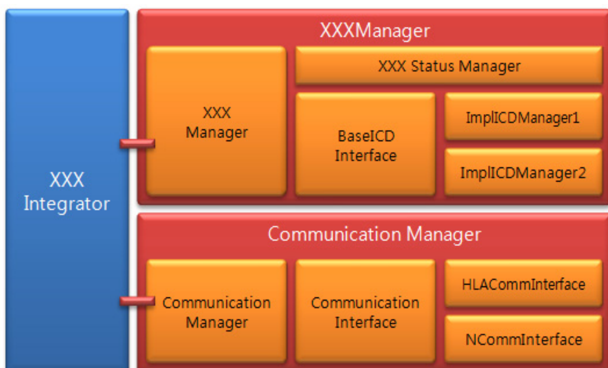

그림 5 개발 컴포넌트 Prototype 구조

컴포넌트 단위인 Manager는 Integrator 컴포넌트를 통해 통합되어 하나의 어플리케이션으로 구현된다. 사례로 소개할 Communication Manager 컴포넌트는 RTI 서비스를 제공하는 HLACommInterface 모듈과 TCP/UDP 서비스를 제공하는 NCommInterface 모듈로 구성된다. DDS 서비스는 현재 구현 중에 있다. 패턴 구조와 비교하면 CommunicationManager는 Context 클래스를 구현하고, CommunicationInterface는 Strategy 추상클래스를 정의한다. 통신 매체인 HLACommInterface와 NCommInterface는 ConcreteStrategy 클래스를 구현한다. 적용 구조와 구현 코드 예제는 그림 6, 7과 같다.

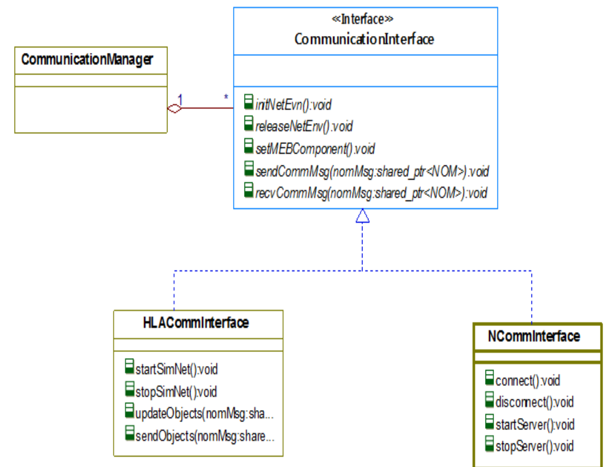

그림 6 전략 패턴이 적용된 구조

```
// Ncomm 객체를 Strategy로 선택
CommunicationInterface* commInterface;
commInterface = new NCommInterface;

//
CommunicationManager::recvMsg()
{
    ... ...
    commInterface->sendCommMsg();
    ... ...
}
```

NCommInterface(Strategy)의 sendCommMsg(Algorithm)을 수행

그림 7 전략 패턴이 적용된 코드

CommunicationInterface 추상클래스의 *sendCommMsg*와 *recvCommMsg* 함수는 패턴의 AlgorithmInterface 함수로서 CommunicationManager는 해당 함수를 통해 통신 서비스에 접근하게 된다. 따라서 동일한 알고리즘 즉, 분산 통신 기능을 제공하는 DDS 서비스가 Communication Interface 추상클래스에서 제공하는 기능들만 구현하면 개발 컴포넌트들은(Context) 코드 수정 없이 서비스를 사용

할 수 있다. 추가적으로 그림 5의 BaseICDInterface 추상 클래스를 구현하는 ImplICDManager 클래스 또한 전략 패턴을 적용했다.

5. 결론

디자인 패턴의 행동 패턴 중 하나인 전략 패턴은 동일 알고리즘 군에 대한 추상클래스를 정의하고, 해당 추상클래스의 서비스를 구현한 객체들을 동적으로 변경할 수 있는 이점을 제공한다. 본 논문은 전략 패턴을 적용하여 재사용 및 확장성을 늘릴 수 있는 프레임워크의 구현 컴포넌트를 소개했다. 좋은 설계는 요구사항의 변경에 빠르게 대처하는 것이며, 다양한 패턴 중 전략 패턴은 이러한 문제를 유연하게 대처할 수 있는 해법을 제공했다.

참고문헌

[1] Len Bass, Paul Clements, Rick Kazman, "Software Architecture in Practice, 2nd", Addison-Wesley, 2003

[2] Gamma, E., Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995

[3] Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson, "Building Application Frameworks: Object-Oriented Foundations of Framework Design", Wiley, 1999

[4] 심준용, 이용헌, 김세환, “컴포넌트 재사용을 위한 DLL 플러그인 프레임워크 설계”, 한국정보처리학회 추계 학술발표대회 논문집 제17권 제1호, 2010