브리지 패턴을 사용한 커넥터 아키텍처의 커넥션 메커니즘 확장

°채정화* 전형수* 유철중* 장옥배*
전북대학교 컴퓨터과학과
{jhchae, hsjeon}@cs.chonbuk.ac.kr
{cjyoo, okjang}@moak.chonbuk.ac.kr

Extension of Connection Mechanism on Connector Architecture using the Bridge Pattern

°Jung-Hwa Chae* Hyung-Su Jeon* Cheol-Jung Yoo* Ok-Bae Chang*
Dept. of Computer Science, Chonbuk National University

요 약

J2EE(Java™ 2 Platform, Enterprise Edition) 커넥터 아키텍처(Connector Architecture)는 J2EE 플랫폼을 다양한 EIS와 연결하기 위하여 공통 클라이언트 인터페이스(CCI; Common Client Interface)를 정의한다. CCI의 커넥션 메커니즘은 추상 팩토리 패턴(Abstract Factory Pattern)을 따르고 있다. 추상 팩토리 패턴을 이용함으로써 커넥터 아키텍처는 각기 상이한 커넥션과 커넥션 팩토리가 독립적인 벤더들에 의해 독립적으로 구현될 수 있게 한다. 브리지 패턴(Bridge Pattern)은 추상화 정도와 구현에 따라 추상 클래스와 구현 클래스를 별도의 클래스로 구현하여 이들이 동적으로 조합되도록 한다. 본 논문에서는 추상 팩토리 패턴을 따르고 있는 CCI의 커넥션 메커니즘을 브리지 패턴을 적용하여 확장된 커넥션 메커니즘을 제시한다. 추상 팩토리 패턴은 클래스의 생성과 관련있는 생성 패턴(*Creational Pattern*)인 반면 브리지 패턴은 구조 패턴(Structural Pattern)이다. 즉, 대행하는(delegation) 클래스의 행위들을 중계해 준다. 구조 패턴은 새로운 기능을 구현하기 위해 객체를 구성하는 방식에 초점을 두며, 실행 시에 객체 컴포지션 구조를 변경할 수 있어 이를 통해 유동성과 확장성을 추가할 수 있다.

1. 서론

J2EE 커넥터 아키텍처는 J2EE 플랫폼을 다양한 EIS와 연결하기 위한 표준을 정의하는 API이다[1,2]. 즉, 커넥터 아키텍처는 J2EE 플랫폼을 이기종의 EIS와 연결하기 위한 표준 API로써 EIS와의 연결을 위한 클라이언트 API인 공통 클라이언트 인터페이스(CCI)와 시스템 레벨 소프트웨어 드라이버인 시스템 컨트랙트(System Contracts)를 정의한다. EIS 벤더는 CCI와 시스템 컨트랙트를 EIS에 맞게 구현한 리소스 어댑터(Resource Adaptor)를 제공한다[1,2]. 리소스 어댑터는 이 어댑터를 사용하는 비즈니스 로직에 대해 CCI라는 공통적인 인터페이스를 제공하여 단일한 로직으로 ERP 및 TP 시스템에 접근하도록 한다.

CCI의 커넥션 메커니즘은 추상 팩토리 패턴을 따르고 있다. 추상 팩토리 패턴은 클래스의 생성에 관계하는 생성 패턴이다. 반면 브리지 패턴은 객체를 생성하는 방식에 초점을 두는 구조 패턴으로서, 구체적인 구현에 대한 상속관계와 세부적인 구현에 의존하지 않는 추상화에 대한 상속관계가 서로 분리되는 경우 사용된다.

본 논문에서는 생성 패턴인 추상 팩토리 패턴을 따르는 CCI의 커넥션 메커니즘을 구조 패턴의 하나인 브리지 패턴을 이용하여 적용해 봄으로써 확장된 커넥션 메커니즘을 제시하고자 한다.

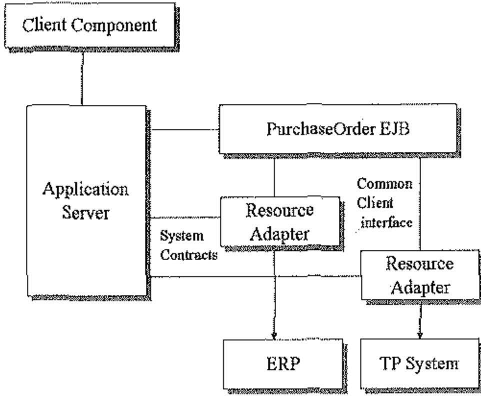
본 논문의 구성은 다음과 같다. 2장에서는 J2EE 커넥터 아키텍처의 개괄적인 내용과 CCI의 커넥션 메커니즘에 적용된 추상 팩토리 패턴을 살펴본다. 3장에서는 CCI의 커넥션 메커니즘에 브리지 패턴을 적용하여 결과를 도출한다. 4장에서는 결론 및 향후 연구 방향에 대해서 언급한다.

2. 관련연구

EIS 벤더는 커넥터 아키텍처를 따르는 표준 리소스 어댑터를 제공한다. 리소스 어댑터는 이 어댑터를 사용하는 클라이언트 애플리케이션, 웹 컴포넌트, *Enterprise* JavaBeans 컴포넌트 등과 같은 엔터프라이즈 애플리케이션에 대해 CCI라는 공통적인 인터페이스를 제공한다. 엔터프라이즈 애플리케이션은 CCI를 통해 단일한 로직으로 ERP 및 TP 시스템과 같은 EIS(Enterprise Information System)에 대한 접근을 일반화한다. 또한 애플리케이션 서버는 리소스 어댑터에 정의되어 있는 시스템 컨트랙트를 구현함으로써 어떠한 리소스 어댑터도 애플리케이션 서버에 플러그인되어 동작할 수 있도록 한다([그림 1])[1].

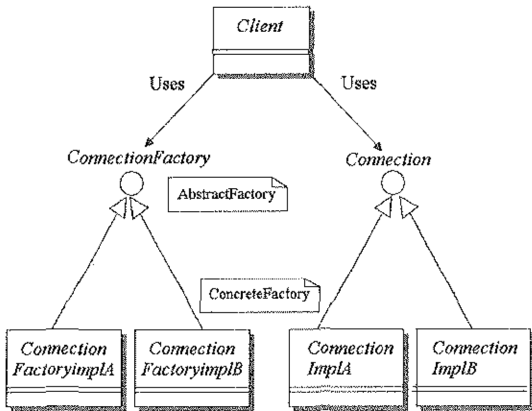
CCI에는 EIS에 연결할 수 있는 커넥션 인터페이스

(Connection Interfaces), 특정 기능을 호출할 수 있는 인터렉션 인터페이스(Interaction Interfaces) 및 특정 정보를 주고받을 수 있는 데이터 표현 기능(Data representation-related Interfaces)과 EIS의 리소스 어댑터의 정보를 얻을 수 있는 메타 데이터 기능을 담당하는 메타 데이터 인터페이스(Metadata-related Interfaces)가 있다.



[그림 1] 커넥터 아키텍처에서의 EIS 개발 구조도

CCI의 커넥션을 얻는 메커니즘인 커넥션 인터페이스는 추상 팩토리 패턴을 따르고 있다([그림 2]). 추상 팩토리 패턴의 *AbstractFactory* 클래스는 사용자 인터페이스를 대표하는 각 추상 클래스의 인스턴스를 생성하는 메소드를 정의한다[4,5]. AbstractFactory 클래스와 이를 상속받는 ConcreteFactory 클래스를 이용해 다른 환경에서 각기 동일한 기능을 수행하는 구체적 클래스를 구성하게 된다.



[그림 2] 추상 팩토리 패턴을 적용한 CCI의 커넥션 메커니즘

추상 팩토리 패턴에서 팩토리 기능을 하는 클래스, 즉 AbstractFactory 클래스는 객체 생성 과정을 캡슐화하므로 클라이언트 클래스를 실제 기능 구현을 담당하는 클래스에서 분리시킬 수 있다. 결국 클라이언트 클래스는 추상적인 인터페이스를 통해 생성된 객체를 이용하게 된다. AbstractFactory 클래스는 프로덕트 그룹 전체를 생성하므로 전체 프로덕트 클래스를 한 번에 변경할 수 있다.

3. 브리지 패턴을 사용한 커넥터 메커니즘의 확장

CCI의 커넥션을 얻는 메커니즘은 추상 팩토리 패턴을 따르고 있다. [그림 2]에서 *ConnectionFactory* 클래스는 커넥션을 얻는 통로 역할을 수행하는 인터페이스이며 JNDI(Java Naming and Directory Interface)를 통해 얻어진다. 팩토리 구현 클래스를 얻을 때는 얻고자 하는 클래스 명을 명시한다. 각각의 팩토리 구현 클래스인 *ConnectionFactoryImplA*, *ConnectionFactoryImplB*는 getConnection() 메소드를 통해 각각에 맞는 *ConnectionImpl*을 돌려준다. 즉, *ConnectionFactory*는 추상 팩토리 역할을 하며, *Connection*은 추상 프로덕트의 역할을 하게된다. 커넥터 아키텍처는 추상 팩토리 패턴을 이용함으로써 각기 상이한 *Connection*과 *ConnectionFactory*를 독립적인 벤더들이 독립적으로 구현할 수 있게 한다.

커넥터 아키텍처를 구현한 리소스 어댑터는 각기 다른 EIS 벤더가 제공하고 이것은 애플리케이션 서버에 배치된다[1]. 엔터프라이즈 애플리케이션은 다양한 EIS에 접근하기 위하여 이를 실행 시에 사용한다. 이러한 실행 관점에서 보면 객체 컴포지션 구조를 실행시에 변경할 수 있는 특징을 갖는 구조 패턴인 브리지 패턴을 적용하면 커넥터 아키텍처의 커넥션 메커니즘에 유동성과 확장성을 추가할 수 있게 된다.

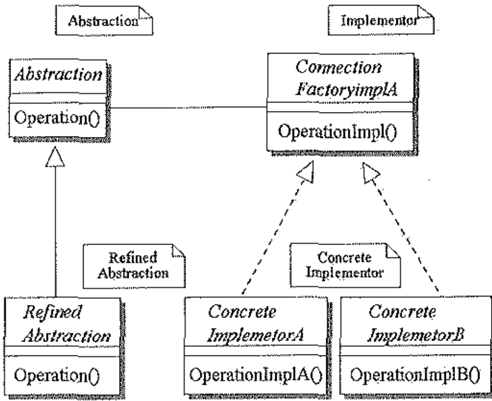
커넥터 아키텍처의 커넥션 인터페이스들이 어떻게 브리지 패턴에 적용되어 커넥션 메커니즘을 확장시킬 수 있는지 살펴보면 다음과 같다.

브리지 패턴은 추상화 정도에 의한 계층으로 구분되어 있고 또 이를 구현하는데 있어 몇 개의 계층으로 나누는 경우에 유용한 패턴이다. 추상화 정도와 구현에 따라 추상 클래스와 구현 클래스를 별도의 클래스로 구현해 이들이 동적으로 조합되도록 한다.

브리지 패턴이 적용될 수 있는 상황은 다음과 같다. 첫째, 추상화 단계에 따라 계층 구조가 존재하는 동시에 구현 관점에서 계층이 존재하는 경우, 이들 관점을 하나의 클래스 계층으로 묶기 위해 사용할 수 있다. 둘째, 추상화된 클래스의 실제 구현과 관련이 없는 공통된 로직을 재 사용할 때 이용할 수 있다. 셋째, 추상화된 여러 클래스가 하나의 구현을 공유하려 할 때 이용할 수 있다. 결국 브리지 패턴은 추상화된 클래스를 각각의 구현에서 독립시킬 수 있다. 또한 추상화된 클래스와 그 구현 부분은 서로 다른 클래스 구조를 갖도록 구성되며, 이를 통해 한 클래스 구조의 변경이 다른 클래스 구조에 영향을 미치지 않고 수행될 수 있다[4,5].

[그림 3]은 브리지 패턴을 도식화한 것인데, 여기서 중요한 관계는 추상 클래스인 *Abstraction* 클래스와 Implementor

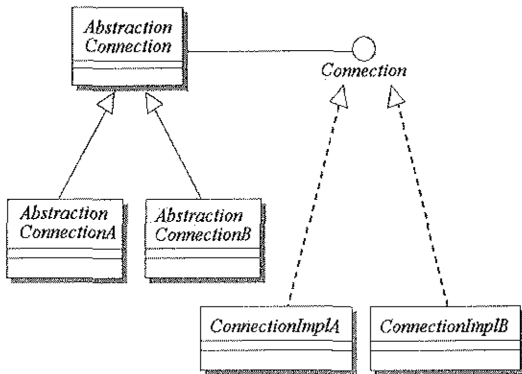
인터페이스 관계이다. Implementor를 구현하는 구현 클래스들인 *ConcreteImplementorA*, *ConcreteImplementorB*가 존재할 수 있다. *Abstraction* 클래스의 모든 행위들은 Implementor인 *ConnectionFactoryA*가 대리(delegation)하게 된다. 이렇게 하여 *ConnectionFactoryA*와 그 구현 클래스들에 상관없이 *Absraction* 클래스를 정제(refine)시킬 수 있다.



[그림 3] 브리지 패턴의 클래스 다이어그램

커넥션 아키텍처의 커넥션 인터페이스에 브리지를 만들어 이를 도식화하면 [그림 4]와 같다. 먼저 *Connection* 인터페이스에 대한 *AbstractionConnection* 클래스를 만든다.

이 *AbstractionConnection* 클래스는 *Connection의* 위임자(delegator) 역할을 한다. 클라이언트 입장에서는 어떤 커넥션 클래스의 인스턴스가 생성되는가는 알 필요가 없으며, 이것은 *AbstractionConnectionA* 또는 *AbstractionConnectionB* 등에 의해서 결정된다.



[그림 4] 브리지 패턴을 사용한 커넥터 아키텍처의 커넥터 메커니즘 확장

추상 팩토리 패턴을 이용하면 여러 클래스 팩토리들을 동일한 코드로 조작할 수 있다. 여기서 추상 팩토리 패턴은 생성 패턴이라는 점에 주목할 필요가 있다. 즉, 클래스의 생성에 관계한다.

생성 패턴은 클래스 정의와 객체 생성 방식을 구조화 및 캡슐화하는 방법을 제시한다. 즉, 객체 생성 과정을 추상화시킨다는 특성을 갖고 있으며, 객체가 서로 연동되는 방법을 내부 구조에 숨긴다. 이로 인해 생성 패턴을 이용한 설계 구조는 다소 복잡해질 수 있다.

반면 구조 패턴은 생성 패턴과 달리 새로운 기능을 구현하기 위해 객체를 구성하는 방식 자체에 초점이 맞춰져 있다. 구조 패턴의 특징은 실행 시에 객체 컴포지션 구조를 변경할 수 있는데, 이를 통해 유동성과 확장성을 추가할 수 있다.

브리지 패턴은 구조 패턴으로서 Abstraction 클래스의 행위를 대리하는 Implementor 클래스의 행위들을 중계해준다. [그림 4]에서 *AbstractionConnection* 클래스는 *Connection* 인터페이스의 모든 행위들을 위임한다.

4. 결론

본 논문에서는 추상 팩토리 패턴을 따르고 있는 CCI의 커넥션 메커니즘을 브리지 패턴을 이용하여 적용해 봄으로써 확장된 커넥션 메커니즘을 제시하였다.

추상 팩토리 패턴은 클래스의 생성과 관련있는 생성 패턴이며, 브리지 패턴은 객체를 구성하는 방식에 초점을 두는 구조 패턴이다. 구조 패턴은 생성 패턴과 달리 새로운 기능을 구현하기 위해 객체를 구성하는 방식 자체에 초점을 둔다.

J2EE 환경의 엔터프라이즈 애플리케이션은 다양한 EIS에 접근하기 위하여 리소스 어댑터를 실행 시에 사용한다. 이러한 실행 관점에서 보면 객체 컴포지션 구조를 실행시에 변경할 수 있는 특징을 갖는 구조 패턴인 브리지 패턴을 적용하면 커넥터 아키텍처의 커넥션 메커니즘에 유동성과 확장성을 추가할 수 있게 된다.

향후 연구과제로는 CCI의 커넥션 인터페이스에 추상 팩토리 패턴을 적용한 경우와 브리지 패턴을 적용한 경우를 비교 분석하여 정량적인 결과를 도출해 보는 것이다.

참고문헌

[1] 채정화, 유철중, 장옥배, "J2EE 커넥터 아키텍처에서의 리소스 어댑터 배치 방법", 2000 한국정보과학회 추계 학술발표논문집, Vol. 27. No. 2, 2000.

[2] Sun Microsystems, Inc., "JavaTM 2 Platform Enterprise Edition Specification," vol. 2, 1999.

[3] Sun Microsystems, Inc., "J2EE Connector Architecture Specification," v1.0, June 1, 2000.

[4] Patterns in Java Volume 1, Mark Grand, Wiley Press, 1999.

[5] Java DESIGN PATTERNS, James W. Cooper, Addison Wesley Press, 2000.