CORBA를 이용한 제품 정보 관리 시스템의 설계와 개발

정 철주　이 상덕
한국전자통신연구원 컴퓨터·소프트웨어연구소
Email: {cjeong,lsd}@etri.re.kr

Design and Implementation of A PDM System Using CORBA

Cheoljoo Jeong　Sangduck Lee
Computer & Software Technology Laboratory
Electronics and Telecommunication Research Institute

요 약

본 논문에서는 CORBA와 Java를 이용하여 분산 객체 서버 형태의 응용 프로그램인 제품정보관리 시스템을 설계하고 구축한 사례를 소개한다. 이 시스템은 Visibroker 3.4 for Java와 JDK 1.2.2를 이용하여 개발되었으며 데이타베이스로는 Oracle 8i를 사용하였다. 특히, 지속성 객체 (persistent object) 의 지원을 위하여 추가의 추상화 계층 (abstraction layer) 를 구현하고 있으며 전체 시스템은 객체 지향 프레임워크 (object-oriented framework) 으로 설계되었다. 본 문서에서는 제품정보관리 시스템 뿐 아니라 다른 분산객체서버의 설계 및 구현에도 참고가 될만한 설계 가이드라인들을 제시한다.

제 1 절　서론

제품정보관리 (Product Data Management, 이하 PDM) 시스템은 제품의 구성 정보, 부품 설계 정보, 부품 변경 관리 (change management) 등의 서비스를 지원하는 시스템이다. 본 논문에서는 CORBA와 Java를 이용하여 설계하고 구현한 PDM 시스템의 예를 소개한다. 소개되는 PDM 시스템은 크게 기본 기능 (로그인 관리, 파일 관리), 조직관리 (사용자 관리, 부서 관리, 회사 관리, 사용자그룹 관리), 부품 관리 (부품 관리, 공급업체 관리), 문서 관리 (문서 관리, 폴더 관리, 분류체계 관리), 그리고 결재 관리, 전자 게시판 관리, 전자 우편 관리 등을 위한 서브시스템들로 이루어져 있다. 각 관리 시스템은 하나 이상의 관리 대상을 가진다. 예를 들어 사용자 관리 모듈은 '사용자' 객체들을 관리하게 되며, 부품 관리 모듈은 '부품 마스터' 객체와 '부품' 객체들을 관리하게 된다.

본 시스템에서는 별도의 CORBA 서비스를 사용하지 않고 객체 지속성 (object persistence), 동기화 (synchronization), 버전 관리 (version control) 등의 서비스를 설계 및 구현하였으며, 이를 위해 다양한 설계 패턴 [4] 을 사용하였다. 또, 실행 효율을 위하여 객체 풀 (object pool) 을 사용하였으며, 시스템의 확장을 용이하게 하기 위하여 전체 시스템을 객체 지향 프레임워크 (object-oriented framework) 으로 설계하였다.

본 논문의 구성은 다음과 같다. 먼저 2절에서는 시스템 설계 상의 중요한 결정 사항과 결정에 대한 근거를 제시한다. 그리고 3절에서는 전체 시스템의 대략적인 객체 모형을 설명하고, 4절에서는 이 모형을 이용하여 실제 분산 객체 서비스가 어떻게 제공되는지를 설명한다. 마지막으로 5절에서는 결론과 더불어 향후 연구 방향을 제시한다.

제 2 절　설계 상의 주요한 결정 사항

2.1　분산 객체

본 시스템에서는 실제 객체들이 서버 내에 존재하게 된다. 예를 들어, 사용자 관리 시스템에는 실제 `User` 클래스의 인스턴스들이 메모리 상에 위치하게 된다. 현재, 상당수의 CORBA-기반 시스템에서 몇 개의 '관리자 객체'들만을 메모리 상에 위치시키는 것과는 달리 관리자 객체 외에 관리되는 객체 (managed object) 들까지 서버에 생성되어 클라이언트의 메시지에 대한 서비스를 제공한다.

사실, 이와 같이 실제 객체들이 서버 상에 생성되어 존재하는 것이 당연하나 모든 객체들이 이와 같이 서버 상에 생성될 경우, 메모리 부족 현상이 발생하기 때문에 상당수의 CORBA 서버들은 다른 접근 방식을 사용하고 있다*. 본 시스템에서는 이와 같은 문제점을 극복하기 위해 객체 풀 (object pool) 을 이용하여 동시에 서버 메모리 상에 위치하는 서버 객체들의 수를 제한시키는 방법을 사용하였다. 이 방법은 장점과 단점을 가지게 되는데, 이는 실제 객체가 서비스하도록 함으로써 DB나 파일 접근을 필요로 하지 않기 때문에 실행 속도가 빨라진다는 점과 제대로 된 객체 지향 설계를 가능케 한다는 점이다.

대신, 일치성 (consistency) 과 같은 문제를 해결해 주어야 하므로 구현의 복잡도가 높아지게 된다. 자세한 사항은 다음 절에서 다루기로 한다.

2.1.1　객체 풀

본 시스템은 내부에서 객체 풀들을 유지하도록 설계되었는데, 객체 풀들을 통해 얻을 수 있는 잇점들은 다음과 같다.

*예를 들어, `struct`를 통해 객체의 모든 정보를 클라이언트에 넘겨 주며 모든 서버의 서비스는 관리자 객체들을 통하도록 하는 방식

- 객체 생성 (Object construction) 과 초기화에 필요한 시간을 절약할 수 있다.
- 객체가 수정될 때마다 이 객체의 수정 사항을 지속성 저장매체에 저장할 필요가 없으므로 DBMS 나 파일 시스템 접근 횟수를 줄일 수 있다.

객체 풀은 객체 지속성과 연관하여 객체들의 캐시 역할을 한다. 각 관리자 객체들은 `Object getObject(Key)`와 같은 연산을 지원하는데[†] 이 연산의 구현은 만약 해당 키의 객체가 객체 풀 내에 있을 경우에는 이 객체를 풀에서 찾아 그 객체 참조 (object reference) 를 돌려주고, 그렇지 않을 경우에는 지속성 저장매체에서 이 객체의 정보를 읽은 후, 새로운 CORBA 객체를 만들어 이에 대한 참조를 돌려준다.

객체 풀의 가장 큰 효용은 DB 접근과 같은 시간이 많이 걸리는 연산들의 횟수를 줄여주는데 있다. 그러나, 앞에서 설명했듯이 본 시스템에서는 서버 객체의 수를 조절하는 역할도 한다.

2.1.2 객체의 일관성의 유지

이제 하나의 객체에 대해 객체 풀과 지속성 매체 두 군데에 이 객체의 상태가 유지될 수 있다. 이 결과로 두 상태간의 불일치가 발생할 수 있는데, 이 불일치로 인한 파국을 막아주는 것은 매우 중요하다. 이를 위해 다음의 한 가지 원칙이 사용된다.

> 하나의 객체의 상태가 객체 풀과 지속성 매체 두 군데에서 유지될 때, 객체 풀에 존재하는 객체의 상태가 항상 우선적으로 고려된다.

이렇게 두 상태간의 불일치가 문제가 되는 경우는 몇가지 예가 있는데, 대표적인 것이 `getAllArticles()`와 같이 직접 DB에서 읽어오는 것이 유리한 연산들이다. 풀에 모든 `Article` 객체들이 들어 있는 것이 아니므로 DB의 `ArticleTbl`을 검색하여 검색 결과를 돌려주는 것이 바람직한데, 이 경우, 돌려지는 값은 풀에 있는 객체의 값과 다른 값을 가질 수 있기 때문이다.

이를 해결하는 방법은 여러가지가 있을 수 있겠지만 본 구현에서 사용한 방법을 설명하자면 다음과 같다. 모든 게시글들의 목록을 가져온 후, 이 게시글 중, 객체 pool에도 존재하는 것들이 있는지 확인한다. 만약 그러한 경우에는, 해당 객체의 상태가 `_MODIFIED`인 경우, 게시글의 목록에서 해당 게시글을 이 객체의 정보로 덮어 쓴다 (overwrite). 풀 객체들의 내용이 반영된 게시글들의 목록을 클라이언트에게 전달한다.

2.2 지속성 객체

지속성 객체 (Persistent object) 란 객체가 메모리에서 사라진 이후에도 객체의 상태, 또는, 필드의 값이 사라지지 않고 유지되는 객체를 의미한다. 일반적으로 객체의 상태는 DB나 파일과 같은 매체에 저장이 된다.

응용 프로그램 수준에서 객체의 지속성를 지원하기 위해서는 객체의 생성주기 (lifetime) 에 대한 제어를 응용프로그램에서 해주어야 한다. 즉, 추가의 계층을 두어, 지속성 서비스를 지원해야 하는데, 현재 CORBA 서비스에 지속성 서비스가 존재하지만 본 시스템에서는 직접 구현하고 있다.

[†]실제로는, 게시판 관리 시스템의 경우, `Article getObject(int articlekey)`와 같은 이름을 가진다.

2.2.1 DB 키 관리

이제, 하나의 객체를 판별 (identify) 하기 위해서는 객체의 클래스와 해당 클래스의 인스턴스들 내에서 특정 객체를 판별하는 키가 필요하다. 본 시스템에서 키는 지속성 객체의 저장을 위해 사용하는 DB에서의 테이블 키를 그대로 사용하고 있다.

실제, 각 `ObjectImpl` 클래스의 객체들은

```
String  getObserverName();
```

이라는 연산을 가지게 되는데, 이 연산 자신의 클래스의 이름과 해당하는 테이블의 키 값을 결합 (concatenate) 한 결과를 돌려주는 연산으로 구현된다.

2.3 Tie 메카니즘의 사용

Java의 경우, 다중 상속 (multiple inheritance) 를 지원하지 않으므로 CORBA 응용 프로그램의 개발시 코드 재사용 (code reuse) 에 큰 문제점이 있다. 이는 Java의 객체 구현 (object implementation) 이 이미 `omg.orb.CORBA.Object`를 상속받기 때문에 다른 클래스를 상속받을 수 없다는 점이다. 물론 새로운 클래스 C를 만들어 C가 `omg.orb.CORBA.Object`를 상속하게 한 후, 이 C를 상속하면 되기는 하지만 이 경우, CORBA에 독립적인 수퍼 클래스를 가질 수 없게 된다. 이는 개념적으로 큰 손실이 되는데, 이를 해결하기 위해서 사용되는 방법이 객체의 조합 (composition, delegation) 을 이용하는 방법이다 (자세한 내용은 [7]을 참고할 것).

제 3 절 시스템의 설계

3.1 로그인의 관리

본 시스템의 특징 중의 하나가 로그인의 관리에 있다. 상당수의 시스템에서 로그인 관리가 사용자 객체에 몇 개의 필드를 두어 관리되는 것과는 달리, 로그인이 '객체'의 형태로 존재하게 된다. 본 시스템에서 하나의 로그인은 한명의 사용자가 '로그인'해서 '로그아웃'하기 전까지, 이 사용자의 작업에 관련된 모든 정보를 나타낸다. 실제 시스템에서는 A라는 사용자가 로그인을 하면 하나의 `Login` 객체가 생성이 되는데, 이 사용자의 작업에 대한 로그, 이 사용자가 다른 객체에 대해 얻은 락 (lock) 등 다양한 정보가 이 `Login` 객체 내에 저장이 된다. `Login` 객체는 지속성 객체가 아닌데, 만약 사용자가 로그아웃을 하거나 오류에 의하여 이 사용자의 클라이언트와 서버 간의 통신이 끊기거나 서버 오류 (server failure) 가 발생하는 경우, 해당 `Login` 객체는 완전히 사라지게 된다.

Class `Login`

```
public class Login implements LoginOperations,
                              ObserverI {
   ...
   public synchronized void addLock(double lock) ...
   public void removeLock(double lock) ...
   public synchronized void checkLock(double lock) ...
   public void unsubscribe(ObserverI o) ...
   public void notifyObservers() ...
   ...
}
```

위의 operation 중 addLock(), removeLock(), checkLock()은 개별 ObjectImpl 객체에 대한 락 관리를 위해 필요한 연산이고 나머지 두 연산은 ObserverI 인터페이스를 구현 (implements) 하는데서 필요한 연산이다.

3.2 락 관리

하나의 CORBA 객체 구현, 예를 들어, 키 34를 가지는 하나의 ArticleImpl 객체 o를 생각해보자. o를 동시에 두 사용자가 수정을 위해 접근할 가능성이 존재한다. 이때 발생할 수 있는 문제들을 방지하기 위해 o를 수정하려는 클라이언트는 먼저 이 객체에 락을 걸어야 한다. ArticleImpl 클래스는 ObjectImpl 클래스를 상속하며 ObjectImpl 클래스는 아래에서 설명되는 바와 같이 addLock(), removeLock() operation이 제공되므로 클라이언트에서는 이 연산들을 사용하면 된다.

Login 클래스의 addLock() 등의 연산은 client에 의해 사용되는 것이 아니고 'ObjectImpl의 addLock() 등의 연산의 구현'을 위해 사용된다. 만약, 사용자가 o에 대하여 락을 걸게 되면, 시스템에서는 LockGenerator를 통해 unique한 실수 하나를 생성한다. 이 실수는 두 군데에 저장이 되는데, 하나는 해당 ObjectImpl 객체 내에, 그리고하나는 락을 건 클라이언트에 해당하는 Login 객체 내에 저장이 된다.

이제, 추후, 실제로 객체를 수정하는 연산이 이루질 때마다, 서버 측에서는 해당 클라이언트의 로그인이 가진 수와 이 객체의 수가 동일한 지를 검사함으로써 이 클라이언트가 이 객체를 수정할 수 있는지의 여부를 판단한다.

3.3 오류의 처리

시스템 수행 도중 다양한 이유로 인하여 클라이언트나 서버의 오류가 발생할 수 있다. 이 모든 경우에 대하여 대책을 마련하는 것은 극히 어려우나 몇 가지 중요한 오류에 대해서는 오류 핸들러를 준비하는 것이 바람직하다.

본 절에서는 클라이언트 오류가 발생하는 경우에 대한 핸들러의 구현에 대해 설명하겠다. 클라이언트 오류는 클라이언트 프로그램의 사용자가 강제로 클라이언트 수행을 중단시키거나 통신 오류에 의하여 서버 객체가 자신에 연결된 클라이언트와 연결이 끊어지는 경우 발생한다.

본 시스템에서는 대부분의 CORBA ORB에서 제공하는 이벤트 핸들러 서비스를 사용한다.

Observer 패턴을 이용한 자원 수거 A라는 사람이 로그인하면 서버에서는 A에 대한 Login 객체 (예를 들어, $login_A$)가 생성된다. 이미 설명했다시피, 이 $login_A$에는 A가 접근하게 되는 다양한 CORBA 객체들에 대한 락 등의 정보가 저장된다.

중요한 것은, A가 특정 CORBA 객체 o에 대하여 어떠한 메시지를 보낼 경우, 이 메시지에 대한 메소드의 수행은 DB 접근을 유발할 수 있다는 것이다. 그런데, 만약 DB 접근이 수행되는 도중 클라이언트 오류가 발생하는 경우에는 어떻게 해야 하는가? 본 시스템 (그리고, 대부분의 다른 시스템) 에서는 DB 커넥션 (connection) 을 별도로 관리해주고 있다. 즉, DB 접근마다 새로운 커넥션을 만들어내기보다는 미리 만들어진 커넥션들의 풀로부터 커넥션을 얻어서 사용하는 방법을 사용하고 있다. 결국, DB 접근 도중 클라이언트 오류가 발생하면 자동으로 커넥션의 release가 이루어져야 하는 것이다.

이러한 경우에 대한 대처를 위하여 ObserverI와 ObservableI의 두 인터페이스를 정의한다 (Observer 패턴에 대한 자세한 내용은 [4]을 참고하기 바란다).

```
interface ObserverI {
   String getObserverName();
   void doAction();
}

interface ObservableI {
   void subscribe(ObjectImpl o);
   void unsubscribe(ObjectImpl o);
   void notifyObservers();
}
```

본 시스템의 구현에서 Login 클래스는 ObservableI 인터페이스를 구현 (implements) 하도록 정의되며, 임의의 ObjectImpl 클래스는 ObserverI 인터페이스를 구현 (implements) 하게 된다.

따라서, LoginImpl 클래스의 구현시 만약 이상 상황이 발생하면 적당한 곳에 (실제 구현시에는, Visibroker의 이벤트 핸들러 서비스 내에 위치)

```
LoginImpl.notifyObservers();
```

를 부르면, 해당 Login 객체가 가지고 있던, 모든 자원는 수거 (deallocate) 된다.

실제 시스템에서는, ObserverI를 구현하는 ObjectImpl 클래스[‡] 내에 이에 관련된 코드들을 미리 구현하여 AritlceImpl이나 UserImpl과 같은 실제 클래스에서는 이 코드를 상속만 받으면 되도록 되어 있다.

3.4 추상 클래스 ObjectImpl

먼저 모든 객체 구현은 추상 클래스인 ObjectImpl을 상속한다. ObjectImpl은 지속성 객체가 지원해야할 다양한 연산들을 정의하고 있으며, 락킹 (locking) 의 단위가 되기도 한다.

```
abstract class ObjectImpl {
   final static int _NOT_MODIFIED = 0;
   final static int _MODIFIED     = 1;
   final static int _NEW          = 2;

   final static int _UNLOCKED     = 0;
   final static int _LOCKED       = 1;

   // state variables
   protected int _objectStatus;
   protected int _lockStatus;

   // important operations
   void getLock(Login login) { ... }
   void releaseLock(Login login) { ... }
   boolean isOkToMutate(Login login) { ... }
   void save(Login login) { ... }

   // misc operations
   abstract DBAccessor getDB();
}
```

본 시스템의 설계에서 ObjectImpl 객체는 다음과 같은 특징을 가진다.

[‡] ObjectImpl 클래스 역시 추상 (abstract) 클래스이다

- ObjectImpl 객체는 수정 여부를 나타내는 상태를 가지는데, 이 정보는 _objectStatus에서 유지된다.
- ObjectImpl 객체는 락킹의 단위가 된다. 이를 위해 현재 이 객체가 락이 걸려있는지의 여부를 나타내는 변수 (_lockStatus) 를 가지며 락킹을 위한 두개의 연산 getLock(Login)과 releaseLock(Login)을 가진다.
- ObjectImpl 객체는 현재의 상태를 지속성 저장매체에 저장할 수 있다.

가장 중요한 두 가지의 연산은 isOkToMutate(Login)과 save(Login)이다. isOkToMutate()는 인자로 주어진 Login 객체의 주인이 이 객체를 수정할 권한이 있는지를 검사한 후, 그 결과를 참/거짓 값으로 돌려주는 연산이다. 또, save(Login)은 현재 객체의 상태를 지속성 저장매체에 저장하는 연산이다. 이 연산은 현재의 객체 상태 (_objectStatus) 의 값에 따라 적당한 저장 작업을 수행한다.

3.5 지속성 객체 구현의 예

위의 ObjectImpl 클래스를 상속하여 전자게시판의 각 게시글을 나타내는 ArticleImpl 클래스를 구현한 예는 다음과 같다.

```
class ArticleImpl extends ObjectImpl {

   void setArticleBody(Login login, String body) {
      if (isOkToMutate(login)) {
         _body = body; // update
      }
   }
}
```

3.6 추상 클래스 ObjectImpl의 구현

3.6.1 연산 save(Login)의 구현

지속성 저장매체로서 DBMS를 사용한다고 가정하겠다. save() 연산은 현재 객체의 상태 _objectStatus의 값에 따라 다른 행위를 보인다. 만약 이 객체가 새로 생성된 객체이면, DB에 insert 명령을 사용해야 할 것이고 그렇지 않고 이미 DB에 존재하는 객체이고 다만 객체의 상태가 수정된 것이라면 update 명령을 사용해야 할 것이다.

그리고, 이 save() 연산이 추상 연산이 되지 않고 이 연산의 메소드를 하위클래스 (subclass) 에서 그대로 상속받을 수 있도록 하기 위하여 수상 연산 getDB()를 추가했다. 이 연산의 메소드는 ObjectImpl 클래스를 상속받는 하위클래스에 의해 제공된다. 이 getDB()는 DBAccessor 클래스의 인스턴스를 결과로서 돌려주는데 아래에서 볼 수 있듯이 이 클래스는 두 개의 연산 insert()와 update() 연산을 지원한다. 결국, save() 연산은 DBAccessor 객체를 얻어와서 (getDB()를 통해) 이 객체에 insert 또는 update 메시지를 보내면 되는 것이다.

```
abstract class DBAccessor {
   public abstract void insert(Login login);
   public abstract void update(Login login);
}
```

3.6.2 연산 isOkToMutate(Login)의 구현

isOkToMutate() 연산은 사용자가 ObjectImpl 객체를 수정하기 전에 수정할 권리를 가지는지를 검사해주는 연산이다. 이 연산에서 해주어야 할 일은 다음과 같다:

- 이 ObjectImpl 객체가 락이 걸린 상태인지 검사한다.
- 인자로 넘어오는 Login의 사용자가 이 ObjectImpl 객체에 대해 락을 걸었는지 확인한다.

중요한 점은 락을 거는 연산, getLock()에서 사용자가 해당 ObjectImpl에 대하여 락을 걸, 즉 수정을 할 권한 (authorization) 이 있는지를 검사해주어야 한다는 것이다.

3.7 버전 관리

추상 클래스 VersionObjectImpl 버전 관리를 위하여 ObjectImpl 클래스를 상속하여 다음과 같은 클래스를 정의한다.

```
abstract class VersionObjectImpl extends
                                  ObjectImpl {
   final static int _CHECKED_IN  = 0;
   final static int _CHECKED_OUT = 1;
   final static int _BASELINED   = 2;
   final static int _REVISED     = 4;

   // state variables
   protected int _objectStatus;
   protected int _vcStatus;
   protected int _lockStatus;

   // version control operations
   void checkin(Login login) ...
   ApprovalDataStruct checkout(Login login) ...
   void baseline(Login login) ...
   ApprovalDataStruct revise(Login login) ...

   // factory methods
   abstract void doCheckin(Login login);
   abstract ApprovalDataStruct doCheckout(Login login);
   abstract void baseline(Login login);
   abstract ApprovalDataStruct revise(Login login);
}
```

버전 관리는 checkout 등 4 개의 연산을 통해 지원된다. 이 연산들은 doCheckout 등 4 개의 팩토리 메소드 (factory method) [4] 를 사용하여 구현되는데, 각 팩토리 메소드는 VersionObjectImpl 클래스를 상속받는 실제 객체 구현에서 구현된다.

_vcStatus의 관리 _vcStatus는 현재 VersionObjectImpl이 어떤 상태에 있는지를 나타내는 변수이다. 이 변수가 가질수 있는 값은 다음과 같다:

- _CHECKED_IN: checkin() 연산이 성공적으로 끝난 경우, 이 상태를 가지게 된다. 이 상태에서는 checkout()과 baseline() 연산 만이 적용될 수 있다.
- _CHECKED_OUT: checkout() 연산이 성공적으로 끝난 경우, 이 상태를 가지게 된다. 이 상태에서는 checkin() 연산 만이 적용될 수 있다.

- _BASELINED: baseline() 연산이 성공적으로 끝난 경우, 이 상태를 가지게 된다. 이 상태에서는 revise() 연산 만이 적용될 수 있다.
- _REVISED: revise() 연산이 성공적으로 끝난 경우, 이 상태를 가지게 된다. 이 상태에서는 checkin() 연산 만이 적용될 수 있다.

3.8 결재 관리 모듈의 설계 및 구현

본 시스템에서는 버젼 관리와 결재 관리가 서로 결합된 형태로 구현되어 있다. 즉, checkin() 연산이 수행되면 바로 수정한 내역이 반영되는 것이 아니라 일단 결재를 타게 되며 결재가 완료된 후에야만 수정한 결과가 반영된다. 이를 위하여 본 시스템에서는 checkout() 연산의 리턴 타입이 ApprovalDataType이 되도록 하였다.

```
// Approval.idl
struct ApprovalDataType {
   long dataType;
   long dataKey;
};
```

이 IDL struct에서 dataType은 해당 결재가 문서에 대한 것인지 아니면 부품에 대한 것인지를 알려준다. 그리고 dataKey는 해당 결재 문서에 대한 키 값을 알려준다.

3.9 checkin() 연산의 구현

실제 checkout() (또는, revise()) 과 checkin() 연산이 이루어졌을 때, 실제 처리되는 일들은 다음과 같다.

체크아웃

1. 현재 부품의 마스터 번호가 P000이라 하자. 그리고 P000의 번호를 가진 VersionObjectImpl 객체가 m (m은 "부품 마스터" 객체이다) 이라 할 때, 이 m은 여러 개의 버젼을 가질 수 있다. 버전에 따라 달라지는 정보는 실제 $p_0, p_1, \cdots$와 같은 "부품" 객체들에 저장된다. 가장 최근의 부품이 p_k이고 p_k의 리비젼 (revision) 과 차수 (sequence) 가 각기 r, s일 때, m.checkout()가 실행되면 먼저 p_k와 동일한 정보를 가진 임시 "부품" 객체 p'_k이 생성된다. p'_k의 리비젼과 차수는 각기 $r, s+1$이 되며 unique한 부품 id를 부여받는다.
2. 돌려주는 값은 '부품'임을 나타내는 dataType 값과 p'_k의 부품 id 값을 가진 dataKey 값을 가진 ApprovalDataType struct 변수이다.

체크인 사용자는 돌아온 ApprovalDataType 내의 dataKey 값을 가지고 새로이 생성된 임시 객체 p'을 getPart(partid)의 연산을 이용하여 얻는다. 이 객체를 수정한 후, p'.checkin() 연산을 통해 수행한다. 이 때, checkin()의 인자로서 결재자 목록과 같은 정보가 넘어간다.

이 연산이 수행된 후에는 하나의 결재 객체, 즉 ApprovalImpl 객체가 생성이 되는데, ApprovalImpl은 다음과 같이 정의된다.

```
public class ApprovalImpl extends ObjectImpl
                implements ApprovalOperations
{
    private int _approvalId;
    private ApproverStruct[] _approvers;
    private ApprovalDataStruct _approvalData;

    public static final int _IN_PROGRESS = 0;
    public static final int _REJECTED    = 1;
    public static final int _APPROVED    = 2;

    private int _approvalStatus;
    private int _approverCursor;

    public void approve(Login login);
    public void reject(Login login);
    public void comment(Login login, String comment);
}
```

approve() 연산의 구현시, 최종 결재자에 의한 호출일 경우, 결재 행위가 완료되므로 모종의 조치가 이루어져야 한다. 예를 들어, checkin시 생성된 임시 객체를 정식 객체로 바꾸어주는 작업이 필요한데, 이를 위해, 임의의 VersionObjectImpl 객체는 abstract 연산인 doActionForApproval()을 구현해야 한다 (비슷한 이유로, doActionForRejection()도 구현해야 함).

제 4 절 결론

본 논문에서는 CORBA와 Java를 이용하여 설계/구현한 PDM 시스템을 소개하였다. 본 시스템에서는 Observer, Factory Method 등의 패턴을 통해 시스템의 확장과 변경 (evolution) 이 용이하도록 설계되었는데, 이 패턴들을 이용하여 개발된 객체 풀이나 지속성 객체를 위한 추상화 계층은 PDM 시스템이 아닌 다른 분산 객체 시스템에서도 사용될 수 있는, 특정 도메인에 국한되지 않는 개념들이다.

본 시스템을 개발하는 도중, CORBA가 매우 쉽게 분산 객체 서버를 구현할 수 있는 환경을 제공함에도 불구하고, 실행 효율 및 분산 자원 수거 (특히, 분산 가비지 컬렉션) 등에 많은 문제점이 있음이 발견되었다. 이를 극복할 수 있는 다양한 분산 객체 개발 기법과 가이드라인에 대한 연구가 앞으로 이루어져야 하리라고 생각한다.

참고 서적

[1] E. J. Choi and Y. Kwon. An efficient tree version control method. In *Proceedings of 23rd KISS Spring Conference*, pages 685–688, April 1996.

[2] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–280, June 1998.

[3] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, MA, 1997.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[5] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, MA, 1999.

[6] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[7] D. Pedrick, J. Weedon, J. Goldberg, and E. Bleifield. *Programming with VisiBroker*. John Wiley & Sons, New York, NY, 1998.