EJB Command Pattern

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

Document title:	EJB Command Pattern
Document file name:	EJB_Command_Pattern_jstorm_1.0_final
Revision number:	<1.0>
Issued by:	<이용원>
Issue Date:	<2002/2/23>
Status:	final

Content Information

Audience	EJB 중급
Abstract	EJB 클라이언트는 유즈 케이스를 완료하기 위해서 비즈니스 로직을 실행해야 한다. EJB에서 클라이언트와 결합하고, 하나의 트랜잭션과 한번의 네트워크 호출을 하는 간단한 방법으로 유즈 케이스의 비즈니스 로직을 개발자는 어떻게 구현할 수 있을까?
Reference	http://www.theserverside.com/resources/patterns_review.jsp
Benchmark information	

Table of Contents

EJB Command Pattern

EJB 클라이언트는 유즈 케이스를 완료하기 위해서 비즈니스 로직을 실행해야 한다.

EJB에서 클라이언트와 결합하고, 하나의 트랜잭션과 한번의 네트워크 호출을 하는 간단한 방법으로 유즈 케이스의 비즈니스 로직을 개발자는 어떻게 구현할 수 있을까?

EJB 시스템을 설계할 때 중요한 아키텍쳐적인 결정 요소는 비지니스 로직을 어느 곳에 두느냐 하는 것이다. 유스 케이스의 비지니스 로직을 도메인 모델의 적절한 메소드에 위임하거나 여러 개의 다른 엔티티빈이나 세션빈(워크 플로우)으로 처리하는 로직을 실행하는 것이다.

클라이언트(서블릿, 애플릿 등)에 비지니스 로직을 두는 것은 성능이나 유지보수에 심각한 부작용을 낳는다("Session Façade" 패턴에서 설명한 바 있다). 이러한 문제는 패턴을 적용함으로써 해결할 수 있는데, 비지니스 로직을 세션빈의 메소드에 두고 세션빈의 각 메소드는 특정 단위 업무 또는 유스 케이스에 매핑한다. 이렇게 함으로써, 서버상의 객체 모델은 클라이언트로부터 감싸고, 유스 케이스는 하나의 트랜잭션과 한번의 네트워크 호출로 실행된다.

"Session Façade" 패턴 자체는 EJB 개발에 있어서 주요 산물이다. 그러나 "Session Façade"가 가진 단점도 포함한다. 대규모의 프로젝트의 경우에 있어서 클라이언트가 "Session Façade"를 직접 호출하는 것은 클라이언트 팀과 서버 팀 간의 의존성을 야기할 수 있고, EJB 에 높은 결합도 때문에 클라이언트 코드가 복잡하게 된다. ("Business Delegate" 패턴에서 설명한 바 있다.) 이러한 문제는 "Business Delegate" 사용함으로써 완화될 수 있고, 그것은 EJB 계층에 대한 모든 접근을 캡슐화한 객체 계층을 추가하는 것이다. "Business Delegate" 는 클라이언트와 서버 간의 의존성을 최소화하여 클라이언트 코드를 간단하게 할 수 있다.

"Business Delegate" 패턴과 협력하여 "Session Façade" 패턴은 서버의 상세한 구현과 클라이언트간의 결합도를 줄이는 형식으로 비지니스 로직을 작성하는 가장 좋은 방법을 제공한다. 하나의 트랜잭션에서 한번의 네트워크 호출로써 유스 케이스를 실행할 수 있다. 늘 그런 것처럼 흥정은 있다.

① 더 느린 개발프로세스.

유즈 케이스 로직(자주 변경될 수 있는)이 세션빈에서 실행하기 때문에, 유스 케이스가 변경되어야 할 때마다, 세션 빈의 메소드는 유스 케이스의 변경을 구현해야 한다. (예를 들면, 메소드의 파라미터가 추가된다, 또는 부가적인 애트리뷰트을 리턴한다). 세션빈을 변경하는 프로세스는 사소한 일이 아니다 - 4개의 다른 파일(홈 인터페이스, 리모트 인터페이스, 빈 클래스, 디플로이먼트 디스크립터) 을 편집해야 할 뿐만 아니라, EJB 서버에 다시 배치해야 하고 가능한 서버를 재시작 해주어야 한다. 부가적으로, 클라이언트에 변경된 세션 빈을 캡슐화한 "Business Delegate"도 변경되어야 한다.

② 대형 프로젝트의 경우 업무 분담이 복잡해진다.

프로젝트에서 개발자들이 분업하는 전략에 의존할 때, 프로젝트가 진행됨에 따라 빈번한 변경이 주제가 될 수 있으므로, "Session Façade" 는 어떤 다른 팀이나 개발자 사이의 논쟁이 되어 자주 병목이 발생한다.

③ 서버측 리소스가 단지한 팀에 의해서만 제어된다.

확정해서 디플로이된 EJB 를 운용하는 세트를 가진 큰 회사에서, 기존의 클래스에 어떤 변경을 반영하기 위한 다른 프로젝트를 작업하는 다른 팀에 대해 어려운 일이 될 수 있다.

.

간단히 말하자면, "Session Façade"와 "Business Delegate"를 적용해서 개발하는 것은 변경-디플로이-테스트의 순환 과정이 오래 걸릴 수 있다. 그래서, 대형 프로젝트에 있어 병목현상을 가져올 수 있다. 문제의 요점은 비즈니스 로직이 세션 EJB의 계층에 놓이고 있다라는 것이다. 그것은 개발하는 꽤 무거운 무게일 수 있다.

그러므로,

한번의 네트워크 호출로 실행하고 EJB 계층을 Façade 역할을 하는 EJB 와 클라이언트의 결합도를 줄여주는 경량 커맨드 빈에 비지니스 로직으로 감싸는 "Command" 패턴 을 사용하라.

커맨더 빈은 Gang of Four가 저술한 “Design Patterns” 에 기술된 것처럼 get, set, execute 메소드를 가진 일반 자바 클래스이다. EJB에 적용되면, “Command” 패턴은 “Session Façade”와 “Business Delegate” 와 같은 장점을 얻을 수 있는 경량 솔루션을 제공한다: EJB 계층의 객체 모델을 숨겨주는 “Façade” 는 EJB와 클라이언트의 결합도를 완전히 줄여주고, 하나의 트랜잭션화 한번의 네트워크 호출로 실행된다. “Command” 패턴은 지역적으로 상호작용하는 클래스를 클라이언트에 제공하는 것에 의해 이들을 수행한다. 그러나, 클라이언트에 투명하게, 리모트 EJB 서버 내에서 사실 실행된다.

“Command” 는 애플리케이션의 개별 단위 작업을 캡슐화 한다. placeOrder, transferFunds등과 같은 유스케이스는 그것의 비지니스/워크플로우 로직을 해당 유스케이스를 위한 특정 “Command” 에 캡슐화했다. (그림 1. 참고)

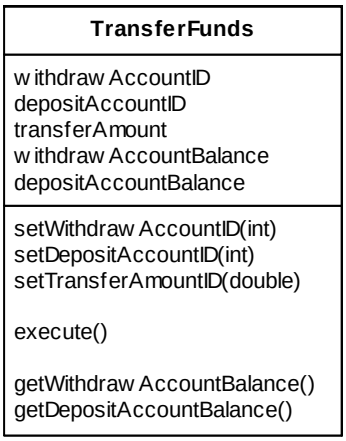

그림1. : TransferFunds Command 의 클라이언트 관점

클라이언트와 “Command” 간의 상호작용은 매우 간단하다. 일단 클라이언트가 “Command”를 얻으면(직접 생성할 수도 있고 “Factory” 로부터 생성할 수 도 있다. 이것은 구현의 문제이다.) “Command”가 유스케이스를 실행하기 위해서 필요한 모든 데이타를 얻을 때까지 간단하게 “Command”의 애트리뷰트을 설정한다. 그런 후 클라이언트는 “Command”의 실행 메소드를 호출하고, “Command”/유즈 케이스 의 실행 결과로부터 필요한 모든 데이타를 검색할 때까지 “Command”의 get 메소드를 실행한다.

클라이언트가 “Command”를 실행할 때 그 이면에는 흥미로운 일이 일어난다. 실제 로컬상에서 실행하지 않고, 리모트 EJB 서버에 “Command”가 전달되고 EJB 서버의 JVM내에서 실행된다. 그것의 유스케이스가 실행되는 동안 “Command” 에 의한 모든 EJB 호출은 EJB 서버 자신의 내부에서 발생된다. “Command”의 실행이 완료되었을 때, 그 결과는 클라이언트에게 리턴되고, 데이터를 검색하기 위해서 get 메소드를 호출할 수 있다. EJB 서버 내에서 “Command” 실행하여, 유즈 케이스는 단 하나의 유즈케이스를 실행할 수 있다. 이 동작의 구현 메커니즘은 이 패턴의 후반에 설명된다.

그림 2.의 transferFunds의 예를 보면, 클라이언트는 인출계좌, 입금계좌의 ID를 설정하고, 송금금액을 설정한다. transferFunds command의 실행 메소드를 실행한 후, 클라이언트는 각 계좌의 최종 잔액을 얻을 수 있다.

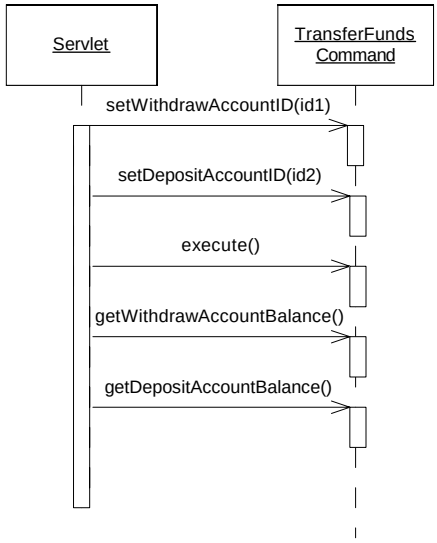

그림 2. TransferFunds Command의 사용

“Command” 패턴을 잘 구현한 것 중에 하나가 아마도 웹 스피어에 적용된 IBM의 “Command” 프레임워크일 것이다. 기본적인 “Command” 프레임워크 구현을 부록에 수록하였으며, 이것은 IBM's 프레임워크의 개념에 부분적으로 바탕을 두고 있다. “EJB Command: 패턴은 여러 가지 방법으로 구현될 수 있으나 어느 경우나 3가지 요소를 가진다.

1. Command Beans.

gets, sets 메소드와 유스케이스를 실행하는데 필요한 비지니스 로직을 포함하는 실행 메소드를 가진 간단한 자바 빈 클래스이다. “Command”로 구현된 transfer funds 유즈 케이스에 대한 코드 예제는 부록을 보자. “Command” 빈은 “Command” 패턴에서 애플리케이션 개발자에 의해 작성되어야 하는 유일한 부분이다. 아래에 설명된 다른 컴포넌트들은 타 프로젝트간에 재사용이 가능한 부분들이다.

2. 클라이언트 측의 라우팅 로직

일반적으로 클래스의 프레임워크 “Command” 를 얻어서 리모트 EJB 서버에 보내는 역할을 맡는다. 일반적으로 이 라우팅 로직은 클라이언트에게 보여지지 않으며, “Command”의 실행 메소드를 호출함으로써 실행된다. 절차 결정 로직/프레임워크는 타 프로젝트간에 재사용이 가능한 일반적인 클래스 집합이다. 부록에서 라우팅 로직은은 CommandExecutor와 EJBCommandTarget 클래스이며, “Command”를 얻어 EJB 서버에 보내는 역할을 한다. 이 모든 것은 클라이언트와 무관하다.

3. 리모트 Command Server.

Command Server는 “Command”를 받아서 실행시키는 서비스이다. EJB 입장에서 보면 “Command” 서버는 파라미터로 “Command”를 갖고 이를 로컬상에서 실행시키는 무상태 세션 빈으로 볼 수 있다. “Command” 서버 역시 타 프로젝트간에 완벽히 재사용이 가능하다. 부록에 “Command” 서버 무상태 세션 빈에 대한 구현이 나와 있다.

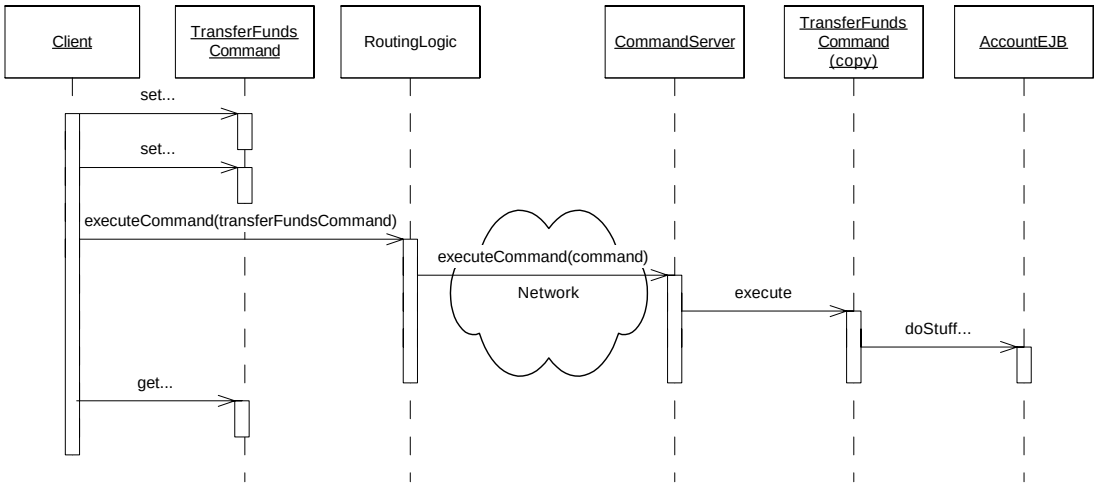

그림 3. “Command” 패턴 상호 작용

위 3가지 컴포넌트와 클라이언트와의 상호작용은 그림 3.과 같다. 부록의 예제 코드를 바탕으로 한 이 예제에서 클라이언트는 CommandExecutor 클래스로 표시된 routing logic 컴포넌트의 executeCommand 메소드를 호출한다. IBM의 Command 프레임워크에서, 클라이언트는 “Command” 자신의 execute를 호출할 뿐이며, 실제적으로 메소드 호출은 “Command”의 수퍼 클래스 (routing logic 프레임워크의 일부분)에 의해 수신될 것이다.

그 이면에서 CommandExecutor는 EJBCommandTarget (routing logic의 일부분이므로 그림 2.에는 나와 있지 않다)에게 위임한다. command를 어떻게 CommandServer 무상태 세션 빈에 보내야 하는지를 EJBCommandTarget이 담당한다.

“Command” 패턴의 장점은 :

① 경량 개발 / 디플로이 프로세스때문에 RAD이 가능.

“Command” 빈 형태로 유스 케이스를 작성하면 세션 빈의 형태일 때보다 배포와 테스트가 상당히 쉽고 빨라진다. 모두 EJB형태로 구현한 경우와 대조적으로 일반 자바클래스에 대한 잦은 변경이 가능하다.

② 프리젠테이션 로직으로부터 비지니스 로직을 분리.

"Command는" "Command" 내부의 비지니스 로직을 캡슐화 하고 클라이언트는 간단히 "Command" 의 인터페이스만 사용함으로써 서버내에서 "Façade" 형태로 작동된다. 이렇게 분리함으로써 클라이언트와 서버를 따로따로 개선할 수 있게 된다.

③ 한 번의 호출로 유스케이스를 실행한다.

실제적으로 "Command" 는 EJB 서버에서 실행되므로, 복잡한 유스케이스를 완료하기 위해서 오직 한번의 네트워크 호출(그리고 트랜잭션)만이 필요한다.

④ EJB 와 클라이언트간의 결합도를 줄인다.

로컬 클래스인 "Command" 빈 에서 볼 수 있는 것처럼 서버의 상세한 구현과 클라이언트 간의 결합도를 완전히 제거한다

⑤ " Command" 는 로컬에서 실행되거나 더미 데이타를 생성할 수 있다.

프로젝트 초기에 내용이 비었거나 혹은 가상 "Command" 를 생성하여 프리젠테이션 계층 개발자가 비지니스 로직/EJB 팀과 독립적으로 코드를 개발하고 테스트할 수 있다.

⑥ 다수의 반환값을 가질 수 있다.

오직 한 객체만을 리턴할 수 있는 세션 빈의 메소드와는 대조적으로 "Command" 는 서로 다른 다수의 'value' 를 생성할 수 있다.

여러 측면에서 "Command" 패턴은 가벼운 인프라스트럭쳐이며 "Session Façade" 와 "Business Delegate" 패턴의 장점을 조합한 최상의 솔루션인 것처럼 보이지만 고려해야 할 중요한 부분도 있다.

① 매우 큰 단위(coarse-grained) 트랜잭션 제어.

"Command" 는 일반 자바 클래스이므로 세션 빈에서 보던 것처럼 자동적으로 특정 트랜잭션 설정이나 격리 수준에서 실행되도록 할 수 있는 방법은 없다. "Command" 는 자신을 실행시키는 CommandServer의 트랜잭션 설정 에서만 실행된다. 다른 JNDI 네임과 트랜잭션 설정(디플로이먼트 디스크립터에서 설정한다)을 가진 다수의 CommandServer 세션빈을 배치하는 것이다. "routing logic" 컴포넌트는 어떤 "Command"를 어떤 CommandServer에서 보내야 하는지 설정되어야 한다. 예를 들면, 하나는 트랜잭션을 실행하는 모든 읽기 전용 command 를 전송할 수 있다. 반면에 update

Command 는 TX_REQUIRE 와 Serializable 격리 수준 으로 실행하는 Command Server 에서 실행해야만 한다.

② 보안을 사용할 수 없다.

비지니스 로직이 command 빈에 내장되므로, SessionContext (isCallerInRole 또는 getCallerPrinciple 같은 보안 검사를 수행하는 비지니스 로직) 같은 EJB 컨테이너의 서비스에 접근할 수 있는 방법이 없다.

③ Error 처리가 번거롭다.

Command 프레임워크는 일반적인 사항이기 때문에, 오직 CommandException만이 발생될 수 있다. 이것은 NoMoneyInAccountException 같은 어플리케이션 예외를 잡아서CommandException으로 감싸야 한다. 클라이언트는 특정 예외에 대해서 "Command" 객체의 내부를 살펴보아야 한다. 예외가 명시적으로 선언되어있지 않으므로 클라이언트는 예외 처리에 대해서 컴파일시에 검사할 수 있는 장점을 잃게 된다.

④ 큰 프로젝트에서는 Command 가 통제하기 힘들어진다.

큰 프로젝트는 수 천 개의 "Command" 로 넘쳐날 수도 있으며, 이들 중 많은 수가 중복되는 비지니스 로직이다. 이것은 비지니스 로직 레이어에 대한 유지보수를 어렵게 만든다. 반면에 "Session Façade" 패턴에서는 유스케이스가 세션 빈의 메소드로 구현되고 적은 수의 세션 빈 에 적절하게 그룹화 될 수 있다. 큰 프로젝트에서는 클래스의 증가가 심각한 문제가 될 수 있다.

⑤ " Command" 서버의 ejb-jar는 다른 EJB와 "Command" 빈과 높은 결합도를 발생한다.

"Command" 빈이 "Command" 서버 세션 빈 환경에서 실행되므로 "Command" 빈이 역시리얼라이즈되고 실행되기 위해서는 "CommandServer" 세션 빈과 함께 배치되어야 한다. 이것은 "Command" 빈이 변경될 때마다 변경을 테스트하기 위해서 "CommandServer" 서버 세션 빈 이 다시 재배치되어야 함을 뜻한다. 또한 "Command" 빈은 비지니스 로직을 사용할 수 있는 어떠한 홈 인터페이스, 리모트 인터페이스, 홈 로컬 인터페이스 또는, 로컬 인터페이스라도 이용할 수 있어야 한다. "CommandServer" 는 어떤 "Command" 빈에 접근되는 다른 EJB로써 같은 ejn-jar 에 디플로이되거나, 접근되는 EJB 의 인터페이스는 "CommandServer" 에 함께 패키지된다.

“ Command” 패턴과 “Session Façade” 패턴 모두 두 가지 중요한 장점을 제공한다 두 경우 모두 “Façade” 형태로 작동되고 한번의 네트워크 호출로 실행된다. “Session Façade”에 비해서 “Command” 패턴의 장점은 EJB와 클라이언트간의 결합도를 줄일 수 있다. (“Session Façade” 패턴과 결합된 “Business Delegate” 패턴에 의해서도 구현될 수 있다) 둘 중 어느것을 택할 것인가에 대해서 “Command”를 노력이 더 들지 않는 세션 빈으로 생각해보는 것이 도움이 될 것이다. 초기개발절차를 빠르게 하여 결과적으로는 유지보수 비용이 더 적게 들고, 시스템이 더 가벼워질수 있다

부록

"Routing Logic" 컴포넌트 (CommandExecutor과 EJBCommandTarget)와 "Command Server "(CommandServerBean), bank account transfer command등을 포함하여 여기에 보인 것은 command 패턴에 대한 매우 간단한 구현이다.

이 출처는 IBM의 Command 프레임워크의 단순화된 버전이다. "Command" 패턴의 개념들을 설명하기 위해 제공된다.

```
package examples.command;

import examples.account.AccountHome;
import examples.account.Account;
import examples.account.ProcessingErrorException;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import javax.ejb.FinderException;
import java.rmi.RemoteException;

public class TransferFundsCommand extends Command implements Serializable
{

    String withdrawAccountID;
    String depositAccountID;
    double transferAmount;

    double withdrawAccountBalance;
    double depositAccountBalance;
```

```
public void execute() throws CommandException
{
   //at this point we are inside the EJB Server
   try {

      InitialContext ctx = new InitialContext();
      AccountHome home = (AccountHome) PortableRemoteObject.narrow
         (ctx.lookup("Account"), AccountHome.class);

      //locate accounts and perform transfer
      Account account1 = home.findByPrimaryKey(withdrawAccountID);
      Account account2 = home.findByPrimaryKey(depositAccountID);

      account1.withdraw(this.transferAmount);
      account2.deposit(this.transferAmount);

      //populate command with final balances
      this.depositAccountBalance = account2.balance();
      this.withdrawAccountBalance = account1.balance();
   }
   catch (Exception e)
   {
      //wrap the exception as a command execption and throw
      //to client for interception
      throw new CommandException(e);
   }
}

public void setWithdrawAccountID(String withdrawAccountID) {
   this.withdrawAccountID = withdrawAccountID;
}

public void setDepositAccountID(String depositAccountID) {
```

```
        this.depositAccountID = depositAccountID;
    }

    public void setTransferAmount(double transferAmount) {
        this.transferAmount = transferAmount;
    }

    public double getDepositAccountBalance() {
        return depositAccountBalance;
    }

    public double getWithdrawAccountBalance() {
        return withdrawAccountBalance;
    }

    public TransferFundsCommand()
    {}
}
```

부록 1. : Transfer Funds Command

```
package examples.command;

import java.io.Serializable;

public abstract class Command implements Serializable {

    public abstract void execute() throws CommandException;

}
```

부록 2. : Command Superclass

```
package examples.command;

import javax.ejb.*;
import java.rmi.RemoteException;
import javax.naming.*;


public class CommandServerBean implements SessionBean {

    SessionContext ctx;

    public void CommandServer() {}

   public Command executeCommand(Command aCommand)
        throws CommandException
   {
```

```
        try
        {
            aCommand.execute();
        }
        catch (CommandException e)
        {
            ctx.setRollbackOnly();
            throw e;
        }

        return aCommand;
    }

    public void ejbActivate() throws EJBException,
      java.rmi.RemoteException {}
    public void ejbCreate() throws CreateException {}
    public void ejbPassivate() throws EJBException,
      java.rmi.RemoteException {}
    public void ejbRemove() throws EJBException,
      java.rmi.RemoteException {}

    public void setSessionContext(final SessionContext p1)
      throws EJBException, java.rmi.RemoteException
    {
        this.ctx = p1;
    }

}
```

부록3: CommandServer Session Bean

```
package examples.command;

public class CommandException extends Exception  {

   Exception wrappedException;

   public CommandException(){}

   public CommandException(Exception e)
   {
      this.wrappedException = e;
   }

   Exception getWrappedException()
   {
      return wrappedException;
   }

   public CommandException(String s) {
      super(s);
   }
}
```

부록 4. : CommandException

```
package examples.command;

interface CommandTarget {
    Command executeCommand(Command aCommand)
        throws CommandException;
}
```

부록 5: CommandTarget Interface

```
package examples.command;

import javax.rmi.PortableRemoteObject;
import javax.ejb.CreateException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.rmi.RemoteException;

public class EJBCommandTarget implements CommandTarget {

   private CommandServerHome serverHome;

   public EJBCommandTarget()
   {
      try
      {
         Context ctx = new InitialContext(System.getProperties());
         Object obj = ctx.lookup("CommandServer");
         System.out.println(obj);
```

```
            this.serverHome = (CommandServerHome)
            PortableRemoteObject.narrow(obj, CommandServerHome.class );
        }
        catch (NamingException e)
        {
            e.printStackTrace();
        }
        catch (ClassCastException e)
        {
            e.printStackTrace();
        }
    }

    public Command executeCommand(Command aCommand)
    throws CommandException
    {
        try
        {
            CommandServer aCommandServer = serverHome.create();
            aCommand = aCommandServer.executeCommand(aCommand);
            return aCommand;

        }
        catch (Exception e)
        {
            throw new CommandException(e);
        }
    }
}
```

부록 6.: EJBCommandTarget

```
package examples.command;

public class CommandExecutor
{
   private static EJBCommandTarget ejbTarget = new EJBCommandTarget();

   //execute command, overwriting memory reference of the passed
   //in command to that of the new one
   public static Command execute(Command aCommand)
   throws CommandException
   {
      //at this point, a real implementation would use a properties file
      //to determine which command target (EJB, Local, Corba, etc) to
      //use for this particular command, as well as which deployed
      //CommandServer to use (inorder to run commands in different
      //under different transaction configurations)

      return ejbTarget.executeCommand(aCommand);

   }

}
```

Appendix X.7: CommandExecutor