컴포넌트 형상관리를 위한 프레임웍

Framework for Component Configuration Management

김 귀 정
건양대학교 IT 학부

Kim Gui-Jung
Division of Information Technology, Konyang University

요약

CBD 개발 방법론의 발전으로 많은 컴포넌트들이 개발되고 있다. 본 연구에서는 컴포넌트들의 재사용을 위해 객체지향 설계 프레임웍이 객체보다 더 좋은 컴포넌트로 활용될 수 있음을 기술하였다. 또한 프레임웍 개념과 변경에 의한 형상관리 방법을 논하였고, 여러 프레임웍에서의 객체 공유와 새로운 프레임웍의 생성 방법에 관하여 기술하였다.

Abstract

Using CBD methodology, Components have been developed. In this paper, we explained that Object-Oriented Design frameworks are increasingly recognized as better components than objects. And we discussed the framework concept and component configuration management method using changes. Finally, we explained object sharing in several frameworks and new framework creation process.

Ⅰ. 서론

컴포넌트 형상관리(component configuration management) 는 시스템 내부의 구성원들 사이의 일치성을 제어함으로서 신뢰도를 증가시키는 역할을 한다. 컴포넌트 기반으로 개발된 패키지에 새로운 컴포넌트가 추가 또는 대체될 때 두 가지 문제점을 해결해야 한다. 첫째 기존의 컴포넌트가 다른 패키지와 연결되어 있을 경우 이를 대체 시킬 때 발생하는 문제점이다. 즉 새로운 버전의 컴포넌트가 대체될 때 변경(changes)과 컴포넌트들 사이의 관계가 불확실하다는 것이다. 둘째 실시간 환경에서의 시스템에 대한 동적 행위는 더 심각한 문제점이다. 즉, 실시간으로 컴포넌트가 대체되면 하나의 패키지는 동작을 하지만 이전의 컴포넌트와 연결된 패키지는 동작하지 않을 수 있다. 이러한 문제점의 해결방안이 바로 컴포넌트 형상관리(CCM)이다[1][2].

따라서 본 연구에서는 컴포넌트 관리에 대하여 분석해보고, 컴포넌트 형상에 관련된 문제점들을 지적하며, 이러한 문제점을 해결할 수 있는 컴포넌트 형상관리에 관하여 기술하였다.

Ⅱ. 컴포넌트 변경(changes)

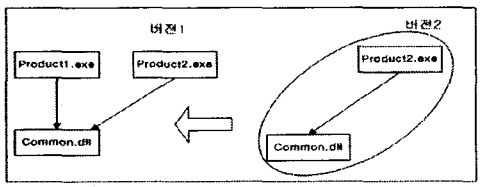
컴포넌트는 공유 라이브러리로 구성된다. 따라서 컴포넌트 사용은 직접 라이브러리를 참조하는 것이 아니라 컴포넌트 인터페이스를 참조한다. 여기서는 컴포넌트의 논리적, 물리적 단계의 변경 정보뿐만 아니라 관계의 정보까지 알아야 한다. 즉, 라이브러리와 인터페이스가 일치되어야 하기 때문에 컴포넌트 형상관리는 두 레벨(라이브러리, 인터페이스)에서 이루어져야 한다[3][4].

일반적으로 dll 파일들이 나오기 이전의 라이브러리는 정적으로 실행파일과 링크되어 있기 때문에 새로운 라이브러리를 위해서는 실행파일과 다시 링크해야하는 문제점이 발생한다. 이것은 라이브러리가 인터페이스 호환성을 갖는다는 의미에 위배된다. 또한 라이브러리와 연결된 모든 실행파일들도 모두 재링크해야만 한다. 이의 해결 방안은 라이브러리는 인터페이스 호환성을 갖으면서 실행파일은 재링크를 하지 않고도 새로운 라이브러리를 공유하는 것이다. 즉, 공유 라이브러리들을 dll로 설계하는 것이다. dll 파일은 실행파일이 필요할

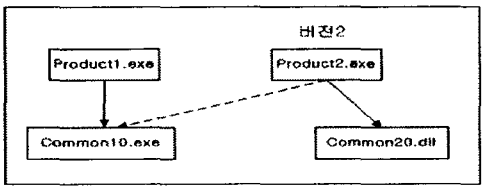
때마다 호출하기 때문에 호환성이 가능하다. 그러나 dll 파일 역시 시스템의 일관성에 대한 새로운 문제점이 발견된다.

(그림 1)은 이 시스템에 Product2.exe가 업그레이드 되면서 버전2의 Common.dll이 인터페이스 호환성을 갖고 있어서 버전1의 Common.dll로 대체될 경우 성공적이지만 이 시스템 내부에 연결되어 있는 Product1은 중단될 수 있음을 보여준다. 이의 해결책은 라이브러리의 다중 버전(multiple version)을 이용하는 것이다. 예를 들면 MFC40.dll, MFC42.dll과 같은 이름으로 사용하는 것이다. (그림 4)에서처럼 이름 충돌 문제를 해결할 수 있다. 그러나 이것 역시 많은 유사 버전들이 생성되면서 제어가 어려운 단점을 갖고 있다. 변경에 의해 영향을 받는 시스템 내부 구성원을 이해하기 위한 내용은 다음과 같다.

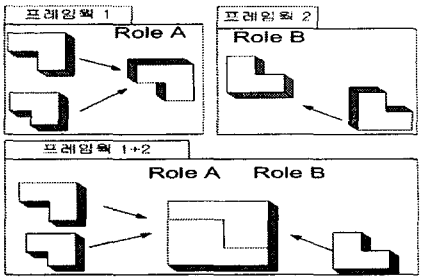
- 컴포넌트와 연결된 버전을 함께 인식
- 직접 또는 간접으로 연결된 의존성을 인식
- 의존범위에 대한 충분한 정보를 인식



▶▶ 그림 1. dll 변경에 의한 Product1의 실행중단



▶▶ 그림 2. Common.dll의 공존



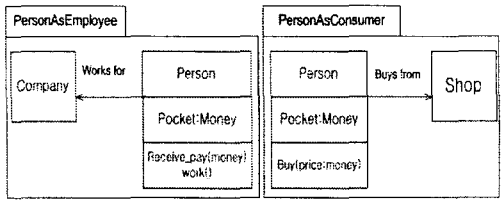
▶▶ 그림 3. 서로 다른 프레임웍에서 다중 역할

Ⅲ. 객체지향 프레임웍

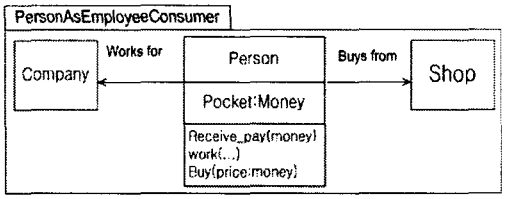
객체지향 설계 프레임웍(OOD Framework)은 소프트웨어 개발에 있어서 객체보다 더 좋은 컴포넌트로 인식되고 있다. 그 이유는 실제 시스템에서 많은 객체들은 여러 상황에서 하나 이상의 역할을 하며, OOD 프레임웍이 이를 해결해 주기 때문이다. 그러나 기존의 OOD 방법들은 한 가지 고정된 기능만을 갖고 있다. 보통 객체는 고정된 역할을 하는 기능을 갖고 있지만 프레임웍은 (그림 3)에서처럼 객체가 서로 다른 프레임웍에서 서로 다른 역할을 하는 내용을 보여주고 있다.

1. 프레임웍의 예

다음의 예는 프레임웍의 개념을 보여주고 있다. (그림 4)는 종업원의 프레임웍을 보여준다. 여기서 person은 회사의 종업원의 역할을 한다. 종업원으로서의 person은 자신이 받는 급여의 양을 나타내는 pocket 속성을 갖고 있고, receive_pay()와 work()의 멤버함수를 갖고 있다. 이제 또 다른 관점의 사람 즉, 소비자로써의 사람을 의미하는 PersonAsConsumer를 생각해보자. 여기서 person은 pocket 속성을 갖지만 buy() 멤버함수를 갖고 있다. 그러면 여기서 두 가지 역할 PersonAs-Employee와 PersonAsConsumer를 모두 포함하는 프레임웍을 만들 수 있다. <그림 4>는 두 역할을 모두 포함하는 프레임웍을 보여준다. 여기서 person은 receive_pay(), work(), buy()의 기능을 모두 할 수 있다.



▶▶ 그림 4. 서로 다른 역할을 하는 Person



▶▶ 그림 5. 두 가지 역할을 하는 Person

2. 형상관리 방법

순수한 객체 대신에 프레임웍의 사용은 여러 장점을 가진다. 그러나 구현에 있어서 여러 복잡성의 문제점이 도출된다. 이것은 프레임웍이 혼합된 형태의 entity를 갖기 때문이다. 즉, 객체들로 구성되는 내부 구조를 갖고 있고, 이 entity들은 다른 프레임웍과 관계가 있기 때문이다. 이와 같은 복합 entity의 정의와 생성은 형상관리 문제점으로 나타난다[5].

2.1 여러 프레임웍에서의 객체 공유

프레임웍 F_1은 객체 O_1과 O_2를 포함하고 있고, 이 객체들 사이의 관계는 R_{12}의 관계가 있다. 프레밍 F_2는 객체 O_1과 O_3을 포함하고 있고 관계는 R_{13}이다. 따라서 객체 O_1은 두개의 프레임웍에 공유된다.

$F_1 = \{ O_1, O_2 ; R_{12} \},$
$F_2 = \{ O_1, O_3 ; R_{13} \}$

여기서 객체 O_1에 대한 새로운 특성을 추가한다고 가정하자. 또한 이 특성은 프레임웍 F_2에 있는 객체 O_2가 갱신된다고 가정하자. 그러면 새로운 버전 $O_1{:}v_2$가 생성된다. 그러면 프레임웍 F_2는 다음과 같다.

$F_2 = \{ O_1{:}v_2O_3 ; R_{13} \}$

그러나 이 버전을 다른 프레임웍과의 연결된 정보 즉 프레임웍 명세서를 고려하지 않으면 이전의 버전으로 되어있는 경우가 된다. 이 경우에 프레임웍 F_2에 있는 객체 O_1이 변경되었다는 사실은 알 수 있지만 F_1에 있는 객체 O_1은 오류가 발생하게 된다. 즉, 실제로 프레임웍 F_1은

$F_1 = \{ O_1{:}v_2O_2 ; R_{12} \}$

으로 변경이 되었기 때문에 F1은 실패할 수 있다.

이와 같은 예측 불가능한 상황을 피하기 위하여 기본적인 형상관리 방법인 객체의 버전관리와 프레임웍 형상관리를 제시하고자 한다.

- 객체는 이름과 버전으로 정의
- 프레임웍은 이름과 버전으로 정의
- 새로운 프레임웍 버전은 프레임웍에 포함되어 있는 객체버전으로 유추

이 규칙은 새로운 객체 버전이 변경될 때 새로운 프레임웍 버전이 형성됨을 의미한다.

$F_{1:vi} = \{ O_{1:vm}O_{2:vn} ; R_{12} \},$
$F_{2:vk} = \{ O_{1:vm}O_{3:vk} ; R_{13} \},$
$F_{1:vi+1} = \{ O_{1:vm+1}O_{2:vn} ; R_{12} \},$
$F_{2:vk+1} = \{ O_{1:vm+1}O_{3:vk} ; R_{13} \}$

여러 프레임웍이 하나의 객체를 공유하고 하나의 프레임웍이 여러 객체를 포함하고 있을 때, 생성된 프레임웍의 수는 폭발적으로 증가할 수 있다. 그러나 형상의 수를 제한하는 것은 가능하다. 일반적으로 개발 프로세스에서 모든 객체의 변경과 버전은 얻을 수 있고 객체 버전에서 프레임웍을 유도할 수 있기 때문에 무분별한 확산은 막을 수 있다.

2.2 새로운 프레임웍 구성

프레임웍 모델에서 기존의 프레임웍 모델에서 새로운 프레임웍 모델을 구성하는 것은 가능하다. 새로운 프레

임웍이 실시간에서 생성되면 선택된 프레임웍들에서 나온 객체들이 새로운 프레임웍을 구성한다. 다음의 예는 두개의 프레임웍 F_1, F_2에서 새로운 프레임웍 F_3을 생성한 예이다.

$F_1 = \{ O_1O_2 ; R_{12} \},$
$F_2 = \{ O_1O_3 ; R_{13} \},$
$F_3 = \{ O_1O_2O_3 ; R_{12},R_{13},R_{23} \}$

그러나 하나의 프레임웍에서의 객체의 변경은 고려할 필요가 없다. F_2에서 새로운 객체 버전을 생성하고, F_1의 같은 객체 버전을 유지시키고자 하면 다음과 같다.

$F_{1:vi} = \{ O_{1:vi}O_{2:vk} ; R_{12} \}$
$F_{2:vj} = \{ O_{1:vl+1}O_{3:vl} ; R_{13} \}$

병합 과정에서는 프레임웍에 있는 같은 객체의 다른 버전들이 병합되는 지를 인식해야만 한다. 이런 경우 두 가지 해결법이 있다.

- 객체의 특정 버전을 선택
  $F_{3:vl} = \{ O_{1:vl+1}O_{2:vk}O_{3:vl} ; R_{12},R_{13},R_{23} \}$
- 두 버전 모두를 선택하여 새로운 프레임웍에서 이들을 일치시킨다.
  $F_{3:vl} = \{O_{1:vl},O_{1:vl+1},O_{2:vk}O_{3:vl}; R_{12},R_{13},R_{23}\}$

두 번째의 경우에는 객체 버전을 인식하고 있어야 한다.

Ⅳ. 결론

컴포넌트를 재사용하기 위한 프레임웍 접근 방법은 보다 많은 가능성을 보여준다. 본 연구에서는 프레임웍 모델을 통한 객체의 버전과 변경에 대하여 기술하였고, 객체의 변경에 대한 새로운 프레임웍의 생성을 보였다. 따라서 프레임웍에 대한 형상관리는 객체들을 혼합함으로써 새로운 프레임웍의 구현이 가능함을 보였다. 이때 필요한 정보는 객체 사이의 관계, 프레임웍의 현재 버전, 이에 대한 변경관리 정보 등이다.

▌참고문헌▐

[1] Magnus Larsson, Ivica Crnkovic, "New Challenges for Configuration Management", Ninth International Symposium on System Configuration Management (SCM-9), Toulouse, France, September 1999.
[2] Ivica Crnkovic, "Component-based Software Engineering - New Challenges in Software Development", Invited talk & Invited report, MIPRO 2001 proceedings Opatija, Croatia , May 2001.
[3] George T. Heineman, William T. Councill, "Component-Based Software Engineering", Addison- Wesley, pp. 485-549, 2001.
[4] Alan W. Brown, K.C. Wallnau, " Engineering of Component-Based Systems", Proceedings of the 2nd IEEE International Conference on Complex Computer Systems, Oct. 1996.
[5] Magnus Larsson, Ivica Crnkovic, "Component Configuration Management", ECOOP 2000 Conference, Workshop on Component Oriented Programming, Nice, France, June, 2000.