컴포넌트 형상관리를 위한 인터페이스 설계에 관한 연구

채은주*, 한정수*, 김귀정*
*천안대학교 정보통신학부
*건양대학교 IT 학부
e-mail: *cobi80@cheonan.ac.kr *jshan@cheonan.ac.kr
**gjkim@konyang.ac.kr

A Study on Interface Design for Component Configuration Management

Eun-Ju Chae*, Jung-Soo Han*, Gui-Jung Kim*
*Division of Information and Communication, Cheon-An Univ.
*Division of IT, Kon-Yang University

요 약

CBD 개발 방법론의 발전으로 많은 컴포넌트들이 개발되고 표준화에 관한 연구가 진행되고 있다. 본 연구에서는 컴포넌트들의 재사용을 위한 컴포넌트 형상관리에 초점을 두었다. 컴포넌트 대체를 위해 입출력, 인터페이스, 행위의 호환성을 고려하였고, 컴포넌트는 버전관리, 형상과 구축관리, 변경관리, 의존관리를 통한 관리 방법을 제시하였다. 또한 인터페이스를 통한 컴포넌트 연결 과정을 기술하였다.

1. 서론

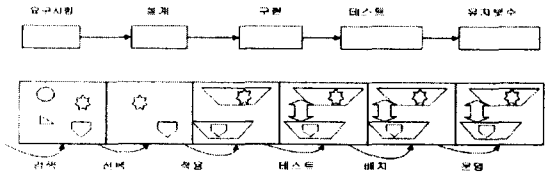
최근 소프트웨어 개발과정이 in-house 형태의 개발에서 표준화된 컴포넌트, 아웃소싱(outsourcing), 상용 컴포넌트(COTS) 등의 새로운 패러다임으로 진행되고 있다. 즉 개발된 최종 시스템은 하나의 고정된 시스템이 아니라 컴포넌트 기반으로 개발되어 다른 시스템과 통합될 수 있고, 자체적으로 갱신될 수 있는 개방형(open) 또는 유동성(flexible) 있는 시스템으로 발전되고 있다. 이와 같은 시스템에서는 시스템이 실시간으로 업그레이드되기 때문에 시스템 형상에 영향을 준다. 따라서 이러한 패러다임은 개발의 효율성을 증가시키지만 시스템 형상의 일관성 또는 신뢰성을 감소시키는 위험이 따른다[1][4].

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 행위에 대한 문제점이다. 즉, 실시간으로 컴포넌트가 대체되면 하나의 패키지는 동작을 하지만 이전의 컴포넌트와 연결된 패키지는 동작하지 않을 수 있다는 것이다. 이러한 문제점의 해결방안이 바로 컴포넌트 형상관리(CCM)이다.

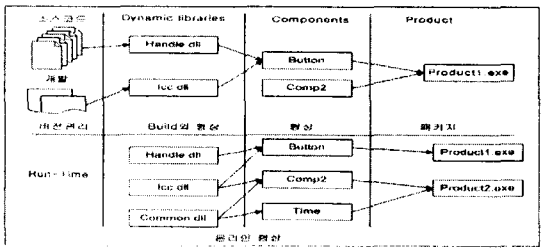
따라서 본 연구에서는 컴포넌트 관리에 대하여 분석해 보고, 컴포넌트 형상에 관련된 문제점들을 지적하며, 이러한 문제점을 해결할 수 있는 컴포넌트 형상관리 방향에 관하여 기술하였다.

2. CBD(Componet Based Development) LifeCycle

CBD와 기존의 개발 생명주기 방법들과의 차이점은 컴포넌트의 재사용성이다. 컴포넌트 재사용을 위해서는 명세화, 이해하기 쉽고, 범용성, 적응성, 재배치 등이 잘 이루어져야만 가능하다. (그림 1)은 폭포수모델과 비교한 CBD 생명주기이다. CBD 생명주기는 컴포넌트를 검색하여 정확한 컴포넌트를 선택한 후 시스템에 적용하여 테스트한 후 재배치되는 과정이다.[2].



(그림 1) 폭포수 모델과 비교한 컴포넌트 생명주기



(그림 2) 컴포넌트 기반 생명주기에서 형상관리

(그림 2)는 개발단계와 런타임에서의 형상관리를 보여주고 있다. 개발단계에서는 소스로부터 라이브러리를 만들고, 컴포넌트는 라이브러리(dll 파일)를 조립하여 생성한다. 그리고 프로덕트는 컴포넌트들의 집합으로 구현된다. 이 과정에서 첫 번째 개발단계에서의 소스 관리는 코드의 버전 정보를 이용하여 이루어진다. Build는 소스코드와 연결하여 진행되고, 끝으로 컴포넌트 조립으로 프로덕트가 생성된다. 이때는 소스코드를 제어하고, 이 코드로부터 전체 시스템을 만들기 때문에 시스템 형상관리를 제어할 수 있다. 그러나 하나의 컴포넌트가 추가, 또는 대체되면 이 제어는 컴포넌트의 기능을 부분적으로 알기 때문에 사라진다. 하지만 컴포넌트 버전제어가 가능하면 버전과 형상관리가 동시에 가능하기 때문에 컴포넌트 사이의 불확실한 관계를 해결할 수 있다.

3. 컴포넌트 형상 관리

3.1 컴포넌트 호환성

컴포넌트의 특성은 독립적이어야 하며 특히, 런타임 컴포넌트는 실행 중에 인터페이스를 통하여 단위 프로그램으로 패키지에 동적으로 추가되는 독립적인 기능으로 시스템에 재배치가 가능해야 한다. 이처럼 컴포넌트의 중요성은 개발과 통합을 위한 기술이 표준화되어야 그 효력을 발휘한다. 현재 애플리케이션 개발의 컴포넌트 기술은 EJB, COM+, ActiveX, CORBA 등이 표준으로 사용되고 있다. 새로운 컴포넌트 버전이 새로운 기능으로 추가되어 기능이 변경될 수 있다. 이때 고려해야 할 사항이 호환성(compatibility)이다. 호환성은 컴포넌트가 대체될 수 있는지를 결정하는 기준이 되며, 이러한 결정은 런타임 환경에서 특히 중요한 요소가 된다. 호환성은 세 단계로 분류할 수 있다[3].

1) 입출력 호환성 - 컴포넌트는 특정 형식으로 입력을 요구하여 정의된 형식으로 출력한다. 즉, 컴포넌트의 내부적 특성은 고려할 필요가 없다. 그 예로서 서로 다른 워드 프로세서가 같은 문서 형식을 출력하는 경우이다.

2) 인터페이스 호환성 - 개발기간이나 런타임 때 인터페이스는 같을 수 있지만 구현은 다른 경우이다. 즉, 같은 인터페이스를 통하여 서로 다른 ActiveX object를 구현하는 경우이다.

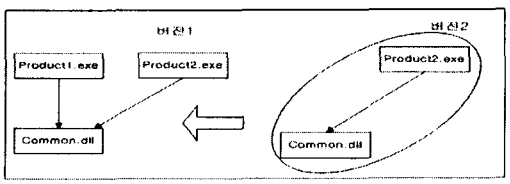
3) 행위 호환성 - 컴포넌트의 내부 특성은 보존되어야 한다. 이와 같은 보존성은 실시간 시스템에 적합하며 가장 중요한 요소이다.

3.2 컴포넌트 변경(component changes)

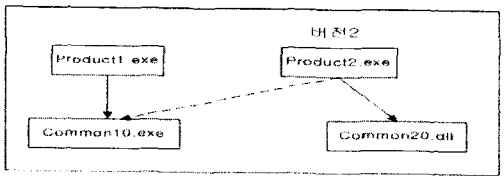
컴포넌트는 공유 라이브러리로 구성한다. 따라서 컴포넌트 사용은 직접 라이브러리를 참조하는 것이 아니라 컴포넌트 인터페이스를 참조한다. 여기서는 컴포넌트의 논리적, 물리적 단계의 변경 정보뿐만 아니라 관계의 정보까지 알아야 한다. 즉, 라이브러리와 인터페이스가 일치되어야 하기 때문에 컴포넌트 형상관리는 두 레벨(라이브러리, 인터페이스)에서 이루어져야 한다.

3.2.1 라이브러리(libraries)

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 재링크 해야만 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 3) dll 변경에 의한 product1의 실행중단



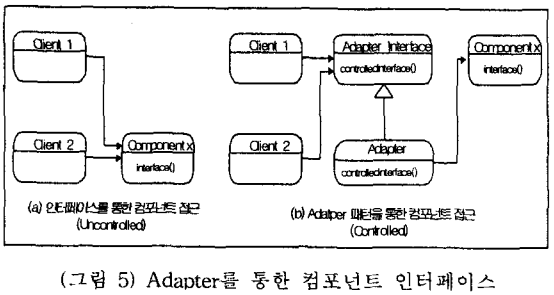
(그림 4) Common.dll의 공존

(그림 3)은 이 시스템에 Product2.exe가 업그레이드되면서 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

3.2.2 인터페이스(*interface*)

인터페이스는 컴포넌트와 사용자 사이의 연결이다. 인터페이스의 역할은 컴포넌트 구현과 분리되는 것이 그 목적이다. 분리를 위하여 Adapter design pattern을 이용하면 가능하다[3]. (그림 5-a)에서 두 client가 하나의 컴포넌트 인터페이스에 의존한다. 컴포넌트와 인터페이스가 변경되면 두 client 모두가 영향을 받는다. Adapter 패턴을 사용하면 (그림 5-b)처럼 client와 컴포넌트사이의 정보 또는 컴포넌트의 인터페이스에 관련된 정보를 이용하여 변경이 가능하다. 즉, Adapter 인터페이스를 통하여 컴포넌트에 접근할 수 있다. 그러면 컴포넌트 의존은 Adapter와의 관계일 뿐이다. 따라서 새로운 버전이나 유사 컴포넌트로 변경될 때 Adapter만 갱신시키면 시스템의 다른 컴포넌트에 영향을 최소화시키면서 목적 컴포넌트와 연결할 수 있다.



(그림 5) Adapter를 통한 컴포넌트 인터페이스

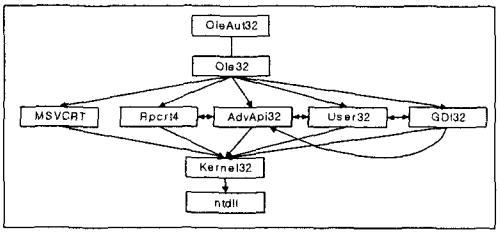
3.3 컴포넌트 형상관리 방법

3.3.1 라이브러리 형상관리

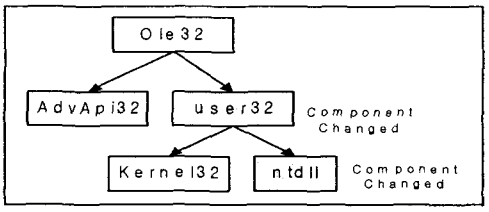
시스템이 새로운 프로그램으로 갱신될 때 수행중인 프로그램은 어떠한 경고 없이 영향을 받을 수 있다. 따라서 갱신되는 컴포넌트들에 대한 인터페이스가 필요하다. 이는 프로그램과 라이브러리 사이의 의존관계를 이용하면 해결될 수 있다. 새로운 프로그램을 설치할 때 다음의 단계로 진행하면 된다.

① 현 시스템의 형상에 대한 스냅 샷을 얻는다.
② 새로운 모듈을 설치한다.
③ 새로운 형상에 대한 스냅 샷을 얻는다.

스냅 샷의 내용은 설치되는 프로그램과 라이브러리에 관한 테이블이 된다. 각 속성들은 날짜, 시간, 크기, 이름, 관계정보 등을 갖고 있다. (그림 6)은 COM 라이브러리의 OleAut32.dll에 대한 의존관계이다. 스냅 샷의 서로 다른 버전은 형상관리 품목으로 취급된다. 이 정보들은 시스템 오류의 원인을 추적하기가 용이하다. (그림 7)은 변경 후의 그래프를 보여준다. 이는 어떤 컴포넌트가 변경되고, 어떤 컴포넌트가 영향을 받는지에 관한 직접적인 정보는 알 수 없지만 컴포넌트의 물리적 표현이 라이브러리이기 때문에 간접 정보는 유용하다.



(그림 6) OleAut32.dll에 대한 의존관계



(그림 7) 변경 후 의존관계

3.3.2 컴포넌트 형상관리

소프트웨어 형상관리(SCM)는 개발과정에서 사용되지만 런타임 시에는 잘 활용되지 못하고 있다. SCM의 주요 원칙은 버전관리, 형상관리, 변경관리이다. 여기에 컴포넌트 관리를 위해서는 이 세 가지 원리와 의존관리가 필요하다[5].

(1) 버전관리(Version Management)

컴포넌트는 이름, 버전, 속성 등을 인식해야 한다. 버전 ID는 새로운 버전으로 갱신될 때 그 변경을 인식해야 하고, 시스템에 통합된 여러 컴포넌트 버전들을 관리한다. 새로운 특성이 추가될 때 이전 버전과 호환되지 않을 경우도 발생한다. 따라서 두 버전을 유지하며 시스템은 동시에 여러 버전이 사용되는 환경을 제공해야 한다. (그림 8)의 IVersion 인터페이스는 버전, 이름, 작성 일자 등에 대한 정보를 보여준다. 각 컴포넌트가 이와 같은 인터페이스를 갖고 있으면 컴포넌트 사이의 의존 정보를 기록할 수 있고, 변경이 가능하다.

```
interface IVersion : IUnkown
{
  HRESULT Name([out, retval] BSTR *name);
  HRESULT Version([out, retval] VERSION *version);
  HRESULT CreationDate([out, retval] Date *date);
  HRESULT TypeOfChange([out, retval] BSTR *name);
  HRESULT History([in] LONG size, [out, size_is(size)]
  HISTORY history[*]);
  HRESULT HasInterfaces([in] LONG numOfElements,
  [out, size_is(numOfElements)] IID interfaces[*]);
  HRESULT UsesInterfaces([in] LONG numOfElements,
  [out, size_is(numOfElements)] IID interfaces[*]);
}
```

(그림 8) 버전 관리 명세서

(2) 형상과 구축 관리

형상과 구축 관리는 특정 버전을 검색, 선택하여 통합시키는 과정이다. 구축(build)은 의존관계 정보를 이용한다. 이 원칙을 런타임 시스템에 적용할 수 있다. 따라서 컴포넌트가 추가되면 실제 이진 코드로 되어 있기 때문에 컴포넌트는 명세서를 갖고 있어야 한다. 이를 이용하여 시스템 형상은 새롭게 만들어진다.

(3) 변경관리(Change Management)

변경관리는 추상화 단계인 논리적 변경에 관한 정보를 제공하며, 새로운 버전 생성 때 중요한 역할을 한다. 모든 컴포넌트 버전은 이전 버전과의 차이점에 관한 정보를 가져야 한다. 그러나 이 정보는 자동 생성을 할 수 없기 때문에 개발자가 제공해야 한다. 이는 컴포넌트 변경에 따른 시스템의 이상 유무를 판단할 수 있다. 시스템과 충돌 가능성은 입출력, 인터페이스, 행위 호환성으로 판별할 수 있다.

(4) 의존관리(Dependency Management)

컴포넌트가 추가될 때는 물리적인 이진 컴포넌트가 전달되기 때문에 컴포넌트 사이의 의존 관계 정보는 전달되지 않는다. 따라서 컴포넌트가 시스템에 어떤 영향을 미치는지에 관한 정보를 알아야 한다. 예를 들어 COM에서는 컴포넌트가 윈도우 레지스트리를 통하여 참조되는 컴포넌트들을 찾는다. 레지스트리는 설치된 모든 컴포넌트들에 대한 *interface id, class id, library location*과 같은 데이터들이 저장되기 때문이다. 따라서 컴포넌트 사이의 관계는 처음 설치될 때 정해지고, 클라이언트는 레지스트리에 있는 서버 컴포넌트를 키 값으로 찾게 되는 것이다. 그러면 COM은 클라이언트 메모리로 관련된 컴포넌트를 로드시킨다.

4. 결론

본 연구는 실시간 환경에서 컴포넌트 형상 관리의 필요성에 관하여 기술하였다. 컴포넌트 기반 시스템에 새로운 컴포넌트가 추가 또는 대체될 때 기존의 컴포넌트가 다른 패키지와 연결되어 있을 경우 변경과 의존관계의 불확실성, 또한 실시간 환경에서의 시스템에 대한 동적 행위에 관한 문제점 등을 분석하였다. 그리고 그 문제점의 해결방안인 컴포넌트 형상관리(CCM) 방법에 관하여 논하였다. 컴포넌트 변경은 라이브러리와 인터페이스로 나누어 설명하였고, 컴포넌트 형상관리는 기존의 SCM에 컴포넌트 관계성을 추가하여 버전관리, 형상과 구축 관리, 변경관리, 의존관리 4가지로 구분하여 컴포넌트 관리기법을 제안하였다.

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.