마이크로소프트 컴포넌트 기술의 발전과 동향

특집 10

목 차

1. COM과 닷넷 컴포넌트 아키텍처
2. 서비스 지향성과 윈도우 커뮤니케이션 파운데이션
3. 소프트웨어 팩토리와 도메인 특화 언어
4. 서비스 지향 개발을 위한 도메인 특화 언어
5. 결 론

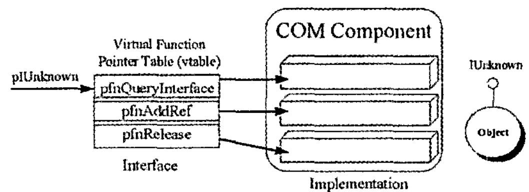
김 명 호
(한국마이크로소프트)

1. COM과 닷넷 컴포넌트 아키텍처

1.1 COM (Component Object Model)

마이크로소프트가 설계하고 구현한 대표적인 컴포넌트 아키텍처로 COM[1]이 있다. COM은 내부 구현을 모르더라도 운영 환경에 관계 없이 언어 중립적 방법으로 사용될 수 있는 컴포넌트의 실현을 가능하게 한다. 실제로 윈도우 시스템의 일부 기능과 다수의 마이크로소프트 애플리케이션은 COM 객체로 구현되어 있어서 이 기술을 활용하는 다양한 애플리케이션 출현의 토대를 제공하였다. 또한 COM 개념을 명시적으로 표현할 수 있는 Visual Basic, Object Pascal 등의 언어와 Visual Studio, Borland Delphi 등의 강력한 개발 환경은 COM의 급속한 보급의 견인차 역할을 하였다.

COM은 컴포넌트 재사용과 상호운용을 위한 바이너리 표준으로, 식별자인 GUID(Globally Unique Identifier), 재사용과 상속, 프로그래밍 모델, 라이프사이클 추상화, 객체 수준 보안 등을 지원하는 완전한 모델로 설계되었다. 모든 COM 컴포넌트는 참조계수 지원을 위한 AddRef와 Release, 인터페이스 조회를 위한 QueryInterface 메소드를 포함하는 최상위 수준 인터페이스 IUnknown을 반드시 구현해야 한다(그림 1).



(그림 1) IUnknown 인터페이스의 구조

COM은 포함/위임(containment/delegation)과 조합(aggregation)을 재사용 수단으로 하며, 대다수의 객체지향 언어에서 지원되는 코드 상속은 그 위험을 회피하기 위해 지원하지 않는다.

COM은 COM 객체, 자동화(Automation), 컨트롤(Control), 복합 다큐먼트(Compound Document),

DAO/ADO/OleDB와 같은 프로그래밍 인터페이스 등의 수단으로 사용될 수 있다. 이들 가운데 자동화는 특히 폭넓은 언어에 의해 채택되고 사용되어 왔다. 자동화를 위한 애플리케이션은 그 애플리케이션이 제공하는 객체 모델을 "형식 라이브러리"(type library)로 제공해야 한다. 이를 위하여 마이크로소프트는 Microsoft Office를 위시한 많은 애플리케이션의 객체 모델을 문서화하여 공개하였다. 이에 따라 내부 구현을 모르더라도 엑셀(Microsoft Excel) 애플리케이션을 구동하고 데이터베이스나 인터넷으로부터 수집된 정보로 내용을 채운 후 특정 형식으로 구성하고 그 결과를 인쇄하는 과정을 자동화 할 수 있는 코드를 지원하는 임의의 언어로 작성할 수 있게 되었다. 이러한 장점을 인식한 다른 많은 개발자들도 그들이 개발한 애플리케이션의 객체 모델을 공개함으로써 자동화 기반의 COM 컴포넌트 재사용을 적극적으로 지원하였다.

이후 윈도우 시스템에는 시스템 경계를 넘어 다른 윈도우 기반 시스템에 있는 COM 컴포넌트도 호출할 수 있는 기능인 DCOM 기술과, COM을 기반으로 트랜잭션 의미론과 보안, 자원 관리 등을 제공하는 MTS(Microsoft Transaction Server) 아키텍처도 개발되었다. MTS는 자바 진영 Java EE의 EJB Container 설계에 직접적인 영향을 주었다. DCOM과 MTS는 분산, 트랜잭션 시스템을 위한 컴포넌트 기술의 집합체인 COM+로 발전하게 된다.

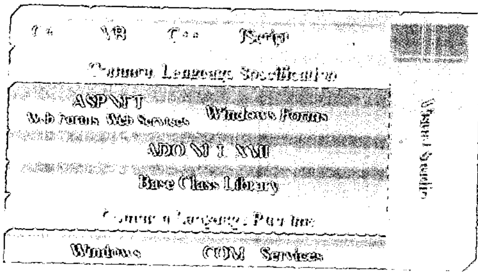
1.2 컴포넌트 아키텍처로서의 닷넷

COM 및 관련 기술은 당시의 첨단 아이디어와 기술, 관행 등을 조합한 컴포넌트 아키텍처와 그 구체적 구현을 제공하고, 실제 사용자의 생태계를 형성함으로써 매우 성공적인 컴포넌트 기술로 존속해 왔다. 그러나 COM은 다른 컴포넌트 기술들과 마찬가지로 몇 가지 중요한 문제가 있었다. COM 컴포넌트를 개발하려면 언어 중립적인 인터페이스를 작성해야 하는데 이에는 IDL(Interface Definition Language)이 사용되었다. 한편 IDL로 인터페이스를 정의한 후 실제 구현에는 C++와 같은 개발 언어를 사용하여 구현해야 한다. 결국 개발자는 단일 목적을 위해 성격이 서로 다른 두 언어를 학습하고 사용해야 하는 불편이 있었다. 또한 컴포넌트의 사용자는 객체 모델의 표현을 위해 사용되는 형식 라이브러리에 의존하는데 만일 형식 라이브러리의 내용이 실제 구현과는 다르게 기술된 경우 심각한 오류에 직면하기도 한다. 버전 관리에도 문제가 있어서 설치된 컴포넌트가 삭제될 경우 이에 의존하는 애플리케이션이 동작하지 않는 사례가 빈번히 발생하였는데 이를 흔히 DLL Hell이라고까지 하였다. 개발 언어에도 프로그래밍 모델, 데이터형의 구조, 실행 환경, 기본 라이브러리의 차이가 커서 언어 중립성을 목표로 하면서도 임피던스의 차이를 극복하기 어려운 경우가 자주 있었다.

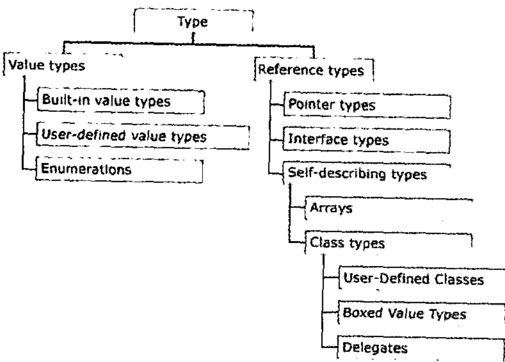
"닷넷"(.NET)은 COM의 이러한 문제점들을 해결하고, 유사한 목적을 지원하면서도 서로 다른 형식으로 제공되던 내부적 기술들을 통합-단일화하며, 마이크로소프트와는 다른 기술을 선택한 외부 시스템이나 컴포넌트와도 효과적으로 상호운용 할 수 있는 플랫폼으로 개발되었다. 즉 COM의 문제점 해결이 닷넷이 출범한 주요 계기 가운데 하나라는 것이다.

닷넷은 언어 중립적이고 안전한 실행 환경인 CLR(Common Language Runtime), 임의의 닷넷 기반 언어에서 사용할 수 있는 근래의 보편적 애플리케이션 도메인을 지원하는 표준 라이브러리 등으로 구성된 "닷넷 프레임워크"(.NET Framework)를 그 핵심으로 하고 있다(그림 2).

닷넷을 위한 언어는 대표적인 개발 언어인 C#, Visual Basic, C++ 이외에도 다수의 다른 언어들이 있으며, 이들은 CLS(Common Language Specification)를 통해 자유로운 상호운용이 가능하다. (그림 3)은 닷넷 CLS에서 데이터 형의 개



(그림 2) 닷넷의 아키텍처



(그림 3) CLS의 데이터 형 계층구조

괄적인 계층구조를 보인 것이다.

즉 닷넷에서는 진정한 의미의 언어 중립성이 보장되면서도 공통의 라이브러리와 CLS를 통해 언어 간 파워의 차이가 거의 없어서 언어의 선택은 개발 경험이나 취향의 문제일 뿐이다.

닷넷에서는 임의의 닷넷 언어를 사용하여 컴포넌트를 개발할 수 있으며, 별도의 IDL이나 형식 라이브러리를 제공하지 않아도 리플렉션을 활용하여 데이터 형의 문제를 자동으로 해결한다. 또한 병행 실행(side-by-side execution)을 지원하여 버전 관리의 효율성을 도모하고 DLL Hell 문제도 해결하였다. 후방 호환성을 위하여 기존의 COM이나 COM+ 기능도 래퍼(wrapper)를 사용하여 별다른 수정 없이 계속하여 활용할 수 있도록 배려하고 있다.

마이크로소프트는 과거 CORBA, DCOM, Java RMI 등이 분산 컴포넌트의 표준을 두고 경쟁하였으나 결국은 바이너리의 고유 기술로는 원활한 상호운용이 어렵다는 판단 하에 XML 기반의 메시징을 기반으로 하는 웹 서비스를 닷넷에서 적극 지원하고 이를 위하여 업계의 여러 경쟁자와도 협력하고 있다. 웹 서비스는 대규모로 분산화된 컴포넌트 기반의 소프트웨어 개발을 위한 근래의 전략적 키워드인 서비스 지향성을 실현하는 핵심 수단이 되었다.

닷넷은 마이크로소프트에 의해 윈도우 환경에 최초로 구현되었으나 ECMA와 ISO에 의해 닷넷 프레임워크의 주요 부분인 CLI(Common Language Infrastructure)와 C# 언어가 표준화되어 개방형 시스템이 되었다[2]. 닷넷(정확히는 CLI와 C#)의 또 다른 대표적인 구현으로는 노벨(Novell)사에 의해 공급되는 Mono가 있다[3]. 닷넷은 진화를 거듭하여 윈도우 비스타(Windows Vista)부터는 닷넷 프레임워크 3.0이 기본 API가 될 예정이다.

이후부터는 서비스 지향성, 도메인 특화 언어를 사용하는 소프트웨어 팩토리 개발 방법론, 이를 구체적으로 실현한 Visual Studio Team Edition의 기능들을 차례로 소개하기로 한다.

2. 서비스 지향성과 윈도우 커뮤니케이션 파운데이션

서비스 지향성(Service-orientation)은 컴포넌트 소프트웨어, 메시지 처리 미들웨어, 분산 객체 컴퓨팅의 경험 등을 활용하여 객체 지향성을 보완한 것이다. 객체 지향 개발이 상호 연관된 클래스 라이브러리로부터 구축되는 "애플리케이션"에 치중하는 반면, 서비스 지향 개발은 독자적인 서비스로부터 구축되는 "시스템"에 치중한다. 여기에서 서비스란 메시지라는 수단을 사용하여 커뮤니케이션 할 수 있고 필요에 따라 상태를 유지하기도 하는 임의의 프로그램을, 클라이언트란 메시지를 사용하여 다른 서비스를 사용하는 또 다른 임의의 프로그램을, 시스템은 시간이 경과

함에 따라 지속적으로 변화할 가능성이 있는 서비스의 집단을 각각 의미한다. 결국 서비스 지향성은 유연성과 독자성, 가변성의 중요성이 강조되는 오늘날의 상황을 위한 본질적인 개념의 전환을 의미한다. 문제는 이러한 개념의 차이가 개발 경험에 있어서도 심오한 영향을 미친다는 것이다. Don Box는 서비스 지향 개발이 다음 4대 교의에 기초하고 있다고 설명한다[4]:

교의 1: 경계가 명확하다.
(Boundaries are explicit.)
교의 2: 서비스는 독자적이다.
(Services are autonomous.)
교의 3: 스키마와 계약을 공유한다.
(Share schema & contract, not class.)
교의 4: 호환성은 정책에 의해 결정된다.
(Compatibility is determined based-on policy.)

많은 사람들에 의해 다양하게 설명되는 서비스 지향 패러다임의 기술적 핵심이 4대 교의로 압축될 수 있는 이유는 이 패러다임의 근본 목적이 불필요한 가정에 의존하지 않고 단순성으로 회귀함으로써 보다 유연한 시스템을 얻고자 하는 것이기 때문이다. 그러나 이러한 수준의 단순성은 많은 서로 다른 해석이 난무하는 원인이 되기도 한다.

2.1 경계가 명확하다.

서비스 지향 애플리케이션은 흔히 넓은 지역, 여러 신뢰 기관, 서로 다른 실행 환경에 산재한 서비스들로 구성되며, 서비스 사이의 상호작용은 필연적으로 경계를 교차하게 된다. 경계를 교차하는 데는 복잡도나 성능 관점에서 볼 때 큰 비용이 수반될 수 있다. 서비스 지향 설계는 이러한 비용을 인식하여 경계 교차에 프리미엄을 부과한다. 이를 위하여 프로그램의 모든 기능을 자동적으로 외부에 노출하는 대신 자발적이고(opt-in), 명시적으로(explicit) 선언된 기능에 한하여 경계 외부와의 상호작용을 허락한다. 또한 경계 교차 커뮤니케이션의 비용 때문에 서비스 지향성은 비용 요소가 은폐되는 묵시적 메소드 호출보다는 이를 분명히 드러내는 명시적 메시지 전달 모델에 기초한다. 이로 인하여 다양한 메시지 전달 시나리오를 효과적으로 구사할 수 있으며, 보다 높은 수준의 동시성을 실현할 수도 있다.

2.2 서비스는 독자적이다.

서비스의 독자성은 서비스들을 배포하고 버전 관리하는 여러 측면에 중대한 영향을 미치게 된다. 서비스 지향 개발에서는 모든 서비스를 일괄적으로 배포하지 않는다. 개별 서비스는 물론 원자적으로 배포되지만 전체 시스템의 총괄적 배포 상태는 정지 상태에 있는 것이 아니다. 예를 들어 어떤 서비스는 이를 사용하는 어떤 시나리오나 애플리케이션도 개발되지 않은 시점에 먼저 배포될 수도 있다. 서비스 지향 애플리케이션은 관리자나 개발자의 직접적인 개입이 없이도 시간에 따라 진화할 수 있다. 결국 서비스 지향성은 서비스간 상호작용의 복잡도를 감소시켜 편재성을 증진시키기 위한 모델인 셈이다.

독자적 서비스로 구성된 시스템은 고장과 보안 이슈에도 새로운 대처 방법을 요구한다. 서비스는 클라이언트가 어떠한 통지도 없이 불능 상태에 빠질 수 있음을 가정해야 한다. 시스템의 무결성을 유지하기 위하여 서비스 지향 설계는 트랜잭션이나 이중화 배포 등과 같이 부분 고장 모드에 대처하는 여러 기술을 사용하여야 한다. 또한 많은 서비스가 공용 네트워크에 배포될 수 있으므로 서비스 개발자는 악의적인 목적을 지닌 모든 비정상적 입력에 대처하여야 하며, 사용자 인증과 권한 부여와 같은 보안 기능을 고려하여야 한다.

2.3 스키마와 계약을 공유한다.

객체 지향 프로그래밍은 개발자들이 클래스

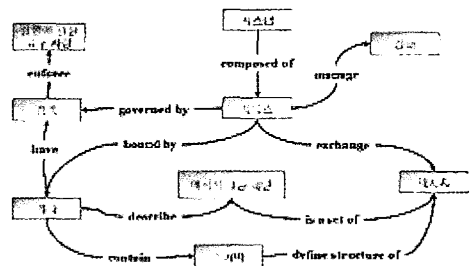
형태로 새로운 추상화를 작성하는 것을 권장한다. 다수의 현대적 개발 환경은 새로운 클래스의 작성을 용이하게 할뿐만 아니라 클래스의 개수가 증가하더라도 개발 과정을 보다 단순하게 하는 여러 수단을 제공한다. 클래스는 구조와 동작을 단일 구문에 표현할 수 있는 편리한 추상화를 제공한다. 애플리케이션의 모든 구성 요소를 한꺼번에 실험하고 배포할 수 있는 객체 지향 개발에서는 구조와 동작, 특히 상속에 의존하여 코드까지 공유하는 관행이 효과적인 개발 전략일 수 있다.

개별 서비스의 독자성이 강조되는 서비스 지향 개발에서는 현실적으로 이러한 수준의 친근성을 결코 기대할 수 없으므로 구조와 동작을 엄격하게 분리하는 전략에 의존한다. 즉 서비스는 자신이 전달하거나 전달받을 수 있는 메시지의 구조(스키마)와 메시지 교환 패턴을 정의한 계약만을 외부에 공개한다. 스키마는 데이터 형이나 클래스 대신 메시지의 유효성 여부를 자동으로 판독하고 검증할 수 있는 형태로 표현한다. 서비스를 위한 스키마와 계약은 넓은 범위의 시간과 위치에서 참조될 수 있으므로 극도의 안전성을 요구한다.

2.4 호환성은 정책에 의해 결정된다.

객체 지향 설계에서는 구조적 호환성과 의미적 호환성이 종종 혼동되지만 서비스 지향성에는 이들을 별도로 취급한다. 즉 구조적으로 호환되는 서비스라 하더라도 의미적 호환성이 보장되지 않으면 상호작용을 허락하지 않는다. 구조적 호환성은 계약과 스키마에 의존하며 의도적으로 혹은 자동적으로 검증할 수 있다. 한편 의미적 호환성은 정책 형태로 표현된 기능과 요구사항의 명시적 선언에 의존한다. 모든 서비스는 자신의 기능과 요구사항을 머신 판독이 가능한 형태의 정책 식(policy expression)으로 표현한다. 정책 식은 해당 서비스의 정상적 동작을 위하여 반드시 만족해야 하는 조건과 보장사항(흔히 단언이라고 함)을 나타낸다. 정책 단언은 전세계적으로 유일하고 안정적인 명칭으로 확인된다. 정책 단언은 그 단언의 정확한 해석을 위한 매개변수를 포함할 수도 있다.

(그림 4)는 지금까지 설명한 서비스 지향 패러다임의 교의들을 정리한 개념도이다.



(그림 4) 서비스 지향성의 개념도

이 그림이 뜻하는 바의 핵심은 서비스 지향 시스템 개발에서 경계 외부에서는 제시한 요소에만 의존하여야 하며, 경계 내부에서는 이러한 요소들을 구현하는 기술의 선택에 완전한 자유가 존재한다는 뜻이다. 즉 서비스 사이의 경계의 의미를 명확히 하고, 경계를 넘나드는 연산은 최소한의 상호 의존성만 가지도록 하는 것이 서비스 지향 개발의 근본적인 개념이다.

2.5 윈도우 커뮤니케이션 파운데이션

닷넷 웹 서비스(ASP.NET Web Services)와 윈도우 커뮤니케이션 파운데이션(Windows Communication Foundation, WCF)은 메시지를 교환하는 서비스로 구성된 시스템을 효과적으로 구현할 수 있는 프로그래밍 모델과 프레임워크를 제공한다. 이들은 서비스의 구현에 개입된 모든 클래스와 메소드를 자동적으로 외부에 공개하지 않고 개발자가 자발적이고 명시적인 방법으로 선언한 것만을 공개의 대상으로 한다. 경계

의 선언에는 애트리뷰트(attribute)가 사용된다. 다음 예제는 WCF를 사용하여 서비스를 작성한 것이다.

```
using System.ServiceModel;
[ServiceContract]
public class ExposeMe
{
    [OperationContract]
    public int Method1() { . . . }
    public int Method2() { . . . }
    [OperationContract]
    private int Method3() { . . . }
}
```

ExposeMe 클래스에는 "ServiceContract" 애트리뷰트가, Method1과 Method3 메소드에는 "OperationContract" 애트리뷰트가 각각 사용되었다. 이들의 의도는 명시적으로 선언된 클래스와 메소드 이외에는 외부에서 참조할 수 없도록 하는 분명한 경계를 설정하기 위한 것이다. 특기할 사항은 Method2는 public 메소드이지만 공개되지 않았으며, Method3는 private 메소드임에도 불구하고 공개하였다는 것이다. 이는 서비스의 경계가 상속과 다형성, 재정의를 위한 클래스의 경계와는 구별되는 것임을 강조하기 위한 것이다.

이처럼 WCF는 서비스 지향 패러다임을 명시적으로 지원하는 프레임워크를 제공하기 위하여 설계되었다. WCF는 윈도우 비스타 기본 API인 닷넷 프레임워크 3.0의 일부이며 후방 호환성을 위해 기존 시스템을 위한 닷넷의 확장으로도 제공될 예정이다.

3. 소프트웨어 팩토리와 도메인 특화 언어

프로그래밍 언어 중립성과 효과적인 재사용을 목표로 하는 컴포넌트 기술이 오랜 기간 개발, 보급되어 왔음에도 불구하고 아직도 재사용은 기대치에 이르지 못한 것이 현실이다. 이에는 많은 원인이 있지만 대표적으로는 전체적 개발, 지나친 일반성, 일회성 개발, 그리고 프로세스 미성숙 등이 주요 이유로 지적된다.

3.1 기존 재사용 기법의 문제점

3.1.1 전체적 (monolithic) 개발

대다수의 소프트웨어는 상업적으로 인정받을 만한 수준의 조립에 의한 개발이 거의 이루어지지 않고 일일이 처음부터 완전하게 개발되고 있다. 플랫폼 종속적 프로토콜에 의존하는 컴포넌트는 플랫폼 경계에서의 조립을 근원적으로 어렵게 한다. 현재의 컴포넌트 명세와 패키징 기술은 컴포넌트와 그들의 상호작용과 관련된 특성을 충분히 표현하지 못하여 조립에 의해 개발되는 소프트웨어의 특성을 예측하기 어렵다. 또한 적극적인 캡슐화의 결과 컴포넌트의 경계가 너무 일찍 고정되어 변형을 금지하며, 리팩토링 하기도 어렵다. 기존 컴포넌트를 재사용하는 것이 보다 비용-효과적임에도 불구하고 본능적으로 컴포넌트를 재 작성하는 경우도 허다하다.

3.1.2 지나친 일반성

전형적인 비즈니스 애플리케이션은 작은 부분에 한해 범용 3세대 언어가 필요함에도 불구하고 대부분의 소프트웨어가 그러한 언어에 종속되어 개발되고 있다. UML과 같은 모델링 언어도 스케치를 위한 수단으로는 적합하지만 모델-기반 개발이나 실행 언어로는 부적합하다. UML은 너무 일반적이어서 특정 도메인에 관한 상세한 정보를 표현하기 어려우며, 확장 기능이 취약하여 특화 하는 것도 어렵다. 플랫폼 독립적인 모델로부터 코드를 생성하는 CASE 도구들은 플랫폼 기능을 활용할 수 없고, 개발자가 아키텍처나 코드 생성을 제어할 수도 없어서 낮은 품질의 코드가 생성되는 경우가 허다하며, 생성된 코드와 수작업에 의한 코드가 효과적으로 조합되지도 않

는다. 도구가 유지하는 메타데이터는 라이프 사이클의 자동화에 충분히 사용될 수 있음에도 불구하고 이들이 효과적으로 통합되지 않고 있다.

3.1.3 일회성 개발

소프트웨어 개발 프로젝트는 대개 손익분기점과 인도 시기에 집중하므로 전체적인 아키텍처는 지적 호기심이나 학술적인 것으로 치부되는 경우가 많다. 기존 자산을 분석하고 평가하는 노력이 거의 없으며, 새로이 생성된 자산이 다른 문맥에서도 재사용될 수 있는지 확인하는 데는 거의 관심을 기울이지 않는다. 그 결과 대부분의 프로젝트에서는 재사용 가능한 컴포넌트를 확인하고, 다른 프로젝트에서도 사용할 수 있도록 문서화하고 패키지화하는 사후 검토를 거의 하지 않는다.

3.1.4 프로세스의 미성숙

소프트웨어 산업은 약속한 기간과 경비 내에 소프트웨어 제품을 인도하지 못해 왔다는 것이 일관된 관측이다. 이러한 문제에 대한 해결 시도는 크게 두 선택으로 나누어진다. 한 선택은 변경 가능성을 담보로 하여 복잡성을 제어하는 것으로 "전통적" 프로세스를 포함하여 RUP, Waterfall 등이 이에 해당한다. 다른 선택은 복잡성을 담보로 하여 변경을 제어하는 것으로 SCRUM, XP와 같은 "민첩한"(agile) 방법론이 이에 해당한다. 자동화는 잘 정의된 프로세스만 자동화할 수 있기 때문에 재사용을 위해서는 프로세스 자체가 먼저 발달하여야 한다.

이러한 문제들과, 소프트웨어 재사용의 경제적 모델에 대한 잘못된 이해로 인하여 제조업과 같이 보다 성공한 산업 분야에 비해 소프트웨어의 대량생산에는 많은 장애가 존재해 왔다.

3.2 소프트웨어 팩토리

특정 도메인을 위한 소프트웨어를 개발할 때 그 도메인과 관련된 지식이 흔히 개발되는 소프트웨어에 내포된다. 문제는 이러한 지식이 소프트웨어 내에 감추어져 있어서 이를 포함하고 있는 소프트웨어의 원래 영역 밖에서 재사용할 수 없다는 것이다. 소프트웨어 팩토리(Software Factories, SF)는 특정 도메인에 관한 기존의 전문화된 지식과 그 도메인 내에서 개발된 애플리케이션들을 수집하여 재사용 가능한 생산 자산으로 변환하기 위한 적극적인 시도이다[5]. 이러한 지식은 유사한 형태의 다른 애플리케이션들을 위한 청사진을 생성하는데 사용할 수 있다. 결국 소프트웨어 팩토리는 특정 도메인 내에 존재하는 가치 있고 재사용 가능한 생산 자산으로부터 애플리케이션을 생성하는 것을 근본 목적으로 하고 있다.

지금껏 매우 숙련된 애플리케이션 개발자나 아키텍트라 하더라도 구현 수준 작업에 많은 시간을 사용해야 했다. 초급 개발자는 대개 적절한 도메인 지식이 부족하여 중견 개발자로부터의 지도가 요구된다. 중견 개발자는 자신의 지식뿐만 아니라 그 지식의 구현까지 함께 전달하게 마련이다. 이러한 개발 방법은 초기의 가내 수공업과 마찬가지 패턴으로 단일 작업자가 개별적인 요구가 있을 때마다 특화된 솔루션을 일일이 생성하는 것과 마찬가지이다. IT 산업에 종사하는 대부분의 사람들은 어떤 형태의 표준과 모듈화가 소프트웨어 개발의 효과적 산업화를 가능하게 하는 수준의 재사용에 핵심인 것에 동의하고 있다. 다만 표준화와 모듈화를 달성하는 방법에 대해서는 동의하지 않고 있다. 소프트웨어 팩토리는 소프트웨어가 모듈화되어 재사용 가능한 자산이 되도록 하는 프로세스를 규정하기 위한 노력을 강조한 것이다.

소프트웨어 팩토리는 모델과 패턴, 도메인 특화 언어, 생산 라인 등의 요소로 구성된다.

3.2.1 모델과 패턴

모델은 프로젝트의 여러 개념에 대한 일관성

있는 표현을 유지할 수 있다는 점에서 매우 중요하다. 예를 들어 모델에서 Person 객체를 그리고 조작하기는 쉬운 일이지만 동일한 객체를 보다 저 수준에서 조작하려면 그 객체가 클래스 파일이나 테이블과 열로 표현되어 있어서 보다 어렵거나 불편할 수 밖에 없다.

모델은 소프트웨어 시스템의 핵심 추상화와 개념을 표현하고 조작하는 역할 뿐만 아니라 그로부터 구현에 관한 구체적 결과물을 생성하는 작업에 효과적으로 사용될 수 있어야 한다. 생성된 결과물을 수정하면 그 내용이 모델에도 반영될 수 있도록 의미적 연관성과 상호작용(round-trip)이 가능한 것이 바람직하다.

3.2.2 도메인 특화 언어

도메인 특화 언어(Domain Specific Language 혹은 DSL)는 예전부터 도메인을 위한 개념의 추상화를 제공하는 수단으로 사용되어 왔다. 지금까지는 SQL이나 HTML과 같이 널리 사용되고 일반적인 관심사를 다룰 수 있어서 비용-효과적인 DSL만 관심을 끌어 왔으며, 보다 수직적인 다른 도메인을 위한 DSL을 생성하는 것은 비용적으로 시도하기 어려웠다.

그러나 최근 여러 업체가 DSL 생성을 위한 도구나 플러그-인을 발표하였다. 이러한 DSL이 생성되고 나면 이를 활용하여 보다 높은 수준에서 컴포넌트를 사용하고 보다 높은 수준의 재사용을 활용하여 작업할 수 있다.

3.2.3 소프트웨어 생산 라인

소프트웨어 생산 라인(Product Line)은 거의 완성품에 가까운 제품을 위해 구성, 조립, 패키지될 수 있는 컴포넌트의 집합들을 의미한다. 결과 제품은 개발자에 의해 프로젝트 측면에서 매우 특화된 측면을 위한 조정만 요구하여야 한다. 이를 위하여 소프트웨어는 분명하고 재사용 가능한 컴포넌트들로 주의 깊게 분할되어야 하며, 이러한 컴포넌트들은 조화로운 방법으로 손쉽게 맞출 수 있어야 한다. 소프트웨어 생산 라인에서 임의의 프로젝트는 활용하고자 하는 컴포넌트를 선택하고, 구성에 따라 애플리케이션을 생성하여야 한다.

지금껏 여러 업체들이 소프트웨어 팩토리와 유사한 목적으로 도구와 컴포넌트를 제공하려고 시도해 왔지만 도구가 유연하지 못하거나 고유 형식을 사용하여 대부분 실패하고 말았다. 이러한 경향은 개선되고 있어서 영향력 있는 업체들이 소프트웨어 팩토리 방법론을 지원하는 컴포넌트를 발표할 준비가 되어가고 있다.

마이크로소프트는 Visual Studio 2005의 발표와 함께 DSL을 생성하는 것뿐만 아니라 이들을 IDE에 통합하는 것까지 가능하게 하는 여러 부가기능과 플러그-인을 제공할 예정이다. 개발자들은 IDE 내에서 특정 도메인을 위한 언어를 조작하고 사용할 수 있다. 그 실증적 사례로 웹 서비스 연결성, 논리적 데이터센터, 서비스 배치 검증을 위한 DSL을 Team Architect Edition에 기본적으로 제공한다. 또한 Borland(UML 2.0), Unisys(여러 산업), Siemens(의료 이미징 기기 소프트웨어), Kinzan(웹 개발), Nationwide(금융 산업) 등을 위시한 여러 업체가 마이크로소프트와 협력하여 Visual Studio에서 사용될 수 있는 DSL을 개발할 것을 선언한 바 있다. 또한 Sun Microsystems도 Project Ace라는 명칭 하에 자체적인 소프트웨어 팩토리 기술 개발을 위해 노력하고 있다.

DSL/SF는 UML 기반의 MDA 방법론과 긴장 관계에 있다. 마이크로소프트의 입장은 UML이 도메인을 고려하지 않은 일반적 추상화를 추구하기 때문에 모델 기반의 "정밀한" 개발을 위한 수단으로 적합하지 않다는 것이다. 즉 UML의 보편성으로 인한 문서화와 의견교환 수단으로서의 가치는 충분히 인정하며 이를 위하여 Visual Studio 내에 UML 도구를 제공하기도 하지만, 특정 도메인의 의미론을 충실하게 반영하여 효과적으로 개

발을 지원하는 도구로는 간주하지 않는다.

4. 서비스 지향 개발을 위한 도메인 특화 언어

소프트웨어 팩토리는 특정 도메인에 국한되지 않는 일반적 개발 방법론이다. 그러나 Visual Studio 2005 Team Edition for Software Architects(VS/SA)[6]에는 서비스 지향 개발을 위한 여러 DSL이 기본적으로 제공되고 있어서 서비스 지향성의 직접 지원 의도를 분명히 하고 있다.

VS/SA가 제공하는 도메인 특화 언어 기반의 도구들은 아키텍트와 개발자들이 서비스 지향 애플리케이션을 구성하고 이를 목표하는 데이터센터를 기준으로 배치하는 것을 준비할 수 있도록 적극 지원하기 위한 것이다. 이 목적을 위해 다음 세 도구를 포함하는 분산 시스템 디자이너 Distributed System Designers(DSD)가 제공된다.

1) System Designer (SD): 애플리케이션 아키텍트와 개발자들이 여러 서비스와 이들을 사용하는 컴포넌트들을 설계, 구축, 구성하는 것을 보조한다.

2) Logical Datacenter Designer (LDD): 인프라 아키텍트와 운영팀이 구성의 제약조건, 논리적 토폴로지, 애플리케이션에 적용될 정책 등을 포함하는 데이터센터 아키텍처의 설명을 보조한다.

3) Deployment Designer (DD): 개발자와 운영팀이 애플리케이션을 배치하기 전에 그들의 데이터센터 아키텍처와 제약조건 등을 기준으로 검증하는 것을 보조한다.

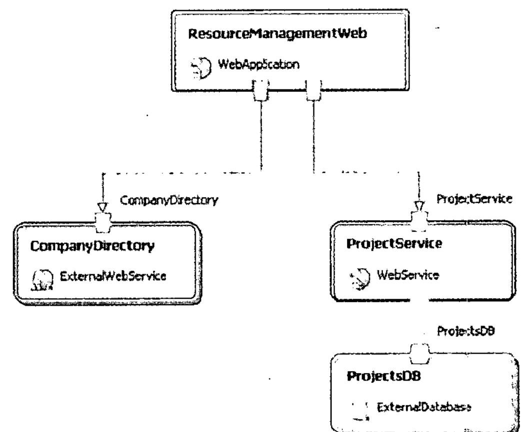
이들은 개별 도구로서의 가치도 중요하지만 높은 수준의 상호작용을 통하여 아키텍트, 개발자, 운영팀 사이의 의견 교환과 일관성 유지를 지원하는 것에 주목할 필요가 있다. 예를 들어 애플리케이션 개발자가 제약조건을 설정하여 데이터센터 설계에 반영되도록 요구할 수 있기도 하지만 데이터센터 설계자가 제약조건을 설정하여 애플리케이션이 그 조건을 만족하는가를 검증하도록 활용할 수도 있다.

4.1 System Designer (SD)

애플리케이션이 제공하는 서비스들과 애플리케이션 사이의 연결을 설명함으로써 애플리케이션의 설계적 측면을 정의하기 위한 도구이다. 아키텍트는 이 도구를 사용하여 애플리케이션의 골격을 정의하고, 설정과 설계를 조사할 수도 있다. 이러한 애플리케이션들로부터 하나 이상의 시스템이 구성될 수 있으며, 구성된 시스템의 유효성을 배치 환경의 요구사항을 기준으로 검증할 수 있다.

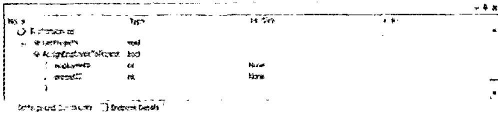
필요한 경우 이 도구는 개발자들이 나중에 완전히 구현하여야 할 코드 골격을 생성할 수도 있다. 애플리케이션이 일단 생성되고 나면 설계와 코드의 일관성이 도구에 의해 자동으로 유지된다. 코드로부터 역공학을 통하여 설계를 구성할 수도 있다. 기존 시스템을 배치 이전에 검증하고자 하는 경우에 역공학의 필요성이 주로 발생한다.

(그림 5)는 SD를 사용하여 외부 데이터베이스를 사용하는 ASP.NET 웹 서비스 하나와 외부 웹 서비스를 사용하는 웹 애플리케이션을 위한 설계를 작성한 사례를 보인 것이다.



(그림 5) SD를 사용한 애플리케이션 설계

웹 서비스의 상세한 내용을 정의하려면 그 서비스의 종단점(청색으로 채워진 사각형)을 클릭하면 된다. (그림 6)은 ProjectService의 상세설정 윈도우를 보인 것이다.

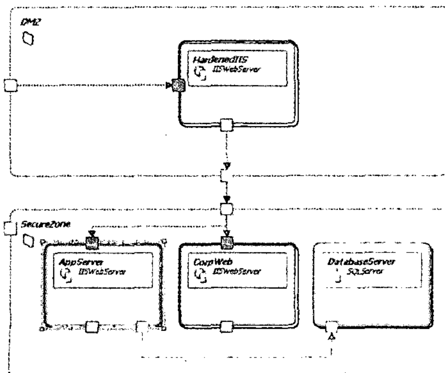


(그림 6) 종단점 상세 설정 윈도우

서비스를 구현하려면 해당하는 웹 서비스 컴포넌트를 우측버튼으로 선택하여 Implement를 선택하면 된다. 그 결과 Solution Explorer에는 그 서비스를 위한 프로젝트가 새로이 생성되며, Code 폴더에는 구현을 위한 코드의 골격을 수록한 파일이 생성된다. Solution Explorer에서 이 파일의 명칭을 선택하여 코드 에디터를 호출할 수 있다 코드 에디터에서 구현을 변경하면 그 내용이 서비스에 동시에 반영된다.

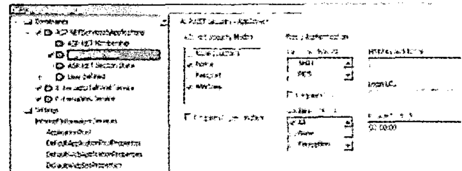
4.2 Logical Datacenter Designer (LDD)

목표 배치 환경에 관한 중요한 정보를 개발자에게 제공하기 위한 도구이다. (그림 7)은 이 도구를 사용하여 논리적 데이터센터를 작성한 예를 보인 것이다.



(그림 7) LDD를 사용한 데이터센터 설계

이 도구를 사용하여 논리적 데이터센터 다이어그램 컴포넌트의 제약조건을 설정할 수도 있다. (그림 8)은 AppServer 컴포넌트의 Settings and Constraints 윈도우를 사용하여 ASP.NET 보안을 설정하는 예를 보인 것이다.



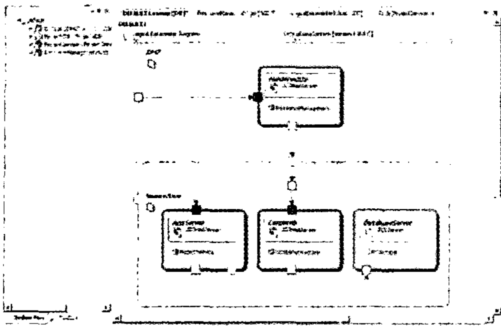
(그림 8) ASP.NET 보안 제약조건의 설정

4.3 Deployment Designer (DD)

애플리케이션 설계의 컴포넌트들을 논리적 데이터센터의 컴포넌트에 매핑 하여 가상적으로 배치를 실험하고 검증하기 위한 도구이다. 먼저 논리적 데이터센터로 구성된 실험적 배치를 위한 DD 인스턴스를 생성한다. 가상적인 배치는 SystemView에 있는 컴포넌트(서비스)를 DD 윈도우 컴포넌트에 드래그-앤-드롭 방식으로 수행할 수 있다.

드래그-앤-드롭이 항상 가능한 것은 아니다. 예를 들어 HardenedIIS 컴포넌트가 웹 서비스를 호스트 하지 않도록 설정된 경우 ProjectService를 드래그하여 이 컴포넌트에 드롭 하려는 시도는 근원적으로 차단된다. (그림 9)는 배치를 완료한 상태를 보인 것이다.

가상적 배치가 끝난 후 배치 다이어그램에서 우측 버튼을 클릭하여 Validate를 선택하면 애플리케이션 설계와 데이터센터 설계 사이의 제약조건 만족 여부를 검사한다. 제약조건을 만족하지 않는 컴포넌트가 있으면 에러가 표시된다. 에러 메시지를 선택하면 SD 윈도우에 문제를 유발하는 애플리케이션 컴포넌트가 E 라는 붉은 색 마크와 함께 나타난다. 그 컴포넌트의 Settings



(그림 9) DD를 사용한 가상적 배치의 결과

and Constraints 기능(그림 6에서 소개한 종단점 상세 설정 윈도우에 있음)을 사용하여 제약조건을 만족하도록 설정한 후 다시 검증을 시도할 수 있다.

5. 결 론

컴포넌트 기술은 단일 프로그래밍 모델과 사실상 특정 프로그래밍 언어에 종속되었던 객체지향 기술과는 달리 프로그래밍 모델과 언어 중립적인 높은 수준의 재사용을 위한 효과적인 해결책으로 제시되었다. 그러나 기술의 복잡성과 과도한 주도권 및 표준 경쟁으로 기대만큼 큰 마켓 모멘텀을 가지지 못하였다.

마이크로소프트는 COM 기술을 개발하여 운영체제의 주요 부분과 다수의 애플리케이션에 적용함으로써 이론 수준이 아닌 개발자와 사용자들에 의해 폭넓게 사용되는 실제적인 활용 사례와 컴포넌트 생태계를 구축하였다.

닷넷은 COM의 정신을 계승하되 언어 중립성과 안정성, 생산성 등을 크게 강화한 새로운 컴포넌트 아키텍처를 제공하였다. 닷넷은 또한 XML 기반의 웹 서비스를 근원적으로 제공함으로써 다른 컴포넌트 아키텍처 기술과의 효과적인 상호운용과 서비스 지향적 개발을 적극 지원한다. 특히 윈도우 비스타 시스템의 기본 API 닷넷 프레임워크 3.0의 구성요소인 WCF는 서비스 지향적 개발을 명시적으로 표현하는 강력한 추상화와 도구들을 포함하고 있다.

마이크로소프트는 컴포넌트 기술을 위한 개발 도구와 개발 방법론도 제공한다. 닷넷을 위한 개발도구인 Visual Studio는 컴포넌트와 웹 서비스를 위한 다양한 지원 기능이 포함되어 있다. 또한 도메인 특화 언어 기반의 소프트웨어 팩토리를 통해 컴포넌트와 모델기반 개발 방법론의 실효성을 한층 더 강화한 새로운 개발 방법론을 제시하고 있다.

참고문헌

[1] Don Box, Essential COM, Addison-Wesley International, 1997.

[2] ISO Standards: ISO/IEC 23270(C#), ISO/IEC 23271 (CLI Partition I-IV), ISO/IEC 23272 (Partition IV XML File).

[3] Project Mono, Novell, http://www.mono-project.com/Main_Page.

[4] Don Box, "A Guide to Developing and Running Connected Systems with Indigo," http://msdn.microsoft.com/msdnmag/issues/04/01/Indigo/default.aspx#S1.

[5] Jack Greenfield and Keith Short, Software Factories: Assembling applications with patterns, models, frameworks, and tools, Wiley, 2004.

[6] Visual Studio 2005 Team Edition for Software Architects, http://msdn.microsoft.com/vstudio/teamsystem/products/arch/default.aspx.

저자약력

김 명 호

1989년 한국과학기술원 전산학과 졸업 (공학박사)
1989년~1999년 동아대학교 컴퓨터공학과 교수
1999년~2002년 비트웹 기술이사
2002년~2003년 모하비소프트 대표
2003년- 현재 한국마이크로소프트 최고기술임원 (NTO)
관심분야 : 애플리케이션 플랫폼, 분산 시스템
이 메 일 : mhkim@microsoft.com