.NET 기술 구성요소

이복영* · 박지환**

1. Common Language Runtime(CLR)

CLR은 .NET의 핵심이다. .NET Framework의 핵심적인 두 가지 구성요소 중 하나로 각각의 언어로 작성 되어진 Application이 작동하고 서로 협력할 수 있는 실행환경 이라고 할 수 있다. CLR이 이러한 환경을 제공할 수 있는 이유는, CLS (Common Language Specification), 즉 CLR에서 동작할 수 있는 .NET Application을 만들기 위해 지켜야 할 규칙들의 묶음이라고 할 수 있다. 누구든지 .NET Application을 만들고자 할 경우 이 규칙을 따라야 한다. 이것만 지킨다면 .NET으로 만들어진 어떠한 컴파일러로 작성된 어떤 프로그램이라도 다른 .NET Application과 함께 실행되어 질 수 있다.

CLR에 관련된 중요한 개념 중에 하나가 Managed Code라는 개념이다. 이는 CLR의 환경하에서 실행될 수 있도록 작성되었으며, 따라서 CLR의 보호 아래에서 실행되면서 관리되는 Code를 말한다.

◆CLR을 따르는 Managed Code는 다음과 같은 규칙을 따른다.

- Component의 Interface와 Type을 기술하는 Meta Data를 읽는다.
- Code Stack의 실행
- Exception 처리
- Security 정보 검색

◆CLR의 설계 목표는

- 개발의 단순화
  Code의 재사용을 위한 표준을 정의한다. Garbage Collection과 기타 메모리 관리 기능을 포함하는 광범위한 서비스를 제공한다.
- 응용프로그램 배포의 단순화
  컴포넌트는 레지스트리(Registry)에 등록하는 대신 Meta Data를 사용한다.
  하나의 시스템에서 같은 이름을 가진 다양한 버전의 컴포넌트의 등록을 지원한다.
  단순히 복사만 함으로써 설치되고, DEL 명령어로 삭제할 수 있다.
- 개발언어의 지원
  개발언어와 도구에 대한 다양한 Base Class를 지원한다.
- 다양한 언어를 지원
  모든 .NET 언어에 사용되는 CTS를 정의한다.
- 프로그래밍 모델 단일화
  Common Framework에서 언어와 도구를 구성한다. (예를 들면 ASP.NET, C#,

*(주)필라넷 기술이사
**부경대학교 전자컴퓨터정보통신공학부 교수

VB.NET은 모두 동일한 Base Class를 사용한다.

CLR은 .NET 응용프로그램을 개발하고자 하는 개발언어의 코드가 반드시 따라야 하는 규칙을 정해 놓고 있다. 사실은 CLR 외에도 .NET언어가 따라야 하는 몇 가지 규칙이 더 있는데, CTS (Common Type System), CLS(Common Language Specification)이 .NET 언어가 따라야 하는 규칙이다. 이러한 규칙의 묶음은 모든 Application이 어떤 언어로 작성되었는지 상관없이 전체적으로 어떻게 통일된 방식으로 작동해야 하

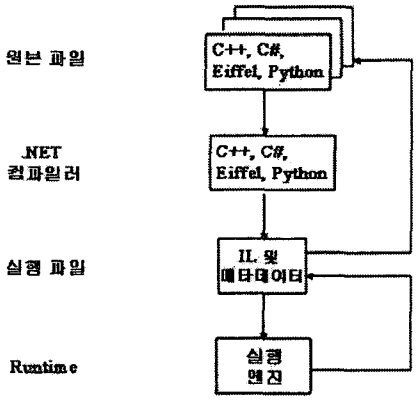


그림 1. CLR(Common Language Runtime)의 실행 구조

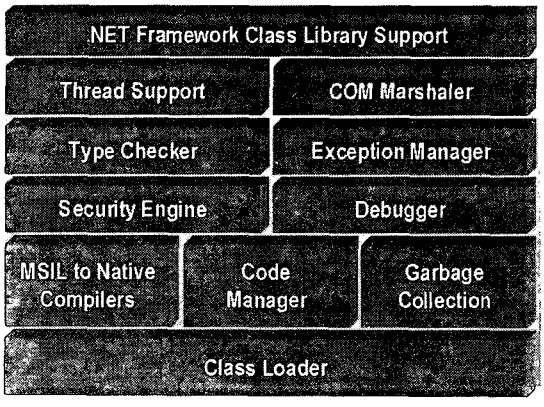


그림 2. CLR의 구조

는가를 명시해 주는 것이다. .NET Application의 통일된 동작 방식은 .NET의 핵심이라고 할 수 있다. 그러나, 일반적인 개발자들은 이러한 규칙들을 컴파일러를 작성하는 사람들한테 맡겨도 무관할 것이다. CLR에 대하여 .NET Framework에서도 상세하게 다루어 질 것이다.

CLR의 구조를 좀 더 세부적으로 살펴보면

◦ Class Loader

메모리에 Class를 로딩하고, 실행을 위해 준비하는 것으로, Assembly, Name, Version 정보, Reference 정보 같은 것들이 포함되어 있는 컴포넌트의 메타 데이터를 읽어 들여 동작한다. Class Loader는 컴파일해서 실행하기 전에 Code를 Check하는 중요한 기능을 수행한다. 예를 들면, .NET Application이 CLR 환경에서 실행 될 때 Evidence를 제출하게 되는데 CLR은 이를 근거로 Code의 보안 사항을 Check한다.

◦ MSIL과 Native 컴파일러

MSIL은 서로 다른 프로그래밍 언어로부터 다양한 정보를 컴파일러에 의해 해석하여 생성된 명령어들의 집합이라고 할 수 있다. MSIL은 Code Stack을 실행해서 생성하게 된다. 이는 완전히 컴파일된 이진 Code가 아니기 때문에 컴파일러 중간언어로 간주된다.

실제로 MSIL Code로 컴파일 되는 부분은 프로그래밍 언어로 작성된 순수 Source Code이다. MSIL Code는 반드시 실행되기 전에 Native Code로 변환 되어야 한다. CLR은 Native 컴파일러에게 이러한 작업을 위임한다. 컴파일러는 여러 종류가 있는데 대표적인 분류는

- JIT(Just In Time) 컴파일러: MSIL을 분석하고, 필요할 경우 Native Code로 바꾸는 최적화된 컴파일러이다.

- 기본 방식의 컴파일러: Assembly는 완전히 Native Code로 변환되고, JIT(Just In Time) 컴파일러에 비해 최적화가 덜 되어 있다.

- EconoJIT: 제한된 자원을 가진 시스템을 위해 특별히 설계되었다. JIT(Just In Time) 컴파일러와의 차이는 Code Pitching이라는 기술과 결합되었다는 것이다.

• Garbage Collection과 메모리 관리

CLR은 관리형 힙(Managed Heap)을 사용하여 참조형을 메모리에 위치 시키고 관리한다. 관리형 힙은 메모리를 동적으로 분할하고 오직 관리형 Code만이 관리형 힙에 접근할 수 있도록 통제한다. 관리형 Code는 참조를 사용해서 동작하도록 설계되어 있으며, 이러한 작동 방식은 참조가 형식 안정성을 보장 할 수 있게 한다. Object 참조 그래프에 의해 참조가 깨어진 Object는 GC(Garbage Collection)에 의해 메모리에서 자동적으로 해제 된다. .NET은 해제된 객체 참조에 대한 소멸자를 제공하는데 관리형 Code에서는 Dispose();, Close(); 메소드를 사용하고, Unmanaged Code에서는 주로 Finalize(); 메소드를 제공하는데 이는 COM 객체와 같은 Unmanaged 자원에 대한 Garbage Collection을 수행하기 위한 Code에 사용한다.

다시 말해서 Runtime에서는 개체가 더 이상 사용되지 않을 때 해당 개체를 해제하면서 자동으로 개체 레이아웃을 처리하고 개체에 대한 참조를 관리한다. 이 자동 메모리 관리 기능은 응용 프로그램의 가장 일반적인 오류 중 두 가지인 메모리 누수와 잘못된 메모리 참조 문제를 해결한다.

보안과 관련하여, 관리되는 구성 요소의 장점으로 인터넷, 기업 Network 및 로컬 컴퓨터 등을 포함하는 여러 요소에 따라 신뢰도를 다르게 부여할 수 있다. 이는 관리되는 구성 요소가 동일한 활성 응용 프로그램에 사용되는 경우에도 파일 액세스 작업, 레지스트리 액세스 작업 또는 기타 중요한 기능을 수행할 수 있을 수도 있고 수행할 수 없을 수도 있음을 의미한다.

이러한 방법의 예로 Microsoft .NET Framework Configuration에서 설정하는 내용에 따라 신뢰도를 다르게 부여해서 응용프로그램의 실행을 제어할 수 있다. Runtime에서는 Code 액세스 보안을 적용한다. 예를 들어, 사용자는 웹 페이지에 포함된 실행 파일이 화면에 애니메이션을 재생하거나 음악을 재생할 수는 있지만 자신의 개인 데이터, 파일 시스템 또는 Network에 액세스할 수는 없는 것으로 신뢰할 수 있다. 따라서 Runtime의 보안 기능은 소프트웨어의 합법적인 인터넷 배포를 가능하게 하는 매우 강력한 기능을 수행한다.

Runtime에서는 개발자의 생산성도 향상시킨다. 예를 들어, 프로그래머는 자신이 선택한 개발 언어로 응용 프로그램을 작성하면서도 다른 개발자가 다른 언어로 작성한 Class 라이브러리 및 구성 요소를 완벽하게 사용할 수 있다. 이는 Runtime을 대상으로 하는 모든 컴파일러 공급업체에게서 마찬가지로 적용된다. .NET Framework를 대상으로 하는 언어 컴파일러에서는 기존 응용 프로그램에 대한 마이그레이션 프로세스를 매우 쉽게 함으로써 .NET Framework의 기능을 해당 언어로 작성된 기존 Code에서 사용할 수 있도록 한다.

Runtime은 미래의 소프트웨어를 위해 디자인되었지만 현재 또는 과거의 소프트웨어도 지원한다. 관리되는 Code와 관리되지 않는 Code 간의 상호 운용성을 통해 개발자는 필요한 COM Component및 DLL을 계속 사용할 수 있다.

Runtime은 성능 향상을 목적으로 디자인되었다. CLR에서는 다양한 표준 Runtime 서비스를

제공하지만 관리되는 Code는 절대 해석되지 않는다. JIT(Just In Time) 컴파일이라는 기능을 사용하면 모든 관리되는 Code는 해당 Code가 실행되는 시스템의 고유 기계어 Code로 실행될 수 있다. 동시에, 메모리 관리자는 조각화된 메모리가 발생할 가능성을 제거하고 메모리의 참조 집약성을 높여 성능을 더 향상시킨다.

마지막으로, Runtime은 Microsoft® SQL Server 및 IIS(인터넷 정보 서비스) 같은 고성능의 서버측 응용 프로그램을 통해 Hosting될 수 있다. 이 인프라를 사용하면 관리되는 Code로 비즈니스 논리를 작성하면서, Runtime Hosting을 지원하는 업계 최고 엔터프라이즈 서버의 뛰어난 성능을 얻을 수 있다.

2. .NET Framework

.NET Framework은 고도로 분산된 Internet 환경에서 Application 개발을 단순하게 하는 새로운 개념의 분산 컴퓨팅 Platform이다. .NET Framework은 이전에 있었던 프로그래밍 문제점에 대한 해결책을 미리 만들어 놓은 Microsoft사의 운영체제 제품이다. 이는 아래와 같은 목적을 위해 디자인되었다.

- 소프트웨어적인 면에서 프로그램의 Code가 로컬로 저장 되어서 실행되든, 또는 로컬로 실행되지만 Internet을 통해 분산되든, Remote로 실행되든 상관 없이 일관된 개체 지향 프로그래밍 환경을 제공한다.
- 소프트웨어의 배포 및 버전 관리의 충돌을 최소화하는 방법으로 Code의 실행 환경을 제공한다.
- 내부를 알 수 없거나 일부 신뢰할 수 있는 타사에서 만든 Code를 포함하여 .NET Framework를 기반으로 작성된 Code가 작성된 Code가 영향 받지 않고 안전하게 실행될 수 있는 Code 실행 환경을 제공한다.
- 스크립트의 Parsing, 또는 interpret 환경의 성능 문제를 제거하는 방법으로 Code 실행 환경을 제공한다.
- Windows 기반 응용 프로그램 및 Web 기반 응용 프로그램 같은 다양한 형식의 응용 프로그램에서 개발자가 일관된 작업환경을 가질 수 있도록 한다.
- .NET Framework를 기반으로 작성된 Code가 .NET Framework를 기반으로 하지 않는 다른 모든 Code와 통합될 수 있도록 모든 Data의 통신을 산업 표준을 토대로 해서 Build한다.

.NET Framework에는 운영체제 서비스에서부터 시스템 수준 Class(BCL: Base Class Library), 추상 Class들과 같은 것들이 계층구조를 이루고 있다.

.NET Framework의 구성요소들을 살펴보면

- CLR(Common Language Runtime)
- BCL(Base Class Library)

  Framework에서 다른 Class로 상속되고 확장될 수 있으며, 기본적인 기능을 가지는 Class들의 모음이다. 예를 들면 System.Object 혹은 System.IO와 같은 Class들을 들 수가 있다.
- 확장 Class Library

  특정 부분 개발에 초점을 맞춘 Class Library이다. BCL에서 확장되며, 특정 종류의 응용프로그램 개발을 쉽고 빠르게 하기 위하여 설계 되었다. ASP.NET, ADO.NET, XML.NET등과 관련된 Class들을 예로 들 수가 있다.
- CLS(Common Language Specification)

.NET Framework을 지원하는 언어들이 따라야 하는 규정을 정의한다.

- 다양한 프로그래밍 언어
- Visual Studio .NET

  .NET Framework에서 프로그램 작성을 위한 통합개발환경이다.
- Windows와 COM+

  기술적으로는. NET의 일부분이 아니지만 현재 .NET Framework SDK를 설치하여 운영하기 위한 필수 요구사항이다.

.NET Framework는 윈도우 API를 캡슐화한 객체 지향 Class Library로 윈도우 운영체제의 기능에 광범위하게 접근할 수 있는 기능을 제공한다. 물론 .NET Framework은 윈도우 API보다 더 많은 요소들로 구성되어 있다.

- .NET Framework은 CLR(Common Language Runtime)
- .NET Framework Class 라이브러리
- 언어와 개발도구

라는 세 개의 핵심적인 기본 구성 요소로 이루어져 있다. .NET Framework의 계층은 윈도우 API보다 훨씬 더 기능이 추상화 되어 있다.

CLR은 Application 실행 시 Code를 관리하는 Agent로서 CLR이 제공하는 기능은 다음과 같다.

- Type 안정성 검사, 메모리 관리, Garbage Collection, Exception 처리 등의 방법으로 실행 Code를 관리한다.
- 유형 안정적인 CTS를 제공하여 Code가 안정적이다.
- 윈도우 API와 COM 상호운영 서비스를 포함하여 시스템 자원에 대한 접근성을 제공한다.
- 동일한 예외처리와 언어간 디버깅과 같은 다중언어를 지원한다.

동시에 엄격한 Type 안전성 및 다른 형태의 Code 정확성을 유지하므로 Security 및 견고성을 보장한다.

사실, Code 관리의 개념은 CLR의 기본 원칙이다. .NET Framework의 다른 기본 구성 요소인 Class 라이브러리는 다시 사용할 수 있는 형식으로 Collection의 일부로서 광범한 개체 지향 Collection이다. 기존 Command Line 또는 GUI(그래픽 사용자 Interface) 응용 프로그램에서부터 ASP.NET에서 제공하는 Web Forms 및 XML Web services 같은 최신의 혁신적인 기능을 기반으로 하는 응용 프로그램에 이르기까지 다양한 응용 프로그램을 개발하는 데 사용할 수 있다.

.NET Framework는 CLR을 해당 프로세스로 로드하고, Managed Code의 실행을 시작하는 Unmanaged 구성 요소에 의해 Hosting되는 관리되는 기능과 관리되지 않는 기능을 모두 사용하는 소프트웨어 환경을 만들 수 있다. .NET Framework에서는 몇 가지 Runtime 호스트를 제공할 뿐만 아니라 타사 Runtime 호스트의 개발도 지원한다.

예를 들어, ASP.NET에서는 Runtime을 Hosting하여 관리되는 Code에 대해 확장 가능한 서버 측 환경을 제공한다. ASP.NET은 Runtime과 함께 직접 작동하여 Web Forms 응용 프로그램 및 XML Web services를 활성화다. Web Forms 응용 프로그램과 XML Web services에 대한 내용은 이 항목의 뒷부분에서 설명한다.

Internet Explorer는 MIME 형식 확장의 형식으로 Runtime을 Hosting하는 관리되지 않는 응용 프로그램의 예이다. Internet Explorer를 사용하여 Runtime을 Hosting하면 HTML 문서에 관리되는 Component또는 Windows Forms 컨트롤을 포함시킬 수 있다. 이러한 방식으로 Runtime을 Hosting하면 Microsoft® ActiveX® 컨트롤과 유사한 관리되는 이식 가능 Code가 가능해지며,

관리되는 Code에서만 제공되는 일부 신뢰된 실행 및 보안 격리된 파일 저장소 등의 크게 향상된 기능을 사용할 수 있게 된다.

다음 그림에서는 응용 프로그램 및 전체 시스템에 대한 CLR과 Class 라이브러리의 관계를 보여 줍니다. 이 그림에서는 관리되는 Code가 보다 큰 아키텍처에서 작동하는 방식도 보여 준다.

컨텍스트에 있는 .NET Framework

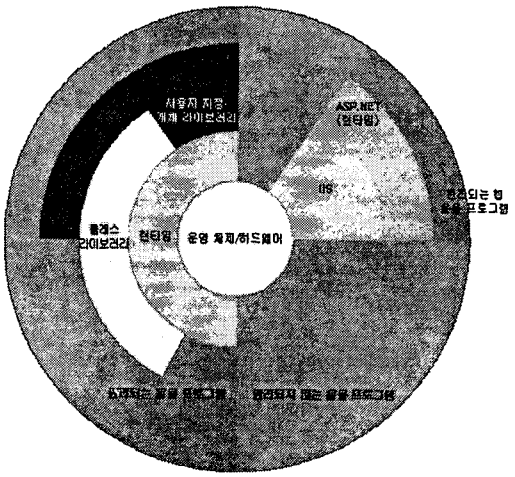


그림 3. CLR과 Class 라이브러리 관계

.NET Framework Class 라이브러리

.NET Framework Class 라이브러리는 CLR과 강력하게 통합된 재사용 가능한 형식의 컬렉션이다. Class 라이브러리는 개체 지향적이며, 사용자 고유의 관리되는 Code는 이 라이브러리에서 제공하는 형식에서 기능이 파생될 수 있다.

Class 라이브러리를 사용하면 .NET Framework 형식을 사용하기가 훨씬 쉬워질 뿐만 아니라 .NET Framework의 새로운 기능을 익히는 데 필요한 시간도 줄어든다. 또한 타사 구성 요소가 .NET Framework의 Class와 쉽게 통합될 수 있다.

예를 들어, .NET Framework 컬렉션 Class에서는 사용자 고유의 컬렉션 Class를 개발하는 데 사용할 수 있는 여러 Interface를 구현한다. 사용자가 정의한 컬렉션 Class는 .NET Framework의 Class와 쉽게 결합될 수 있다.

개체 지향 Class 라이브러리에서와 마찬가지로, .NET Framework 형식을 사용하면 문자열 관리, 데이터 수집, 데이터베이스 연결, 파일 액세스 등의 작업을 비롯한 일반적인 범위의 프로그래밍 작업을 수행할 수 있다. 이러한 일반적인 작업 외에도 Class 라이브러리에는 특수화된 다양한 개발 시나리오를 지원하는 형식이 포함되어 있다. 예를 들어, .NET Framework를 사용하면 다음과 같은 종류의 응용 프로그램 및 서비스를 개발할 수 있다.

- 콘솔 응용 프로그램
- 스크립팅 또는 Hosting된 응용 프로그램
- Windows GUI 응용 프로그램(Windows Forms)
- ASP.NET 응용 프로그램
- XML Web services
- Windows 서비스

예를 들어, Windows Forms Class는 Windows GUI 개발을 매우 단순하게 하는 재사용 가능한 형식의 포괄적인 집합이다. ASP.NET Web Form 응용 프로그램을 작성하는 경우 Web Forms Class를 사용할 수 있다.

.NET Framework Class들은

클라이언트 응용 프로그램 개발

클라이언트 응용 프로그램은 Windows 기반으로 프로그래밍 된 기존 응용 프로그램 스타일과 가장 가깝다. 이러한 종류의 응용 프로그램은 사용자가 작업을 수행할 수 있는 창 또는 폼을 데스크톱에 표시하는 응용 프로그램이다. 클라이언트

응용 프로그램에는 워드 프로세서, 스프레드시트 등의 응용 프로그램과 데이터 입력 도구, 보고서 도구 등의 사용자 지정 업무 응용 프로그램이 포함됩니다. 클라이언트 응용 프로그램에서는 대개 창, 메뉴, 단추 및 기타 GUI 요소를 사용하며 파일 시스템 등의 로컬 리소스와 프린터 등의 주변 기기에 액세스할 수 있다.

다른 종류의 클라이언트 응용 프로그램은 웹 페이지처럼 인터넷을 통해 배포되는 기존의 ActiveX 컨트롤(현재는 관리되는 Windows Forms 컨트롤로 대체)이다. 이 응용 프로그램은 다른 클라이언트 응용 프로그램과 매우 유사다. 즉, 기계어로 실행되고 로컬 리소스에 액세스할 수 있으며 그래픽 요소를 포함한다.

과거에 개발자들이 이러한 응용 프로그램을 만들 때는 C/C++과 MFC를 함께 사용하거나 C/C++과 Microsoft® Visual Basic® 등의 RAD (신속한 응용 프로그램 개발) 환경을 함께 사용했다. .NET Framework에서는 이러한 기존 제품의 기능을 하나의 일관된 개발 환경에 통합하여 클라이언트 응용 프로그램의 개발을 매우 단순하게 한다.

.NET Framework에 포함된 Windows Forms Class는 GUI 개발에 사용하도록 디자인되었다. 쉽게 변화하는 비즈니스 요구를 수용하는 데 필요한 융통성이 있는 명령 창, 단추, 메뉴, 도구 모음 및 기타 화면 요소를 쉽게 만들 수 있다.

예를 들어, .NET Framework에서는 폼과 연관된 시각적 특성을 조정하는 간단한 속성을 제공한다. 일부 경우에는 내부 운영 체제에서 이러한 특성을 직접 변경하는 것을 지원하지 않으며, 이러한 경우 .NET Framework에서는 자동으로 폼을 다시 만든다. 이는 .NET Framework에서 코딩을 보다 간단하고 일관되게 만들어 개발자 Interface를 통합하는 여러 가지 방법 중 하나다.

ActiveX 컨트롤과 달리 Windows Forms 컨트롤에는 사용자의 컴퓨터에 대한 일부 신뢰할 수 있는 액세스가 있다. 따라서 이진 또는 기계어 실행 Code에서 다른 리소스를 액세스하거나 손상시키지 않고도 사용자의 시스템에 있는 GUI 요소 및 제한된 파일 액세스 등의 일부 리소스에 액세스할 수 있다. Code 액세스 보안으로 인해, 사용자의 시스템에 한 번은 설치해야 했던 대부분의 응용 프로그램을 이제는 웹을 통해 안전하게 배포할 수 있다. 응용 프로그램은 웹 페이지처럼 배포되면서도 로컬 응용 프로그램의 기능을 구현할 수 있다.

서버 응용 프로그램 개발

관리되는 측면의 서버측 응용 프로그램은 Runtime 호스트를 통해 구현된다. 관리되지 않는 응용 프로그램에서는 CLR을 Hosting하여 관리되는 사용자 지정 Code에서 서버의 동작을 제어할 수 있도록 한다. 이 모델에서는 호스트 서버의 성능 및 확장성을 확보하면서 CLR 및 Class 라이브러리의 모든 기능을 제공한다.

다음 그림에서는 기본 Network 스키마와 서로 다른 서버 환경에서 실행되는 관리되는 Code를 보여 준다. 응용 프로그램 논리가 관리되는 Code를 통해 실행되는 동안 IIS 및 SQL Server 등의 서버에서는 일반적인 작업을 수행할 수 있다.

서버측 관리되는 Code

ASP.NET은 개발자가 .NET Framework를 사용하여 웹 기반 응용 프로그램을 만들 수 있는 Hosting 환경이다. 그러나 ASP.NET은 단순한 Runtime 호스트 이상의 기능을 한다. 즉, ASP.NET은 관리되는 Code를 사용하여 웹 사이트 및 인터넷 분산 개체를 개발하기 위한 완전한 아키텍처이다. Web Forms 및 XML Web services는 모두 IIS 및 ASP.NET을 응용 프로그램의 게시 메

커니즘으로 사용하며, .NET Framework의 Class를 지원하는 컬렉션이 있다.

웹 기반 기술에서 발전한 중요한 요소인 XML Web services는 일반적인 웹 사이트와 유사한 분산 방식의 서버측 응용 프로그램 구성 요소이다. 그러나 웹 기반 응용 프로그램과 달리 XML Web services 구성 요소는 UI가 없으며 Internet Explorer 및 Netscape Navigator 같은 브라우저의 대상이 되지 않는다. 대신, XML Web services는 기존의 클라이언트 응용 프로그램, 웹 기반 응용 프로그램 또는 다른 XML Web services 등의 다른 응용 프로그램에서 사용할 수 있도록 디자인된 다시 사용할 수 있는 소프트웨어 구성 요소로 구성된다. 따라서 XML Web services 기술은 응용 프로그램 개발 및 배포를 고도로 분산된 인터넷 환경으로 빠르게 이행시킨다.

이전 버전의 ASP 기술을 사용한 적이 있으면 ASP.NET 및 Web Forms에서 제공하는 향상된 기술을 쉽게 알 수 있을 것이다. 예를 들어, .NET Framework를 지원하는 모든 언어로 Web Forms 페이지를 개발할 수 있다. 또한, Code에서 동일한 파일을 HTTP 텍스트와 공유하는 것이 아직 가능하기는 하지만 이제 더 이상 그럴 필요가 없다. Web Forms 페이지는 다른 관리되는 응용 프로그램과 마찬가지로 Runtime을 많이 사용하므로 기계어로 실행될 수 있다. 반면, 관리되지 않는 ASP 페이지는 항상 스트립팅되고 해석된다. ASP.NET 페이지는 모든 관리되는 응용 프로그램과 마찬가지로 Runtime과 상호 작용하므로 관리되지 않는 ASP 페이지보다 더 빠르고 기능적이며 개발하기가 쉽다.

.NET Framework에서는 XML Web services 응용 프로그램의 개발 및 소비를 보조하는 Class 및 도구의 컬렉션도 제공한다. XML Web services는 SOAP(원격 프로시저 호출 프로토콜), XML(확장 가능한 데이터 형식) 및 WSDL(웹 서비스 설명 언어) 등의 표준을 기반으로 빌드된다. .NET Framework는 이러한 표준을 기반으로 빌드되어 Microsoft의 솔루션이 아닌 솔루션과의 상호 운용성을 향상시킨다.

예를 들어, .NET Framework SDK와 함께 제공되는 웹 서비스 설명 언어 도구에서는 웹에 게시된 XML Web services를 쿼리하고 해당 서비스의 WSDL 설명을 구문 분석한 다음 사용자의 응용 프로그램을 XML Web services의 클라이언트로 만드는 데 사용할 수 있는 C# 또는 Visual Basic 소스 Code를 생성한다. 소스 Code에서는 SOAP 및 XML 구문 분석을 사용하여 모든 내부 통신을 처리하는 Class 라이브러리의 Class에서 파생된 Class를 만들 수 있다. 이 Class 라이브러리를 사용하여 XML Web services를 직접 소비할 수도 있지만 SDK에 포함된 웹 서비스 설명 언어 도구 및 기타 도구는 .NET Framework를 사용한 개발을 용이하게 한다.

사용자 고유의 XML Web services를 개발하고 게시하는 경우 .NET Framework에서는 SOAP, WSDL, XML 등의 모든 내부 통신 표준에 맞는 Class 집합을 제공한다. 이러한 Class를 사용하면 분산 소프트웨어 개발에 필요한 통신 인프라에 신경 쓸 필요 없이 서비스의 논리를 중점적으로 처리할 수 있다.

마지막으로, 관리되는 환경의 Web Forms 페이지와 마찬가지로 XML Web services는 IIS의 확장 가능한 통신을 사용하여 기계어 속도로 실행된다.

3. NET Object들

Namespace

Namespace는 관련된 Class들을 논리적인 그

룹으로 묶은 컨테이너와 같은 것으로, Class의 그룹화는 응용프로그램 작성시 소스 Code에서 해당 Class를 검색하고 참조하는 작업을 매우 쉽게 해준다. 또한 Namespace를 사용해서 이름 충돌의 문제를 해결 할 수가 있다. 이전 윈도우에서는 프로그래밍 할 때에 원하는 함수를 하나 찾기 위해서 알파벳 순서로 정리되어 있는 운영체제의 함수를 모두 찾아야 했다. 초기 윈도우에서는 운영체제에서 지원하는 모든 함수가 몇 백 개 밖에 되지 않아 함수를 알파벳순으로 정리하여도 별 무리가 없었다.

32비트 윈도우에서는 약 5000여 개의 함수를 지원하게 되어 그냥 알파벳순으로 정리하는 것이 불가능하게 되었다. 그래서 거대한 리스트를 다루는 방법으로 리스트를 작은 논리 그룹으로 나누어 서브트리로 나누게 되었다.

.NET Framework에서는 운영체제의 함수와 객체를 정리할 수 있는 방법을 제공하는데, 이러한 방법은 개발자가 작성한 함수와 객체의 이름이 다른 개발자가 작성한 것과 중복되는 것을 막는데 똑같이 적용되어 진다. .NET Framework에서는 Namespace라는 개념을 사용하는데 이는 소프트웨어의 함수를 기준으로 나눈 하나의 고유한 영역으로 .NET Framework에서 처음 나오는 새로운 개념이 아니라, 객체 지향 프로그램에서 오랫동안 사용하고 있던 개념이다. 그러나, 함수나 객체의 이름을 고유하게 하는 것뿐 아니라, .NET Framework에서 사용하는 것은 시스템의 전체적인 함수를 Namespace라는 개념으로 정리하고 있다.

.NET Framework의 모든 CLR 객체는 System이라는 Namespace의 일부이다. 예를 들면 Console 윈도우에 대해 입출력을 처리하는 함수를 포함하고 있는 Class인 Console Class는 System.Console이라는 이름으로 Namespace가 더 작게 나뉘어 진다. Console 화면에 출력을 담당하는 Write함수의 완전한 이름은 System.Console.Write가 된다.

System Namespace는 여러 개의 분리된 DLL로 구현되어 있으며, 개발에 필요한 모든 DLL을 모두 포함 시켜야 한다. 개발자가 C#으로 개발한다면 using 키워드를 사용해서 소스코드에서 선언해 주어야 한다. 그렇다고 해서 해당 DLL을 컴파일러나 링커가 자동으로 참조되는 것은 아니다.

Namespace를 포함 시키는 것은 그 Namespace에 포함되어 있는 함수를 호출할 때에 짧은 이름을 사용할 수 있게 해준다. 만약 Namespace의 이름이 매우 길고 자주 사용한다면 긴 Namespace를 짧은 이름의 Namespace로 대체할 수 있다.

```
using System;
using Test = System.Web.UI.Control;
```

이라고 별칭을 써서 선언한다면 향후 Test만으로 해당 Namespace를 대신할 수 있다.

Assembly

.NET Framework는 .NET의 모든 코드, 리소스, 메타데이터에 대해서 Assembly를 사용한다. 즉 .NET Runtime이 수행하는 모든 코드는 Assembly안에 들어 있다. 또한 .NET의 모든 보안에 관련된 사항과 Namespace 알아내기, 버전 관리 기능 등 모든 사항은 Assembly 단위로 이루어 진다.

Assembly는 Application의 코드와 리소스를 갖고 있는 EXE 파일이나 DLL파일의 논리적인 집합이다. Assembly는 또한 Manifest를 갖고 있는데 이는 Assembly 내부에 들어 있는 코드와 리소스를 설명하는 메타데이터이다. Visual Studio .NET과 같은 툴을 사용할 경우 각 프로젝트는 단일 Assembly에 해당한다. 그러나, As-

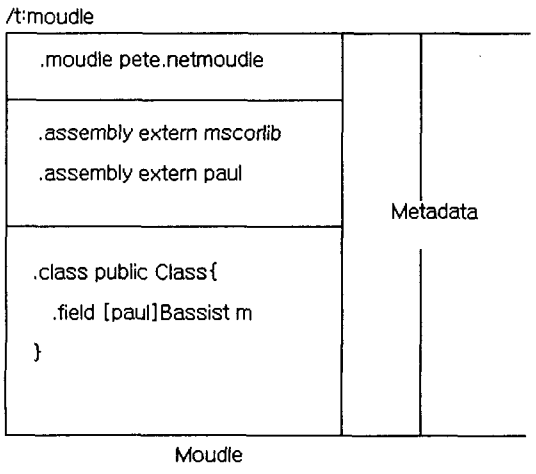
sembly가 종종 단일 파일에 들어 있지만 동일한 Directory에 들어있는 여러 개의 파일을 모아서 하나의 Assembly를 만들 수도 있다.

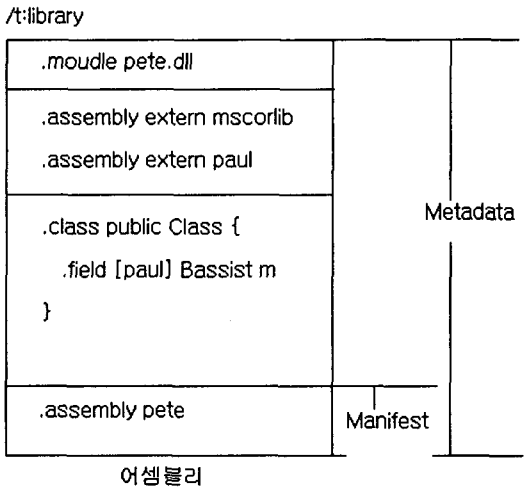
우리가 C# 컴파일러를 사용하여 Application을 컴파일할 때, 실제로 Assembly를 생성한다. 하지만 하나의 Assembly에 여러 개의 모듈을 넣거나, 버전 관리와 같은 Assembly의 장점을 취하려고 하지 않는다면 Assembly를 생성한다는 것을 눈치채지 못할 것이다.

.NET은 EXE나 DLL(컴파일 시 옵션: /t:library 스위치 사용)을 빌드 할 때마다 Manifest를 가진 Assembly를 생성하게 된다. 또한 /t:module 옵션을 사용하여 Manifest가 없는 DLL 모듈을 생성할 수도 있다. 이것은 비록 논리적으로 DLL이라 하더라도 Assembly에 속하지 않는다.

이러한 모듈은 Application을 컴파일할 때 /t:addmodule 옵션을 사용하거나, Assembly생성 툴(al.exe)을 사용해서 Assembly에 추가시켜야 한다.

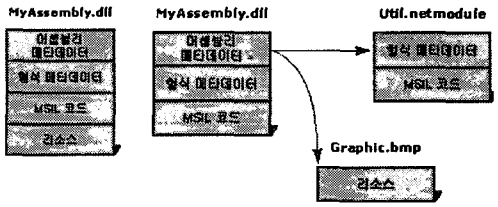
Assembly의 Manifest는 다른 방법으로 저장될 수 있다. 단일 파일의 Assembly는 컴파일할 때 그 결과물인 PE 파일 안에 병합되어 있겠지만, 여러 파일로 구성된 Assembly에서는 하나의 부





속품으로써 Manifest를 생성하여 Assembly를 생성할 수 있다.

이러한 내용을 그림으로 나타내면 다음과 같다. 그림에서 볼 수 있듯이 하나의 파일로 구성된 Assembly든지 다중 파일로 구성된 Assembly든지 반드시 하나의 Assembly에 하나의 Manifest를 가지게 된다.



단일 파일 Assembly 다중 파일 Assembly

Manifest안에 저장되어 있는 정보의 내용을 보면 다음과 같이 분류할 수 있다.

- Assembly의 이름: 문자로 이루어진 Assembly이름
- 버전정보: 네 개의 파트로 나누어진 문자열
- 공유이름과 서명된 Assembly Hash
- 파일: Assembly에 존재하는 모든 파일을 포함하는 리스트

• 참조되는 다른 Assembly들: Manifest의 Assembly로부터 직접 참조되는 모든 외부 Assembly

• 형식: 모듈에 매핑된 Assembly안의 모든 형의 리스트

• 보안허가: Assembly에 의해 명시적으로 거절되는 모든 보안허가 리스트

• 사용자 정의 Attribute:

• 제품정보: 회사, 상표, 제품, 저작권 같은 정보

Assembly는 .NET Framework로 프로그래밍 할 때의 기본적인 요소로서 다음과 같은 기능을 수행한다.

• Common Language Runtime에서 실행하는 코드를 포함하고, PE(이식 가능한 실행) 파일에 들어 있는 MSIL(Microsoft Intermediate Language) 코드는 연관된 Assembly Manifest가 없으면 실행되지 않는다. 각 Assembly는 **DllMain**, **WinMain** 또는 **Main** 중 하나의 진입점만 가질 수 있다.

• 보안 경계를 구성한다. Assembly 단위에서 권한을 요청하고 허가한다. 형식 경계를 구성한다. 모든 형식의 ID에는 해당 형식이 저장된 Assembly의 이름이 있다. 한 Assembly의 범위 내에서 로드 된 MyType 형식은 다른 Assembly의 범위 내에서 로드 된 MyType 형식과 다르다.

• 참조 범위 경계를 구성한다. Assembly의 Manifest에는 형식을 확인하고 리소스 요청을 완료하는 데 사용되는 Assembly 메타데이터가 들어 있다. Manifest는 Assembly 외부에서 노출된 리소스와 형식을 지정하며, 이 Manifest가 종속된 다른 Assembly를 열거하기도 한다.

• 버전 경계를 구성한다. Common Language Runtime에서 Assembly는 버전 관리가 가능한 가장 작은 단위이며 동일한 Assembly에 들어 있는 모든 형식 및 리소스는 동일한 버전을 갖는다. Assembly의 Manifest는 다른 종속 Assembly에 대해 지정하는 버전 종속성을 설명한다.

• 배포 단위를 구성한다. 응용 프로그램이 시작될 때에는, 응용 프로그램에서 초기에 호출하는 Assembly만 있어야 한다. 지역화 리소스, 유틸리티 클래스가 들어 있는 Assembly 등의 기타 Assembly는 필요할 때 검색할 수 있다. 이렇게 하면 응용 프로그램을 간단하고 크기가 작은 상태로 다운로드 할 수 있다.

• Side-by-side(여러 버전을 동시에 실행) 실행이 지원되는 단위다.

Assembly는 정적이거나 동적일 수 있다. 정적 Assembly에는 Assembly의 리소스(비트맵, JPEG 파일, 리소스 파일 등)나 .NET Framework 형식(인터페이스 및 클래스)이 포함될 수 있으며 이러한 Assembly는 디스크에 있는 PE 파일에 저장된다.

.NET Framework를 사용하여 동적 Assembly를 만들 수도 있는데, 동적 Assembly는 메모리에서 직접 실행되며 실행되기 전에 디스크에 저장되지 않는다. 동적 Assembly는 실행된 후에 디스크에 저장할 수 있다.

Assembly를 만드는 방법에는 여러 가지가 있다. Visual Studio .NET과 같이 이전에 .dll 또는 .exe 파일을 만들 때 사용하던 개발 도구를 사용하거나 .NET Framework SDK에 제공된 도구를 사용하여 모듈이 있는 Assembly를 다른 개발 환경에서 만들 수도 있다. 또는 Reflection.Emit와 같은 Common Language Runtime API를 사용하여 동적 Assembly를 만들 수도 있다.

Assembly는 응용 프로그램의 배포를 단순화하고 Component기반 응용 프로그램에서 발생할 수 있는 버전 관리 문제를 해결하기 위해 디자인

되었다.

최종 사용자나 개발자에게는 현재의 Component기반 시스템에서 발생하는 버전 관리 문제나 배포 문제는 흔한 일이다. 경우에 따라서는 컴퓨터에 새 응용 프로그램을 설치했을 때 기존 응용 프로그램이 갑자기 작동하지 않기도 한다. 또한 개발자들은 COM 클래스를 활성화하는 데 필요한 모든 레지스트리 항목을 일관되게 유지하는 작업에 수많은 시간을 보내기도 한다.

.NET Framework에서는 Assembly를 사용함으로써 배포와 버전관리 문제가 해결되었다. Assembly는 자체 설명(Self Description)되고 레지스트리 항목에 종속되지 않는 구성 요소이기 때문에 다른 응용 프로그램에 전혀 영향을 주지 않고 설치하는 것을 가능하게 한다. 이러한 특징으로 응용 프로그램 제거 및 복제 작업도 단순화할 수 있다.

버전 관리 문제

현재 Win32 응용 프로그램에는 두 가지 버전 관리 문제가 있다.

◦ 버전 관리 규칙은 응용 프로그램을 이루는 Component간에 통신하거나 운영 체제에서 일괄적으로 적용될 수 없다. 현재 방식은 이전 버전과의 호환성에 기반하는데, 이러한 호환성을 유지하는 것이 쉽지 않다. 일단 게시된 인터페이스 정의는 정적이어야 하고 특정 코드는 이전 버전과 호환 가능해야 한다. 또한, 일반적으로 코드는 특정 코드의 단일 버전만이 컴퓨터에 존재하고 실행되도록 디자인되었다.

◦ 함께 빌드된 Component집합과 Runtime에 존재하는 집합 사이의 일관성을 유지할 수 없다.

이 두 가지 버전 관리 문제로 인해 DLL 충돌이 발생하는데, 하나의 응용 프로그램을 설치할 때 이전 버전과 호환되지 않는 특정 소프트웨어의 Component또는 DLL이 함께 설치되면 기존 응용 프로그램을 사용할 수 없게 된다. 이러한 문제가 발생하면 시스템 자체에서는 문제를 진단하거나 고치지 못한다.

DLL 충돌 문제 해결

이러한 문제에 대한 연구를 통해 Microsoft® Windows® 2000에는 DLL 충돌을 부분적으로 수정하는 기능이 제공된다.

◦ Windows 2000에서는 클라이언트 응용 프로그램을 만들 때 종속되는 .dll 파일이 응용 프로그램의 .exe 파일이 들어 있는 디렉터리에 위치하도록 할 수 있다. 또한 특정 구성 요소를 검색할 때, 정규화된 경로나 일반 경로에서 검색하기 전에 .exe 파일이 들어 있는 디렉터리에서 구성 요소를 먼저 검색하도록 구성할 수 있다. 이렇게 하면 다른 응용 프로그램에서 설치하고 사용하는 구성 요소와는 독립적으로 새 구성 요소를 사용할 수 있다.

◦ Windows 2000에서는 운영 체제와 함께 제공된 파일을 System32 디렉터리에 따로 잠금으로써 다른 응용 프로그램을 설치할 때 실수로 이들 파일이 바뀌는 것을 방지한다.

Common Language Runtime에서는 Assembly를 사용하여 DLL 충돌을 완전하게 해결할 수 있는 방법을 찾고 있다.

3.5 Assembly 해결책

버전 관리 문제를 비롯하여 DLL 충돌을 일으키는 문제를 해결하기 위해 Runtime에서는 Assembly를 사용하여 다음을 수행한다.

◦ 서로 다른 소프트웨어 구성 요소에 대해 개발자가 버전 규칙을 지정할 수 있도록 한다.

◦ 버전 관리 규칙을 적용할 수 있는 인프라를

제공한다.

• 특정 소프트웨어 구성 요소의 여러 버전을 동시에 실행(side-by-side 실행)할 수 있도록 하는 인프라를 제공한다.

ILDASM.exe를 가지고 Assembly의 목록을 볼 수 있다. Assembly의 배포에 관한 문제를 살펴보면 Assembly를 전용으로 둘 것인지 공개용(전역)으로 할 것인지를 신중히 고려해야 한다. Assembly를 전용으로 두고자 하면 해당 DLL을 사용하는 Client가 들어 있는 디렉토리의 하위 bin 폴더 안에 Assembly를 복사하면 된다. 하지만 Assembly를 공유하려고 할 경우 .NET Framework의 Global Assembly Cache(GAC)에 Assembly를 넣어 놓고 공유할 수 있도록 한다. GAC는 \\WINNT\assembly\ 디렉토리에 있다. GACUtil.exe를 이용하면 GAC에 넣고 Assembly의 속성을 볼 수도 있다. 공유 Assembly는 고유한 이름을 확보하기 위하여 공개키 암호화를 사용한다.

Memory관리

메모리 관리 작업에 대한 경험은 개발 환경에 따라 다를 수 있으므로 상황에 따라서는 CLR에서 제공하는 자동 메모리 관리 기능에 맞춰 프로그래밍 방식을 변경해야 할 수도 있다. 프로그램에 버그가 발생하는 가장 큰 이유는 메모리 관리 작업을 수작업으로 하기 때문이다. C++과 같은 예전 프로그램방식은 프로그래머가 생성한 객체는 프로그래머가 직접 지우게 한다. 그러나 이렇게 하면 두 가지 중요한 문제가 발생하게 된다.

첫째는 프로그래머가 객체를 생성한 이후에 지우는 것을 잊어버리는 경우가 발생하고, 이렇게 하면 프로세스의 전체 메모리 공간이 점점 없어지게 되어 결국 충돌이 발생하게 된다.

둘째로 프로그래머가 객체를 직접 지우고 나서 실수로 그 메모리에 접근하는 것이다. 삭제된 객체의 메모리를 구성하고 있는 트랜지스터는 잘못된 값을 가지고 계속 돌아갈 것이다.

이렇게 되면 프로그래머는 Business Logic에 신경을 쓰는 것이 아니라, 리소스 관리에 신경을 쓰게 된다. Microsoft사는 .NET CLR의 일부로 자동 메모리 관리 메커니즘을 만들었다.

NET Framework Garbage Collection(GC)는 응용 프로그램의 메모리 할당 및 해제를 관리한다. 사용자가 **new** 연산자를 사용하여 개체를 만들 때마다 Runtime에서는 Managed Heap에서 해당 개체에 메모리를 할당하며 Managed Heap에 주소 공간이 남아 있으면 새 개체에 필요한 공간을 계속하여 할당한다. 그러나 메모리는 한정되어 있기 때문에 시간이 지나면 Garbage Collection이 수집을 수행하여 일부 메모리를 해제해야 한다. Garbage Collection의 최적화 엔진은 할당되는 메모리에 기반하여, 가비지를 수집할 가장 좋은 시기를 파악한다. 가비지를 수집할 때 Garbage Collection는 응용 프로그램에서 더 이상 사용하지 않는 Managed Heap에서 개체를 확인하고 메모리를 회수하는 데 필요한 작업을 수행한다. Garbage Collection은 자신이 수행하고 싶을 때 아무 때나 수행하지만 프로그래머는 직접 Garbage Collection이 발생하도록 할 수 있다. 프로그래머는 System.GC.Collect 함수를 호출하여 강제적으로 Garbage Collection을 수행할 수 있다. 예를 들어, 파일을 저장한 후에 메모리 찌꺼기를 지우고 싶을 때나, 큰 연산을 수행하기 전에 메모리를 비우고 싶다면 Garbage Collection을 프로그램에 삽입할 수 있다. 그러나 대부분의 경우 Garbage Collection이 원하는 때에 작업을 수행하도록 내버려 둔다.

Garbage Collection은 다음 단계로 구성된다.

1. Garbage Collection이 관리되는 코드에서 참조된 관리되는 개체를 찾는다.

2. Garbage Collection이 참조되지 않은 개체를 종료한다.

3. Garbage Collection이 참조되지 않은 개체를 해제하고 해당 메모리를 회수한다.

이러한 자동 메모리 관리 기능은 한가지 문제를 가지고 있다. 객체가 소멸될 때 일어나야 하는 정리 작업은 어떻게 하는가 이다. C++ 내부에서는 객체의 소멸자 내부에서 마지막 정리 작업을 한다.

CLR은 Garbage Collection에서 Finalizer의 개념을 지원한다. Finalizer라는 것은 객체가 Garbage Collection 될 때 호출되는 메소드이다. 프로그래머는 이 메소드를 오버라이드 하여 자신만의 Finalize라는 메소드를 작성한다. 객체가 Garbage Collection될 때 Garbage Collection 쓰레드는 해당 객체가 Finalize메소드를 가지고 있다는 것을 알아채고, Finalize 메소드를 호출한다. 그렇게 해서 마지막 정리 작업을 수행한다. Finalize 메서드는 Dispose 메서드가 호출되지 않은 경우에 리소스를 정리하며, 이 메서드는 관리되지 않는 리소스를 정리할 때만 사용해야 한다. 관리되는 리소스는 Garbage Collection이 자동으로 정리하기 때문에, 관리되는 개체에 대해서는 Finalize 메서드를 구현할 필요가 없다.

Client가 리소스를 즉시 해제하기를 바라는 경우 해당 객체는 Dispose라는 메소드를 가지고 있다. 이 때에는 객체에 대해 Dispose를 수행한 후에 해당 객체에 접근하려는 Client에 대해서도 처리를 해 주어야 한다. Type의 Dispose 메서드는 형식에 포함된 모든 리소스는 물론 부모 형식의 Dispose 메서드도 호출하여 해당 기본 형식의 모든 리소스도 해제해야 한다. 부모 형식의 Dispose 메서드는 해당 형식에 포함된 모든 리소스는 물론 그 부모 형식의 Dispose 메서드를 호출하며, 이 패턴은 기본 형식의 계층 구조 전체에 전파된다. 리소스가 항상 제대로 정리되기 위해서는 Dispose 메서드를 여러 번 호출한 경우에도 예외가 throw되지 않아야 한다.

Dispose 메서드는 삭제하는 개체에 대해 GC.SuppressFinalize 메서드를 호출해야 한다. 개체가 현재 종료 대기열에 있는 경우 GC.SuppressFinalize 메서드는 Finalize 메서드가 호출되는 것을 방지한다. 위에서 말했듯이, Finalize 메서드는 리소스를 많이 소모하기 때문에 Dispose 메서드를 통해 개체가 이미 정리된 경우에는 Garbage Collection에서 개체의 Finalize 메서드를 호출할 필요가 없다.

COM과의 상호운용

새로운 소프트웨어 플랫폼의 성공여부는 보다 나은 Application을 개발할 수 있는 방법론을 제시 하는 동시에 현존하는 플랫폼과 얼마나 잘 통합 되었는가에 있다. 윈도우는 1993년부터 Application간의 통신을 위해 COM에 의존해 왔다. 윈도우 환경에 돌아가는 코드는 COM과 밀접하게 연결되어 있다. .NET Framework은 COM을 지원해야 한다. .NET Client는 COM서버를 사용하고, COM Client는 .NET 서버를 사용한다. 새로운 .NET Code는 COM Code와 상호운용 되어야 한다.

개체는 COM(Component Object Model)을 통해 다른 구성 요소 및 호스트 응용 프로그램에서 해당 기능을 사용할 수 있다. COM 개체는 오랜 기간 Windows 프로그래밍의 기초였고 .NET 플랫폼에 사용된 CLR(공용 언어 런타임)에 맞게 응용 프로그램을 개발하면 많은 이점을 얻을 수 있다.

결국에는 .NET 플랫폼 응용 프로그램이 COM

으로 개발한 응용 프로그램을 대신할 것이다. 그 때까지는 COM 개체를 사용하거나 만드는 데 Visual Studio .NET을 이용해야 할 경우도 있다. COM 또는 *COM interop*와 상호 운용성이 있기 때문에 상황에 맞게 .NET 플랫폼으로 전환하면서 기존의 COM 개체를 사용할 수 있다.

관리되는 코드 및 데이터

.NET 플랫폼용으로 개발된 코드는 *Managed Code*라고 하며 CLR(공용 언어 런타임)에 사용되는 메타 데이터를 포함한다..NET 응용 프로그램에 사용되는 데이터는 *관리되는 데이터*라고 한다. 이는 .NET 런타임에서 메모리 할당이나 회수, 형식 검사와 같은 데이터 관련 작업을 관리하기 때문이다. 기본적으로 Visual Basic .NET에서는 관리되는 코드와 데이터를 사용하지만 interop 어셈블리를 사용하여 COM 개체 데이터 및 관리되지 않는 코드에 액세스할 수 있다.

어셈블리

어셈블리는 .NET Framework 응용 프로그램의 기본 빌드 블록으로, 하나 이상의 파일을 포함하는 단일 구현 단위로 빌드, 버전 지정 및 배포되는 기능의 모음이다. 각 어셈블리에 어셈블리 Manifest가 하나씩 있다.

형식 라이브러리(Type Library) 및 어셈블리 Manifest

형식 라이브러리(Type Library)는 멤버 이름이나 데이터 형식과 같은 COM 개체의 특성에 대해 기술한다. 어셈블리 Manifest는 .NET 응용 프로그램에 대해 동일한 기능을 수행한다. 어셈블리 Manifest에는 다음과 같은 정보가 들어 있다.

- 어셈블리 ID, 버전, Culture 및 디지털 서명
- 어셈블리 구현을 구성하는 파일
- 어셈블리에서 내보내는 형식 및 리소스를 포함하여 어셈블리를 구성하는 형식 및 리소스
- 다른 어셈블리에 대한 컴파일 시점에 대한 종속성
- 어셈블리의 올바른 실행을 위해 필요한 권한

형식 라이브러리(Type Library) 가져오기 및 내보내기

Visual Studio .NET에 포함된 Tlbimp 유틸리티를 사용하여 형식 라이브러리의 정보를 .NET 응용 프로그램으로 가져올 수 있다. Tlbexp 유틸리티를 사용하면 어셈블리에서 형식 라이브러리를 생성할 수 있다.

Interop 어셈블리

Interop 어셈블리는 관리되는 코드와 관리되지 않는 코드 사이에서 다리 역할을 하는 .NET 어셈블리로, COM 개체 멤버를 해당하는 .NET 관리 멤버에 매핑한다. Visual Basic .NET을 사용하여 만든 Interop 어셈블리는 상호 운용성 마샬링과 같이 COM 개체에 대한 많은 작업을 처리한다.

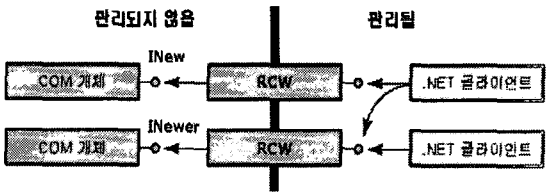
상호 운용성 마샬링

모든 .NET 응용 프로그램은 사용된 프로그래밍 언어에 관계 없이 개체 간의 상호 운용성을 허용하는 공통 형식 집합을 공유한다. COM 개체의 매개 변수 및 반환 값에는 관리되는 코드에서 사용되는 것과 다른 데이터 형식이 사용되는 경우가 종종 있다. *상호 운용성 마샬링*은 COM 개체와의 이동에 있어 해당하는 데이터 형식으로 매개 변수 및 반환 값을 패키지로 만드는 프로세스이다.

.NET Client는 RCW(Runtime Callable Wrapper)를 통해 COM 서버에 접속할 수 있다. RCW는 COM객체가 .NET 객체에게는 마치 로컬의 .NET 객체인 것처럼 보이게 하고, .NET Client는 COM 객체에게 마치 COM Client인 것처럼 보이

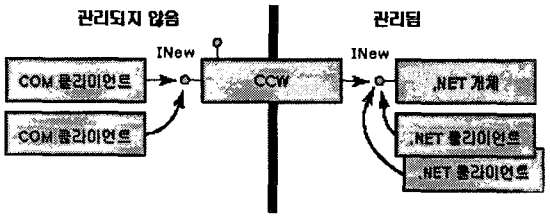
게 포장하여 이 COM 객체와 .NET의 CLR /환경 사이의 중재 역할을 해준다.

.NET Client 개발자는 두 가지 방법으로 RCW를 발생 시킨다. 하나는 VS.NET을 사용해서 Reference해 오는 것이고, 또 다른 하나는 TlbImp.exe라는 유틸리티를 사용하는 것이다.



RCW를 통한 .NET Client/COM 객체의 상호작용

COM으로부터 .NET 객체를 사용하는 것은 .NET Framework에서는 CCW(COM Callable Wrapper)를 통해 지원된다. CCW는 .NET 객체를 포장하고 .NET 객체가 COM Client에게 마치 COM 객체인 것처럼 만들어 중재한다.



CCW를 통한 .NET 객체/COM Client의 상호작용

CCW를 가지고 작업하려면 .NET 컴포넌트는 Strong Name으로 서명되어야 한다. 그렇지 않으면 CLR은 컴포넌트를 구별하지 못할 것이다. 또한 .NET 컴포넌트는 반드시 GAC(Global Assembly Cache) 안에 있어야 하고, COM Client가 사용할 수 있도록 만드는 .NET Class는 파라미터가 없는 생성자만을 제공해야 한다. 또한 COM Client가 .NET 객체를 찾을 수 있게 하려면 COM 객체에 필요한 레지스트리 Entry를 만들어야 한다. 이 Entry는 .NET SDK에 포함되어 있는 RegAsm.exe를 사용하여 만들 수 있다.

.NET에서의 Transaction

분산 시스템에서 데이터의 무결성을 보호하려면 Transaction이 반드시 필요하다. COM+, COM, MTS는 Transaction을 처리하는 객체를 쉽게 작성할 수 있도록 자동으로 지원한다. 프로그래머는 자신의 객체에 Transaction이 필요함 이라고 표시하면 COM+ 객체가 활성화 될 때 자동으로 Transaction을 만들어 주었다.

객체는 데이터베이스를 변경하기 위해 COM+ 방식의 Transaction을 지원해주는 Microsoft SQL Server와 같은 프로그램인 COM+ 리소스 매니저(RM:Resource Manager)를 사용하였다. 그러고 나면 객체는 결과에 만족하는지 여부를 COM+에게 알려준다. 참여한 모든 객체가 만족하면 COM+는 Transaction을 승인하고, 리소스 매니저에 변경된 값을 저장하도록 한다. 만약 객체 중 하나라도 연산에 실패하면 COM+는 Transaction을 중지하고 시스템의 상태를 Transaction 시작하기 이전의 상태로 바꾼다.

순수한 .NET 객체는 Transaction에 참여할 수 있다.

현재 Microsoft의 Transaction 처리는 COM에 기반을 두고 있다. .NET 객체는 COM과의 상호 운용성을 이용하여 Transaction에 참여한다. Transaction에 참여하고 있는 .NET 객체는 Transaction의 결과에 만족하는지 알려 주어야 한다. .NET 객체는 System.EnterpriseService.ContextUtil이라는 시스템 객체에서 자신의 Context를 찾아 낼 것이다. 이 시스템 객체는 COM+가 Transaction 만족도를 설정하는 방법과 똑같이 .NET 객체의 Transaction 만족도를 설정한다.

트랜잭션 이라는 개념은 일련의 연속된 작업에 따라 데이터베이스의 결과가 달라지는 문제를 해결하기 위해 개발되었다. 이러한 문제가 발생하는 원인은 연속된 작업으로 인해 이전 작업 결과가 바뀔 수 있기 때문이다. 이러한 경우에는 어느 한 작업이 실패하면 작업 결과의 상태를 예측할 수 없게 된다.

이러한 문제를 해결하기 위해 트랜잭션은 완벽한 최종 결과를 얻을 수 있도록 일련의 작업을 그룹화한다. 이 경우, 모든 작업이 성공하여 커밋, 즉 데이터베이스에 쓰여지거나 전체 트랜잭션이 실패하거나 둘 중에 하나이다. 트랜잭션을 취소하는 것을 Rollback이라고 한다. Rollback을 하면 변경 사항이 복구되고, 데이터베이스가 트랜잭션 이전 상태로 돌아간다.

예를 들어, 자동화된 뱅킹 트랜잭션으로 A 계좌에서 B 계좌로 송금하는 경우 자금을 정확히 처리하려면 A 계좌에서 출금하고 B 계좌로 입금하는 작업이 모두 성공해야 하며 그렇지 않은 경우 전체 트랜잭션이 실패해야 한다.

트랜잭션에는 다음과 같은 내용을 나타내는 ACID 속성이 있어야 한다.

• **원자성(Atomicity)** : 트랜잭션은 가장 작은 작업 단위이며 한 번만 실행된다. 즉, 작업이 모두 수행되거나 모두 수행되지 않는다. All or Nothing

• **일관성(Consistency)** : 트랜잭션은 하나의 일관된 데이터 상태를 또 다른 일관된 데이터 상태로 변환하여 데이터의 일관성을 유지한다. 트랜잭션으로 바운딩된 데이터는 의미 구조가 변경되어서는 안 된다.

• **격리(Isolation)** : 트랜잭션은 격리 단위로서, 각 트랜잭션은 동시 트랜잭션과는 별도로 발생한다. 따라서 하나의 트랜잭션에서 다른 트랜잭션의 중간 단계를 절대로 볼 수 없어야 한다. 한 트랜잭션 안의 전체 총액은 같아야 한다.

• **영속성(Durability)** : 트랜잭션은 또한 복구 단위이기도 하다. 트랜잭션이 성공하면 시스템이 충돌하거나 종료되어도 업데이트 내용은 그대로 유지된다. 트랜잭션이 실패하면 시스템은 트랜잭션을 커밋하기 전 상태로 남아 있게 된다.

OLE DB또는 ODBC의 트랜잭션을 지원할 수 있다.

트랜잭션은 논리적인 하나의 단위로 성공 또는 실패하는 일련의 관련 작업들의 집합이다. 트랜잭션 처리 개념으로 볼 때 트랜잭션은 두 가지로 처리 되는데 Commit 되거나 중단되어 Rollback 될 수 있다. 트랜잭션이 커밋되는 경우, 모든 트랜잭션 참가 요소는 데이터의 변경 내용이 모두 변하지 않도록 해야 한다. 변경 내용은 시스템 정지 또는 기타 예측하지 못한 일들이 발생하는 경우에도 유지되어야 한다.

참가 요소 하나라도 이에 실패하면 트랜잭션 자체가 이루어지지 않는다. 이 경우 트랜잭션 범위 내의 데이터 변경 내용은 모두 특정 설정 지점으로 롤백된다.

트랜잭션에서는 여러 개의 작업이 함께 바인딩 된다. 예를 들어, ASP.NET 페이지에서 두 가지 작업을 수행하는 것으로 가정하면, 첫 번째는 데이터베이스에 새 Table을 만드는 작업이고, 두 번째는 지정된 개체를 호출하여 데이터를 수집하고 서식을 설정하여 새 Table에 삽입하는 작업이다. 이 두 작업은 서로 아주 밀접히 관련되어 있어서 새 Table을 데이터로 채울 수 없는 한, 새 Table을 만들지 않아도 된다. 하나의 트랜잭션 범위 내에서 이 두 작업을 실행하면 두 작업이 연결된다. 두 번째 작업이 실패하는 경우 첫 번째 작업은 새 Table을 만들기 전의 지점으로 롤백된다.

이 예제에서처럼 트랜잭션은 데이터베이스 또

는 메시지 대기열과 같은 하나의 데이터 리소스로 한정될 수 있다. 일반적으로 이러한 데이터 리소스에서는 로컬 트랜잭션 기능이 제공된다. 데이터 리소스에 의해 제어되는 트랜잭션은 효율적으로 쉽게 관리할 수 있다.

트랜잭션은 여러 개의 데이터 리소스에 연결될 수 있다. 이와 같이 분산된 트랜잭션을 사용하면 서로 다른 시스템에서 발생하는 여러 개의 개별 작업을 하나의 성공 또는 실패 동작으로 통합할 수 있다.

Distributed Transaction이란 분산 데이터, 즉 하나 이상의 네트워크로 연결된 컴퓨터 시스템의 데이터를 업데이트하는 트랜잭션을 말한다. 분산 시스템에서 트랜잭션을 지원하려면 OLE DB 트랜잭션 지원보다는 Microsoft .NET Framework를 사용해야 한다.

트랜잭션 처리를 사용하면 트랜잭션에 있는 메시지가 순서대로 단 한 번에 배달되어 대상 대기열에서 제대로 검색되도록 할 수 있다. Message Queue 구성 요소를 사용하여 트랜잭션 방식으로 메시지를 주고 받을 수 있다. 트랜잭션에 있는 메시지를 보낼 때 일련의 관련 메시지를 그룹화한다. 트랜잭션에 포함된 메시지는 보낸 순서대로 모두 한꺼번에 배달되며(*커밋된* 트랜잭션), 문제가 발생하면 자동으로 메시지 전체가 배달되지 않는다(*중단된* 트랜잭션).

응용 프로그램에서 다음과 같은 두 종류의 주요 트랜잭션을 만들 수 있다.

◦ 두 개 이상의 메시지 대기열 리소스, 즉 메시지 대기열 엔터프라이즈에 속한 두 대기열 간에 메시지를 전송하는 데 사용하는 *내부 트랜잭션*

◦ 대기열과 데이터베이스 같은 다른 리소스 간에 메시지를 전송하는 데 사용하는 외부 트랜잭션

◦ 내부 트랜잭션과 외부 트랜잭션은 트랜잭션 프로그래밍 모델, 트랜잭션을 제어하는 데 사용하는 리소스 관리자 및 트랜잭션 가능 패턴이 서로 다르다.

내부 트랜잭션

내부 트랜잭션은 MessageQueueTransaction 클래스의 인스턴스를 만들어 MessageQueue 구성 요소의 인스턴스와 연결하는 방식으로 실행되는 가장 단순한 종류의 트랜잭션이다. 내부 트랜잭션에서는 하나 이상의 메시지 대기열 사이에 메시지가 전송되며 발생하는 동작을 메시지 대기열의 트랜잭션 코디네이터가 제어한다. 외부 트랜잭션은 자동 또는 암시적 트랜잭션인 반면, 내부 트랜잭션은 프로세스의 이러한 단계를 직접 제어하기 때문에 수동 또는 명시적 트랜잭션이다.

내부 트랜잭션의 프로그래밍 모델은 매우 간단하다. 즉, **MessageQueueTransaction** 클래스의 Begin 메서드를 호출하여 이 클래스의 인스턴스를 송신 또는 수신 메서드로 전달한다. 그런 다음, Commit을 호출하여 트랜잭션의 변경 내용을 대상 대기열에 저장한다.

내부 트랜잭션은 메시지 대기열이 아닌 데이터베이스와 같은 리소스를 사용하여 트랜잭션을 수행할 수 없다는 점에서 기능에 제한이 있다. 트랜잭션에서 데이터베이스와 상호 작용하려면 외부 트랜잭션을 사용해야 한다. 그러나 이러한 제한 있음에도 불구하고 내부 트랜잭션은 외부 트랜잭션보다 성능이 더 좋다.

내부 트랜잭션과 외부 트랜잭션의 경우 모두 트랜잭션으로 표시된 대기열로 메시지를 보내야 한다.

외부 트랜잭션

대기열과 다른 형식의 리소스 간에 메시지를 보낼 경우에는 일반적으로 외부 트랜잭션을 사용한다. 예를 들어, 대기열에서 메시지를 검색하여 데이터베이스로 보낼 수 있고 그 반대로 할 수도

있다. 외부 트랜잭션은 몇 가지 점에서 내부 트랜잭션과 다르다.

• 외부 트랜잭션은 메시지 대기열 시스템의 일부가 아닌 코디네이터를 사용한다. 대개 MS DTC(Microsoft Distributed Transaction Coordinator)를 코디네이터로 사용한다. MS DTC는 트랜잭션에서 필요한 리소스를 모으고 트랜잭션의 모든 작업이 성공인지 또는 실패인지를 결정하는 프로세스를 하나의 단위로 제어한다.

• 외부 트랜잭션은 좀더 복잡한 프로그래밍 모델을 사용한다. 단순히 **Begin, Commit** 및 **Abort**를 호출하는 대신, 외부 트랜잭션을 나타내는 특성을 지정하여 COM+ 서비스와 함께 해당 구성 요소를 등록한다. 또한, 트랜잭션 형식에 대한 매개 변수가 있는 특수한 송신 및 수신 메서드 형식을 사용하고 이 필드를 Automatic으로 설정해야 한다.

분산 TP(트랜잭션 처리) 시스템은 분산 환경에서 서로 다른 종류의, 트랜잭션 인식 리소스에 연결되는 트랜잭션을 용이하게 하기 위해 고안되었다. 분산 TP 시스템에서 지원하는 응용 프로그램에서는 MSMQ(Microsoft Message Queue) 대기열에서 메시지를 검색하거나, Microsoft SQL Server 데이터베이스에 메시지를 저장하거나, Oracle Server 데이터베이스에서 해당 메시지에 대한 기존의 참조를 모두 제거하는 것과 같은 다양한 작업을 하나의 트랜잭션 단위로 결합할 수 있다. 이러한 작업들은 여러 개의 데이터 리소스에 연결되기 때문에 모든 리소스 간의 데이터 일관성을 유지하려면 분산 트랜잭션에 ACID 속성을 적용해야 한다.

분산 TP 시스템은 서로 협력하는 몇 개의 Entity로 구성된다. 이러한 Entity들은 논리적 단위로 동일한 컴퓨터에 상주할 수도 있고 서로 다른 컴퓨터에 상주할 수도 있다.

TP 모니터

TP(트랜잭션 처리) 모니터는 트랜잭션 인식 응용 프로그램과 리소스 컬렉션 사이에 있는 소프트웨어다. TP 모니터는 운영 체제 작업을 최대화하고, 네트워크 통신을 능률적이게 하며, 여러 데이터 리소스에 액세스하는 여러 응용 프로그램에 여러 클라이언트를 연결한다.

다중 사용자로 구성된 분산 환경을 관리하는 응용 프로그램을 만드는 대신 하나의 트랜잭션 요청으로만 구성되는 응용 프로그램을 작성하도록 한다. TP 모니터를 사용하면 응용 프로그램을 필요한 만큼 확장시킬 수 있다.

MS DTC(Distributed Transaction Coordinator)는 Microsoft Windows 2000용 TP 모니터이다.

트랜잭션 관리자

분산 트랜잭션에서, 관여된 각 리소스에는 해당 컴퓨터에 대해 들어오고 나가는 트랜잭션을 추적하는 로컬 TM(트랜잭션 관리자)이 있다. TP 모니터는 하나의 TM에 로컬 TM 간의 모든 동작을 조정하는 추가 작업을 할당한다. 트랜잭션 동작을 조정하는 TM을 루트 또는 조정 TM이라고 한다.

TM은 모든 트랜잭션 처리 기능을 조정 및 관리하지만 데이터를 직접 관리할 수는 없다. 데이터 관련 동작은 리소스 관리자가 처리한다.

리소스 관리자(Resource Manager)

리소스 관리자는 데이터베이스의 지속적인 또는 영속적인 데이터, 영속적인 메시지 대기열 또는 트랜잭션 파일 시스템을 관리하는 시스템 서비스 이다. 리소스 관리자는 데이터를 저장하고 오류를 복구한다.

SQL Server와 MSMQ에서는 분산 트랜잭션에 참여하는 리소스 관리자가 제공된다. Oracle,

Sybase, Informix, IBM(IBM DB2의 경우) 및 Ingres에서도 자사의 데이터베이스 제품을 위해 호환 가능한 리소스 관리자를 제공한다.

리소스 분배자

리소스 분배자는 공유될 수 있는 비 영속적인 상태를 관리한다. 예를 들어, ODBC 리소스 분배자는 데이터베이스 연결 풀을 관리하면서 연결이 더 이상 필요하지 않을 때 각 연결을 해제시킨다.

.NET Framework 개체가 자동 트랜잭션에 참여할 수 있도록 하려면, Windows 2000 구성 요소 서비스를 사용하여 .NET Framework 클래스를 등록해야 한다. 그러나 모든 트랜잭션이 자동적인 것은 아니다. 트랜잭션을 프로그래밍할 때 수행되는 동작은 선택한 트랜잭션 모델에 따라 달라진다. 공용 언어 런타임(CLR)에서는 수동 트랜잭션 모델과 자동 트랜잭션 모델이 모두 지원된다.

Code Access Security

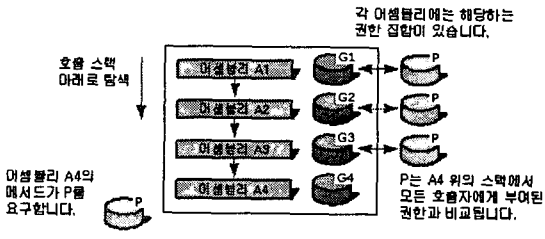
PC 시대 초창기에는 소프트웨어를 구매할 때 소프트웨어는 Box에 싸여 전산용품 가게의 진열대에 있었다. 이는 제조회사에서 해당 소프트웨어에 대한 보증을 포함하고 있는 것이다. 가끔 소프트웨어에 버그가 있었지만 자기 회사의 마케팅적 입장에서 바이러스를 고의로 퍼트리는 일은 없을 것이다.

하지만 오늘날은 더 이상 소프트웨어를 상점에서 구매하지는 않는다. 대부분의 소프트웨어를 인터넷을 통해서 다운로드 받는다. 웹을 통해서 소프트웨어를 배포 하는 것은 매우 대단한 것이기는 하지만 그만큼 보안의 문제가 증가하게 되었다.

사용자의 입장에서 보면 웹으로부터 다운로드 한 소프트웨어의 코드가 안전한지의 여부를 알아내기란 거의 불가능하다. 웹 코드를 안전하게 만들기 위해서 Microsoft사 에서 처음으로 시도했던 것은 1996년 ActiveX SDK와 함께 소개된 Authenticode 시스템이다. Authenticode는 사용자가 ActiveX 컨트롤을 다운로드 할 때 사용자가 다운로드 한 소프트웨어가 진짜 제조회사에서 만든 코드인가를 맞는지 확인하고, 출시된 이후로 변경된 사실이 없음을 확신할 수 있도록 제조회사에서 다운로드 하게 될 코드에 전자서명을 넣도록 하는 것이다.

Authenticode는 일단 코드가 설치되면 그 코드로부터 시스템을 지킬 방법은 없었다. Authenticode는 보안시스템이 아니라 책임소재만 알려주는 시스템인 것이다. 사실 해당 코드의 각 부분이 주어진 신뢰도 레벨에 따라서 연산을 제한 하는 것이다.

.NET CLR은 Code Access Security라는 것을 제공한다. 이것은 시스템관리자가 Assembly에 들어있는 신뢰도의 정도에 따라 관리된 코드 Assembly의 권한을 명시해 주는 것이다. 이렇게 관리된 코드가 설치 될 해당 시스템의 CLR을 호출하여 엑세스 할 때 CLR은 해당 Assembly에 그런 권한을 부여 했는지를 체크한다.

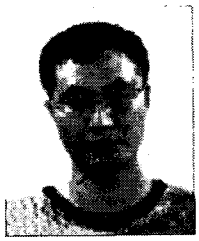


CLR은 COM과 같은 관리되지 않는 코드의 활동을 다스릴 수 없다. COM 객체는 CLR이 아니라 운영체제를 가지고 직접 처리한다. 그러나 관리되지 않는 코드에 액세스할 수 있는지의 여부는 시스템관리자가 설정하기 나름이었다.

시스템관리자는 보안 정책을 설정하게 되는데, 이것은 어떤 어셈블리가 어떤 형태의 연산을 할

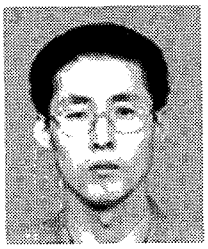
수 있고, 할 수 없는지를 알려주는 설정 규칙이다. 이러한 권한은 세 가지 레벨(엔터프라이즈. 머신, 사용자)로 설정될 수 있다. 하위 레벨 설정 내용은 상위 레벨에서 대체되어 더 엄격해 질 수 있다.

시스템 관리자는 XML 기반의 설정파일을 편집하여 코드 액세스 보안을 설정한다.



이 복 영

- 연세대학교 교육과학대학 학사 졸업
- 1999년 다우 교육원 전임강사 (VB,Visual C++, MFC, ASP, COM+)
- 2001년~현재 : ㈜웹타임 전임강사
- 현재 : (주)필라넷 기술이사
- 자격사항
  Microsoft Certified Trainer (MCT)
  Microsoft Certified Solution Developer (MCSD)



박 지 환

- 1990년 3월 일본 요코하마국립대학 전자정보공학과 졸업 (공학박사)
- 1994년 9월~1995년 3월 일본 동경대학 생산기술연구소 방문연구
- 1998년 1월~1998년 2월 일본 전기통신대학 방문연구
- 1999년 7월~1999년 8월 Monash University, Austrailia, Visiting Research
- 2001년 2월~2001년 3월 STA Fellowship, Communication Research Laboratory, Japan
- 1996년 4월~현재 동경대학 생산기술연구소 협력연구원
- 1990년 3월~현재 부경대학교 전자컴퓨터정보통신공학부 교수
- 1997년 3월~현재 한국통신학회 부호 및 정보이론 연구회 운영위원
- 1997년 3월~현재 한국통신정보보호학회 이사 및 영남지부 부지부장
- 1999년 3월~현재 한국정보처리학회 논문지 편집위원
- 관심분야: 멀티미디어 압축 및 응용, 정보보호 및 암호학
- e-mail: jpark@pknu.ac.kr