서비스 지향 아키텍처(SOA) 및 컴포넌트 기반 개발을 사용하여 웹 서비스 응용프로그램 빌드

Rational Software 백서

Alan Brown,
Simon Johnston,
Kevin Kelly

Rational®
the software development company

목차

소개

엔터프라이즈 규모의 소프트웨어 시스템 빌드는 복잡한 작업입니다. 오랜 기간의 기술 발전에도 불구하고 오늘날 정보 시스템의 요구는 기업이 업무에 중요한 소프트웨어 솔루션을 디자인, 구성 및 전개할 수 있는 능력의 한계까지 도달했습니다. 특히 완전히 새로운 시스템이 디자인되는 경우는 거의 드뭅니다. 오히려 일반적인 소프트웨어 설계자의 타스크는 일반적으로 기존 데이터 저장소를 활용하는 새 비즈니스 로직을 설명하거나 인터넷 브라우저 또는 휴대용 장치와 같은 새 채널을 통해 기존 데이터 및 트랜잭션을 나타내거나 중복된 비즈니스 활동을 지원하며 이전에 연결이 끊긴 시스템을 통합하는 등 기존 솔루션의 수명을 확장시키는 타스크입니다.

현재 Microsoft 및 IBM 과 같은 벤더가 소프트웨어 개발자를 지원하기 위한 상용 소프트웨어 하부 구조 제품을 제공하고 있습니다. 이러한 제품은 각각 .NET 및 WebSphere 제품군에서 지원하는 소프트웨어 개발 접근 방식의 핵심 역할을 합니다. 두 가지 접근 방식 모두 분배된 서비스로부터의 시스템 어셈블리에 초점을 맞춥니다. 그러나 서비스로부터의 엔터프라이즈 규모 솔루션 빌드에 새로운 사항이 있습니까? 컴포넌트 기반 시스템의 교훈을 서비스 지향 아키텍처(SOA)의 구성에 어떻게 적용할 수 있습니까? 새로운 소프트웨어 하부 구조 제품에 배치할 고품질 시스템을 빌드하기 위한 가장 효과적인 접근 방식은 무엇입니까? 이 문서의 주제는 이러한 중요 질문으로 구성됩니다.

최근, 소프트웨어 엔지니어링 커뮤니티의 관심사항은 대부분 독립적이고 재사용가능한 기능 콜렉션에서 대규모 소프트웨어 시스템을 어셈블할 수 있는 개념을 지원하는 디자인 접근 방식, 프로세스 및 도구에 초점을 맞추고 있습니다. 일부 기능은 이미 사용할 수 있어 내부에서 구현되거나 써드파트가 제공하고 있으며 나머지 기능은 작성해야 합니다. 이러한 경우, 이러한 모든 요소를 일관적인 단일 완전체(whole)로 통합하도록 전체 시스템을 고안하고 디자인해야 합니다. 오늘날 이러한 예는 *컴포넌트 기반 개발(CBD)*에서 찾을 수 있습니다. CBD 는

Microsoft .NET 플랫폼 및 J2EE(Java 2 Enterprise Edition) 표준과 같이 IBM WebSphere 및 Sun iPlanet 제품이 지원하는 기술적 접근 방식으로 실현되는 개념입니다.

추가 고려사항은 운영 체제를 일반적으로 여러 시스템에 분배하여 성능, 가용성 및 확장성을 개선해야 한다는 것입니다. 엔터프라이즈 솔루션은 하드웨어 콜렉션에서 실행되는 기능성을 조정해야 합니다. 그러한 시스템을 고안하기 위한 한 가지 방법은 상호작용 *서비스*의 콜렉션으로 작성되도록 고려하는 것입니다. 각 서비스는 잘 정의된 기능성 콜렉션에 대한 액세스를 제공합니다. 전체 시스템은 이러한 서비스 간의 상호작용 세트로서 디자인되고 구현됩니다. 기능성을 서비스로 나타내는 것이 유연성의 비결입니다. 이를 통해 다른 기능성(일반적으로 서비스로 구현)에서 실제 위치에 관계 없이 자연스럽게 다른 서비스를 이용할 수 있습니다. 시스템은 새 서비스를 추가하여 전개됩니다. 결과 *서비스 지향 아키텍처(SOA)*는 시스템을 구성하는 서비스를 정의하고 특정 동작을 실현하기 위해 서비스 간에 발생하는 상호작용에 대해 설명하며 특정 기술을 사용하여 서비스를 하나 이상의 구현으로 맵핑합니다.

서비스는 비즈니스 기능성을 캡슐화하지만 서비스 상호작용 및 통신을 용이하게 하려면 일부 서비스 간 하부 구조 양식이 필요합니다. 서비스는 단일 시스템에서 구현하거나 근거리 통신망의 컴퓨터 세트에 분배하거나 여러 회사 네트워크에 광범위하게 분배할 수 있으므로 이 하부 구조는 다양한 양식이 가능합니다. 특히 흥미로운 경우는 서비스가 통신 메커니즘으로 인터넷을 사용하는 경우입니다. 결과 *웹 서비스*는 보다 일반적인 서비스의 특성을 공유하지만 서비스 간 상호작용에 대한 불안전하고 충성도가 낮은 공용 메커니즘을 사용하므로 특별한 주의가 필요합니다.

지금까지 소프트웨어 산업은 대부분 웹 서비스 및 해당 상호작용을 구현하기 위한 기본 기술에 초점이 맞추어져 왔습니다. 그러나 웹 서비스를 엔터프라이즈 규모 솔루션으로 쉽게 어셈블할 수 있도록 디자인하는 가장 효과적인 방법과 관련한 관심사항이 대두되고 있습니다. 반대로, 웹 서비스로 구성되는 엔터프라이즈 규모 소프트웨어 솔루션의 아키텍처를 구성하기 위한 올바른 사례 및 도구에 대한 관심은 현저히 줄어들고 있습니다. 복잡한 구조 디자인의 경우와 같이, 고품질 솔루션은 잘 알려진 여러 디자인 기법, 구조 패턴 및 스타일에서 지원하는 아키텍처를 일찍 결정한 결과입니다. 이러한 패턴은 확장성, 신뢰성 및 보안과 같은 공통 서비스 문제를 해결합니다.

이 문서는 엔터프라이즈 규모 소프트웨어 솔루션에 대한 서비스 및 서비스 지향 아키텍처(SOA)를 심층적으로 이해할 수 있는 컨텍스트를 제공합니다. 특히 보다 명확한 소프트웨어 컴포넌트 개념과 관련하여 서비스를 탐색하며, 현재 컴포넌트 기반 개발 사례가 서비스 지향 아키텍처(SOA) 구현을 위한 시도 및 테스트 기반을 제공하는 방식에 대해 설명합니다. 인터페이스 기반 디자인은 서비스 및 컴포넌트 디자인 모두에 대한 키로 강조표시됩니다. 또한 이 두 디자인 모두로 표시된 인터페이스에는 두 디자인을 구분하는 특정 제한조건 및 기준이 필요합니다. UML(Unified Modeling Language)[1]은 논리 및 구현 디자인 모두와, 컴포넌트 및 서비스 디자인 모두에 대한 특정 패턴에 대해 설명하는 도구로 사용됩니다.

마지막으로 이 문서는 일반적인 서비스 구현과 관련된 문제를 탐색하기 위한 도구로서의 웹 서비스에 초점을 맞춥니다. 특히 이 문서는 웹 서비스의 특정 디자인 패턴으로서의 서비스 지향 아키텍처(SOA)에 대한 일반 안내를 해석하는 방법에 대해 설명합니다. 이 문서에서 설명하는 모든 패턴은 Rational_ XDE 의_ 고유 패턴 및 코드 템플리트 기능을 사용하여 구현되었으며 Rational Developer Network[8]를 통해 공개되었습니다.

서비스 지향 아키텍처(SOA)의 개념

서비스 지향 아키텍처(SOA)란 공개되어 검색 가능한 인터페이스를 통해 일반 사용자 응용프로그램 또는 다른 서비스에 서비스를 제공하도록 소프트웨어 시스템을 디자인하는 방법입니다. 일반적으로 서비스는 별도 비즈니스 기능을 보다 효과적으로 나타내는 방법, 즉, 비즈니스 프로세스를 지원하는 응용프로그램을 효과적으로 개발하는 방법을 제공합니다.

서비스 지향 아키텍처(SOA)는 새로운 개념은 아니지만 이 시점에서는 새로운 웹 서비스 기술로 인해 중요합니다. 예를 들어, 다음은 2000 년에 출간된 서적에서 인용한 내용으로, 서비스 지향 아키텍처(SOA)의 가치에 대한 설명입니다.

> *서비스 지향 솔루션_ 응용프로그램은 잠재 사용자에게 잘 정의된 인터페이스를 제공하는, 상호작용 서비스의 독립 세트로서 개발되어야 합니다. 마찬가지로, 응용프로그램 개발자가 서비스 콜렉션을 검색하고 관심있는 서비스를 선택하며 원하는 기능을 작성하기 위해 어셈블할 수 있는 지원 기술을 사용할 수 있어야 합니다. [2]*

이 문서의 목적에 따라 고려해야 할 서비스 정의는 다음과 같습니다.

> *서비스는 일반적으로 덜 세분화되고 검색 가능한 소프트웨어 엔티티로서 구현됩니다. 이 엔티티는 단일 인스턴스이며 결합이 약한(일반적으로 비동기) 메시지 기반 통신 모델을 통해 응용프로그램 및 기타 서비스와 상호작용합니다.*

많은 경우, 서비스 용어는 컴포넌트 기반 개발을 설명하는 데 사용되는 용어와 거의 동일합니다. 그러나 아래 그림 1 과 같이 웹 서비스 내부의 요소를 정의하는 데 사용되는 특정 용어가 있습니다.

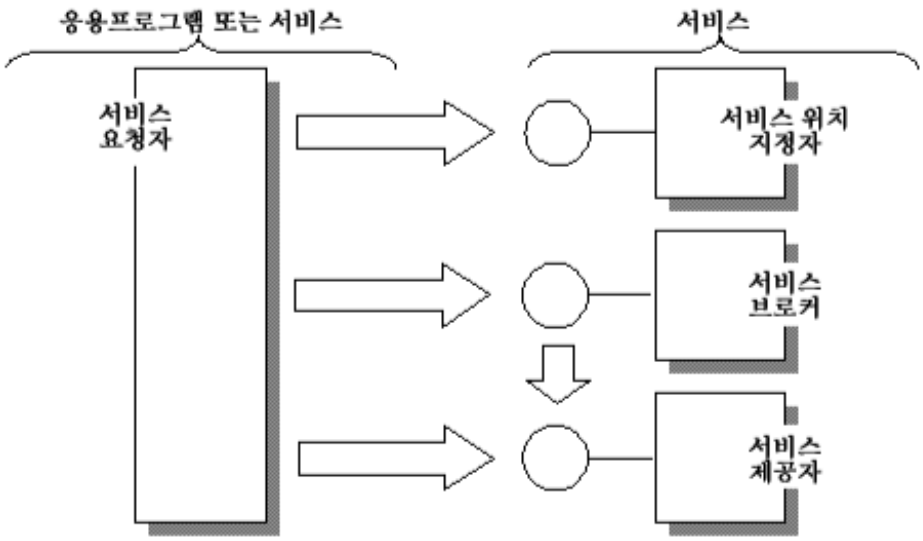

그림 1 – 서비스 용어

- **서비스** — 논리 엔티티. 하나 이상의 공개된 인터페이스로 정의되는 계약

- **서비스 제공자** — 서비스 스펙을 구현하는 소프트웨어 엔티티
- **서비스 요청자** — 서비스 제공자를 호출하는 소프트웨어 엔티티. 기존의 "클라이언트"와 같은 의미의 용어입니다. 그러나 서비스 요청자에는 일반 사용자 응용프로그램 또는 다른 서비스가 포함됩니다.
- **서비스 위치 지정자** — 레지스트리 기능을 수행하는 특정 유형의 서비스 제공자로서, 서비스 제공자 인터페이스 및 서비스 위치 찾아보기를 허용합니다.
- **서비스 브로커** — 하나 이상의 추가 서비스 제공자로 서비스 요청을 전달할 수 있는 특정 유형의 서비스 제공자

이 서비스 설명 및 사용 컨텍스트에 따라 일련의 제한조건이 적용됩니다. 또한 효율적인 서비스 사용을 위해 몇 가지 상위 레벨 우수 사례를 제안합니다. 다음은 효과적인 서비스 사용을 위한 몇 가지 중요 특성입니다.

- **덜 세분화됨** — 서비스에 대한 오퍼레이션은 일반적으로 보다 많은 기능을 포함하도록 구현되며 컴포넌트 인터페이스 디자인과 비교하여 대규모 데이터 세트에 대해 수행됩니다.
- **인터페이스 기반 디자인** — 서비스는 별도 정의된 인터페이스를 구현합니다. 이러한 경우 여러 서비스에서 공통 인터페이스를 구현할 수 있으며 단일 서비스에서 여러 인터페이스를 구현할 수 있습니다.
- **검색 용이** — 서비스는 디자인 시간과 런타임에서 고유 ID 뿐만 아니라 인터페이스 ID 및 서비스 유형으로 모두 찾을 수 있어야 합니다.
- **단일 인스턴스** — 필요에 따라 컴포넌트를 인스턴스화하는 컴포넌트 기반 개발과 달리, 각 서비스는 여러 클라이언트가 통신할 수 있는 항상 실행되는 단일 인스턴스입니다.
- **약한 결합** — 서비스는 XML 문서 교환과 같이 종속성이 약하고 결합되지 않은 메시지 기반의 표준 방법을 사용하여 다른 서비스 및 클라이언트에 연결됩니다.
- **비동기** — 일반적으로 서비스는 비동기 메시지 전달 접근 방식을 사용하지만 필수사항은 아닙니다. 실제로, 때때로 많은 서비스가 동기 메시지 전달을 사용합니다.

인터페이스 기반 디자인 및 발견성과 같은 이러한 기준 중 일부는 컴포넌트 기반 개발에도 사용됩니다. 그러나 J2EE 또는 .NET 와 같은 컴포넌트 아키텍처를 사용하여 개발된 응용프로그램과 서비스 기반 응용프로그램을 구분하는 이러한 속성의 합계입니다.

인터페이스 기반 디자인

컴포넌트 및 서비스 개발 모두 인터페이스 디자인은 소프트웨어 엔티티가 해당 정의의 핵심 파트를 구현하고 표시하는 방식으로 수행됩니다. 따라서 "인터페이스"의 개념은 컴포넌트 기반 및 서비스 지향 시스템 모두의 올바른 디자인을 위한 핵심 요소입니다. 다음은 인터페이스 관련 몇 가지 핵심 정의입니다.

- **인터페이스** — 논리적으로 그룹화되지만 구현을 제공하지 않는 공용 메소드 서명 세트를 정의합니다. 인터페이스는 서비스 요청자와 제공자 간 계약을 정의합니다. 인터페이스 구현은 모든 메소드를 제공해야 합니다.
- **공개된 인터페이스** — 클라이언트가 동적으로 발견하는 레지스트리에서 사용할 수 있으며 고유하게 식별되는 인터페이스[3]
- **공용 인터페이스** — 클라이언트가 사용할 수 있지만 공개되지 않은 인터페이스로서, 클라이언트 파트에 대한 정적 지식이 필요합니다.

- **이중 인터페이스** — 일반적으로 인터페이스는 특정 인터페이스가 다른 인터페이스에 의존하는 쌍으로 개발됩니다. 예를 들어, 클라이언트 인터페이스는 일부 콜백 메커니즘을 제공하므로 클라이언트는 요청자를 호출하는 인터페이스를 구현해야 합니다. 이 개념은 웹 서비스에서 도입된 개념입니다.

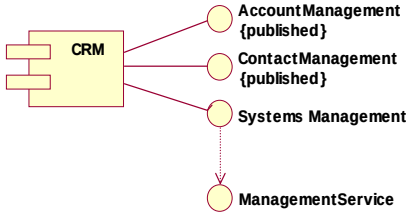

그림 2 – 구현된 서비스

그림 2 는 고객 관계 관리(CRM) 서비스의 UML 정의를 보여줍니다. 이 서비스는 UML 컴포넌트로 표시되며 AccountManagement, ContactManagement 및 SystemsManagement 인터페이스를 구현합니다. 이 중 처음 두 인터페이스만 공개된 인터페이스이며 마지막 인터페이스는 공용 인터페이스입니다. SystemsManagement 인터페이스와 ManagementService 인터페이스가 이중 인터페이스를 구성합니다. CRM 서비스는 이러한 인터페이스를 원하는 수만큼 구현할 수 있습니다. 또한 서비스(또는 컴포넌트)는 이 기능을 통해 동작 구현에 있어 유연성을 허용하는 클라이언트에 따라 여러 가지 방식으로 동작을 수행할 수 있습니다. 또한 특정 클라이언트 클래스에 다른 서비스 또는 추가 서비스를 제공할 수도 있습니다. 일부 런타임 환경에서는 이러한 기능을 사용하여 단일 컴포넌트 또는 서비스에서 동일한 인터페이스의 다른 버전을 지원할 수도 있습니다.

인터페이스 동작

Java 또는 C#과 같은 언어나 IDL 과 같은 언어의 인터페이스 정의는 메소드 서명 세트를 제공합니다. 이 정의는 해당 "방식"에 대한 안내가 없이 "대상"을 제공합니다. 예를 들어, 그림 3 의 보안 인터페이스를 고려할 때, 이 인터페이스 구현을 호출하는 클라이언트가 세 가지 공용 메소드를 호출할 수 있습니다.

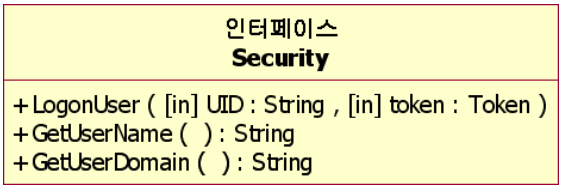

그림 3 – UML 의 인터페이스

"대상"을 정의하는 것만으로는 사용자가 로그온할 때까지 클라이언트가 GetUserName( ) 또는 GetUserDomain( )을 호출할 수 없다는 사실을 나타낼 수 없습니다. 다음 상태 머신은 이 종속성 또는 동작을 보여줍니다. 이러한 유형의 제한조건은 일반적으로 인터페이스 기반 디자인에 대한 문서에 포함됩니다. 그러나 프로그래밍 언어에서는 지원되지 않으므로 인터페이스 구현자가 동작 스펙을 준수하는지 확인할 수 없습니다.

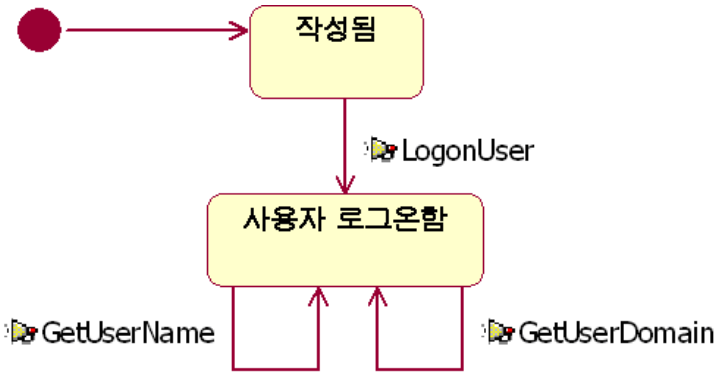

그림 4 – 인터페이스 동작

그러나 서비스 협업을 통해 비즈니스 프로세스를 실현하도록 보다 쉽게 통합되고 구성될 수 있다는 기대를 가지고 비즈니스가 서비스 지향 시스템으로 점차 이동하고 있습니다. 결과적으로 인터페이스 동작 정의의 개념과, 보다 중요한 관련 인터페이스 세트 동작에 대한 업계 관심이 증가하고 있습니다. 그러나 현재 이에 대한 표준 접근 방식은 거의 없습니다.

한 가지 접근 방식은 이 문서에서 소개한 디자인 모델을 사용하는 것입니다. 이러한 모델은 UML 과 같이 표준화된 언어로 정의되어 서비스 인터페이스 간 상호 종속성을 문서화합니다. 이러한 모델은 공유 및 일반화되어, 도입 시 특정 표준을 파생시키는 데 사용될 수 있습니다.

또한 Rational 은 이 문제점에 적용할 수 있는 자산을 패키지하고 공유하는 메커니즘을 제공하는 재사용가능한 자산 스펙(RAS)을 후원합니다. 예를 들어, RAS 메커니즘을 사용하여 서비스 세부사항을 분배하는 경우, 해당 동작에 대해 설명하는 모델을 패키지할 수도 있습니다. 이러한 모델에서는 시퀀스 다이어그램을 사용하여 인터페이스 호출 간 필수 상호작용을 표시할 수 있습니다.

서비스 지향 시스템 아키텍처 구성

소프트웨어 엔지니어링에서 새로 개발하는 경우, 이전 프로젝트에서 사용한 것과 동일한 기법 및 도구를 적용할 수 있는 것으로 쉽게 가정할 수 있습니다. 컴포넌트와 서비스는 단지 유사할 뿐 실제로는 다르다는 사실에 대해 이미 언급했습니다. 즉, 디자인 기준과 디자인 패턴이 서로 다릅니다. 이 섹션은 중요한 실제 결과에 대해 설명합니다. *결론은 서비스로 변환된 올바른 컴포넌트가 모두 올바른 서비스를 작성하는 것은 아니라는 점입니다.*

응용프로그램 디자인 계층화

이전 솔루션으로 새 문제점을 해결하려는 경향은 새로운 것이 아닙니다. 마찬가지로, 개발자가 컴포넌트 기반 시스템을 작성하기 시작하면서 객체 지향 개발과 관련한 자신의 경험을 유사한 문제점에 적용하기 위해 시도해 왔습니다. 또한 경험이 축적됨에 따라 객체 지향 기술 및 언어가 효과적인 컴포넌트 구현 방법으로 인정받게 되었습니다. 그러나 이 때 결정 및 구현을 통한 절충을 이해해야 합니다. 즉, 다양한 동작을 구현하기 위한 상속 및 집계 관련 절충과, 클래스 라이브러리를 단일 C++ 응용프로그램에 대한 기반이 아닌 컴포넌트 세트에서 사용할 수 있도록 재디자인하는 것에 대해 이해해야 합니다.

마찬가지로, 전형적인 컴포넌트 기반 응용프로그램이 반드시 전형적인 서비스 지향 응용프로그램을 작성하지는 않지만,*컴포넌트는 가장 효과적인 서비스 구현 방식입니다.* 응용프로그램 아키텍처에서 서비스로 수행되는 역할을 이해하면 회사의 컴포넌트 개발자 및 기존 컴포넌트를 효과적으로 활용할 수 있습니다. 이러한 전이(Transition)를 작성하려면 서비스 지향 접근 방식이 추가 응용프로그램 아키텍처 계층을 의미함을 이해해야 합니다. 아래 그림 5 는 응용프로그램 이용자와의 관계 진전에 따라 보다 구체적이지 않은 구현을 제공하기 위해 기술 계층을 응용프로그램 아키텍처에 적용할 수 있는 방법을 보여줍니다. 이 시스템 파트를 나타내기 위해 "응용프로그램 에지"라는 새로운 용어를 사용합니다. 이 용어는 서비스가 기존의 컴포넌트 디자인을 사용한 내부 재사용 및 컴포지션으로 시스템의 외부 보기를 효과적으로 나타낼 수 있는 방법임을 반영합니다.

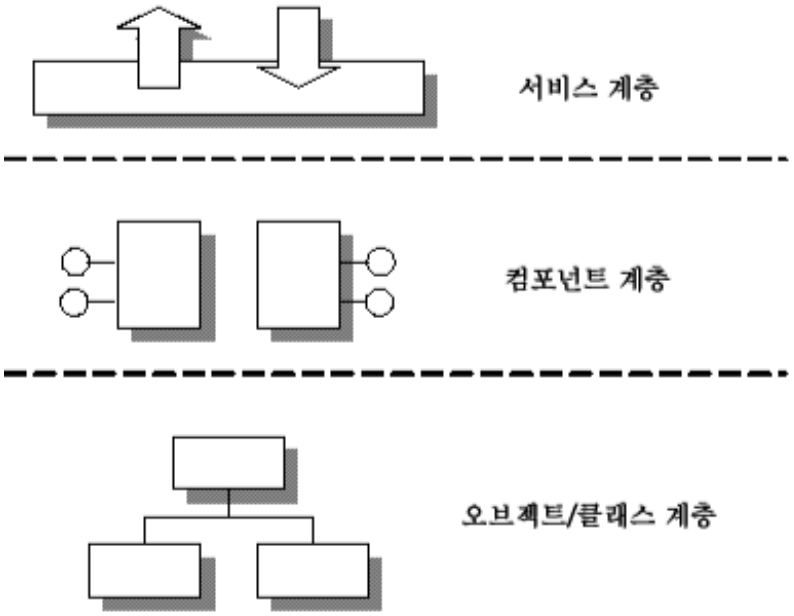

그림 5 – 응용프로그램 구현 계층

일반적으로 객체 지향 사고에서 컴포넌트 지향 사고로의 전환을 위해서는 개발자가 이 신기술과 관련 요구사항을 이해하는 데 6 – 18 개월이 소요됩니다. 그러나 서비스 지향 시스템으로의 전환은 보다 빠른 시일 내에 이루어질 수도 있습니다. 이를 위해서는 개발자가 서비스 지향 응용프로그램 지원하의 컴포넌트 개발 및 재사용을 허용하는 디자인 결정사항, 절충 및 도전 과제를 이해해야 합니다.

예제 고객 모델

컴포넌트와 서비스가 응용프로그램 실현을 위해 상호작용하는 방식을 설명하기 위해 아래 그림 6 의 UML 클래스 다이어그램에 정의된 대로 고객 관계 정보를 관리하는 예제를 고려합니다.

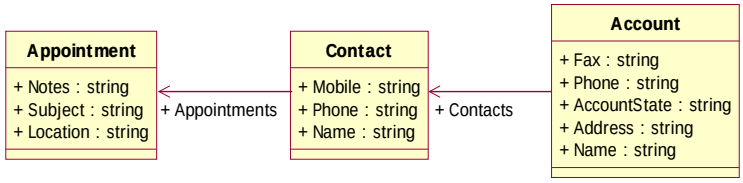

그림 6 – 논리 고객 모델

다음 섹션에서는 개발자가 이러한 논리 모델을 컴포넌트 기반 응용프로그램 및 서비스 기반 응용프로그램에 규정하는 방식에 대해 설명합니다. 이 모델은 시스템 동작에 대해서는 설명하지 않습니다. 이러한 변환 단계 중 상당 수는 자동으로 생성됩니다. Rational Software 는 응용프로그램 아키텍처를 모델링하고 해당 패턴을 수집 및 적용하며 전체 개발 라이프사이클에서 이러한 모델/코드 아티팩트를 관리하는 도구를 제공합니다.

컴포넌트 기반 디자인

예제 논리 모델이 컴포넌트 디자인으로 변환되는 방식을 이해해야 합니다(예: COM 또는 J2EE 의 경우). 아래 그림 7 은 모델의 컴포넌트 기반 디자인을 보여줍니다.

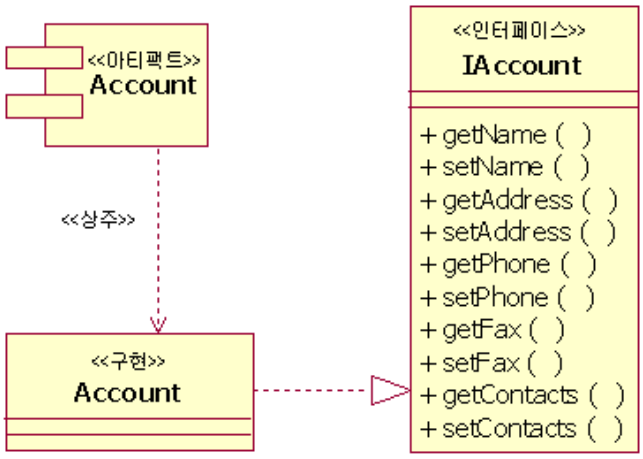

그림 7 – 일반 컴포넌트 다이어그램

위의 인터페이스에는 한 가지 *핵심* 디자인 기능이 있습니다. 즉, 기존 컴포넌트 플랫폼에 대한 공통 패턴이 있습니다. 분석 클래스의 각 속성에 대해 두 가지 오퍼레이션(값을 설정하는 하나의 오퍼레이션과 값을 리턴하는 하나의 오퍼레이션)을 제공해야 합니다. 로컬 컴포넌트의 경우, 메소드 호출 오버헤드를 무시할 수 있으며 원격 오브젝트의 경우, 오버헤드를 최소화하도록 원격 프로시저 호출(RPC) 메커니즘이 최적화됩니다. 대부분의 경우, 클라이언트는 응용프로그램에 특성 서브세트만 필요하므로 필요에 따라 액세스할 수 있습니다.

서비스 지향 디자인

위의 디자인 모델은 각 컴포넌트 인스턴스가 단일 오브젝트를 나타내는 컴포넌트 구현을 이해하기 위한 올바른 방법을 보여줍니다. 예를 들어, CRM 의 각 개인 연락처에게 있어 데이터베이스는 논리적으로 별도 컴포넌트가 됩니다. 따라서 컴포넌트 ID 는 연락처 ID 와 연결됩니다.

그러나 서비스마다 자원 세트를 관리하는 단일 인스턴스가 존재하므로 대부분의 경우 해당 상태는 Stateless 입니다. 즉, 서비스는 유형 또는 유형 세트의 인스턴스를 작성하고 관리할 수 있는 *관리자* 오브젝트로 인식해야 합니다. 이러한 경우 인스턴스 상태를 나타내는 *값 오브젝트*를 실제로 단순 직렬화된 상태인 오브젝트로 이용하는 디자인 패턴(컴포넌트 간 전송에 대한 상태가 지속되는 분산 시스템의 공통 패턴)이 생성됩니다. 한 가지 흥미로운 결과는 그림 7 의 모델과 같이 컴포넌트 정의를 사용하고 서비스로 변환하는 규칙을 정의할 수 있는 경우 이 직렬화를 패턴으로 구현할 수 있다는 것입니다. 이러한 패턴의 작성 및 재사용은 Rational XDE 를 사용하여 가능합니다.

이처럼 제공자에서 요청자로의 상태 전달은 컴포넌트 상태를 검색하기 위해 여러 소규모 오퍼레이션보다 대규모 단일 오퍼레이션을 사용함을 의미합니다. 대부분의 서비스는 원격 서비스이므로 이러한 특징은 원격 서비스에 대한 네트워크 사용 및 대규모 값 오브젝트 처리 시 요청자 동작과 관련하여 특정 의미가 있습니다. 또한 요청자에게 일부 엔티티의 상태 사본이 제공되며 이 사본이 최신 사본이 아닌지 여부를 확인해야 한다는 점에서도 의의가 있습니다. 최신 정보가 아닌 정보가 제공될 수 있는 주식 시세 또는 일기 예보를 검색하는 경우 이를 허용해야 합니다. 또한 데이터 유형도 고려해야 합니다. 예를 들어, 주식 시세 데이터가 일기 데이터보다 빨리 쓸모없게 됩니다(stale). 여기서 설명하는 아키텍처에서는 요청자가 상태 사본을 허용해야 합니다. 자세한 내용은 이 문서 후반부의 "덜 세분화된 상호 작용의 영향" 섹션을 참조하십시오.

이러한 요구사항을 고려할 때 특정 서비스 구성을 예측할 수 있습니다. 그림 8 의 모델 단편은 이러한 패턴을 디자인 레벨에서 설명할 수 있는 방식을 보여줍니다. 즉, 컴포넌트가 공개하는 인터페이스와 인터페이스가 조작하는 값 오브젝트를 보여줍니다.

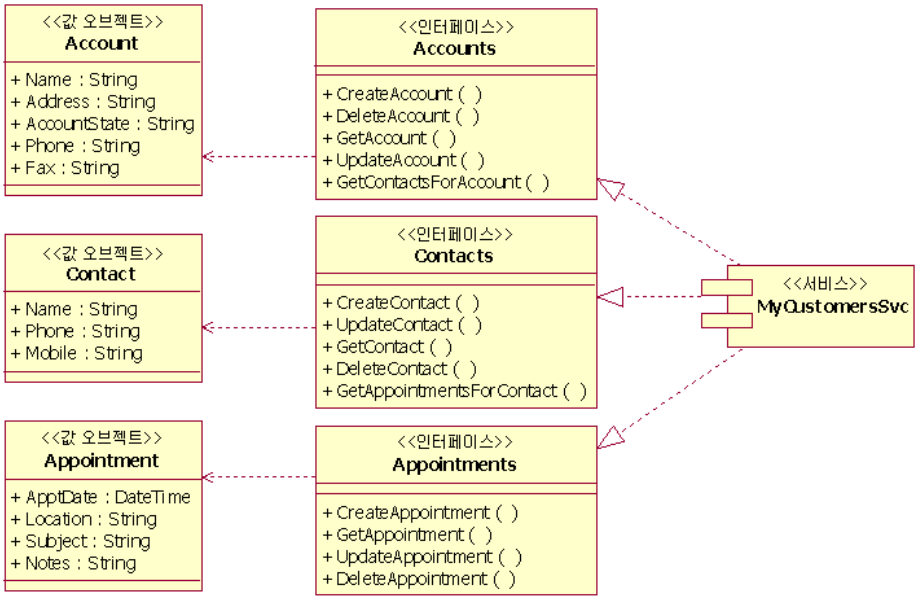

그림 8 – 일반 서비스 지향 디자인

지금까지 서비스 디자인 방법에 대해 설명했으며 이제 이 특정 디자인의 일부 속성에 대해 알아봅니다. 먼저, 제공자 MyCustomerSvc 에서 해당 상호작용에 대한 요청자로의 단순 "get" 또는 "set"보다 값 오브젝트에 많은 정보가 전달되며 이로 인해 네트워크 대역폭에 영향을 줄 수 있습니다. 그러나 웹 서비스 특성을 고려할 때, 서비스 구현에서 사용되는 프로토콜은 컴포넌트 구현에서 사용되는 프로토콜과 많은 차이가 있습니다. 이러한 플랫폼은 설계자 또는 정보

엔지니어에게 네트워크에 영향을 주지 않고 각 값 오브젝트의 컨텐츠를 극대화할 수 있는 값 오브젝트 및 해당 컴포지션을 주의깊게 선택해야 하는 추가 부담을 줍니다.

서비스 지향 디자인의 캐싱

이전 섹션에서 소개한 제공자에서 요청자로의 "오래된(stale)" 정보 사본 전달에 대한 개념에 대해 다시 생각해봅니다. 예를 들어, 주식 포트폴리오 관리 응용프로그램을 개발하는 경우, 웹 서비스에 현재 주식 가격과 각 주식에 대해 계속해서 질문하지 않습니다. 주식에는 3 - 5 자 데이터를 전달하고 가격에는 5 - 7 자 데이터를 전달합니다. 이러한 경우 네트워크 및 서비스 제공자가 허용할 수 없는 로드가 가해질 수 있습니다. 요청자는 기호 목록을 전달하거나 서비스에 포트폴리오 ID 를 전달하고 각 주식에 대한 모든 정보를 검색하여 전체 포트폴리오의 컨텐츠를 요청해야 합니다. 이제 사용자가 단순히 단일 기호에 대한 갱신을 요청한 경우, 지나친 행동으로 보일 수 있습니다. 그러나 이제 요청자는 결과를 캐시할 수 있으며 이후 사용자가 다른 기호에 대한 갱신을 요청하는 경우 요청자가 캐시에 만족할 수 있습니다. 이제 요청자에 대한 작업은 데이터의 "임대차" 지속 기간을 식별하는 것입니다. 따라서 주식 시세 서비스의 지연 시간이 20 분인 경우 포트폴리오에 대해 25%의 수익을 제공하고 결과를 5 분 동안 캐시할 수 있습니다.

이 패턴은 정보 시스템에서 반복 표시됩니다. 사용자가 주문 관리 시스템에서 주문을 검색할 때마다 해당 사용자에게 주문 사본이 효과적으로 제공됩니다. 시스템에서 주문에 대한 추가 액세스를 차단하지 않은 경우 다른 사용자가 동시에 해당 주문을 갱신할 수 있기 때문입니다. 이제 웹 서비스 제공자가 요청자와의 상호작용의 일부로 실제로 캐시 또는 임대차 지속 기간을 식별할 수 있어야 합니다. 이러한 문제는 MSMQ(Microsoft Message Queue Server) 및 IBM MQSeries 와 같이 메시지 제한시간 및 만기 시간이 일상적으로 관리되는 메시지 전달 시스템에서 잘 해결됩니다.

이 문제점에 대해서는 이 문서 후반부에서 재검토하며 이 문제에 대처하기 위한 요청자 및 제공자 개발에 대한 특정 안내를 제공합니다.

XML 웹 서비스 응용프로그램 디자인

이제 XML 웹 서비스는 일상적으로 사용되는 용어이며 웹 서비스 배치 및 개발을 위한 플랫폼 및 도구와 더불어 벤더 지원 또한 증가하고 있습니다. 웹 서비스 스택의 요소에 대해서는 이미 다른 문서에서 설명했으므로 이 문서에서는 면밀히 검토하지 않습니다. 다음 정의는 W3C(World Wide Web Consortium) 웹 서비스 아키텍처 작업 그룹에서 인용한 정의입니다.

> *웹 서비스는 URI 로 식별되는 소프트웨어 응용프로그램으로서, 해당 인터페이스 및 바인딩을 XML 아티팩트로 정의, 설명 및 발견할 수 있습니다. 또한 웹 서비스는 인터넷 기반 프로토콜을 통한 XML 기반 메시지를 사용하여 다른 소프트웨어 응용프로그램과의 직접 상호작용을 지원합니다.*[4]

서비스 지향 아키텍처(SOA) 용어를 설명할 때 그림 1 에 오늘날 웹 서비스 사용 관련 기술이 적용되는 방식에 대해 간략히 설명합니다.

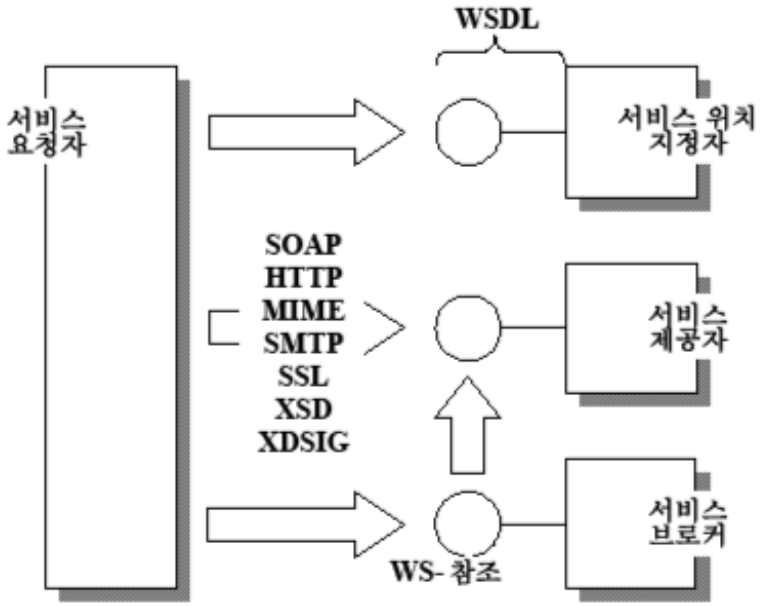

그림 9 – XML 웹 서비스 표준

먼저, 실제 웹 프로토콜인 HTTP 를 통한 SOAP(Simple Object Access Protocol)로 모든 웹 서비스가 XML 메시지를 사용한다고 잘못 알고 있는 경우가 있습니다. 이는 사실과 다릅니다. 웹 서비스 메시지는 XML 을 사용할 수 있지만 2 진 데이터를 전송합니다. 일반적으로 SOAP 헤더를 사용하지만 메시지 본문에 SOAP 인코딩을 사용하지 않아도 됩니다. 또한 HTTP 를 전송 수단으로 사용할 수 있지만 SMTP 또는 다른 수단도 사용할 수 있습니다. 웹 서비스의 설명 및 발견에 대해 잘 정의된 두 가지 표준, 즉 WSDL(Web Services Definition Language)과 UDDI(Universal Discovery, Description and Integration)가 존재합니다.

이러한 형식 및 전송 프로토콜의 유연성은 웹 서비스의 현재 문제 중 하나입니다(상호운용성). SOAP 형식, 엔벨로프(envelope), 전송 프로토콜 등에 대한 선택사항을 고려할 때 두 가지 구현 방법으로 정보를 전송하는 방법에 대해 알아봅니다. 이를 위해서는 WS-I(Web Services Interoperability) 그룹이 필요합니다. 이 그룹은 기존 및 최신 표준의 사용법에 대한 안내를 업계에 제공합니다. 벤더 또한 IBM WebSphere Studio Application Developer Integration Edition 과 같은 유연한 웹 서비스 개발 환경을 제공하여 도움을 줍니다. 이 제품은 필요에 따라 가장 빠르거나 올바른 세트를 사용할 수 있도록 여러 형식 및 전송 프로토콜을 포함하는 웹 서비스를 작성합니다.

웹 서비스 디자인 및 구현 패턴

웹 서비스는 응용프로그램의 기능적 요구사항과 관련한 분석 및 디자인 프로세스를 실제로 변경하지 않습니다. 예를 들어, 보험 청구액 처리 응용프로그램은 계속 보험 청구액을 처리해야 합니다. 여기서는 비기능적 요구사항 영역의 여러 제한조건 및 잠재 문제에 대해 설명합니다. 다음 섹션은 이처럼 가능한 일부 위험에 대해 설명합니다.

성능 및 신뢰성

HTTP 및 SOAP(원래 속도가 느리고 신뢰할 수 없음) 기반 아키텍처에서 웹 서비스 성능, 신뢰성 및 확장성에 필요한 기능을 제공할 수 있는지 여부에 대한 질문이 자주 제기됩니다. 먼저, "느리고 신뢰할 수 없음"을 정의해야 하며 하루 업무 마감 시 신뢰할 수 있는 전송도 신뢰할 수 없는 수단에 의존함을 이해해야 합니다. 엔터프라이즈 규모 솔루션의 아키텍처를 구성하고 디자인하는 경우, 항상 기능적 및 비기능적 요구사항을 고려하여 비즈니스 목적을 지원하는 올바른 절충을 수행하고 결정을 내려야 합니다.

예를 들어, HTTP 를 통한 SOAP 를 사용하는 경우, 항상 메시지 수신 확인 및 보안에 대한 추가 기능을 제공하는 응용프로그램 레벨 프로토콜 및 상호작용을 빌드할 수 있습니다. 그러나 특정 서비스가 동일한 보안 또는 응용프로그램 컨텍스트에서 통신을 수행하는 것으로 고려하는 경우, HTTP 와 다른 수단을 사용할 수 있습니다. 그림 10 의 예제를 고려하십시오.

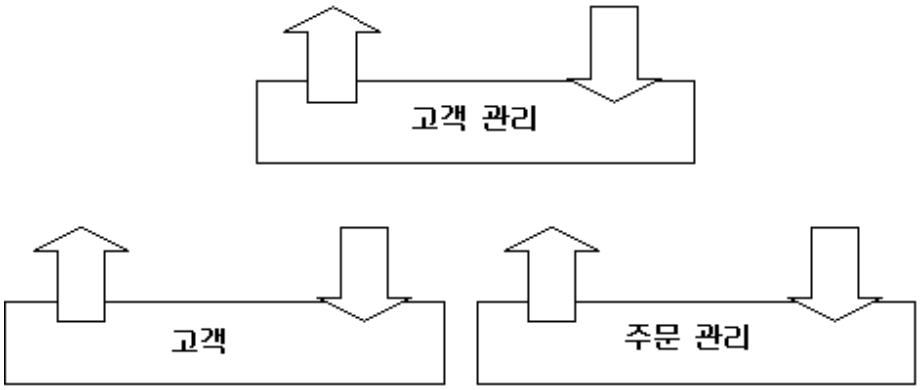

그림 10 – 서비스 종속성

모든 외부 클라이언트는 기본적으로 고객 관리 서비스와 상호작용하지만 두 개의 내부 서비스와도 상호작용합니다. 여기서는 이러한 내부 서비스 통신을 위해 HTTP 및 SOAP 의 유연성이 필요한 이유를 결정해야 합니다. 이와 관련하여, 성능이 고객 관리와 고객 간의 상호작용을 위한 핵심 요구사항인 것으로 가정합니다. 이러한 경우, 2 진 인코딩 형식 및 고성능 특성을 제공하는 컴포넌트 RPC 통신(예: Microsoft .NET Remoting 또는 Java RMI)을 사용하도록 결정할 수 있습니다. 반면에 고객 관리에서 주문 관리로 주문을 이행하기 위한 핵심 요구사항은 보장된 전달이므로 보다 우수한 신뢰성을 위해 성능을 절충하는 경우 대기열 지정 기술(예: IBM MQSeries 또는 MSMQ)을 사용하여 메시지를 전달할 수 있습니다.

웹 서비스가 단순 모델과 단순하면서도 유연한 프로토콜 세트를 제공하는 경우라도 이러한 선택사항의 제한을 받지 않습니다. WSDL 에는 이미 SOAP 및 HTTP GET/PUT 모두에 대한 바인딩이 포함되므로 요청자에게 추가 선택사항을 제공해야 합니다. 예를 들어, 단일 서비스가 메시지 대기열 바인딩 *및* SOAP 바인딩을 사용하여 메시지를 나타낼 수 있으므로 요청자는 사용하기에 보다 적합한 바인딩을 선택할 수 있습니다. 이러한 경우, 메시지 대기열을 사용하지만 HTTP 대화를 보증하는 서비스가 없으면 변환 제공자가 또한 보증 서비스 레벨과 같은 인센티브를 제공할 수 있습니다.

초기에 결정해야 하는 또 다른 디자인 결정사항은 메시지 교환이 멱등(idempotent) 또는 상호(commutative) 방식인지, 또는 두 가지 모두인지 여부입니다. *멱등*이란 동일한 메시지가 여러

번 도착하고 각 경우마다 *조치가 수행되는* 경우에도 잘못된 영향이 없음을 의미합니다. *상호*란 두 개의 관련 메시지가 잘못된 영향 없이 임의의 순서대로 도착할 수 있음을 의미합니다. 서비스 디자인에서 메시지 교환을 최소한 멱등으로 식별하는 경우, 신뢰성이 보다 낮은 전송이 보다 매력적이고 비용도 저렴한 옵션입니다.

보안(이 문서에서는 설명하지 않음)은 실제로 단순하고 저렴한 선택사항에서 복잡하고 많은 비용이 소요되는 선택사항에 이르기까지 여러 가지 선택사항으로 구성되는 것과 같이 성능, 신뢰성 및 확장성에 대한 디자인 목적은 여러 가지 결정사항을 수반합니다. 즉, 필요한 비용, 확보 가능한 예산 등을 고려해야 합니다. 서비스는 많은 솔루션을 제공하며 기존 개발 접근 방식만큼 많은 선택사항도 존재합니다.

비동기 동작 및 대기열 지정을 통한 확장성

서비스 지향 아키텍처(SOA)에 대한 소개에서 언급한 대로 웹 서비스를 본질적으로 비동기식으로 만드는 것이 바람직합니다. 웹 서비스와 연관된 추가 전송 오버헤드 및 서비스는 본질적으로 원격 서비스라는 예측으로 인해, 요청자가 응답을 대기하는 데 소요되는 시간을 줄여야 합니다. 비동기 서비스 호출과 별도 리턴 메시지를 통해, 제공자는 계속 응답하고 요청자는 계속 실행할 수 있습니다. 이는 동기 서비스 동작이 잘못되었음을 의미하는 것이 아니며 경험상 특히 통신 비용이 크거나 네트워크 지연을 예측할 수 없을 때 비동기 서비스 동작이 바람직함을 나타냅니다.

그림 11 의 예제를 참조하십시오.

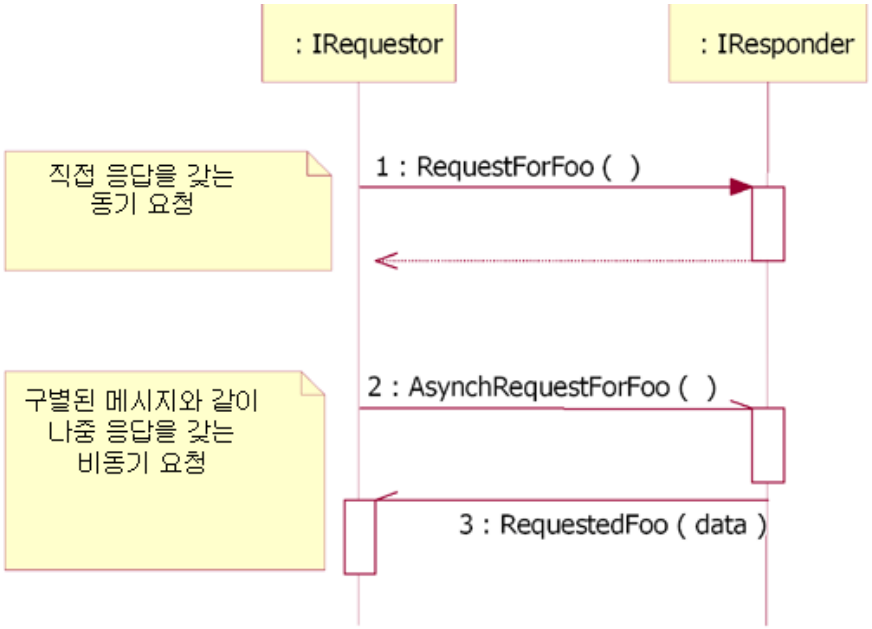

그림 11 – 동기 대 비동기

그림 11 에서 설명하는 동작은 확장성이 우수한 웹 서비스를 구현하기 위한 중요 단계입니다. 비동기 서비스 호출을 작성함으로써 제공자가 여러 작업자 스레드를 사용하여 여러 클라이언트 요청을 처리할 수 있습니다. 비동기 오퍼레이션 모드를 지원하기 위해서는 클라이언트로 신속하게 리턴하는 것 이외에 추가 작업이 필요합니다. 먼저, 이중 인터페이스를 지정해야 합니다. 즉,

요청자는 리턴된 메시지를 승인할 수 있는 인터페이스를 구현하는 서비스에 대한 리턴 주소로 전달해야 합니다. 이는 관계자 간 대화에서 상태를 관리해야 함을 의미합니다. 이 작업을 위한 다양한 방법에 대한 정보는 웹 서비스를 기반으로 하지 않는 웹 세션 디자인을 참조하십시오.

그러나 확장 정도는 제한적입니다. 높은 로드를 예상하는 서비스의 경우, 요청자를 청취하는 파트와 요청 자체를 지원하는 파트를 분리해야 합니다. 이는 이미 잘 알려진 패턴으로서, 메시지 대기열을 사용하여 서비스 모양과 서비스 구현을 분리합니다. 그림 12 의 다이어그램은 대기열을 사용하여 구현과 요청을 분리하기 위해 제공자를 구현하는 방법을 보여줍니다.

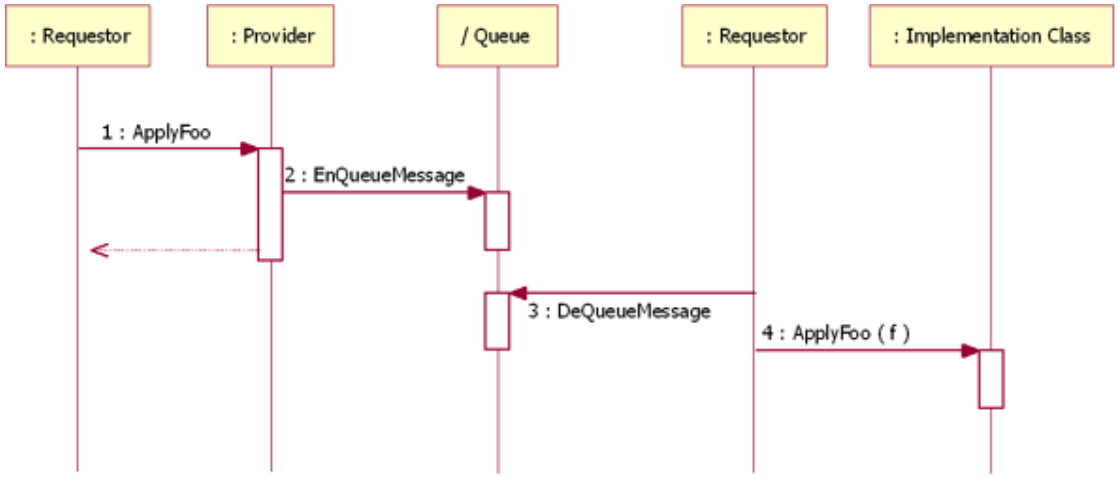

그림 12 – 대기열 지정 구현

이러한 패턴은 .NET 용 MSMQ 와 J2EE 용 JMS(Java Message Queue Service) 또는 메시지 구동 Bean 과 같이 해당 플랫폼에서 제공하는 서비스를 제공하여 .NET 및 J2EE 모두에서 쉽게 구현할 수 있습니다. 이러한 경우, 개발자에게는 보다 단순한 확장성 모델이 제공됩니다. 이 모델은 요청에 대한 동기화로 스레드 세트를 처리하지 않고 단지 대기열 리스너를 더 추가하여 여러 시스템의 경우에도 대기열에서 메시지를 선택할 수 있습니다.

정보 임대차 재검토

정보 임대차는 일반적으로 주택 또는 자동차와 같은 자산을 임대차하는 것보다는 도서관에서 책을 대여하는 관점에서 이해할 수 있습니다. 요청자는 암묵적으로, 서비스를 요청할 때마다 정보 사본을 요청하며 항상 특정 시간의 상태 스냅샷만 제공됩니다. 이는 명시적으로 이해하고 설명하지 않는 경우 문제점이 될 수 있습니다. 한 가지 전략은 제공자에게 정보와 함께 *만기* 시간을 제공하는 것입니다. 또는 요청자가 정보가 계속 유효한지 여부를 물어 잠재적으로 임대차 기간을 연장할 수 있는 임대차(도서관 서적과 유사) 권한이 있는 "티켓"을 받아, 서버에서 데이터를 다시 검색하지 않고 임대차를 재설정할 수 있습니다.

이제 이는 HTTP, SOAP 또는 전송 프로토콜 중 하나로 해결할 수 있는 기본 문제로 볼 수 있습니다. 브라우저 및 방화벽으로 페이지를 캐시할 수 있는 HTTP 캐싱 시맨틱을 재사용할 수 있지만 제공자가 제어할 수 있는 사항은 아니며 요청자는 HTTP 를 전송 수단으로 사용할 수 없을 수 있습니다. 한 가지 옵션은 문서 교환에 해당 지원을 빌드함으로써 요청자와 제공자 간 메시지가 클라이언트에 대한 임대차 정보를 인코딩하는 것입니다(그림 13 참조).

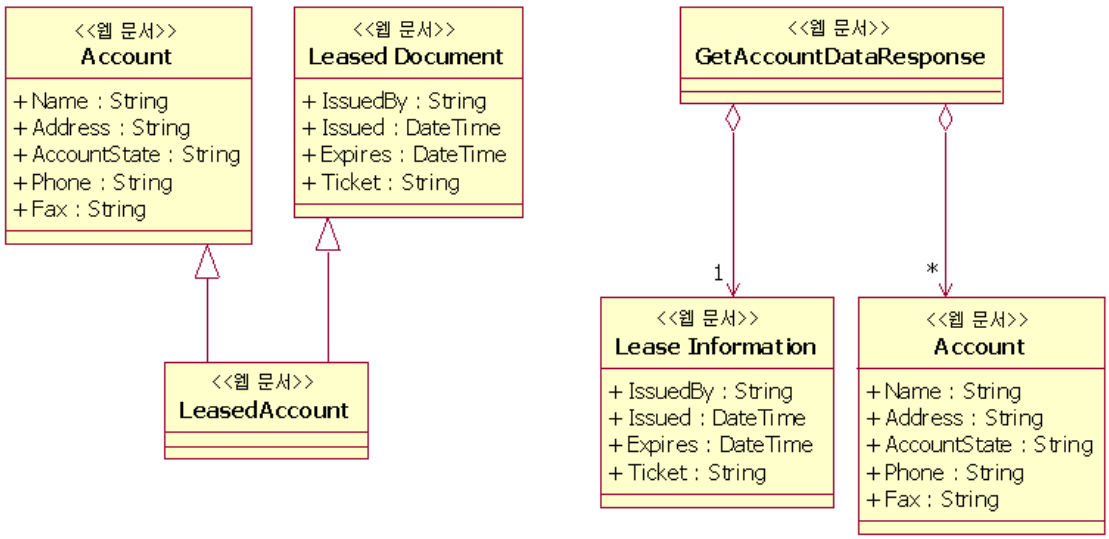

그림 13 – 두 가지 정보 임대차 구현

그림 13 은 정보 임대차 패턴의 두 가지 대체 구현을 보여줍니다. 첫 번째 구현은 계정 XML 문서를 특수 양식으로 특수화하기 위해 상속을 사용하는 구현입니다. 특수 양식은 계정뿐만 아니라 임대차된 문서이기도 하므로 추가 정보를 포함합니다. 두 번째 대체 구현에서는 임대차 정보를 계정과 함께 응답 메시지의 별도 파트로서 리턴합니다. 두 가지 접근 방식 모두 올바른 방식이지만 데이터는 구조적으로 다르며 선택사항은 일반적으로 스타일, 상속 대 집계 중 하나입니다.

결과 웹 서비스 디자인 모델

그림 14 는 웹 서비스 디자인 및 개발에 대한 특정 UML 프로파일을 사용하여 그림 7 의 일반 서비스 디자인을 모델링할 수 있는 방법을 보여줍니다. 프로파일은 매우 단순하여 "WebService" 및 "WebDocument"에 대한 두 가지 새 *스테레오타입* (기존 UML 언어에 대한 확장)만 소개합니다. 이미 UML 에 제공된 인터페이스 시맨틱을 재사용함으로써, WSDL 에 정의된 대로 공개된 서비스 측면을 쉽게 시각화할 수 있습니다.

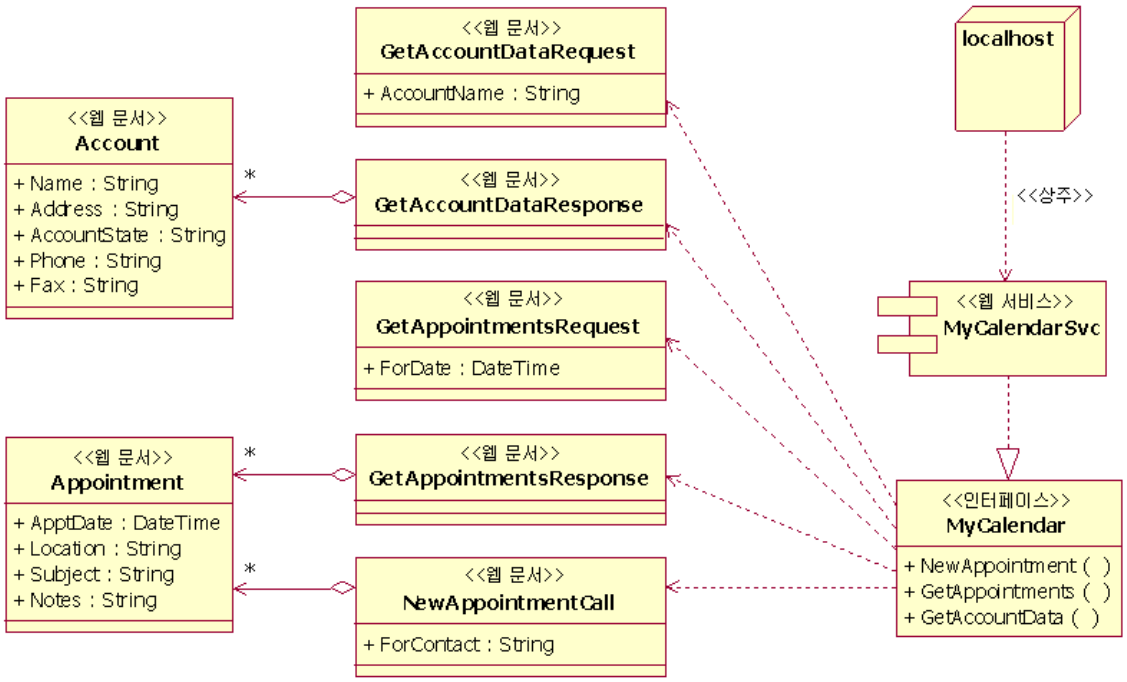

그림 14 – XML 웹 서비스 디자인

아래 표는 그림 14 의 프로파일 요소가 WSDL 의 MyCalendar 서비스에 대한 아티팩트와 관련되는 방식을 보여줍니다.

WSDL 아티팩트	UML 요소	주석
서비스	"WebService"	서비스는 하나 이상의 인터페이스를 실현하며 특정 위치에 상주하는 UML 컴포넌트로 표시됩니다. "reside" 관계는 실제 URL 위치 정보를 캡처합니다.
portType	인터페이스	각 portType 은 하나 이상의 서비스로 실현되는 UML 인터페이스로 표시됩니다. 실현(realization) 관계는 바인딩 정보를 캡처합니다.
메시지	"WebDocument"	각 메시지는 UML 클래스로 표시됩니다. 메시지 및 파트 구조를 모델링하려면 XML 스키마 및 UML 로부터의 맵핑이 필요합니다.
파트	속성 또는 연관 종료점	메시지의 각 파트는 "WebDocument"의 UML 속성 또는 다른 "WebDocument"에 대한 연관으로 표시될 수 있습니다.
주소 위치	노드	노드는 서비스가 상주하는 서버를 나타냅니다. 노드는 상주 서비스 세트를 식별하며 서비스는 여러 노드에 상주합니다.

이러한 방식으로 웹 서비스를 디자인하는 경우, 데이터 교환을 정의하는 문서와 서비스에서 지원하는 인터페이스를 모두 재사용합니다. 이는 엔터프라이즈 규모 솔루션을 디자인할 때 핵심 기능입니다. 계정 문서와 상호작용하는 모든 서비스는 동일한 문서 정의를 기반으로 하는 것이

바람직합니다. 또한 그림 2 의 공개되지 않은 운영 인터페이스(예: SystemsManagement)를 이 영역의 전문가가 정의할 수 있으며 솔루션에서 공통적으로 구현할 수 있습니다.

결론

일부에서는 웹 서비스를 소프트웨어 업계의 혁명으로 기대하지만 웹 서비스의 기능이 지나치게 과장되어 있다고 보는 견해도 있습니다. 일반적인 신기술의 예에서와 같이 진실은 그 중간 단계 정도로 이해할 수 있습니다. 지금까지의 경험으로는 서비스 지향 응용프로그램의 아키텍처상에 근본적인 차이가 있습니다. 그러나 기존 컴포넌트 기반 개발 기법은 서비스 구현에서도 계속 유효합니다.

그림 6 의 계정, 담당자, 약속 모델에서와 같이 단일 도메인 모델에서 컴포넌트 및 서비스 구현을 모두 도출할 수 있습니다. 다른 핵심 결론은 다음과 같습니다.

- 서비스 모델은 *기존* 컴포넌트 기반 모델에서 도출할 수 있습니다.
- 서비스 지향 모델로의 변환을 지원하기 위해 공통 패턴 및 디자인을 적용할 수 있습니다.

이러한 공통 패턴 및 디자인은 일반화, 자동화 단계를 거쳐 Rational XDE 와 같은 모델에서 모델로 및 모델에서 코드로 변환 도구를 사용하여 인스턴스화할 수 있습니다(아래 그림 참조).

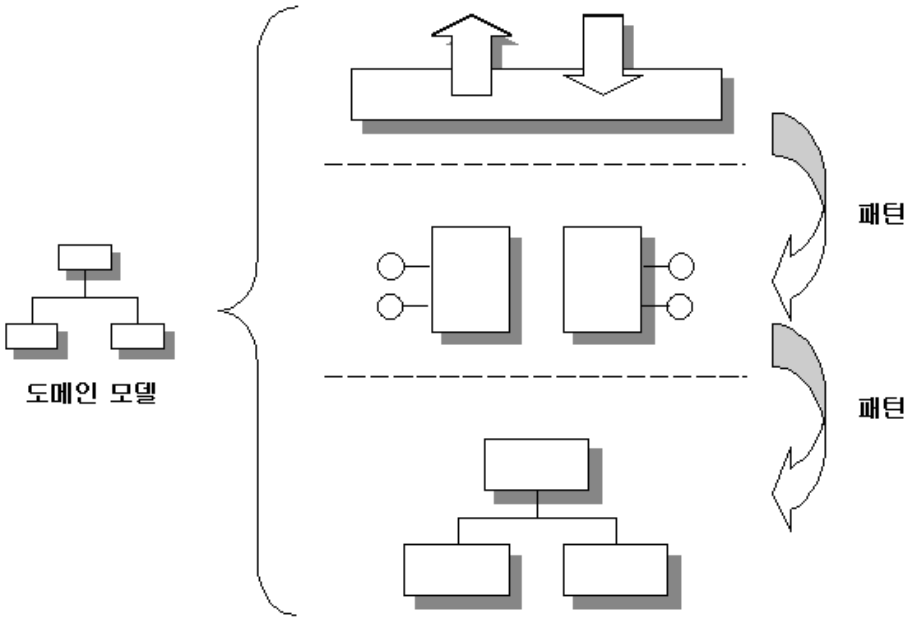

그림 15 – 서비스 구현 접근 방식

이 경로를 사용하는 경우 많은 이점이 있습니다. 즉, 개발자 오버헤드를 줄일 수 있으며 구현 예측 가능성이 향상됩니다. 또한 입증된 패턴을 재사용함으로써 품질이 향상됩니다. "서비스 지향 아키텍처(SOA) 사용"과 같은 우수 사례를 전문 지식에 추가하고 이러한 우수 사례를 Rational XDE 와 같은 자동 도구 세트에 규정함으로써 서비스 구현을 위한 올바른 선택과 절충 결정을 보다 빠르고 올바르게 수행할 수 있습니다.

참조 문헌

[1] 자세한 정보는 http://www.rational.com/uml/을 참조하십시오.
[2] Large-Scale, Component-Based Development by Alan W. Brown, Prentice Hall, 2000
[3] Public versus Published Interfaces by Martin Fowler, IEEE Software, March/April 2002 (Vol. 19, No. 2)
[4] W3C Web Services Architecture Requirements; http://www.w3.org/TR/2002/WD-wsa-reqs-20020429
[5] Web Services Security, Version 1,0; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-security.asp
[6] Web Services Referral Protocol, Draft; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-referral.asp
[7] XML Web Services Basics; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-referral.asp
[8] Rational Developer Network http://www.rational.net/