



special report 2

웹과 서비스, 그리고 아키텍처의 조화 SaaS로 가는 길

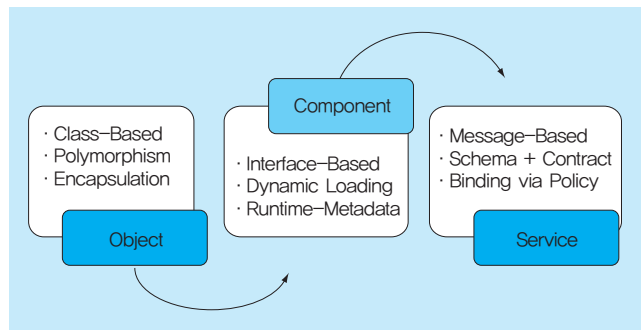
SaaS(Software as a Service)를 우리는 과연 어떠한 시선으로 바라봐야 할까? 이 글에서는 서비스(Service)의 기원과 벤더들 간의 숙명을 건 경쟁, 그리고 그 안에 담긴 정치적 배경을 살펴본다. 아울러 SaaS 애플리케이션을 개발하는 과정에서 고려해야 할 기술적인 측면과 아키텍처적인 측면도 함께 설명한다.

필자가 SaaS(Software as a Service)를 처음 접한 것은 2003년 IEEE Computer에서 발표한 'Turning Software into a Service'라는 논문에서였다. 대형 벤더들은 SaaS를 Delivery Platform 또는 ASP(Application Service Provider)의 진화 모델로 설명하고 있지만 SaaS는 Delivery Platform 이상의 많은 의미들을 포함하고 있다. 필자는 이 글에서 SaaS가 우리 컴퓨팅 환경에 미치는 영향과 벤더간의 숙명을 건 이해관계, 다른 기술과의 관련성, 그리고 SaaS를 구축하기 위해 개발자들이 준비해야 되는 것 등을 이야기하고자 한다. 지금부터 논문에 소개된 조언들과 필자의 부족한 지식을 합쳐 독자 여러분에게 그 내용을 전달해 본다.

서비스와 SOA 정의

'Software as a Service'를 말 그대로 번역하면 '서비스로서의 소프트웨어'이다. 웬지 모르지만 서비스(Service)라는 단어를 보니 SOA(Service Oriented Architecture)와 관련이 있을 것 같다. 따라서 여기서 먼저 '서비스가 무엇인가?'라는 질문에 초점을 맞춰 생각해 볼 필요가 있다.

'웹서비스(Web Service)의 아버지'이자 마이크로소프트의 통합 통신 프레임워크인 Windows Communication Foundation(WCF)의 설계자인 Don Box는 객체(Object)와 컴포넌트(Component)의 연장선에서 서비스를 설명하고 있다.



〈그림 1〉 재사용 패러다임의 진화 서비스

〈그림 1〉은 한 시대를 풍미한 객체(Object)와 컴포넌트(Component), 그리고 최근 종종 듣는 서비스가 나타나 있다. 다양한 측면에서 이들의 의미를 살펴볼 수 있겠지만 여기서는 재사용의 관점에서 바라본다.

1980년대부터 1990년대 중반의 객체지향 시대에 가장 큰 화두는 역시 재사용이었다. 상속과 조합이라는 기법을 어떻게 적절히 조합해 좋은 객체들을 만들 수 있을지를 많은 개발자들과 학자들이 고민했다. 하지만 객체의 재사용 범위를 생각해 보자. 객체의 재사용 범위는 언어에 종속적이다. C++로 만든 객체는 C++

손영수 indigoguru@hotmail.com | 데브피아 Architecture 사립과 Microsoft MVP로 활동 중이며, 소프트웨어공학 스터디인 Devpia Eva의 리더이다. 여전히 실력이 부족하다고 자평하면서도 자신의 지식을 남과 나눌 때는 누구보다 '부자'라는 자부심을 가지고 온라인과 오프라인에서 왕성한 활동을 펼치고 있다. Pattern 전도사가 되기 위해 노력중이고, 이와 함께 PLOP과 같은 Pattern 학회를 국내에 만들기 위해 힘 쏟고 있다.

에서 사용할 수 있고, 자바에서 만든 객체는 자바에서만 활용할 수 있다. 따라서 이와 같이 언어 종속적인 재사용성을 극복하기 위해 1990년대 초부터 2000년대 초까지 재사용을 위한 모듈로 컴포넌트가 대두되었다.

컴포넌트는 언어 종속적인 재사용성을 탈피하기 위해 IDL (Interface Description Language) 같은 메타데이터 (Metadata)를 활용하므로 다양한 언어에서 사용할 수 있다. 실제로 마이크로소프트 플랫폼에서는 COM(Component Object Model)으로 만들지만 하면 Visual Basic이나 Visual C++과 같은 언어에 상관없이 사용할 수 있었다. 그러나 컴포넌트 역시 재사용 범위가 플랫폼에 종속적이라는 문제점이 제기되었다. COM이라는 기술은 마이크로소프트 플랫폼에만 적용될 수 있고, 자바 역시 JVM이 설치된 자바 플랫폼에서만 재사용할 수 있었다. 결국 많은 개발자와 학자들은 언어와 플랫폼에 종속되지 않는 재사용 가능한 모듈을 꿈꿔왔다.

이것이 바로 서비스이다. 다시 말해 서비스는 '언어와 플랫폼에 독립적인 재사용 가능한 모듈'이라고 정의할 수 있다. 이 얼마나 명쾌하고 깔끔한 정의인가?

서비스의 정의를 내린 김에 잠시 SOA의 의미를 파악해 보자. SOA(Service Oriented Architecture)를 말 그대로 정의하면 '서비스의 세상을 실현시키기 위한 아키텍처'이다. 서비스들만 조합해 하나의 애플리케이션을 만드는 서비스 기반의 플랫폼을 구축하는 것이 SOA의 궁극적인 목표인데, 그러기 위해서는 보안, 트랜잭션, 통합인증 등 여러 가지 복잡한 문제들을 해결하기 위한 아키텍처를 제공해야 된다. 현재 가장 큰 힘을 얻고 있는 기술이 메타데이터 기반의 표준을 꿈꾸는 WS-*이며, 그 이외에도 Semantic Web Service, OpenSOA의 SCA(Service Component Architecture) 등의 다양한 기술들이 경합하고 있다.

메타데이터 기반의 통합, 어떤 의미인가?

기존의 SOA 세상을 실현하기 위해 많은 기술들이 존재했다. 그 대표주자로 CORBA를 예로 들 수 있다. OMG에서는 CORBA의 기능을 추가하기 위해 함수명과 파라미터까지 다 정의해야 하므로, 각 플랫폼의 특성과 장점을 포기하고 랩핑(Wrapping)하는 등의 부가적인 작업을 요구하게 된다.

하지만 이런 불편함을 간파한 XML 웹서비스의 창시자 Don Box는 주고받는 데이터 포맷인 SOAP와 몇 가지 규약(WSDL, WS-Policy 등)들만을 준수하고, XML 웹서비스의 구현은 여러 벤더들이 각 플랫폼의 특성을 살려 개발할 수 있도록 유도했다. 다시 말해 CORBA와 달리 각 플랫폼을 존중하는 철학이 있었으므로 널리 웹서비스가 이용될 수 있었다.

SaaS의 탄생 배경

SaaS가 요즘 컴퓨팅 환경에 대두된 이유는 무엇일까? 그 이유를 쉽게 이해하기 위해서는 벤더간에 얽혀 있는 플랫폼 전쟁 이야기를 하지 않을 수 없다. 2000년 초에 많은 대형 IT 벤더들은 차세대 성장 동력으로 SOA를 선택했다. 2000년 초만 해도 SOA의 정의는 무척 간단했다. 서비스를 널리 배포하길 원하는 공급자(Publisher)가 UDDI와 같은 중계자(Broker)에 자신의 서비스를 등록하면, 특정 서비스에 관심이 있는 고객(Consumer)은 중계자를 통해 자신이 원하는 서비스를 찾은 후 직접 공급자의 서비스를 이용한다는 것이다. 새로운 애플리케이션을 만들기 위해 모든 기능을 구현하는 것보다는 존재하는 서비스들을 조합해 손쉽게 만들 수 있는 이상적인 세상을 꿈꿔 왔다.

웹서비스 기반 SOA 역시 이러한 아이디어를 차용해 UDDI라는 중계자를 만들고 성공을 꿈꿨다. 그러나 대형 벤더들이 꿈꾸던 SOA 세상은 위기를 맞게 된다. UDDI가 제공하는 키워드 기반의 검색 때문이다. 과연 실세계의 광범위한 도메인에서 실제 내 입맛에 딱 맞는 서비스를 찾는다는 것이 가능할까? 만약 서울에서 중국집에 자장면을 주문했는데 중국 북경에 있는 중국집에서 자장면을 배송하는 서비스를 실행했다면 여러분은 불어터진 자장면을 받게 될 것이다. 결국 벤더들의 꿈인 서비스 기반의 가상 OS를 만드는 작업은 빈약한 키워드 기반의 검색으로 인해 실패하게 된다.

하지만 새로운 성장 동력이 필요한 벤더들은 SOA를 재해석하게 된다. 기존의 메인프레임과 엔터프라이즈 환경에 강세를 보이는 벤더들(IBM, 썬, 오라클 등)은 Business Process Management System과 Enterprise Service Bus를 기반으로 이질적인 컴퓨팅 환경을 극복하고 변화의 유연함과 시스템 안정성을 보장하는 형태인 B2B 위주의 SOA 전략을 추진해 왔다. 반면, 마이크로소프트처럼 컨슈머(Consumer) 시장에 강세를 보이는 회사들은 B2C 기반의 SOA 전략(마이크로소프트의 경우 Software + Service)에 무게를 실어 SOA 시장의 혈전이 펼쳐졌다.

이렇게 기존의 대형 벤더들이 레드오션인 SOA 시장을 장악하기 위해 서로 혈전을 벌이고 있을 때, 예상치 못한 블루오션인 웹 플랫폼에서 비약적으로 성장한 회사들이 속속 등장했다. 인터넷의 길목이라 할 수 있는 검색시장을 장악한 구글과 Social Networking에 강세를 보이는 몇몇 회사들은 기존 IT 벤더들이 주력하고 있는 사업에 직, 간접적인 영향을 미침으로써 대결 구도를 이뤄냈다. 한 예로 구글은 웹 플랫폼을 기반으로 이메일, 오피스 솔루션, 모바일 플랫폼, 클라우드 컴퓨팅(Cloud Computing) 등으로 전통적인 IT 벤더들과 치열하게 경쟁 중이다.

기존 IT 벤더들이 강세를 보이는 SOA 플랫폼과 신흥세력인

웹 플랫폼간의 충돌이 발생하면서 다소 복잡하고 흥미로운 상황이 연출된다. SOA가 성장하기 위해서는 웹 플랫폼의 접근성이 필요하고 웹 플랫폼의 활성화를 위해서는 SOA에서 제공하는 상호운영성의 산물인 매쉬업(Mash-up)을 필요로 한다. 많은 이들이 이 둘이 상생할 것으로 전망하지만, 엄격히 말해 웹2.0이 주가 되는 상황에서 단지 매쉬업의 수단으로 SOA를 사용하는 형태로 플랫폼이 진화하느냐? 아니면 SOA가 주가 된 상황에서 웹2.0의 접근성을 얻는 형태로 진화하느냐에 따라 벤더들의 운명이 갈릴 것이라고 해도 결코 지나치진 않다.

이러한 상황에서 SOA 플랫폼에 강세를 둔 대형 IT 벤더들은 어떻게 대응해야 할까? 그 해답이 바로 이 글의 주제인 SaaS (Software as a Service)이다.

SaaS의 정의와 분류

서비스 전문가 Mark Turner는 『IEEE Computer Journal』을 통해 SaaS를 다음과 같이 정의했다.

"Software deployed as a hosted service and accessed over the Internet(호스팅된 서비스로서 운영되고 인터넷을 통해 접근할 수 있는 소프트웨어)."

아마 가장 좋은 예는 Windows Live Service(<http://get.live.com/wl/all>)와 구글이 제공하는 서비스(<http://www.google.com/intl/ko/options/index.html>)일 것이다.

SaaS는 기존 패키지 소프트웨어가 중요하게 여겼던 소유와 사용의 관점을 분리함으로써, 전통적인 패키징 기반의 소프트웨어와는 다른 형태의 라이선싱 모델을 구축할 수 있다.

예를 들면 서비스를 호출할 때마다 돈을 받는 트랜잭션 기반의 가격정책과 일정한 기간을 두고 소프트웨어 사용 권한을 주는 Subscription 정책, 그리고 서비스는 무료이지만 광고 기반의 수익을 벌어들이는 방법 등이 존재한다. 이것은 서비스 제공자 입장에서는 불법 복제를 원천적으로 방지할 뿐만 아니라 개별 고객의 요구사항에 따라 맞춤 서비스를 제공할 수 있는 장점을 가지고 있어 잠재적인 고객들을 끌어낼 수 있는 장점을 지닌다.

따라서 소비자의 입장에서는 합리적인 구매를 할 수 있다. 예를 들어 사진 편집에 필요한 포토샵의 일부 기능을 이용하기 위해 소프트웨어 전체를 구입해야 하는 상황이 패키지 소프트웨어 환경에서는 흔했지만, SaaS에서는 일부 기능만을 사용할 수 있거나 1년에 단지 몇 차례만 쓰는 사람들을 위해 저렴한 비용으로 소프트웨어가 제공될 수 있다(물론 불법 복제 천국인 우리나라에서는 그리 합리적이지 않을 수도 있다). 전문가들은 향후 SaaS 시장이 크게 세 가지 형태로 분류될 것으로 전망한다.

Enterprise LOB(Line of Business) SaaS

기존의 B2B 영역에서 ASP 시장을 대체 및 확장할 것으로 보이며 대형 IT 벤더들의 혈전이 예상된다. 이미 CRM 기반의 강자인 세일즈포스닷컴(Sales Force.com)이 버티고 있는 분야로, 세일즈포스닷컴은 다양한 기능의 소프트웨어를 사고 팔수 있는 앱익스체인지(AppExchange)를 제공한데 이어, Apex라는 새로운 개발 툴을 제시함으로써 SaaS 플랫폼 회사로 변신을 시도하고 있다. 이에 마이크로소프트 역시 기존 업계 평균 가격의 절반 수준으로 Dynamic CRM 서비스를 제공함으로써 맞대응하고 있다.

웹2.0을 기반으로 한 일반 사용자들을 위한 SaaS

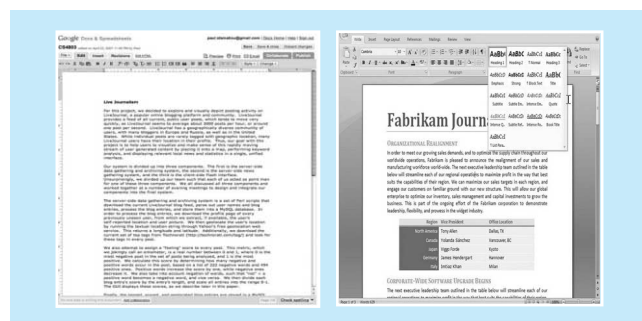
실제 우리가 관심을 가져야 하는 부분은 진입 장벽이 낮은 일반 사용자(B2C)를 위한 웹2.0 기반의 SaaS이다. 웹2.0이 제공하는 접근성을 기반으로 마치 PC에서 수많은 유틸리티(Utility)를 쏟아낸 것처럼 인터넷에 수많은 SaaS 애플리케이션이 쏟아져 나올 것이다. 앞에서 언급한 Windows Live Service와 구글이 제공하는 서비스가 여기에 해당된다.

전통적인 소프트웨어들과 서비스의 조합

마이크로소프트가 주도적으로 밀고 있는 'Software + Service' 전략은 웹2.0 기반의 SaaS와 더불어 기존 소프트웨어들과 서비스를 결합하는 방식에 핵심을 두고 있다. 따라서 일반적으로 많이 사용하는 오피스 제품군 등에 CRM과 같은 서비스들을 결합함으로써 기존 사용자의 경험을 존중하고, 플러그인(Plug-in) 형태로 추가 기능을 제공할 수 있다.

UX와 SaaS의 조합

마이크로소프트, 썬과 같은 회사들은 구글의 성장이 그리 달갑지는 않을 것이다. 우수한 성능을 지닌 구글의 검색엔진은 그 사용자 경험과 선호도가 이미 패터화되어 있어 점유율을 빼앗기가 힘들지만, 웹 플랫폼을 기반으로 이뤄질 구글의 SaaS 독주는 막



(그림 2) Google Docs와 Microsoft Word

으려 할 것이다. 그럼 어떤 방법이 있을까? 아마도 UX(사용자의 경험을 극대화하는)와 맞춤 서비스가 해답이 될 것이다.

혹 워드프로세서가 필요하다면 Microsoft Word나 Google Docs를 사용할 수 있다. 여러분은 어떠한 것을 사용하고 있는가? 그건 상황에 따라 결정된다. 여러분의 PC에 Microsoft Word가 설치되어 있다면 당연히 Word를 사용할 것이다. Google Docs보다 다양한 기능을 제공하고, 이미 여러 번 사용한 경험이 있으므로 기능도 익숙하다. 하지만 여러분이 갖은 출장으로 이동이 잦아 다양한 PC를 사용해야 하는 경우라면 모든 컴퓨터에 Microsoft Word가 설치되어 있는지를 걱정해야 할 것이고, USB 메모리와 같은 저장 매체도 늘 가지고 다녀야 할 것이다. 따라서 이런 번거로움 때문에 기능성보다는 어디서나 쉽게 접근할 수 있는 Google Docs를 선호하게 될 것이다. 이것은 접근성과 기능성의 선택이라고 할 수 있다.

이러한 것을 간파한 기존 IT 벤더들은 SOA 시장의 성장을 그대로 이어가면서 구글의 웹 플랫폼 독주를 견제하기 위해 RIA 기반의 UX 기술과 함께 더 세련된 SaaS를 누구나 만들 수 있는 환경을 제공하고 있다. 이것은 웹 플랫폼 개방성을 이용한 것인데, 한 가지 예로 오픈마루에서 만든 스프링노트(<http://www.springnote.com>)는 Google Docs보다 훨씬 다양하고 강력한 기능을 제공한다. 이처럼 웹의 개방성으로 인해 누구나 쉽게 새로운 신규 서비스를 만들어 진입할 수 있다. 기존 벤더들이 RIA를 쉽게 제공할 수 있는 플랫폼을 제공함으로써 누구나 SaaS 시장에 쉽게 들어올 수 있도록 진입 장벽을 낮춘다면 구글의 검색 기술은 따라잡지 못하더라도 SaaS 시장의 확장을 막을 수는 있다. 그렇기 때문에 기존 IT 벤더들은 UX 플랫폼(마이크로소프트의 WPF와 실버라이트, 어도비의 AIR와 Flex3, 썬의 JavaFX)을 경쟁적으로 제공하고 있다.

그럼 지금부터는 기술적인 관점에서 SaaS 애플리케이션을 만들기 위해 어떠한 점들을 고려해야 할지 언급한다.

성공적인 SaaS 시스템의 구현

SaaS를 기존의 ASP와 많이 비교하게 되는데 그 중 가장 핵심이 되는 키워드가 Multi Tenant Architecture이다.

기존의 ASP 업체는 마치 메모리업체인 샌디스크가 각 고객의 요구 사항에 맞춰 다양한 형태의 메모리 카드를 만들듯이 각각의 고객에 특화된 서비스들을 모두 만들어야 했다. 하지만 <그림 3>과 같은 샌디스크의 메모리 제품은 하나의 모델 형태로 다양한 고객의 요구사항을 수용할 수 있게 했다. 이처럼 하나의 인스턴스에 여러 개의 기능을 제공함으로써 제품 생산 라인이 하나로 줄었고, 훗날 특정 메모리 카드가 사라진다고 하더라도 사업에

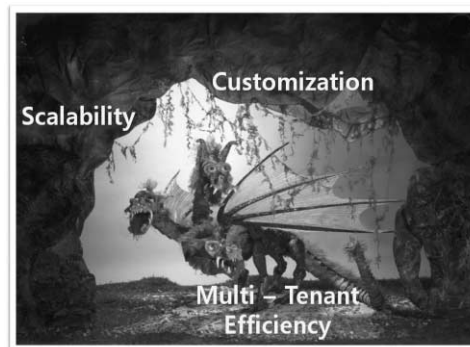


<그림 3> ASP와 SaaS의 차이

별다른 지장이 없을 것이다. 바로 이것이 Multi-Tenant의 장점이다. 하나의 인스턴스(Single Instance)만으로 다양한 고객의 요구사항(Multi-Tenant)을 만족시킴으로써 안정적인 수익을 얻을 수 있는 것이다.

하지만 시스템을 구축해야 하는 우리는 성공적인 SaaS 애플리케이션을 만들기 위해 <그림 4>와 같은 골치 아픈 세 개의 머리를 가진 괴물(?)을 만나야 한다.

- Multi-Tenant Efficiency(Multi Tenant 구조를 고려한 설계)
- Customization(사용자에게 최적화된 서비스 제공)
- Scalability(사용자 증가를 고려한 확장 가능한 구조 고려)



<그림 4> SaaS 시스템을 구축할 때 만나는 머리가 세 개인 드래곤

SaaS를 제공할 때 얻을 수 있는 또 하나의 장점은 long tail 원칙에 의거해 잠재적인 고객을 끌어낼 수 있다는 점이다. 기존의 ERP, MIS, 사무 자동화와 같은 소프트웨어들은 대기업처럼 소프트웨어에 많은 돈을 지출할 수 있는 회사들만 구축할 수 있었다. 하지만 SaaS 서비스의 활성화로 인해 시스템 운영비용, 구축비용이 감소한다면 진입 장벽이 낮아질 뿐만 아니라 Multi-Tenant를 지원하기 때문에 중소기업과 같은 곳에서도 충분히 회사 시스템에 특화된 소프트웨어를 사용할 수 있으니 새로운 시장이 창출될 수 있다.

하지만 고객들의 각기 특화된 요구사항을 잘 반영할 수 있는 도메인에 대한 전문적인 지식과 고객의 효율성을 증진하기 위한



최적의 인프라 설계라는 두 마리 토끼를 잡아야 하는 어려움이 따른다. 그리고 웹이란 특성상 훗날 우리가 만든 SaaS 애플리케이션의 사용자가 갑자기 늘어나는 것을 가정해 Google File System(GFS)과 같이 쉽게 확장할 수 있는 구조 역시 고려해야만 한다. SaaS의 장밋빛 미래 뒷면에는 날카로운 가시가 함께 숨어있는 것이다. 그럼 이런 문제들은 어떻게 해결해야 할까? 필자 역시 SaaS 애플리케이션을 구축한 실제 경험은 없지만 이론적인 차원에서 큰 해답을 제시해 본다.

SaaS 구축의 핵심, 메타데이터 지향적인 설계

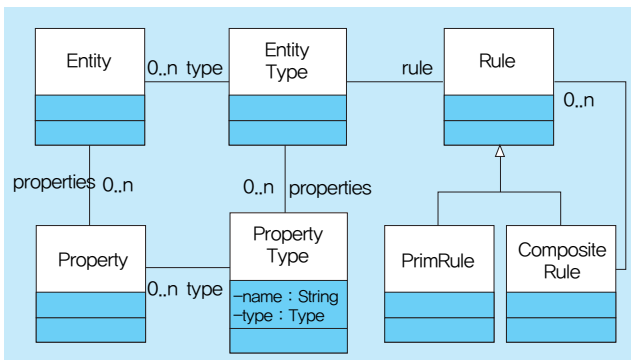
고객의 다양한 요구사항을 쉽게 반영하고 변화에 유연하게 대처하기 위해서는 어떻게 해야 할까? 사용자에게 개별적으로 최적화된 서비스를 제공하기 위해 필요한 것은 무엇인가? 무엇보다 많은 일들이 다이내믹하게(Runtime시에) 일어나는 것에 대응할 수 있어야 하고, 개별 사용자의 요구사항을 정확하게 파악한 후 적절히 표시해 줄 수 있어야 한다.

이것에 대한 현실적인 대답은 메타데이터 지향적인 설계이다. 흔히 사용하는 XML 설정 파일과 같은 메타데이터들을 활용해 런타임시에 변화에 적용할 수 있는 AOM(Adaptive Object Model), Reflection, Component Configurator와 같은 메타데이터 아키텍처에 대한 지식을 습득하고 여러분의 경험들을 적절하게 조합할 수 있는 능력을 갖춰야 한다.

AOM 아키텍처

소프트웨어공학 컨퍼런스인 OOPSLA에서 2001년 발표한 모델이다. 메타데이터로 객체, 속성(Attribute), 관계, 행위, Business Rule 등을 표현함으로써 Runtime시에 변화하는 요구사항들을 시스템의 중지 없이 동적으로 적용할 수 있는 시스템을 구축하는 것이다(마소에 연재가 진행 중인 장선진 필자의 'AOM 이용한 SaaS 기반 애플리케이션 구축' 참조).

AOM의 핵심은 기존의 객체 지향적인 구조를 어떻게 메타데



(그림 5) AOM의 대표 구조-Rule을 포함한 Type Square

이터 지향적으로 잘 표현하고 모델링하느냐에 있다(이러한 문제는 대부분 ORDB를 통해 많이 연구되어 왔다).

먼저 <그림 5>에서 EntityType과 Entity를 보자. 객체지향의 전통적인 상속관계를 새롭게 표현한 모델로서, 최상위의 부모 클래스가 있고 이것을 상속받은 수많은 SubClass가 있는 것을 Type Object라는 형태로 표현한 것이다. 이것은 새로운 클래스들이 부모 클래스의 인스턴스로서 런타임시에 계속 추가될 수 있음을 의미한다. 만약 병원 환자들을 관리하는 프로그램을 구현한다면 객체지향에서는 부모 클래스인 환자 클래스와 수많은 자식 클래스인 산부인과 환자, 외과 환자, 내과 환자 등으로 세분화할 수 있을 것이다. 이러한 관계를 환자종류라는 EntityType과 구체적인 Entity인 환자들(산부인과, 외과, 내과)로 표현할 수 있다.

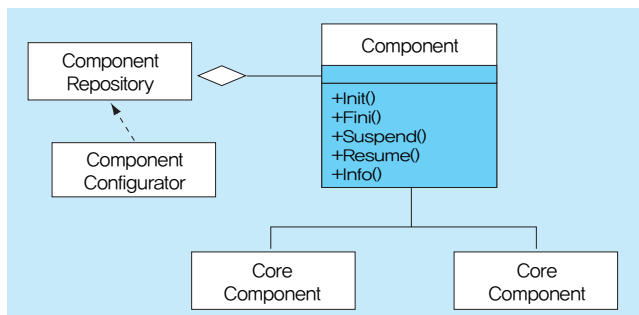
그럼 객체의 행위적인 부분들을 어떻게 표현해야 할까? 이것들은 <그림 5>의 오른쪽에 있는 Rule Object를 이용해 해결한다. RuleObject의 좋은 예는 Micro-workflow이다. 객체의 행위는 기본적인 룰(Primitive Rule)과 반복, 분기, 순차와 같은 룰들의 조합으로 쉽게 표현할 수 있다. 이 룰들은 현재의 BPMS로 대처할 수 있으며 이것을 통해 변화에 유연하게 고객의 요구를 반영할 수 있다.

SaaS의 관리와 확장을 위해 일일이 메타데이터를 수작업으로 생성하는 메타데이터 파일의 '지옥'에서 벗어나려면 Type Square와 Rule들을 쉽게 정의하고 생성 및 관리할 수 있는 유저 인터페이스(User Interface)를 제공해야 한다. 현재 많은 벤더들이 SaaS를 강조하고 있지만 이러한 것들을 충분히 지원할 수 있는 시스템을 제공하지 못하는 상황이다.

Component Configurator Pattern

아키텍처 패턴(Architectural Pattern)을 다룬 『POSA1』 서적을 통해 알려진 패턴으로, 런타임시 사용하는 컴포넌트들을 제어하고 시스템 중지 없이 컴포넌트를 추가, 제거, 교체할 수 있다.

<리스트 1>과 <리스트 2>는 Component Configurator 패턴의 간단한 샘플이다. 설정 파일(<리스트 1>)의 Protocol이 HTTP면



(그림 6) Component Configurator

그 정보를 읽어 HTTP 프로토콜로 메시지를 전송하고 FTP 프로토콜로 설정되어 있다면 FTP 프로토콜로 메시지를 전송할 수 있다. 이렇게 함으로써 런타임 시 객체의 속성을 정의하면 변화에 유연하게 대처할 수 있다.

그러나 새로운 프로토콜이나 로깅 정책이 추가되어야 한다면 <리스트 2>의 소스 코드도 바뀔 수밖에 없다. 이러한 문제를 해결하기 위해 Component Configurator 패턴은 Reflection 패턴과 병행해 사용해야만 한다. Reflection은 Runtime시에 객체를 생성하는 패턴으로, 이미 닷넷(.NET)과 자바 플랫폼에서 지원하고 있다. 아울러 Observer 패턴과 조화롭게 활용한다면 시스템을 증진하지 않고도 Runtime시에 해당 모듈을 교체할 수 있다.

<리스트 1> Component Configurator 설정 파일

```
<xml version ="1.0">
<Components>
  <!? Protocol 정보 선택 -->
  <Protocol>
    HTTP
  </Protocol>

  <!? Log 정보 선택 -->
  <Log>
    XML
  </Log>

  <!? 보안 알고리즘 선택 -->
  <SecuAlgorithm>
    DES
  </SecuAlgorithm>
</Components>
```

<리스트 2> Component Configurator를 적용한 샘플 코드

```
static void main()
{
  CMessage *pMessage = new CMessage();
  if (GetComponentInfo("Protocol") == HTTP)
    pMessage->Protocol = HTTP;
  else
    pMessage->Protocol = FTP;

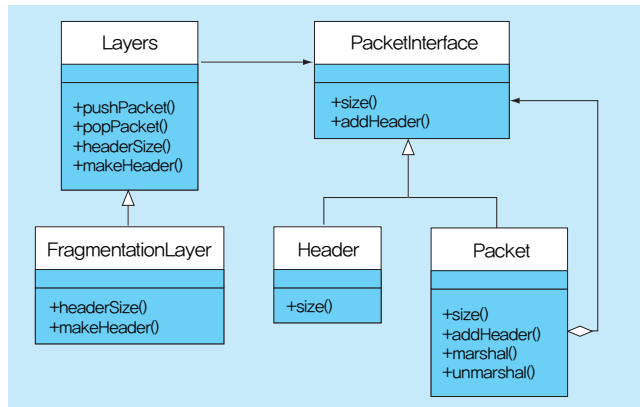
  if (GetComponentInfo("Log") == XML)
    pMessage->Log = XML;
  else
    pMessage->Log = TEXT;

  pMessage->Send("Hello");
  delete pMessage;
}
```

Composite Message Pattern

메시지를 마샬링(marshalling)/언마샬링(unmarshalling)하

는 패턴으로 다양한 프로토콜을 지원하고, 물과 메시지들을 런타임시에 조합할 수 있으므로 메타데이터 지향적인 프로토콜 설계 시 유용한 패턴이다.



(그림 7) Composite Message

송신자(Sender)가 보낸 메시지는 <그림 7>처럼 FragmentationLayer를 하나씩 거쳐 가면서 Header와 Packet 정보를 추가함으로써 최종 메시지를 구성하게 된다. 간략한 예로 최상위 레이어(Layer)에서는 특정 메소드를 호출했고, 파라미터로 어떠한 값이 들어갔다는 메시지가 구성된다. 다음 하위 레이어에서는 메타데이터에 기술되어 있는 암호화 전략(DES, RSA)을 선택해 데이터를 암호화한다. 또 하위 레이어에서는 이 메시지의 트랜잭션을 보장하기 위해 TransactionID를 달게 되고, 메타데이터에 기술된 형태로 하나의 메시지를 구성하게 된다. 이것은 전형적인 레이어 기반의 파이프(Pipe) 구조이다.

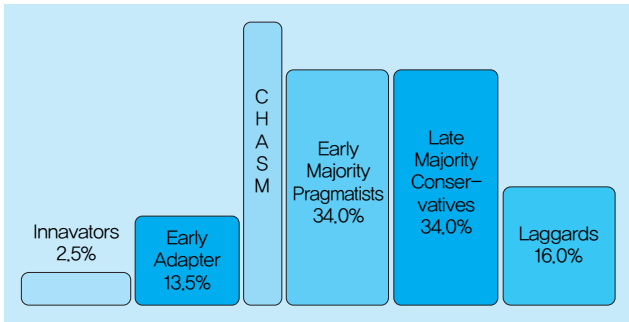
반대로 수신자(Receiver)에서는 설정 파일의 정보를 보고 메시지를 분해하게 된다. 이것은 전형적인 레이어와 필터(Filter)의 조합이다. 최종 분해된 메시지를 이용해 서버의 핸들러를 실행함으로써 결과를 반환하게 된다.

이러한 기술들을 잘 활용하면 고객의 요구사항을 기반으로 적합한 아키텍처를 런타임시에 선택할 수 있다. 실제 구축 사례인 LitwareHR과 AOM의 구현 사례인 의료 프레임워크(IDPH)를 참고하길 바란다.

사용자를 고려한 Customization

추후 언급할 SaaS의 장점인 특정 회사나 개인에게 제공하는 맞춤형 서비스를 제공할 수 있는 구조(Customization)도 가지고 있다. 먼저 고객의 유형을 파악해 보자.

<그림 8>은 일반인들의 제품 구매 유형을 나타낸 것이다. 이노베이터(Innovator)와 얼리어댑터(Early Adapter)로 대변되는 젊은 층의 경우는 새로운 신제품과 기능에 열광하는 사람들이다.



(그림 8) 일반인들의 다양한 구매 유형

집에 MP3 플레이어가 있더라도 마음에 드는 제품이 나오면 언제든 구입하는 고객들인 셈이다. 반면에 오른쪽의 중장년층은 안정성과 가격 등을 고려해 합리적인 구매를 하는 유형들이다. 문제는 이 두 계층 간에 큰 간극(Chasm)이 존재한다는 것이다. 그래서 제품을 만들 때 어느 쪽을 타깃으로 해서 만들 것인지 초기에 결정한 후 설계해야 한다. 하나의 제품이 모든 고객을 만족시킬 수 있다면 좋겠지만 가장 좋은 해답은 개별 고객들에게 특화된 서비스를 제공하는 것이다. 이러한 개별 맞춤화의 중요성을 파악한 마이크로소프트는 Multi User Interface를 쉽게 지원하기 위해 메타데이터 기반의 XAML(WPF)을 만들었다.

SaaS 애플리케이션 성능 향상을 위한 고려사항

마이크로소프트의 SaaS 아키텍트인 Gianpaolo는 성능 향상을 위해 네 가지 팩터를 고려하라고 조언한다.

- Stateless
- 비동기 I/O
- 리소스 풀링
- 동시 접속자 수(Throughput)의 최대화 방법

Stateless

HTTP는 상태 정보를 지속적으로 유지할 수 없으므로, 웹기반의 사용자는 오랜 시간 동안 세션과 작업 진행 중인 상태를 유지할 수 없다. 그러므로 Stateless한 형태를 고려해 성능을 극대화하는 방법을 연구해야 한다. 하지만 현재는 RIA 기술의 발전으로 이러한 제약은 거의 사라졌다고 볼 수 있다.

비동기 I/O(Asynchronous I/O)

비동기 I/O 기술인 닷넷의 APM(Asynchronous Programming Model)과 자바의 NIO를 활용해 기존 동기 모델에 비해 비약적인 성능 향상을 꾀할 수 있다. 하지만 비동기 모델이 모든 경우에 좋은 성능을 보장하지는 못한다. 모든 상황에 최적화된 웹

서버는 존재하지 않음을 파악한 『POSA2』의 저자 Douglas Schmidt's는 주어진 상황에 따라 적합한 전략을 선택할 수 있는 웹서버(JAWS)를 설계, 구축하고 그 실험 결과를 공유했다. 비동기는 작은 사이즈의 데이터(통상 6KB 이하)를 주고받을 경우 오히려 동기보다 성능이 낮아진다는 게 결과의 요점이다. 그렇기 때문에 닷넷 플랫폼은 6KB 미만의 데이터를 비동기 함수로 주고받을 경우 내부에서 동기적으로 처리하도록 했다. 비동기는 큰 사이즈의 I/O 작업을 처리할수록 유리하다는 점을 기억하자.

리소스 풀링(Resource Pooling)

클라이언트가 특정 서비스를 요청할 때마다 서버가 리소스를 생성해 결과를 주는 것보다 미리 자주 사용할 만한 리소스들을 풀링(Pooling)함으로써 생성시 드는 비용을 줄일 수 있는 이점이 있다. 하지만 너무 많은 리소스를 미리 풀링해 놓는 것 역시 과부하이므로 적절한 양만큼 Pooling하는 게 중요하다. 효율적인 풀링을 위해서는 WatchDog의 사용이 필수적이다. WatchDog은 Agent의 성격이 강한 모니터로서, 리소스가 특정 개수만큼 줄어들면 다시 풀의 리소스를 증가시킬 수 있고, 시스템의 리소스를 더 생산할 수 없는 경우 동시 접속자 수를 제한하거나 Message Queue에 쌓아 놓은 후에 처리할 수도 있다.

동시 접속자(Throughput) 수의 최대화

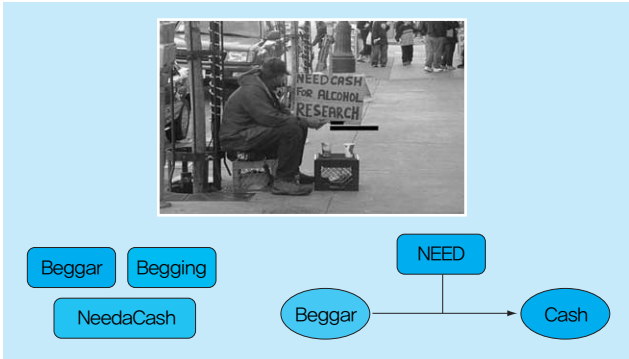
갑자기 증가하는 사용자의 요청을 처리하기 위해 Google File System과 같은 확장 가능한 구조를 갖추고 불필요한 로그, 락킹(Locking)을 제거하는 것이 중요하다. 그리고 부하를 적절히 나눌 수 있는 Load Balancer와 같은 하드웨어 시스템의 도입을 검토할 필요가 있다.

미래를 이끌어 갈 기술들, 그리고 SaaS

그럼 미래를 이끌어갈 어떤 기술들이 존재하고 서로 간에 어떻게 영향을 미쳐 진화하게 될 것인가? 많은 학자들은 향후 10년간 미래를 이끌 차세대 동력으로 시맨틱 웹(Semantic Web), 시맨틱 웹서비스, 그리드 컴퓨팅(Grid Computing)을 선정했고 현재 많은 연구를 진행 중이다.

시맨틱 웹

웹2.0이 사용자 위주의 웹 환경을 만들었다면 웹3.0(Semantic Web)은 똑똑한 웹을 만드는 것이다. 지금의 웹은 아무런 의미 없는 링크로 연결되어 있다. 하지만 '웹의 아버지'인 T. Berners Lee가 초창기 설계한 형태의 웹은 그렇지 않았다. 잘 정의된 의미를 웹에 부여함으로써 똑똑한 웹을 만들기를 바랐다.



〈그림 9〉 Tagging에서 하나의 완전한 문장으로.

지금의 웹은 검색엔진이 만든 인덱스(Index) 정보를 찾거나, 〈그림 9〉의 하단 왼쪽과 같이 거지(Beggar), 구걸(Begging), 구걸하다(NeedaCash) 등으로 태깅(Tagging)을 달 수 있다. 이러한 태깅으로 종전의 홈페이지보다 쉽게 데이터들을 선택하고 관리할 수 있는 장점이 생겼지만 똑똑한 웹을 만들 수는 없다. 그래서 웹3.0은 Description(Rule)들을 이용하게 된다. Rule들은 Resource와 Property의 조합으로 하나의 문장(Statement)을 만들어 〈그림 9〉의 하단 오른쪽과 같이 설명할 수 있다. 물론 Cash는 Money와 동일한 의미라는 것은 물론, 동일한 단어임에도 도메인이 다를 경우에는 전혀 다른 의미로 사용되는 것들을 정의할 수 있으며 계층 관계 등 여러 가지 룰들을 표현해 저장할 수 있다.

시맨틱 웹서비스

SOA의 역사에서 언급했듯이 키워드 기반 검색의 한계를 웹 플랫폼의 접근성으로 어느 정도 해결할 수 있다고 해도 근본적인 해결책이 되지 않는다. 따라서 이를 해결하기 위해 시맨틱 웹과 웹서비스가 결합된 시맨틱 웹서비스(Semantic Web Services)가 탄생했다. 여기에는 웹서비스를 잘 설명할 수 있는 온톨로지(Ontology)가 생겼고 크게 세 가지 구성요소를 가지고 있다.

● Service Grounding

WSDL에서 웹서비스의 실제적인 주소와 데이터/메소드 정보를 얻어와 하나의 프로세스로 맵핑시키는 역할을 한다.

● Service Profile

Grounding된 서비스에 의미를 부여하는 곳으로 지리적 정보(한국에서 중국에 있는 중국집에 음식을 주문하는 상황 방지), 제약, 추천 서비스(내가 원하는 상품이 없을 경우 비슷한 기능 가진 상품 추천), 계층 관계 등을 표시함으로써 똑똑한 웹이 될 수 있다. Profile에 어떠한 정보들을 모아 구성하느냐에 시맨틱

웹서비스의 성패가 달려있다고 볼 수 있다.

● Service Model

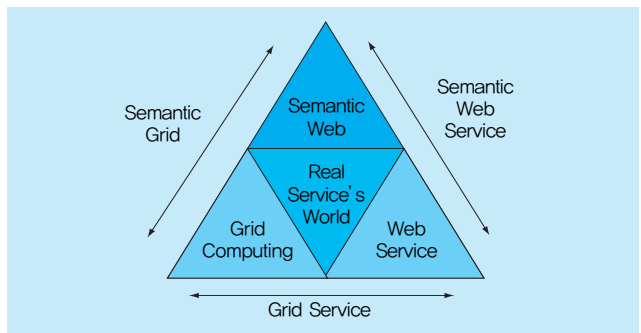
워크플로우(Workflow) 시스템처럼 기본적인 분기, 반복과 같은 서비스의 흐름을 표현하는 곳이다.

그리드 컴퓨팅

유틸리티 컴퓨팅(Utility Computing)으로 대변되는 그리드 컴퓨팅은 클러스터링(Clustering)의 진보된 모델로 볼 수 있다. 클러스터링은 수십, 수백 개의 컴퓨터들을 합쳐 슈퍼컴퓨터의 성능을 낼 수 있게 만든 것으로, 동일한 플랫폼을 가진 컴퓨터들로만 구축할 수 있으며 지역적인 제약을 가지고 있다. 하지만 그리드 컴퓨팅은 XML 웹서비스를 이용해 이러한 문제들을 해결함으로써 누구나 대용량의 계산 능력이 필요할 때 이용할 수 있다. 현재 IBM, 오라클, 썬이 이 시장을 적극적으로 공략하고 있다.

기술들 간의 조우

〈그림 10〉에서 보는 것처럼 세 가지 기술들이 서로 협력하면서 발전하게 된다는 점이 흥미롭다. 시맨틱 웹이 성장하기 위해서는 추론/증명 시 필요한 컴퓨팅 파워를 그리드 컴퓨팅으로부터 얻어 와야 하므로 시맨틱 그리드(Semantic Grid)라는 영역이 탄생한다. 기존의 그리드 컴퓨팅 1세대는 FTP 기반이었지만, 다양한 플랫폼과 대화하기 위해 웹서비스를 이용했고 이 역시 그리드 컴퓨팅의 2세대인 OGSA(Open Grid Service Architecture)가 생겨나는 데 결정적인 공헌을 했다. 그리고 현재의 상호운영성에 초점이 맞춰져 있는 웹서비스에 시맨틱 웹이 결합되면서 시맨틱 웹서비스라는 진화 모델이 앞으로의 컴퓨팅 환경을 바꿀 것이다.



〈그림 10〉 미래를 이끌어갈 기술들

이러한 기술들로 인해 똑똑한 SaaS가 탄생할 것이다. 웹과 서비스가 모두 진화하면서 이 두 가지 측면을 모두 지닌 SaaS는 우리의 삶 깊숙이 들어와 많은 것들을 자동화해 주고 이전에는 상상치 못한 소통들을 가능하게 할 것이다. +