<JSTORM>

GRADY REPORT I : EJB

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

Document title:	GRADY REPORT I : EJB
Document file name:	GradyReport_박현철.doc
Revision number:	<0.2>
Issued by:	<박현철> hpark@kebi.com
Issue Date:	< Dec. 22, 2000>
Status:	continue

Content Information

Audience	J2EE Archirecture와 개발방법론에 관심있는 자바 개발자.
Abstract	EJB의 특성과 개요을 알아보고 여러가지 디자인 패턴에 대해서 살펴본다. 또한 최신 개발방법론인 XP에 대해서도 설명한다.
Reference	박현철(hpark@kebi.com) 님이 제공해주신 문서입니다. 박현철님께 다시한번 감사의 말씀을 드립니다.
Benchmark information	

Table of Contents

컴포넌트에 관한 "Grady Report I: EJB"

(2EE와 애플리케이션 개발 속의 디자인 패턴들)

내용이 워낙 "딱딱"하다보니 나름대로 "격식 없는" 글을 통해 글을 써나가도록 하겠습니다.

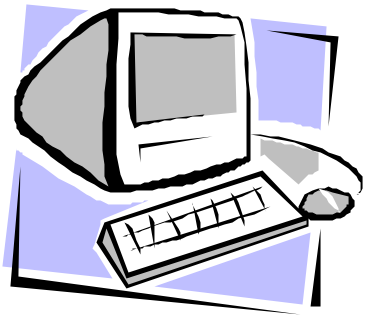
그럼, 부제를 "J2EE(Java 2 Platform, Enterprise Edition)와 애플리케이션 개발 속의 디자인 패턴들"이라 정한 이유에 대해서 먼저 간단히 알아볼까요?

현재의 중/대형 규모 소프트웨어 개발의 여러 특징 중의 하나는 아키텍처 기반이라는 점입니다. 오래 전의 중앙 집중식 컴퓨팅 환경이나 초기의 클라이언트/서버 환경과는 달리, 요즘 같은 매우 다양한 네트웍, 다양한 컴퓨터 자원들을 포함한 광범위한 분산 컴퓨팅 환경에서, 필요한 모든 소프트웨어를 처음부터 개발하는 것은 "나폴레옹"식 접근일 것이기 때문이지요. 기존의 라이브러리, 프레임웍, 컴포넌트를 포함한 다양한 리소스들 뿐만 아니라, 여러 프로젝트를 통한 경험/전략 또는 그 이상의 것까지도 재사용을 하는 것이 필수적입니다. 예를 들면, 분산 환경에서의 Security, Transaction, Naming, Concurrency, Persistence 등과 같은 반복적 이슈를 매번 구현할 수는 없는 일입니다. 따라서 분산 환경을 위한 기존의 아키텍처를 활용하는 것은 매우 의미 있는 일이며, 그 아키텍처들 중 하나가 J2EE에 포함되어있다는 점을 주목할 필요가 있습니다. 즉, J2EE 플랫폼이 제공하는 여러 제품, 표준 등을 살펴보고 그 곳에서 제시되는 아키텍처를 활용하는 것은 선택이 아닌 필수적 상황으로 점차 변하고

있다는 점입니다. 따라서, EJB를 설명하기 전에 EJB를 포괄하는 전체적인 틀로서의 J2EE를 소개하는 것은 의미 있는 일이라고 생각됩니다.

둘째는, 디자인 패턴의 중요성입니다. 사실, 디자인 패턴이 또 다른 앤티 패턴(Anti-pattern)이 될 수도 있지요. 즉, XP(eXtreme Programming) 진영에서 지적하는 것처럼, 디자인 패턴이 최적화된 시스템을 만들기 보다는 막연한 확장 가능성을 위해 많은 시간을 투입하고, 소프트웨어의 복잡도만을 증가시키는 최악의 상황을 연출해 내는 걸림돌이 될 수도 있습니다. (참고로 XP 진영에는 Beck, Fowler와 같은 유명인사들 외에도 디자인 패턴으로 유명한 Gamma를 비롯한 많은 사람들이 이미 합류해 있습니다. 저는 안끼어주더군요. ㅠㅠ) 그럼에도 불구하고 디자인 패턴을 선택한 것은, 단점이 있다 해도 나름대로 많은 장점을 갖고 있기 때문입니다. 이미 많은 숙련된 개발자들은, 다른 사람들과의 커뮤니케이션에 있어 패턴의 사용은 우월감의 표현이라기 보다는, 빠른 시간 내에 정확한 의사 전달을 하기 위한 수단으로 활용하고 있습니다. 즉, 방법론 내부의 객체/컴포넌트 모델링 단계에서 디자인 패턴의 표현은 소프트웨어 아키텍처의 보다 구체적인 이해와 명확한 커뮤니케이션을 위해 필수적인 요소가 되어가고 있지않나 생각합니다.

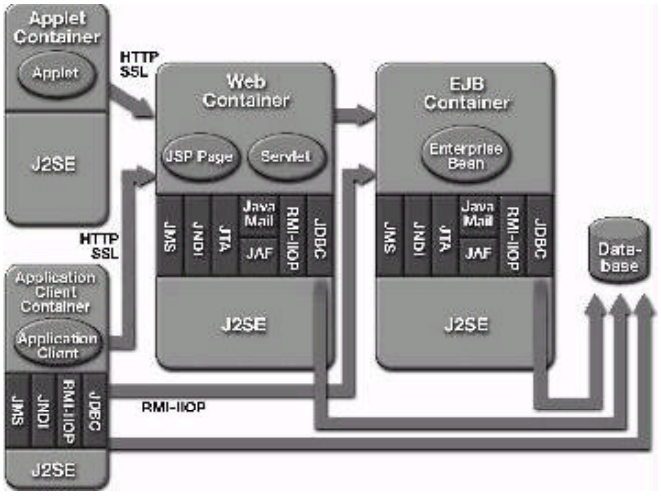
Grady Report는 박현철(hpark@kebi.com) 님이 제공해주신 문서입니다. 박현철님께 다시한번 감사의 말씀을 드립니다.



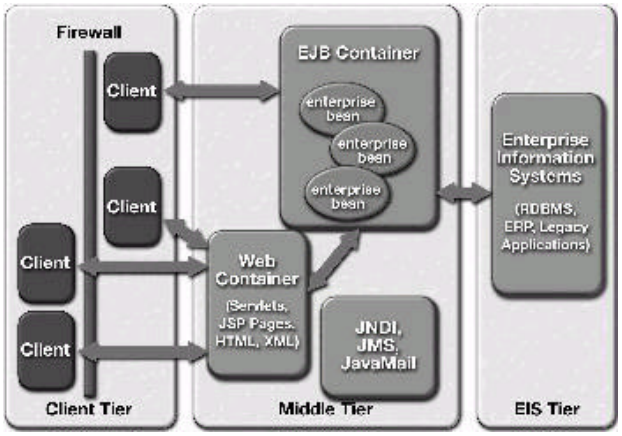
J2EE

우선 J2EE를 말씀드리기 전에, Java Technology를 기반으로 선(Sun)에서 추구하는 전반적인 방향을 짚어보겠습니다. 크게 3가지 즉, Standard Edition인 J2SE, Enterprise 솔루션을 위한 J2EE, 임베디드 솔루션을 위한 J2ME가 있습니다. 구체적인 내용은 틀리지만, 전체적인 방향에 있어 마이크로소프트가 추구하는 .NET 솔루션과 매우 흡사합니다. 즉, "내가 다 먹겠다." 또는 "전세계를 나의 품안에"식 전략이라는 것이지요. 저는 개인적으로 이들의 경쟁을 매우 바람직하게 보고 있습니다. 이들의 새로운 기술적 발전을 따라가기 쉽지는 않지만 제게 일거리를 주니까요...
다음은 J2EE의 귀여운 모습입니다.

귀엽죠? 이 J2EE는 크게 4가지로 나누어집니다. Platform Specification, Reference Implementation, J2EE Blueprint, Compatibility Test Suite. 즉, 단순한 API들의 모임이 아닌 벤더들이 준수해야 할 표준, 불루프린트, 몇몇 제품들을 포함한 다양한 레퍼런스 등을 총칭한 것입니다. 좋은 것은 .NET은 비싼 제품군이 대부분인 반면, J2EE는 공짜라는 점이지요. 나쁜 것은, 써보시면 아시겠지만, 문제가 쪼금 있습니다. AS는 당연히 안되고요...ㅠㅠ 참고로 .NET은 일종의 플랫폼으로 역시 크게 4가지로 분류해볼 수 있습니다: .Net Framework, .Net Enterprise Servers, .Net Building Block Services, .Net Devices



우리나라에 3김이 있듯이, J2EE에는 다음과 같은 3C가 있습니다: Components, Containers, Connectors. 먼저, 컴포넌트부터 볼까요? 위에서 제시한 두 그림과 함께 보세요. J2EE 아키텍처는 상호 연관된 컴포넌트들의 집합입니다. 이들 컴포넌트는 Application Client, Applets, Web Components, Business Compoents로 분류됩니다. 예를 들면, HTML, Servlets, JSPs, EJBs 등이 있다는 것이지요. 컨테이너는 비즈니스 컴포넌트의 생명주기(Lifecycle)를 관리하고, J2EE에서 지원되는 API와 컴포넌트를 위한 환경을 제공합니다. 커넥터는 컨테이너와 EIS(Enterprise Information System)간의 연결 고리 역할을 제공합니다. 쉽게 이야기하면, 브라우저와 서버에 둥둥 떠있는 놈들, 그놈들이 거주하는 소굴, 다른 조직책과의 연결 통로나 무기고(DB)로 통하는 문들이 있다는 말이지요. 다음 그림을 보시면 조금 더 눈에 들어오실 겁니다.



J2EE는 아키텍처가 가져야 할 다음과 같은 중요한 요소들에 대한 포괄적인 가이드라인을 제공합니다. 즉, Availability, Capacity, Extensibility, Flexibility, Manageability, Performance, Reliability, Scalability, Security, Testability를 포함한 여러 중요한 이슈에 대해 다양한 기술적 가능성을 제시하고 있습니다. 예를 들면, Middle Tier중 Web Tier에는 Servlet, JSP를 제공하는 몇 개의 분산된 웹 서버를 배치하고, EJB Tier에는 몇 개의 분산된

EJB 서버를 제공하고, Tier들 사이에는 Load Balancing Router를 배치하고, JNDI를 통한 네이밍 서비스를 제공해서 Flexibility, Reliability, Availability, Scalability, Extensibility, Manageability를 증진시킨다... 등등. 물론 여러 토끼를 함께 잡는 것이 쉽지는 않지만 나름대로의 가능성을 제시하고 있다고 판단됩니다.

J2EE 내부에도 다양한 디자인 패턴이 존재합니다. EJB 부분에서 예를 들면, 리모트 인터페이스들은 프락시의 일종이고, EJBObject는 빈에 대한 하나의 데코레이터이며, EJBHome을 통해 EJBObject를 얻어오는 것은 팩토리 메소드 패턴을 활용한 것입니다. (^^ 이상 Sun, SL-425 3-157에 나오는 이야기였습니다.)

가끔은 표도 나와야 실감이 더 나겠죠? 다음은 J2EE에 있는 API들에 대한 간결한 설명입니다.

API	Description
EJB	Enterprise JavaBeans is a server component model that provides portability across application servers and implements automatic services on behalf of the application components.
JNDI	Java Naming and Directory Interface provides access to naming and directory services such as DNS, NDS, LDAP, and CORBA Naming.
RMI/IIOP	Remote Method Invocation creates remote interfaces for Java-to-Java communications. This extension uses the CORBA standard IIOP communications protocol.
Java IDL	Java Interface Definition Language creates remote interfaces to support Java-to-CORBA communications. Java IDL includes an IDL-to-Java compiler and a lightweight ORB that supports IIOP. (Java IDL is part of the Java 2 platform, Standard Edition, formerly known as the Java Development Kit.)
Servlets and JSP	Java servlets and Java Server Pages are server components that run in a Web server that supports dynamic HTML generation and session management for browser clients.
JMS	Java Messaging Service supports asynchronous communications using either a reliable queuing or publish and subscribe programming model.
JTA	Java Transaction API provides a transaction demarcation API.
JTS	Java Transaction Service defines a distributed transaction management service based on CORBA Object Transaction Service.
JDBC™	JDBC Database Access API provides uniform access to relational databases such as DB2, Informix, Oracle, SQL Server, and Sybase. (JDBC is part of the Java 2 platform, Standard Edition.)
JavaMail	JavaMail provides a protocol independent framework to build mail and messaging applications. (JavaMail requires the JAF API.)
JAF	JavaBeans™ Activation Framework provides standard services to determine the type of an arbitrary piece of data and activate an appropriate JavaBeans component to manipulate the data.

J2EE가 본론이 아닌 관계로, 두서 없는 설명이었지만, 여기서 이만 접겠습니다. 아 조금만, 더 해달라고요? 네... 그럼 약간만 세부적으로 들어가볼까요?

저는 최근에 시스템 아키텍트라는 역할에 많은 관심을 갖게 되었습니다. 이유는 돈을 많이 벌 수 있을 것 같아서입니다. ^^ ...그보다는 중요하다는 생각을 했기 때문이죠. 문제는 소프트웨어 개발과 컴퓨팅 환경 모두에 정통해야한다는 점이었습니다. 앞으로 600박 601일 정도 준비할 생각입니다... 저처럼 이 역할에 관심있는 분들을 위해 간단하나마 아키텍트와 아키텍처를 먼저 보고 지나가도록 하지요. J2EE의 목적과도 아주 근접해있습니다.

시스템 아키텍트와 아키텍처

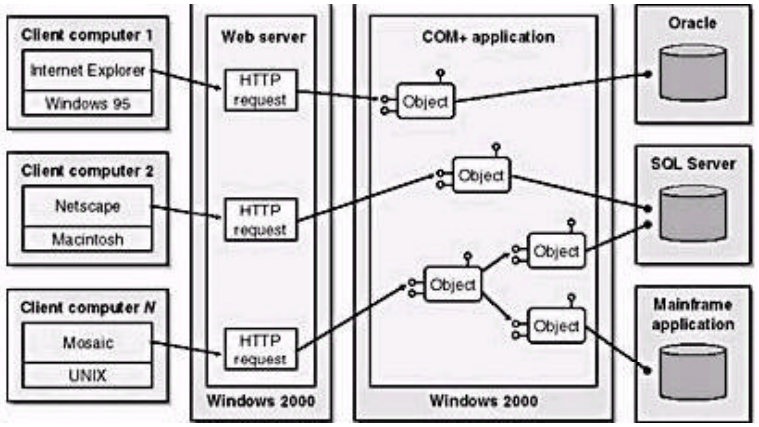
아키텍트는 전지전능해야 합니다. 최소한 슈퍼맨의 옆집에 살아야 합니다. (옆집에 계시는 분이 슈퍼를 경영하고 있음 좋겠네요...) 즉, 고객의 Functional and Non-functional 요구사항을 명확하게 이해하고, 이들 요구사항에 대한 시스템 전체 구조와 세부적 요소들, 그들 간의 상호 관계를 정의할 수 있어야 합니다. 여기서 Function이란 "What the system must do." 이구요. Non-function은 "Fault-tolerance, usability, portability, performance..."와 같은 사항들을 의미합니다. (Large-scale, component-based development 28페이지에 나오는 이야기였습니다.^^) 물론, 아키텍트는 전체 개발 팀에 필요한 역할의 일부에 불과할 뿐, 다른 많은 중요한 역할 - team leads, enterprise modeler, developer, tester, QA staff, configuration experts,... - 도 함께 필요하겠죠.

그렇다면 어떻게 시스템 전체 구조와 세부적 요소들, 그들 간의 상호 관계를 정의할 수 있을까요? 간단합니다. 아키텍처를 정의하면 됩니다. 너무 쉽죠? 그럼 아키텍처를 정의하기 전에 아키텍처의 정의를 유추해보세요. 숙제입니다. 숙제. (바로 위에 다 나와있잖아요.) 이 아키텍처는 상것들에 대한 내용을 표현(Abstract representation)하기 때문에, 시스템의 구성 요소들과 행위를 구현에 관련된 구체적인 내용들까지 포함할 필요는 없습니다. 그럼 어떻게 아키텍처를 잘 정의할 수 있을까요? 소문난 사람 것을 베끼세요. 장땡입니다. 거기에 그것이 만들어졌던 시기와 현재와의 델타(차이)를 반영한다면? 38광땡일 수 있습니다. 문제는 내가 직접 고칠수록 따라지일 확률이 무지 높아진다는 점입니다. 반가운 것은 따라지를 광땡으로 바꿀 수 있다는 점입니다. 쉽지는 않겠죠? 다행인 것은 여기에 그 방법이 나온다는 겁니다. 즉, 미리 사꾸라광과 전두환광을 손목(때에 따라 손바닥)에 숨겨놓아야 하며(요런걸 전문 용어로 "화투를 팜(palm)한다."라고 합니다.), 적시에 그들을 활용하면 됩니다. (연습이 엄청 필요합니다. 잘해야 합니다. 들키면? 죽쑵니다. 또 들키면, 죽습니다.) 그럼 먼저 광을 확보해야되겠죠? 그럼 청바지를 챙겨입고, 서부로 서부로...

자. 광을 확보하기 위한 방법으로 Sun에서 제시하는 3가지 방법이 있습니다. 첫째는 "Capabilities and Design Goals"입니다. 이것은 Non-functional 측면에서 시스템 전반에 걸친 실행가능성에 관련된 요소들을 파악하는 것입니다. 즉, Availability, Capacity, Extensibility, Flexibility, Security, Testability, Scalability, Performance, Reliability, Manageability, Modularity 각각에 대한 Trade-0ff를 고려해야합니다. 둘째, "Process and Artifacts"입니다. 우선 여러 위험(Risk)과 제한사항, 그리고 필요한 가정들을 정의하고 개발에 대한 각 단계와 해당 산출물을 정의합니다. 즉, 평소에 원한 관계에 있던 사람들이 하드디스크나 라우터 속에 엿을 넣거나 광케이블 백본을 톱으로 자를지도 모를 위험 등을 미리 파악하고, 내가 짤리지 않는다는 가정과, 계획단계에서의 고객 접대로 1차는 갈비집, 2차는 그 유명한... 그때의 산출물로 단 한줄짜리 계약서 등을 생각할 수 있겠지요. 물론, 드물게 XP를 기반 Process로 활용하고, UML을 활용한 산출물들을 정의할 수도 있겠죠. 셋째는 "Communication mechanisms"입니다. 이 상황에서 HTTP, HTTPS, RMI, IIOP, Messaging, Transaction 등을 결정하게 됩니다. 광 확보 되었죠?

광이 준비되면, 들키지 않고 배치를 시켜야합니다. 배치시키기 위한 방법으로 두가지가 권고되고 있습니다. 첫번째는, Top-down 접근을 통해, Layers, tiers, services를 파악하고, 적절한 추상화(Abstraction)를 일구어냅니다. 여기서 services란 functional 요구사항들로

부터 나온, 단위 기능들이라고 생각하시면 됩니다. (나중에 use-case 기술서에 조목 조목 나타나겠죠?) 추상화는 "black-boxing"을 통한, 시스템 구성요소들 간의 전체적인 그림이라고 보시면 되겠습니다. 도통하신분들은 하얀상자를 사용하셔도 좋습니다. 뭔말인지 당체(도대체) 이해가 안가신다고요? 제가 J2EE 부분에서 제시한 3번째 그림있죠? 그런 것을 약간 손보시면 좋은 그림이 나올 수 있습니다. MS 쪽이면 아래 그림을 적당히 손보셔도 되구요.



둘째. 둘째가 짱이죠. (참고로 저는 우리집에서 둘째입니다.) 원문으로 볼까요? "Experience can be an architect's best tool." 경험 없으면 하덜덜 말란 야그죠. 이때 경험이란 패턴, 프레임웍, 컴포넌트, 아키텍처, 애플리케이션 등을 포함하는 것입니다. 경험 많으세요? 그럼 아키텍트 하심 좋겠네요.

J2EE 기반 아키텍처 정립

자, 그럼 실습을 해볼까요? 이미 여러분들은 J2EE가 무엇인지, 아키텍트의 역할과 아키텍처에 대한 What과 How를 꿰뚫고 계십니다. 물론 농담입니다. 그럼, 실습을 통해서 꿰뚫어볼까요?

J2EE와 관련해서 우선 4가지를 옆에 준비하세요. 첫째는 스펙들입니다. 제사상에 올릴 목록이 먼저 제시되어야겠지요? 이 스펙에는 당연히 엔터프라이즈 애플리케이션을 만들기 위한 컴포넌트, 프레임웍에 대한 표준이 정의되어 있습니다. 둘째와 세번째로 준비할 것은, 상업용으로 쓸 수는 없지만, 스펙들을 구현해놓은 Reference Implementation과 Compatibility Test Suite를 준비하세요. 물론 상용 제품들을 쓰시면 더욱 좋겠지요. 비싼 샐러드 재료가 없으면, 배추와 무를 준비하셔도 되고, 계량컵이 없으시면 그냥 박아지를 쓰세요. 마지막은 Blueprint입니다. 요리법입니다. "아니, 그걸 다 워디서 구한데요?" 간단합니다. "자바 쩜 썬 쩜 컴"에서 구하실 수 있습니다. 전부 꽁짜에요. 반짝 반짝.

Sun에서 제시하는 레이어(Layer)는 제 입맛에 안맞더군요. 달팽이 요리보다는 닭똥집이 더 땡기거든요. Sun에서는 레이어를 4개로, Application(EJB, JSP등의 Components), Virtual Platform(API, Spec.), Upper Platform(Web server, Application server), Lower

Platform(OS) 나눕니다. 참고로 닭똥집은 다음과 같습니다. User interface layer: 눈 앞에서 알짱거리는(Visual) 컴포넌트, 그리고 연관된 부분을 포함하고 있습니다. Visual JavaBeans, ActiveXs, *.html들, Applets라고 보시면됩니다. Business layer: 웍플로우 또는 비즈니스 로직을 포함한 컴포넌트들이라고 보면 됩니다. 대부분의 EJB 빈들이나, COM/DCOM/COM+ 컴포넌트들에 해당하겠죠? Technical infrastructure layer: JSP엔진이나 Container, MTS 정도 되겠죠? 에러 핸들링, ACL, Security, Recovery 등도 포함입니다. (참고로, Realizing e-Business with Components, 44페이지의 Architecture layering에 의한 분류였습니다.^^)

Sun에서는 티어를 Client tier, Presentation(Web) tier, Business(EJB) tier, Integration(EIS) tier, Resource tier로 나누더군요. 제 입맛에도 딱입니다.

자, 그럼 시작해보자구요. 이들 layer들, tier들과 함께 Security, Manageability, Reliability, Availability, Scalability를 함께 고려해보도록 하겠습니다. 먼저, 웹서버 여러대를 준비하고 JSP와 Servlet을 분산 배치하고, 여러대의 애플리케이션 서버를 둡니다. (MS 쪽에서는 IIS에 ASP+들을 분산 배치시키고, 애플리케이션 서버를 여러대 두면 되겠지요?) 서버 컴포넌트들은 JNDI를 통해 찾아오구요. 자, 이런 그림을 그리셨다면, Flexibility, Reliability, Availability들이 팍팍 올라가는게 보이시죠? 물론, Manageability는 팍팍 떨어지겠지요? 이때 애플리케이션 서버에, EJB 컴포넌트로 가득하거나, COM 컴포넌트를 MTS에 둥 띄워놓았거나, 적절한 서비스팩 또는 Windows2000과 함께 COM+ 컴포넌트를 가득 채워놓았다면? Manageability도 마구 올라갑니다. 믿으세요. 믿습니까? 또한 클라이언트와 웹서버 사이, 웹서버와 애플리케이션 서버 사이에 LBR(Load balancing router)를 두었다면? Capacity, Reliability, Availability 마구 올라갑니다. 저한테 서버와 라우터 살 돈 달라고 하지 마세요. 자기 돈 안들이고, 좋은 아키텍처를 유지하는 방법이 있습니다. *ility를 좋아하는 고객에게 청구서 보내시면 됩니다. 협조 안될 경우에는 트랜잭션에 치명적인 병원균(바이러스)이 침투되거나 서버에 좀이 쓸지도 모른다는 근거없는 루머를 퍼뜨리세요.

자. 이것으로 상위 단계의 실습을 마치셨습니다. 많이 꿰뚫으셨나요? 이제, 아키텍트로 나가시는 길만 남았군요. 넘 개념적 이지요? 그럼 선에서 제공하는 스펙과 각종 문서를 직접 읽어보세요. 그래도 개념적입니다... 일단은 참으세요. 끙.

J2EE Best Practices and Guidelines

자. 사실 이런게 남는 겁니다... 여기에서는 Client tier, Web tier, EJB tier, EIS tier의 신상명세를 간단하게 소개하겠습니다.

먼저, Client tier에서는 Browser, Applet, App(Java, Non-Java)를 생각해볼 수 있습니다. 물론 Applet은 Html에 포장되어 날라오겠죠? 그럼 html, applet, application들의 기본적인 모양을 잠깐 차례대로 볼까요?

```
<!-- Filename: hello.html 地는 애플릿을 담고 있구먼유. 넘 간단하쥬? 짜잘한 태그 없어두 되유. 해보세유. -->
<h1> Hi Grady! </h1><hr>
<applet code=HelloApplet width=100 height=100> </applet>
```

```
// Filename: HelloApplet.java --- 지는 애플릿이구먼유.
import java.applet.Applet;
import java.awt.Graphics;
public class HelloApplet extends Applet
{// 딴 코드 필요 없시유. 몽땅 Applet꺼 쓰면 되냄유.
    public void paint(Graphics g)
{
    g.drawString("Hello Grady!", 10, 20);
}
}

// Filename: HelloApplication.java --- 지두 껴두되남유? 애플리케이션이군먼유.
public class HelloApplication
{
    public static void main(String args[])
    {// 스때릭 메인이 있어야 허쥬. 이 함수가 애플리케이션의 시작 번지네유.
            System.out.println("Hello Grady!");
    }
}
```

그런데, Applet은 초기에 있었던 엄청난 열풍에 비해 지금은 그 중요도가 많이 떨어진게 사실입니다. 오죽하면, IE나 Netscape같은 브라우저들조차도 더이상 지원 안하겠습니까? (JDK 1.x 버전 까지만 지원하지요.) (JDK 1.2 부터 지원하는 Super Signed-applet을 사용하면 몰라도)구닥다리 애플릿에서 DB 연동을 할 경우, 웹서버와 데이터베이스 서버가 다르면 연결조차 안됩니다. 다운로드 문제도 있죠... 하지만, 아직도 Applet은 나름대로의 특별한 솔루션 영역을 갖고있는 애물단지랍니다.
그럼 Html과 application인데, 요즘 웹 시대잖아요. 그럼 결론은? Html. 물론 골치 아픈 script들이 양념으로 들어가야겠지요? 참, XML이 빠졌다구요? 물론, 이거 안하면 깡통찹니다. 망한다는 뜻이지요. 하지만 이건 단순한 Client tier만의 문제 또는 단순한 소프트웨어 개발 만의 문제도 아니고, EDI와 표준에 관련된, 덩치도 크고,... 암튼, 다음에 할께요. 하지만, 중요도는 디아블로2에서 캡 좋은 유니크아이템 정도 됩니다. 댜블로2 모르세요? 스타크만든데서 만든건데... 어느정도인지 알고싶으시면 해보세요. 전 바바리안을 주로 했습니다. 추가팩 나오면 어쌔신으로 해봐야지.^^ 얼렁뚱당. Client tier 끝났습니다. 사실, Client tier가 무지하게 복잡할 수도 있었죠. 그런데, Http가 나온 뒤로 유구한 세월이 흘렀건만, 버전이 도통 안올라가지요? 제가 알고 있는 최신 버전이 1.1입니다. 따라서 브라우저와 웹서버 사이에는 말이 필요없습니다. 물론 음성인식 모듈이 Default로 없기 때문에 말을 알아들을 수는 없지만, 둘 사이에 애매모호한 것이 있을 수 있겠어요? 주고 받는 것들이 명확합니다. 그나마 다행입니다.

다음은 Web tier 입니다. 사실, 이놈은 많이 어려울 수도 있습니다. 그 구성요소들을 나열

해볼까요? Html pages, Servlets, JSP, Nonvisual JavaBean, Visual JavaBean, Custom Actions(Custom tags, Templates) 물론 Jhtml도 있었습니다. Html에서 JSP로 가는 중간 단계였죠. CGI도 있다구요? 그럼 ISAPI나 NSAPI도 있겠네요... 요놈들은 잊어버리세요. 이별의 아픔은 세월이 해결해주겠죠. 서블릿과 다이어트에 성공한 JSP 하나씩 보고 갈까요?

```
// Filename: HelloServlet.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
    res.setContentType("text/html");
    PrintWriter out = new PrintWriter(new
OutputStreamWriter(res.getOutputStream()));
    out.println("<html><head><title>Hi Grady!</title>");
    out.println("</head><body><h1>Hello Grady! Happy New Year!</h1>");
    out.println("</body></html>");
    out.close();
}
}

<!-- Filename: hello.jsp 넘 간단하쥬? 사실은 더 간단할 수도 있지유.-->
<html>
<head>
<title>Hi Grady!</title>
</head>
<body>
    <%@ page language="java" %>
<% out.println("<h1>Hello Grady! Happy New Year!</h1>"); %>
</body>
</html>
```

Java의 코흘리개 시절에는 Visual JavaBean들이 기세 등등했지만, Java가 서버측의 솔루션으로 이동하면서, <jsp:useBean...>의 형태로 사용되는 NonVisual JavaBean들이 오히려 주류가 되었습니다. 이들은 WebTier 내에서 MVC 패턴 중 M 또는 C로서 활약을 하죠.(무슨 소린지 모르시겠죠? 패턴 알고싶죠?^^) 잠시, 불프(Blueprint) 99페이지를 볼까요? Custom actions are portable and reusable... Custom actions are ideal for iterating through data and generating the HTML code needed to render a page. 해석하면 "Custom actions는

기똥찬 놈이다."가 되겠죠? ...죄송합니다. "똥"이란 말을 사용했군요. 그래도 이런 말을 들으면 정신이 말똥말똥해지죠? 참. 이부분에 대한 활용 예(코드)는 불프를 참조하세요.

조금만 더 세부적인 구성요소로 들어가 볼까요? 짤막한 처리를 위한 Scriptlets <%...%>, Expressions <%=...%>, Taglib Directive를 포함한 다양한 Directive들, pageContext와 같은 많은 Implicit Object들... Stop! 에구, 볼게 많아서 좋죠?

한가지, 요부분의 불프에서 잘 이해가 되지 않는 것이 있습니다. 왜 i10n과 i18n에 대한 내용을 다루고 있을까? 특히, 중요해서? 해보시면 아시겠지만, 언어적 특성상 UI를 모든 언어에 맞게 만들면, 인마살상입니다. 보기 않좋다는 뜻이죠. 불프에는 언급이 안되었지만, 큰 프로젝트일 경우, 이 WebTier에는 온갖 핸들러(Message, Debugging, Factory, UserSession, Exception, Workflow, Query(DB 쿼리가 아닌거 아시죠?), Encoding/Decoding, Security/ACL, Statistic, Multimedia...)들이 필요하게 됩니다. Locale은 단지 일부에 불과합니다. 이상 불프의 미스테리였습니다.

자. 다음은 EJB tier군요. 사실, EJB는 J2EE의 한 요소에 불과합니다. 하지만 앙꼬도 빵의 한 요소에 불과하죠? 고무줄도 팬티(바지를 의미하는 것이 아님.)의 한 요소에 불과하죠?

여기서 천기누설이라고나 할까. 저만의 비밀을 한가지. 데이터베이스를 연결할 때 동시 사용자 수를 줄이기 위해, 대부분의 DB 연결은 Pool에서 얻어온 커넥터를 사용하고, 서비스를 받자마자 매정하게 끊어버려서 Pool로 다시 돌려보내면 매우 효율적입니다. 아니. 알고 계시다구요? 이런! 이미 커넥터는 중요한 컴포넌트가 되었군요... 마찬가지 입니다. 애플리케이션 로직들을 각각의 COM/DCOM/COM+, EJB 컴포넌트들로 만들어 놓고, 미리 Pool에 몇 만개정도 만들어 놓고(웹로직의 default 컴포넌트 수가 억단위 이상일 겁니다. 제가 설정한 것이 아닙니다...), 유저가 서비스를 의뢰하면 미리 만들어 놓은 놈을 연결해주고, 유저가 조용하면 다시 풀에 집어넣고... 이렇게 하면, 웹의 사재기 특성상(사재기는 한꺼번에 몰리는 경향이 있지요?) 많은 유저가 일시적으로 몰릴 때, 미리 만들어 놓은 놈으로 서비스를 바로 해주고, 뜸 하면, 다시 재사용하고(일식집에서 회를 받드는 무우채와 접시를 생각하시면 됩니다.), 얼마나 좋습니까?

UI 단의 컴포넌트화는 수십년(?) 전으로 거슬러가겠지만, 서버 단의 컴포넌트화는 제가 알기로는 올해가 겨우 10년째입니다. 아직은 미천하지요. 하지만, Windows 2000 Datacenter Server의 경우 32개의 CPU와 64G 메모리를 지원합니다. 우와 64G! 그렇다면 아직은 미천한 컴포넌트 기반의 애플리케이션이라도 이처럼 빵빵한 OS와 만나면 큰 역할을 할 수 있다는 결론이 나옵니다. 거기에 웹으로 복잡하고, 헤테로지니어스한(한글로 보기에도 어려워보이지요?) 분산환경을 싹 가리면? 이걸 금상첨화라고 하지요.

이 외에도 컴포넌트에는 좋은 점이 많지요. URL을 입력하면, 최신의 Page를 볼 수 있듯이, JDNI를 통해 따끈따끈한 최신 컴포넌트로부터 서비스 받을 수 있고, 멀티 쓰레드 몰라도 "좋아. 좋아"였다가, 모르면 바보 취급하던 시절이 있었지만, 요즘에는 Servlet 엔진이 있는데도, 멀티 쓰레드 프로그래밍하는 놈이 더 바보랍니다. 서버단에 자주 사용되는 정보들에 대한 캐쉬로 엔터티빈을 사용하면 넘 조아! 복잡한 기업의 웍플로우? 세션빈으로 OK! 게다가 개발이 아닌 배치 시점에서 보안이나 트랜잭션을 포함한 여러 특성을 컴포넌트에 부여할 수도 있답니다. 이처럼 Security와 Transaction, Concurrency, Naming,

Persistence, Deployment time configuration 등을 쉽게 제어할 수 있다는 점... 이런데도 컴포넌트 안할 겁니까? "싼 값에 좋은 소프트웨어를 빠른 시간 안에 만들겠다"는 도둑놈(?) 정신에 입각한 소프트웨어 엔지니어링 측면에서도 컴포넌트는 매우 중요한 의미를 갖는다고 생각합니다. (도둑놈(?)이란 표현을 써서 죄송합니다. 날강도(?)..아니 놀부 정도겠지요. 제 개인적인 생각으로는 그나마 돈을 많이 들여야 좋은 품질의 소프트웨어를 빠른 시간 안에, 그것도 겨우 개발하기 때문입니다. 여러 벤더님들은 동의하시지요? 돈을 많이 써야합니다. 자 돈쓰세...@@ ->맞아서 멍든 두 눈.)

여기서 컴포넌트가 갖고있는, 단 하나의 작은 사소한 나쁜 점을 소개해드리겠습니다. 그것은 컴포넌트가 잘 안된다는 점입니다. 97년도부터 속아왔습니다.ㅠㅠ 98년도에도, 99년도에도, 올해도 속았습니다.ㅠㅠ 똑같은 것들이 규모만 커지면 안됩니다. 잘 도는 놈들을 컴포넌트로 분리하면 확률적으로 동작합니다. 컴포넌트가 그만큼 지능적인 것 같아요. 퍼지이론도 알고, 불확정설의 원리도 알고, 카오스나 복잡계 이론도 알고... 파일럿 프로젝트에서는 완벽합니다. 실제 프로젝트에서는 예외 있는(?) 문제아로 등장합니다. 따라서 컴포넌트들의 지능을 죽이는 무립 비법의 구축이 시급합니다. 저도 한초식 하지만 주위에 알고보면, 전문가들 많습니다. 대부분 잡을 수 있습니다. 아! 그런데, 그 부분들이 OS의 커널부분 또는 벤더의 컨테이너 구현부분이라면?ㅠㅠ. 또는 NT에서는 잘 도는데 UNIX로만 옮기면 안된다면? 미들웨어의 Bug라면? (주의. 이때는 저를 찾으시면 절대 안됩니다.)

그럼, 본론으로 들어가겠습니다. EJB는 크게 Session Bean과 Entity Bean으로 나눌 수 있고, Session Bean은 Stateful, Stateless로 나눌 수 있고, Entity Bean은 CMP(Container Managed Persistence)와 BMP(Bean Managed Persistence)로 나눌 수 있습니다. 이상 EJB에 대한 내용이었습니다. ^^ 간단하지요?

```
// Filename: HelloHome.java
import javax.ejb.*;
import java.rmi.RemoteException;
public interface HelloHome extends EJBHome
{
    public HelloObj create () throws RemoteException;
}

// Filename: HelloObj.java
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface HelloObj extends EJBObject
{
    public String getMessage() throws RemoteException;
}
```

```
// Filename: Hello.java
import javax.ejb.*;
public class Hello implements SessionBean
{
    public void ejbCreate() throws RemoteException, CreateException {...}
// getMessage()만 신경써서 구현하고, 나머지는 남들이 만든부분을 보고,
// 적당히 베낀다. ^^
...
}
```

잠깐이나마, SessionBean 코드의 일부를 보았습니다. 사실 위에서 본 Home interface, Remote interface, EJB class 말고도 몇개의 클래스들이 더 필요하지만, 나머지들은 스펙을 구현한 벤더들의 툴들이 몽땅 만들어준답니다. (옴메, 좋은거.) CMP(Container Managed Persistence)인 EntityBean의 경우에는 우닥닥 더 많은 코드가 만들어지겠죠? 두말하면 잔소리(?)입니다. 쿨쿨. 어쨌든 인터페이스, 클래스, JNDI 이름과 함께, 이들 Session 빈에 대한 timeout, stateless 여부, security, transaction 등의 여러 속성들을 ejb-jar.xml(EJB1.0에는 그냥 텍스트 파일이었죠? EJB1.1부터 descriptor가 변했습니다.)에 써넣고 .jar파일을 만들면 준비는 다 된겁니다. 물론, 스펙이 다르거나, 특정 벤더의 제품을 사용하는가에 따라 별도의 과정이나 파일들이 필요하게 됩니다. 두어번만 해보시면 익숙해집니다. 믿음을 가지세요. 믿습니까? 네? 눈에 보이는 것만 믿으신다구요? 사랑이 눈에 보이십니까? 우정이 눈에 보이십니까? 저런, 사랑과 우정을 믿지 않으시는군요. 엄청 좋은건데,...

EntityBean은 PK 클래스만 더 추가하면 됩니다. 물론 그나마 EJB1.1 스펙에서는 Optional로 정의하고 있죠. 이 PK 클래스에는 DB 테이블의 Primary Key(s)에 해당하는 멤버들을 정의하면 됩니다. 그런데, EntityBean의 경우에는 Home 인터페이스의 역할이 매우 중요해집니다. 즉, Home에는 create(..)와 다양한 find..(..) 함수(나중에 요 이름앞에 ejb가 붙으면서, ejbFind..(..)로 만들어지죠.)들을 선언하게 됩니다. 그런데 EJB2.0에서는 더욱 발전된 형태로 변하게 됩니다. 즉, descriptor에 원하는 쿼리문을 넣으면, 2.0에서 추가된 persistence manager라는 놈을 통해 단순 쿼리 이상의 find함수를 구현할 수 있게 된 것입니다. 짜쟌.

자. 그럼 결국, EJB tier는 앞단의 Web tier와 뒷단의 EIS tier 사이에서 워크플로우와 비즈니스 로직을 담고있는 부분이군요... 물론 보안, 트랜잭션 등과같은 시스템 차원의 서비스도 포함되겠죠? 말밥이죠? 뒤에 패턴에서 다시 언급하겠지만, 엔터티 빈은 Facade로서의 세션빈 내부에서만 활용하시는 것이 좋습니다. (에구, 패턴 모르는 설움.) PK에 대한 코드만 잠깐 보고 EntityBean은 끝내도록 하지요.

```
// Filename: InganPK.java -- Ingan이 무어냐구요? 인간은 말이죠... 알겠다구요?
public class InganPK implements java.io.Serializable
{
    // 사실 이 클래스 안만들어도 됩니다. 프라이머리 키가 String 또는
```

```
    // 자바의 기본 타입일 경우
    // .xml에 ...<prim-key-class>InganPK<prim-key-class>..라고만 하셔도 됩니다.
    public String name;
    public InganPK() {}
    public InganPK(String name) {this.name = name;}
    public boolean equals(Object obj) {/* 나 코딩 */}
    …
}
```

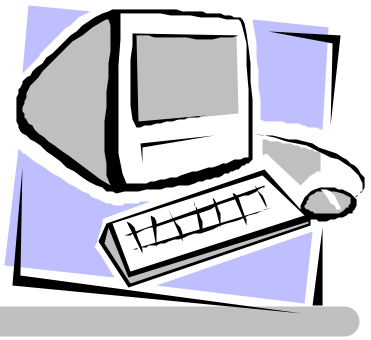
앞에서는 언급을 안했지만, 불프 140페이지에 DAO(Data access objects), 143페이지에 VO(Value objects)에 대한 내용이 있습니다. 이놈들이 뭐하는 놈이냐구요? 정체를 알고 보면 같잖은 놈들이죠. 특별한 놈들이 아니라 늘상 써왔고, 앞으로도 애용해야하는 버스와 지하철 같은 것들이죠. DAO는 데이터베이스관련 로직을 담아놓는 클래스들이고, VO는 EJB tier에서 Web tier로 옮아가는 정보를 담은 클래스들이라고 보시면 90점은 됩니다. 큰 프로젝트의 경우, 이 부분이 엄청 커질 수도 있겠죠. 제 개인적인 생각에, DAO는 Session Bean에서 직접 DB를 핸들링하거나 BMP를 사용해야 할 경우에 쓰시면 딱입니다. VO는 남발하셔도 만수무강에 아무런 지장 없습니다. 형태도 다양하게, 해쉬테이블에도 담고, 벡터에도 담아보고... Serializable 구현해주시는 것만 잊지않으시면 됩니다. 물론 정보의 전달에만 쓰셔야합니다. 관련된 코드는 불프를 참고해주세요.

다음은 EIS tier입니다. 여기서는 기업에서 필요한 기반 정보들을 제공하는 역할을 하죠. 기존(Legacy) 시스템과의 연동이 없을 경우 간단할 수도 있지만, 경우에 따라 J2EE의 범위를 벗어나는 매우 복잡할 수도 있는 부분입니다. 예를 들면, 예전 D사에 있을 시절, 대출 시스템의 애플리케이션 서버로 NT를 사용했었고, 두대의 IBM 호스트의 내부에 DB2와 SAM 파일 형태로 정보계와 회계관련 데이터들이 분산되어 있었습니다. 단순한 정보만의 획득이 아닌, 트랜잭션과 보안의 문제도 매우 중요한 이슈였습니다. 즉, 대출 업무의 특성상 이자계산, 각종 보고서와 같은 비즈니스 로직 뿐만 아니라, 회계정보에 대한 갱신도 필요했었으니까요. 그당시 SNA 게이트웨이와 CICS/NT, DRDA를 사용했었습니다.

J2EE에서는 가급적이면, 보안, 트랜잭션 또는 Scalability에 관한 사항을 EJB의 컨테이너에게 많이 떠넘김으로써 EIS 쪽의 부담을 줄이려하고 있습니다. 좋은 전략이죠. "J2EE에 무슨 EIS 쪽 관련 서비스가 있어? JDBC말고 또 있나?"하고 생각하셨죠? 있습니다. 그것도 많습니다. 열거해볼까요? JDBC 외에 JNDI, JMS, JavaMail, JavaIDL, JNI 등이 있습니다. 그럼 눈치 채셨죠? JMS와 JNI를 보니까, 의외로 골치 아픈 부분이 몰려있다는 것을. 또 예를 하나 들까요? 웹에서 Word나 Excel등의 자료들에 대한 CRUD(읽고 쓰고, 지지고 볶고)가 필요하다고 생각해보세요. 애플리케이션 서버는 UNIX 박스이고요. JNI 쓰셔야합니다. 만만치 않죠... C 또는 C++을 사용해서 .dll도 만드셔야 합니다. VBA 프로그래밍도 익숙하셔야 합니다. J++ 쓰면 간단하다고요? J++은 JDK 1.1.7까지만 지원합니다. 물론, 그나마도 안됩니다. VM이 달라요... 무조건 JNI 쓰셔야합니다.
더 큰 문제는, 기업 환경과 업무에 따라, 썬이 제시하는 것 이외의 많은 미들웨어가 필요

할 수도 있습니다. eLink나 XML이 필요할 수도 있구요. TopLink와 같은 놈들이 필요할 수도 있지요. 여러 RM(Resource manager)과 함께 "Two-Phase Commit"이 필요한 경우, 또는 RDB와 함께 HDB, ORDB, OODB, SAM 도 함께 처리해야하는 경우라면, Windows 2000에서 제시하는 DTC(Distributed Transaction Coordinator)와 같은 놈들도 필요하게 되겠지요. 또한, 이전의 Sync.만의 업무 환경에서 e-mail 통보와 같은 Async.가 필요한 업무들이 늘어나고 있죠? 기존의 업무들과 함께 연동되어야 하죠? 깝깝하시죠? 기존의 ERP 패키지와 연동해야한다면? 데이터 마이닝이나, 데이터 웨어하우징을 생각해보시죠. 더 깝깝하시죠?
결론이 명확하죠? 일반적인 솔루션 또는 해답이 있을 수 없습니다. 그래서 정보를 얻고자하는 Client들을 위해, 복잡한 하부구조를 숨기는, 적절한 인터페이스가 매우 중요합니다. J2EE에서는 이런 역할을 위해 EIS Access Objects(Connection Architecture)라는 말을 사용하고 있습니다. 즉, Access Objects는 비즈니스 컴포넌트들로부터 EIS 시스템과 상호작용하기 위한 복잡한 하부 구조를 숨긴 JavaBean들로, 공통적이면서 일관된 인터페이스를 제공하는 역할을 갖고 있습니다. 하지만, 이들이 직접적으로 트랜잭션이나 보안과 관련된 서비스를 제공하지는 않습니다.

또한 "Acquire and Release"라는 가이드라인을 제시하고 있습니다. 의미는 아시겠죠? DB 커넥션에 관련된 사항입니다.



EJB(Enterprise JavaBeans)

어? 앞에서 했는데 또 하네? 지겹다 생각되시는 분들은 건너뛰셔도 됩니다. 앙꼽니다. 앙꼬. 하지만, 개발자분들은 무조건 보셔야합니다.

자. 별도의 방을 하나 만들었으니, 방의 규모에 맞는 설명이 필요하겠죠? Stub, Skeleton, Proxy 등의 귀에 익숙한 똘마니들을 앞세워, 컴포넌트 시장의 트렌드를 간단히 정리해보겠습니다. 다음 글을 읽어보시지요.

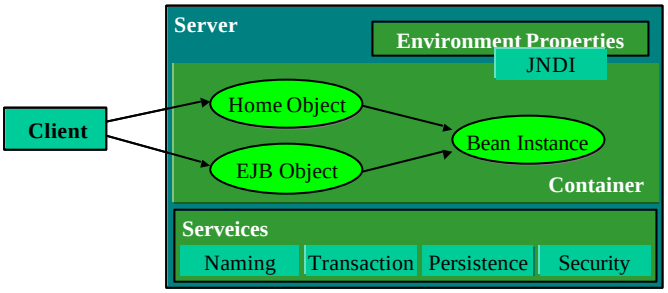
"태고에 RPC(Remote Procedure Call)가 분산 무림 환경을 지배하고 있었으나, 그 후대에 CORBA, COM, EJB가 등장하면서, 무림의 평안은 난지도 쓰레기장에 버려지는 신세가 되었다. 무림인들은 어느 곳에 속해야 할지 고민하면서 갈팡질팡 하고 있었으니. 오호 통재라..." RPC에서는, 무림첩을 받은(서버) 측이나 무림첩을 준(클라이언트) 측 모두를 Stub이라 하였다. "고놈이 고놈인지라, 보낼 때 마샬링과 받을 때 언마샬링을 하건만 되돌아 올 때도 매한가지가 아니더냐. 둘다 Stub 이라 하라." 하지만, CORBA 쪽에서는 새로운 문하생들을 끌어들이기 위해, 무언가 튀는 말을 만들어, 새로운 개념 인양 선전을 해야만 하는 입장이었다. 그래서 이들은 무림첩을 주는 이들은 그냥 Stub 이라 하였지만, 이를 받아들이는 쪽은 Skeleton이라 칭하였다. "본 무림 연맹(OMG)은 이미 무림을 평정하였다. 따라서 모든 무림은 본방의 의도를 따라야 할 것이야. Stub과 Skeleton이라, 이 어찌 기가막히지 않다 할 수 있겠는가." 이 때 COM 쪽 진영에서는 " 어허, 맹랑한 놈들이로고, 우리가 말하려고 한 것을 먼저 공표하다니... 에잉. 어쨌든, 우리는 기존의 RPC를 내맘대로 뜯어고친 ORPC를 사용하는 것처럼, 무림첩을 주는 놈들을 Proxy라 칭하고, 받는 쪽은 그냥 Stub이라 하거라. 왜? 내맘이니까. 에잉. 우쨌든, 그놈들이 암만 까불어도, 우리들은 MTS(Microsoft Transaction Server)로 중생들을 현혹할 것이니라. 웃핫핫하하" 결국 이렇게 되었다. 한편 CORBA 문하생이었던 Sun동자는 뒤로 지속적인 호박씨를 까고 있었으니, 후대에 EJB 문파를 만들기에 이르렀다. " 내, 선배에 대한 도리를 정령 잊지 않으리라. 호박씨 깔판으로 RMI(Remote Method Invocation)를 사용하지만, Stub 과 Skeleton을 그대로 사용하겠노라. 물론 선배와의 유대를 위해 연방 비무(IIOP)도 지원하리라. 그리고, MTS에게 선수를 빼앗겼지만, 내가 중생들에게 EJB 비급(스펙)을 무료로 배포하니만큼, 나의 java 추종자들을 중심으로 무림 천하는 곧 나의 손에 들어올 것은 뻔한 일. 여봐라. 무림 지천에 나의 EJB 비급을 전달하거라. 우와핫핫핫." 이때 한편으로 MS 문파에서는 "큰일났사옵니다. 지금 많

> 은 중생들이 EJB라는 사술에 홀려갈 뿐만 아니라, 심지어는 우리 문하생들 중에서도 이탈자들이 출몰하는 지경에 이르렀나이다. 통촉하소서." "...음. 내 이런 날이 올 줄 알았느니라. 내 Java에 대항하기 위해 미리 C#이라는 것을 만들었고, COM과 MTS를 통합한 COM+ 비급을 이미 완성했느니라. 단, 아직 내공을 6성까지밖에(Windows2000) 연마하지 않아서... 하루 빨리 8성(.Net Studio)의 경지를 지나, 10성(휘슬러)의 경지에 이르러야 할텐데. 서두르다가는 주화입마가 걱정되고... 어쨌든, 무림 천하에 방을 붙이거라. C#을 공표하고, COM+를 연마할 수 있는 미완성 도구들을 무료로 배포하고, 북치고 장구치고, 돈들여 볼거리를 만들지어다." 한편, OMG 본각에서는 "어허, 체면이 말이 아니로고... 무림 천하의 대부분의 문파들이 우리 아래이거늘 중원에 EJB와 COM의 영향력이 그토록 세다는 말이냐. 끌끌, 우째 이런일이. 끙. 우리가 배포한 비급(CORBA 스펙)을 각 문파들 맘대로 해석해서, 기본 초식조차 문파마다 달리하여, 그렇게 본방의 얼굴에 먹칠하더니, 이번에는 사파와 후배에게 권자를 넘겨주게 생겼으니... 우째 이런 일이. 어쨌든, 하루 빨리 CCM(CORBA Component Model)을 완성하여, 기존 문파를 중심으로 사활을 건 한판을 시작하리라."... 이리 하여 다시 한번 중원에는 혈겁의 회오리가 몰아치게 되는데,...

어때요. 대협들께서는 대충 정리가 되셨습니까? 컴포턴트의 역사는 이렇게 흘러왔습니다. 이처럼 역사는 흐릅니다. 왜냐? 시간적인 이슈이기 때문입니다. 무슨말인지 모호하다구요? 좀 거창하게 설명해볼까요? 우주는 "시공간"으로 동적인 개념과 정적인 개념이 지배하는 하나의 궤입니다. 따라서 무언가를 살펴볼 때 시공간적인 틀에 맞춰 살펴봄이 바람직 하죠. 네트웍과 컴퓨터들도 결국은 정적인 구조입니다. 그들은 항상 그곳에 있습니다. 누군가 운동을 위해서 계속 컴퓨터를 들었다 놨다, 네트웍 라인으로 줄넘기를 한다 해도 어쨌든, 확률적으로 존재합니다. 글구, 전기세를 내고 있는 동안, 그 안에 여러 전자기적 흐름이 있습니다. 정력 좋은 전자들의 흐름이 펄스가 되고, 디지럴(digital, 0과 1)이 되고, 다양한 데이러(data, 텍스트, 멀티미디어...나 동영상. 너 사운드.)가 되고, 정보(ex, 백지영은 정보화사회의 희생양이다.)가 되는 것 뿐이지요. 물론 중간에 사람에 의한, 천재 지변에 의한,... 다양한 제어들이 있죠. 이처럼 우리가 늘상 활용하는 네트웍과 컴퓨터도 결국은 정적/동적인 하나의 궤입니다. 개념적으로도 살펴볼 까요? 유즈케이스를 하나의 작은 소우주로 보았을 때, 우리는 그 속에서 여러 흐름과 정적 구조를 살펴볼 수 있게 되는데, 그것을 클래스 다이어그램, 시퀀스 다이어그램 등으로 표현하게 되죠... 당구장이 존재하고, 공들의 흐름이...@@. 화투패와 사람이 존재하고 시간이 흐름에 따라 돈도 따라 흐르고...@@.** (별까지 떳군요.) 네, 그만하겠습니다. 그럼, EJB가 갖고 있는 다양한 정적, 동적인 측면들에는 어떤 것들이 있을까요? 이 말을 꺼내기 위해 거의 2페이지를 할애했군요. 이점을 생각하면서 다음을 계속 보시지요. 무량수불…

EJB Framework

프레임웍이 뭐드라? 아. 다양... 일관된 서비스... 네, 그렇습니다. EJB는 결국 컴포넌트 관련 서비스를 제공하기 위한 프레임웍 역할을 합니다. 구체적인 사항들로, EJB server, EJB container, 각종 EJBs(Session Beans, Entity Beans), Deployment Descriptor, EJB-Jar file 등이 있지요. 그 중 EJB들을 만들기 위해 내부적으로 Home interface, Remote interface, Enterprise bean class가 필요하구요. Home interface는 JNDI를 통해 리모트 객체를 얻기 위해 필요합니다. Remote interface는 Enterprise bean의 서비스를 얻기 위해 필요하죠. Enterprise bean은 실제 서비스를 제공하기 위한 코드들을 포함하고 있죠. 이때 이들을 바탕으로 벤더측에서 만들어주는 Home implementation, Remote implementation 들이 생겨나는데, EJB 컨테이너가 이들을 데리고 고객 서비스를 위해 24시간 컴포넌트들을 항시 대기시키고 있죠. (봉고차 역할을 합니다.) 물론, 벤더마다 다를 수는 있지만, Persistence, Transaction, Security와 EJB 인스턴스들에 대한 생명주기(Lifecycle)를 관리해주고 있죠. EJB 서버는 컨테이너와 약간의 역할 분담(벤더마다 다를 수 있다.)을 하면서, descriptor의 내용과 환경 변수들에 관한 사항들을 포함한 컨테이너와 외부 환경의 중간 역할을 하고 있죠.



각설하고 일단, 요거는 기억하고 넘어가세요. EJB 아키텍처라고 할만한게 있는데, 그 안에는 EJB server, EJB container, Home object, EJB object, Enterprise bean, Descriptor들이 있고, 이들은 서로 적절한 관계를 유지하고 있다고. (클린턴이 말했던, 부적절한 관계는 전혀 없습니다.)

Session Beans

그럼 세션빈을 만들어볼까요? 세션빈을 만들 때 어떤 세션빈을 만들지를 고민하시고 나서, 세션빈을 먼저 만들고, 나중에 인터페이스들을 정의해도 아무런 문제가 없어보이죠? 특히, 세션빈을 실제 구현하다보면 두개로 나누고 싶거나 필요한 서비스를 더 구현하고 싶으실겁니다. 이때 인터페이스를 만들지 않았다면, 문제 될 것이 없죠? 완벽한 Bean 클래스를 먼저 만들고, 인터페이스를 만든다... 설계를 대충하고 나면 그런일이 비일 비재하죠... 그런데, 이거 싸움납니다. 큰 프로젝트의 경우 여러 패키지를 각 팀별로 나

누어 구현하게 되고, 각각의 팀들은 다른 팀이 만들고 있는 컴포넌트를 통해 어떤 정보를 얻어오거나, 어떤 서비스를 대행 시키기도 합니다. 이 때 다른 팀에서 만드는 컴포넌트의 서비스가 요리조리 변하거나, 급기야는 제공하던 서비스말고 다른 서비스를 제공하게 된다면? 예를 들어, 조직에 관련된 컴포넌트를 A 팀이 만들고 있고, B 팀에서는 그 컴포넌트를 사용해서 여러 업무를 처리하고 있었습니다. 그런데, A 팀이 만든 컴포넌트 A-com을 통해 조직 정보를 얻어올 때, 어제까지만 해도 조직의 이름을 넘겨주더니, 오늘은 조직의 코드를 넘겨주는 겁니다. 아무런 통보도 없이. 물론, 이쪽의 컴포넌트는 아무런 동작도 할 수 없습니다. 이름을 기반으로 모든 로직을 만들었었는데,... 그런데 A 팀의 팀장이 대빵입니다. 그래서, B 팀은 코드를 바꾸어야 했습니다. 5명이서 3일동안 날밤 깠습니다. 그런데 하늘도 무심하시지, 이번에는 개발중에 조직의 변화가 있었습니다. 강북/강남 체제였던 조직의 구조가, 지역화 된 것입니다. 조직 내부의 춘추전국시대로 접어든거죠. 따라서 코드 체계가 다 바뀌었습니다. A 팀에서도 난리가 났습니다. B 팀에서는 아무것도 할 수 없었습니다. 왜냐하면, A 팀의 컴포넌트에서 어떤 서비스를 받을 수 있을지에 대한 아무런 정보도 얻지 못했기 때문입니다. 인터페이스가 변하는지, 생기는지, 없어지는지,... A 팀에서는 열심히 컴포넌트를 만들고 있지만, B 팀은 개점 휴업상태입니다... 조직 컴포넌트 때문에, 그야말로 조직의 쓴맛을 보고 있습니다. 이게 몸니까(뭡니까의 어눌한 표현)?

작은 프로젝트는 사실 관리상의 어려움이 별로 없습니다. 하지만, 프로젝트가 커지면, 방법론적 접근이 매우 절실해집니다. 미리 좋은 인터페이스를 만드는 것은 어렵습니다. 하지만, 다른 팀과 함께 큰 프로젝트를 수행한다면, 많은 시간을 들여서 다른 팀에서 안정적으로 사용할 수 있는 인터페이스를 먼저 만들어 주는 일은 매우 중요한 일입니다. 그런데 컴포넌트 기반의 프로젝트가 작은 프로젝트에는 큰 의미가 없습니다. 따라서, 가능하면, 큰 프로젝트에서도 활용할 수 있는 프로세스가 필요합니다. 사실, 앞에서도 팀간의 절차적인 문제에 대해 심각하게 다루어야 할 이슈들이 많았었는데 언급하지는 않았습니다. 관리적인 이슈이니까요. 개발자분들께서는 관심없습니다. 아니 싫어하죠. 이번에도 넘어갈 까 하다가, 하나 정도는 말하고 지나가는 것이 좋겠다 생각이 들었습니다. 그런데, 이런말 나오면, 길어집니다. 쌓인 것도 많으니, 할 말도 많아여라... 하지만, 결국은 흙으로 돌아가는 인생이거늘... 할말무상 싸움회의. 똑똑똑 똑 똑 똑 똑 또르르르르... COM기반의 프로젝트도 마찬가지입니다. 물론, 프로젝트 중간에 인터페이스가 바뀔 수도 있습니다. 하지만 최소화하는데 노력을 기울여야 합니다. 개발자들이 간과하기 쉬운, 컴포넌트 기반 프로젝트의 생산성이 달려있는 문제였습니다.

에고, 삼천포가 아니라 오천포 정도 빠졌네... 다시 시작해보죠. 뭐부터 만든다고요? 인터페이스요. 네 정답입니다. (여기서, 잠깐, "코드" 레벨의 인터페이스가 아니어도 좋습니다. 외부 사용자 들을 위한 API에 대한 설명을 적절하게 명세화하면 그걸로 OK입니다. 인터페이스 코드를 먼저 만들어야 된다는 말은 아니지요. 물론, 코드를 먼저 만들어도, 상관 없습니다.)

본격적으로 시작하기 전에, 잠깐 세션빈이 갖는 두가지 기본 형태인 Stateful과 Stateless에 대해 간단히 설명드리겠습니다. 항간에, Stateless Session 빈은 메소드로만 구성된다라는 루머가 떠돌던데, 그건 아니고 "빈 내의 호출이 일어난 후, 다음 호출 때까지 상태를 유지하지 않는다."는 것 뿐입니다. 즉 필요한 변수를 파라메터로 받아 메소드를 수행하

고, 적절한 결과를 리턴하고는 무슨 일이 있었나 능청을 떨고 있죠. 따라서 재사용이 가능하게 되는겁니다. 이에 반해 Stateful Session 빈은 클라이언트 애플리케이션의 연장이라고 생각하시면 됩니다. 즉, 클라이언트가 요청한 서비스를 수행하고 그 클라이언트와 관련된 상태를 유지하게 되는거죠. 그렇다면, 쇼핑카트같은 놈은 Stateless로 구현해야 할까요, 아님 Stateful로 구현해야 할까요? 클라이언트가 요기조기 돌아다니면서 물건들을 지속적으로 담고 다니죠? 클라이언트 종속이죠? 그래서 Stateful이 적당합니다. 고객의 ID를 받아서 그동안 쇼핑했던 내역을 보여주는 기능에는 어떤 형태가 어울릴까요? 그냥 ID따라 DB에서 쇼핑내역을 갖고와서 클라이언트로 던져버리죠? 단발적이죠? 그럼 Stateless가 적당합니다. (여기서 저는, 적당이라는 표현을 사용했습니다.)

그럼, Home interface부터 알아보도록 하지요. 앞에서 보았던 코드를 다시 볼까요?

```
// Filename: HelloHome.java
import javax.ejb.*;
import java.rmi.RemoteException;
public interface HelloHome extends EJBHome
{
    public HelloObj create () throws RemoteException;
}
```

코드에서 보시는 HelloHome 인터페이스처럼, 모든 Home 인터페이스는 javax.ejb.EJBHome로부터 상속받아야 하며, 적절한 create 함수들을 선언해야 합니다. (Stateless일 경우에는, 인수 없는 create 함수만 존재합니다.) Entity Bean의 Home 인터페이스라면 적절한 find 함수들을 선언해놓아야겠지요? 그런데, 모든 Home 인터페이스들이 상속받고 있는 EJBHome은 무엇일까요? 요놈을 보면, Home 인터페이스의 정체를 좀 더 파악할 수 있겠지요? 여기서는 부모 클래스인 EJBHome을 간단히 보고 넘어가도록 하겠습니다.

```
public interface javax.ejb.EJBHome extends java.rmi.Remote
{
    public abstract EJBMetaData getEJBMetaData() throws
java.rmi.RemoteException;
    public abstract HomeHandle getHomeHandle() throws
java.rmi.RemoteException;
    public abstract void remove(Handle handle)
throws javax.ejb.RemoveException, java.rmi.RemoteException;
    public abstract void remove(Object primaryKey)
throws javax.ejb.RemoveException, java.rmi.RemoteException;
}
```

EJBHome은 분산환경에서 서비스를 하는 놈이죠? 그래서, Remote를 상속하는군요. 그

외, EJBHome 인터페이스 안에 선언된 몇개의 함수들이 보이는 군요. 그 중에 remove(Object primaryKey)는 Entity Bean만을 위한 것입니다. 그런데, 필요한 경우, EJBHome의 getEJBMetaData를 통해 EJBMetaData를 얻을 수도 있습니다. 그럼, 이 EJBMetaData라는 놈은 무엇일까요?

```
public interface javax.ejb.EJBMetaData
{
    public abstract EJBHome getEJBHome();
    public abstract Class getHomeInterfaceClass();
    public abstract Class getRemoteInterfaceClass();
    public abstract boolean isSession();
    public abstract Class getPrimaryKeyClass();
}
```

이 인터페이스를 통해 여러 정보를 얻어올 수 있군요. Home 인터페이스, Remote 인터페이스, PK까지. 그러나 고민하실 필요 없습니다. 스펙을 구현하는 벤더측에게는 매우 중요한 인터페이스이지만, 애플리케이션 개발 단계에서 이 인터페이스를 활용할 경우는 별로 없습니다. 아니, 전혀 안써도, 몰라도 암시롱 안하지요. 단지, 내부적으로 EJB 프레임웍이 어떻게 움직이는가를 이해하기 위해서는 필요한 정보들이겠지요. 아는게 힘!
그럼, 다음은 Remote Object를 알아보기로 합시다.

```
// Filename: HelloObj.java
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface HelloObj extends EJBObject
{
    public String getMessage() throws RemoteException;
}
```

보시는 것처럼, 모든 Remote Object는 반드시 EJBObject로부터 상속을 받아야합니다. 그럼, 이쯤해서 전체적인 상속 또는 구현 구조가 궁금해져야 합니다. EJB1.1 스펙의 82페이지를 보세요. 전체적인 상속 또는 구현 구조를 이해하실겁니다. 어쨌든, Remote Object의 정체를 파악해보기 위해서는 EJBObject를 알아보아야겠군요.

```
public interface javax.ejb.EJBObject extends java.rmi.Remote
{
    public abstract EJBHome getEJBHome() throws java.rmi.RemoteException;
    public abstract void remove() throws java.rmiRemoteException,
javax.ejb.RemoveException;
    public abstract Handle getHandle() throws java.rmi.RemoteException;
    public abstract boolean isIdentical(EJBObject eo) throws
java.rmi.RemoteException;
```

```
    public abstract Object getPrimaryKey() throws java.rmi.RemoteException;
}
```

여기서 isIdentical(EJBObject eo)과 getPrimaryKey()는 Entity Bean 만을 위한 함수선언입니다. 나머지들은 모두, Bean의 타입과는 상관없이 사용할 수 있는 놈들이구요. 마찬가지로 개발하시면서, 이들 함수들을 직접 호출할 필요는 없습니다. 참고로, EJBHome과 EJBObject 모두 Remote를 상속하는군요.
앞에서 우리는 두가지의 핸들을 보았는데요, HomeHandle을 사용해서 JNDI를 통하지 않고도 Home 객체와의 연결을 재설정할 수 있고, Handle을 사용하면 EJBObject와의 연결을 재설정할 수 있답니다. 핸들을 어떻게 구현하는가의 문제는 스펙을 구현하는 벤더측에 따라 달라질 수 있지만, 구현 메카니즘을 몰라도 사용하는데는 지장 없겠죠?
Home과 Remote 인터페이스들을 보았고, 이제 세션빈을 볼 차례죠? 자. 다음은 앞에서 보았던 코드죠?

```
// Filename: Hello.java
import javax.ejb.*;
public class Hello implements SessionBean
{
    public void ejbCreate() throws RemoteException, CreateException {...}
// getMessage()만 신경써서 구현하고, 나머지는 남들이 만든부분을 보고, 적당히 베낀다. ^^
...
}
```

여기에서 주목할 것은 Bean이 SessionBean을 구현하고 있다는 점입니다. 자, 그럼 SessionBean의 정체를 알아볼까요?

```
public interface SessionBean extends EnterpriseBean
{
    public abstract void setSessionContext(SessionContext ctx)
            throw java.rmi.RemoteException;
    public abstract void ejbPassivate() throw java.rmi.RemoteException;
    public abstract void ejbActivate() throw java.rmi.RemoteException;
    public abstract void ejbRemove() throw java.rmi.RemoteException;
}
```

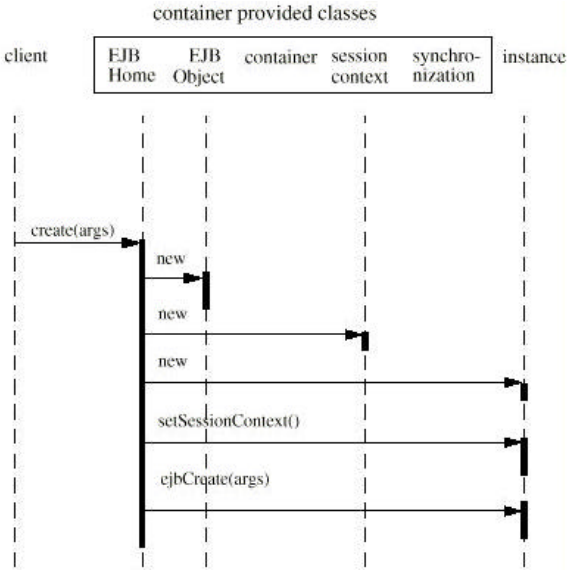
어랄라? 앞에서 보던 것들과는 다른 놈들이 몇개 보이죠? 이들중 setSessionContext (SessionContext ctx)는 매우 중요하답니다. 이놈이 바로 EJB 빈과 컨테이너와의 연결고리입니다. 또한, 요놈을 통해 transaction과 security 상태를 알 수도 있습니다. 조금 더 자세히 보고 넘어갈까요? 이 함수 선언에 하나의 파라메터가 보이시죠? 컨텍스트에는 SessionContext와 EntityContext 의 두개가 있는데 이들은 모두 EJBContext라는 놈으로부터 상속을 받고 있습니다. 그럼 EJBContext 의 생김새를 잠깐 보실까요?

```
public interface EJBContext
{
    public EJBHome getEJBHome();
    public principal getCallerPrincipal();
    public boolean isCallerInRole(String role);
    public UserTransaction getUserTransaction();
    public boolean getRollbackOnly();
    public void setRollbackOnly();
}
```

getCallerPrincipal로부터는 java.security.Principal를 얻어올 수 있고, isCallerInRole 함수를 통해 인스턴스를 호출하는 클라이언트의 역할을 확인할 수 있게됩니다. getUserTransaction을 통해서 새로운 트랜잭션 서비스를 얻어오면, begin, commit, roll back 등을 자체적으로 처리할 수 있습니다. 또한, getRollbackOnly을 통해 RollbackOnly 여부를 검사할 수 있고 setRollbackOnly를 통해 트랜잭션내에서 처리되는 모든 작업이 Rollback되도록 설정할 수 있습니다.
쩝. 슬슬 어려워지기 시작하죠? 조금 전에 SessionContext 가 컨테이너와 빈과의 연결고리라고 했습니다. 증명해보이겠습니다. 우선 빈을 만드는 사람은 SessionBean을 구현하겠지요? 그 중 setSessionContext라는 함수를 구현할때 개발자는 다음과 같은 코드를 작성하게 됩니다.

```
public void setSessionContext(SessionContext sc)
{
    this.sc = sc;
}
```

개발이 끝나고 배치가 되고 EJB 빈들이 메모리에 올라오면, 컨테이너의 내부에 있는 EJBHome 객체는, 클라이언트로부터 create(...)라는 메시지를 받자마자, EJBObject 객체와 SessionContext 를 하나 만들고 나서, 해당 빈 인스턴스를 하나 만듭니다. 그후 그 인스턴스의 setSessionContext(나따끈따끈한세션컨텍스트)를 불러줍니다. 그리고나서 그 인스턴스의 ejbCreate(...)함수를 호출하게 되는 것이지요. 그리고는 EJBObject의 레퍼런스를 반환하게 되고, 클라이언트는 그 리모트 객체를 통해 빈 인스턴스의 서비스를 얻을 수 있게 되는 겁니다. 대충 이해가 되었으면, 스펙을 잠깐 볼까요?



스펙에는 create뿐만 아니라 여러 다른 메시지 호출에 대한 흐름도 자세하게 나와있답니다. 참고해주세요. 물론, 실제 상황은 위의 그림처럼 단순하게 나타나지는 않습니다. 왜냐하면, 벤더들마다 나름대로 풀(Pool)을 구현하게 되고, 그 메카니즘에 의해, 클라이언트의 요청이 있을때마다 빈 인스턴스를 만드는 대신, 풀에 미리 왕창 만들어 놓았던 놈들 중 만만한 놈 하나를 갖고오게 되는 겁니다. 물론 어떤 놈을 만만하게 볼 것인가는 내부 구현 방식에 따라 다르겠죠? "깐데 또깐다" 방식 또는 "안깐데만 골라깐다" 방식 등에서 말입니다.

자. 여기서 세션빈에 대한 설명은 줄이도록 하겠습니다. 사실, Stateful과 Stateless로 구분해서 생명주기를 포함한 다양한 설명을 드리지 못함을 안타깝게 생각합니다. 다음에는 좀더 많은 내용으로 인사드리도록 하지요.

다음에는 엔터티빈입니다. 이미 많은 것을 설명했으니까 다음에서 설명할 것은 별로 없겠지요? 글쎄요... 그럴까요? 어쨌든 넘어가 보도록 하지요.

Entity Beans

자바에서는 현재, 두가지 방법으로 persistence를 지원하고 있습니다. 첫번째는 Object serialization으로 자바 객체를 Byte stream으로 변환하여 저장하거나, 가져오는 방식입니다. 두번째는 자바 객체를 관계형 데이터베이스에 저장하는 방식입니다. 이때 JDBC를 이용하지요. 엔터티빈은 이중 두번째에 해당하겠죠? 맞습니까? 조용...

원래 엔터티빈은 EJB 1.0에서 필수적 요건이 아니었습니다. 그러나, 데이터베이스와의 연계는 기업의 애플리케이션 개발에 있어 너무나 중요한 부분이기 때문에 1.1에서는 바로 필수적인 부분으로 스펙을 바꾸었습니다. 아시겠지만, java 진영에서는 R-DB를 신주모시듯 합니다. 자바를 서버사이드 솔루션으로 자리잡게 해준 주요 요인들 중 하나이니까요. 따라서 엔터티빈도 R-DB를 근간으로 합니다. 혹시, OODB를 쓰시거나, OR 매퍼를 쓰신다면, 이 부분을 우습게 또는 같잖게 보셔도 상관없습니다. 하지만 기본을 중요하게

생각하신다면 결코 지나칠 부분이 아닙니다. 우러러 보셔야지요.
엔터티빈은 R-DB 속의 테이블에 해당합니다. EJB 스펙 2.0 까지는 엔터티빈 하나당 테이블 하나가 매핑됩니다. (향후, 하나의 엔터티빈이 여러 테이블의 내용을 포함하게 될지도 모르겠습니다. 엔터티 빈에 관한 스펙에, Inheritance 또는 Aggregation을 도입한다면 말이죠. 그렇게 되면 더욱 신나는 일들이 많이 생기게 되겠죠?) 엔터티빈의 인스턴스가 포함하는 내용은 한 테이블 속의 레코드에 해당합니다. Home 인스턴스에는 하나 이상의, 다양한 find 함수들이 선언되게 되는데, 이들 함수들의 구현은, 에러가 없다면, 하나 또는 여러개의 레코드들에 대한 내용을 전달하게 됩니다. 이때 구현을 개발자가 직접하게 되면, BMP(Bean-managed persistence)가 되고, 툴에게 맡기면 CMP가 된다고 앞에서 이미 말씀을 드렸습니다.
자. 그럼 엔터티빈의 홈 인터페이스와 리모트 인터페이스는 앞의 세션빈에서 설명한 것으로 대신하고, 바로 빈의 정의로 들어가도록 하겠습니다. 모든 엔터티빈들은 EntityBean라는 인터페이스를 구현해야합니다. EntityBean의 모습을 한번 볼까요?

```
public interface EntityBean extends EnterpriseBean
{
    public abstract void ejbActivate() throw RemoteException;
    public abstract void ejbPassivate() throw RemoteException;
    public abstract void ejbLoad() throw RemoteException;
    public abstract void ejbStore() throw RemoteException;
    public abstract void ejbRemove() throw RemoteException;
    public abstract void setEntityContext(EntityContext ctx)
throw RemoteException;
    public abstract void unsetEntityContext() throw RemoteException;
}
```

setEntityContext와 unsetEntityContext 보이시죠? SessionBean과는 차이가 있지만, 비슷한 맥락이라고만 이해하시고 넘어가도록 하겠습니다. 대신 EntityContext 인터페이스만 잠깐 살펴 보도록 하겠습니다. 왜냐하면 요놈이 getPrimaryKey라는 매우 중요한 함수를 갖고 있기때문이죠. 기억해두시기 바랍니다.

```
public interface EntityContext extends EJBContext
{
    public EJBObject getEJBObject() throw IllegalStateException;
    public Object getPrimaryKey() throw IllegalStateException;
}
```

자. 앞에서 본 EntityBean의 내부에 ejbLoad와 ejbStore 함수가 있었죠? 이 두함수가 바로 데이터베이스에서 레코드를 읽어오거나, 저장을 하는 역할을 한답니다. CMP의 경우에는 이 함수들을 구현할 필요가 없겠죠?
자. 그럼 본격적으로 들어가보죠. 먼저, BMP(Bean-managed Persistence)입니다. 일반적

으로 JDBC를 이용하며 insert, select, delete와 같은 SQL 문을 직접 사용해서 데이터베이스와 상호작용을 하게 됩니다. 사실, 이 부분이 CMP 보다는 복잡하지만, 많은 개발자들에게 오히려 친숙한 부분일 것같아 먼저 살펴보도록 하겠습니다. 즉, 많은 코드를 직접 작성하는 불편함은 있지만, 데이터베이스 관련 개발에 대한 경험이 많다면 CMP 보다 효과적인 빈을 만들 수 있겠지요.

매우 기본적인, 하지만 중요한, ejbCreate 함수부터 살펴보도록 하지요. 여기서 create의 의미가 무엇일까요? 앞에서 하나의 엔터티빈 인스턴스는 하나의 레코드를 의미한다고 했지요? 그럼 당연히 새로운 레코드 하나를 추가하는 것이겠지요? 따라서 create에서는 파라메터로 넘어온 인수들을 바탕으로 작성된 insert 문이 실행되게 됩니다. 이 ejbCreate 함수가 수행되고나서 ejbPostCreate 함수가 실행되는데, 개발자들께서는 무조건 요걸 구현하셔야합니다. 워쩐 코드를 작성하냐구요? 생각나는 것이 없으면 그냥 코멘트라도 넣어주세요. "나 ejbPostCreate 불렀다..."

다음은 ejbRemove인데요. 컨텍스트에 getPrimaryKey 함수가 중요하다고 앞에서 말했지요? 그런걸 어떻게 다 기억하냐구요?... 하기싫음 말구... 어쨌든, 프라이머리 키를 구합니다. 글구, 그놈을 이용해서 delete문을 하나 이쁘게 만들어 ejbRemove안에 놓아두심 됩니다.

ejbLoad는요? getPrimaryKey 함수가 중요하다고 앞에서 말했지요? 왜 했던 말 자꾸 되풀이하냐구요? 쩝, 중요하니까. 어쨌든 기억하심 됩니다. 자. 코스는 전과 동! 프라이머리 키를 구한다. 키를 바탕으로 select 문을 만들어 넣는다. 끝. 그럼 ejbStore를 유추해보세요. 똑같습니다. 키 얻어오고, 단지 달라지는 것은 update 문이 들어가겠지요?

여태 중요한 사항들을 보아왔는데, 지금부터는 더 중요합니다... 엔터티빈에 대한 Home 인터페이스에는 find 함수들이 선언 됩니다. 어떤 형태의 find 함수를 얼마나 많이 만들것인가는 철저히 비즈니스 니즈에 따라 결정될 문제이지요. 이쯤에서 재미있는 규칙을 하나 말씀드리지요. Home인터페이스에 findByXx라는 함수를 선언했다면, 빈 클래스에는 ejbFindByXx라는 함수를 반드시 구현해야 합니다. 이때 findByXx의 리턴은 빈인스턴스에 대한 리모트 객체이고, ejbFindByXx의 리턴은 PK객체가 됩니다. (CMP의 경우 findByXx라는 함수가, Home 인터페이스에 선언되어 있다면, ejbFindByXx라는 함수가 자동으로 만들어집니다.) 워메, 하나도 재미 없어야... 헷갈리기만 하구. 재미 없음 말구.

find 함수는 하나의 레코드만 갖고 오거나, 여러개의 레코드를 반환하게 됩니다. 여러개의 레코드를 반환할 경우, PK 객체에 대한 enumeration 이나 collection에 담아주어야 합니다. 예를 하나만 보고 넘어갈까요? 다음은 주어진 PK에 맞는 인간 하나를 얻어오거나, 어느 정도의 인간성을 갖고 있는 인간들을 얻어오기 위한 Home 인터페이스 선언입니다.

```
public interface InganHome extends javax.ejb.EJBHome // 홈 인터페이스야요.
{
...
    public Ingan findByPrimaryKey(InganPK pk) throws FinderException,
RemoteException;
    public Enumeration findByInganseong(int level) throws FinderException,
RemoteException;
}
```

다음은 Home 인터페이스에 대응되는 BMP 빈 클래스의 구현입니다.

```
public interface InganHome extends javax.ejb.EJBHome // 홈 인터페이스야요.
{
...
    public InganPK ejbFindByPrimaryKey(InganPK pk) throws FinderException,
RemoteException
{
    …
    ps = con.prepareStatement("select name from Ingan where name = ?");
    ps.setString(1, pk.name);
    …
}
    public Enumeration ejbFindByInganseong(int level) throws FinderException,
RemoteException
    {
            …
    }
}
```

자, 그럼 CMP를 해보실까요? 앞에서 만든 ejbFind... 함수들을 구현할 필요 없이, 필요한 함수들만 Home 인터페이스에 선언한다. 끝. 뿅. 짠. 이렇게 간단한 만큼 나쁜 점이 있겠지요? 예를 들면, 데이터베이스와 관련된 코드가 없으니까 Portability가 좋다든가, 컨테이너의 특성을 관리할 수 있기 때문에 Implicit use of cursors, Optimized queries 등이 가능하다든가,... 옴메 나쁜 점이 아니라 오히려 좋은 점들이네. CMP를 애용하세... 하지만, BMP도 나름대로 필요한 경우가 있겠죠?
그런데, 질문 있습니다. 네, 말총머리 학생. ejbCreate도 안만듭니까? 앗, 이렇게 예리할 수가. 네... 사실은 CMP라 해도 꼭 정의해야하는 함수가 하나 있습니다. 바로 ejbCreate 함수죠. 하지만 구현 코드는 BMP와 완죤히 다릅니다. 볼까요?

```
public class InganEJB implements EntityBean // Only 4 EJB 1.1
{
    public String name;
    public int level;
    private EntityContext ctx;
    …
    public InganPK ejbCreate(String name, int level) throws CreateException
// EJB1.0에서는 RemoteException을 던져야한다.
    {
            if(name == null)
            {
```

```
                    throw new CreateException("You. Idiot! The name is
required.");
                }
                this.name = name;
                this.level = level;

                return null;
        }
}
```

오잉? 좀 이상하죠? 파라메터로 넘어온 값들을 이용해서 내부 변수를 초기화 하죠? 성공적으로 코드를 수행했는데도 null을 리턴하죠? 옴메, 맘에 안들어뿌리어... 으미, 답답해 뿐진거. 왜 이렇게 하냐구요? (일러야지.) 스펙에 그렇게 하라고 그랬대요. 참고로 EJB 1.0에 ejbCreate는 return문이 필요없는 void로 되어있었습니다. 다음번 스펙 EJB 3.0에는 어떻게 바뀔까요? Sun도 "이정현"을 좋아하는 것임에 틀림없습니다.

자. 진짜로 끝났습니다. 더 이상의 질문이 없는 관계로, 이상 CMP 였습니다. 저... 질문있는데요. 네, 말씀하세요. CMP인데, ejbRemove, ejbLoad, ejbStore 함수가 구현된 것을 본 적이 있는데요. ejbCreate만 구현하면 된다는 그 거짓말 정말이예요? ... 으예리 하신분이 또 계시군요. 학(급하면 발음이 이렇게 됨, 원래 발음은 학, 아니 확)실히 말씀드리지만 ejbCreate만 구현하면 됩니다. ejbRemove는 보통 비워둡니다. 코딩을 안하셔두 됩니다. ejbLoad, ejbStore의 경우에도 코딩을 안하셔도 되지만, 만약, 데이터베이스와의 연동을 하기 전에 초기화 작업이 필요한 경우에는 코딩을 하셔야겠지요. 즉, 보통의 경우 아무런 코딩도 필요없지만, 코딩 내부적으로 이상한 일들을 하는 경우(가끔 있습니다. 예를 들면 특별한 인코딩과 디코딩을 하는 경우.)에만 필요한 코딩을 넣으시면 되는 거죠. 절대 추천사항 아닙니다. 네, 그럼 더 질문 있으신 분들은 조용히하세요. 네 더 이상 질문이 없군요.^^ 이상 Entity Beans였습니다.

Transactions in EJB

좀 더 많은 IT 지식을 전달하려는 의도와 좀더 쉽고, 전반적인 큰 틀을 이해시키고자 하는 의도와의 조율이 쉽지 않군요. 하지만, 한가지 확실한 것은 이 글을 통해 모든 것을 이해한다는 것은 모래알을 통해 인생의 신비를 깨우치려는 것과 대동소이라는 것이지요. 또한 현명한 독자는 "자신에게 맞는 지식을 알아서 취사선택"할 것이라는 점입니다. 따라서 이 우매한 필자는 현명한 독자를 기대하면서 주제를 Transaction으로 옮겨갈까 생각합니다.

트랜잭션 부분에 대한 설명에 단골 손님으로 등장하는 메뉴가 있죠? 뭡니까? 인출과 잔고 정리죠? 즉, Asynchronous transfer mode의 ATM이 아닌 Automatic teller machine으로서의 ATM을 통해 자기 계좌(Account) 내의 현금을 인출하면(withdrawing), 잔고(balance)에서는 같은 금액이 감소가 되겠죠? (돈만 빼고 잔고는 그대로였음 좋겠죠?) 이 때 이 모든 과정들은 반드시 하나의 트랜잭션으로 묶여져야 합니다. 그러면서 트랜잭션의 속성으로 ACID라는 고전적 개념을 등장시킵니다. 간단히 설명을 드리면, 여기서

A(atomic)는 트랜잭션의 모든 처리가 적용되든지, 아니면 모두 적용되지 않든지 둘 중의 하나를 의미하며, C(consistent)는 트랜잭션에 연관된 데이터에 논리적으로 올바른 변경을 가할 수 있는지를 의미하고, I(isolated)는 중간 상태에서는 다른 어떤 트랜잭션도 트랜잭션 데이터를 볼 수 없다는 것을 의미하고, D(durable)은 트랜잭션이 끝나고 난 후에는 하드웨어 또는 소프트웨어 실패가 발생하더라도 그 결과에 영향을 주지 않는다는 것을 뜻합니다. 하지만, Isolation level에 따라 이들 속성이 어느 정도 바뀔 수 있습니다. 따라서 보다 정확하게 트랜잭션을 제어하기 위해서는, 사용되는 트랜잭션의 유형에 따라 필요한 속성들을 적절하게 할당할 필요가 있습니다. 예를 들면, 금융권의 회계데이터 변경과 같은 중요한 트랜잭션의 경우에는 퍼포먼스도 중요하지만, 데이터의 정확성이 무엇보다 중요하기 때문에 Isolation level을 serializable로 정하고, 선거 현황을 보기위해 들어온 사용자 접속수의 변경과 같은 별로 중요하지 않은 트랜잭션의 경우, 정확성보다는 퍼포먼스를 위해 read uncommitted 또는 read committed로 속성을 정하는 것이 바람직할 것입니다. 이때 주의할 것은 데이터베이스들 또는 버젼 별로 서로 다른 Locking level 정책을 갖고 있기 때문에, 이들 사항을 미리 점검하셔야합니다. 퍼포먼스에 많은 영향을 주기 때문이죠. 더 자세한 사항에 대해서는 데이터베이스 매뉴얼 또는 전문서적들을 참고하시기 바랍니다. 참, 트랜잭션에 대한 기본적인 명령어로, 트랜잭션의 시작을 알리는 begin, 트랜잭션의 변경 부분 등을 DB에 적용하기 위한 commit, 변경된 사항을 되돌리는 rollback이 있죠?

자, 그럼 트랜잭션에서 EJB와 관련된 것들에는 어떤 것들이 있을까요? CMT(Container-managed transaction)과 BMT(Bean-managed transaction)가 있습니다. BMT는 직접 트랜잭션을 코드 내부에서 제어하는 것으로 javax.transaction. UserTransaction을 사용합니다. 요놈을 사용하면, JSP에서도 트랜잭션을 처리할 수 있구요, Session Bean에서도 BMT를 사용할 수 있습니다. CMT를 사용할 경우, 모든 트랜잭션에 대한 제어는 Deployment descriptor에서만 가능합니다. CMT에서는 java.sql.Connection 인터페이스의 commit, setAutocommit, rollback과 같은 함수들을 사용할 수 없으며, javax.transaction. UserTransaction 인터페이스를 사용할 수도 없습니다.

그럼, CMT부터 살펴볼까요? CMT는 말 그대로 컨테이너가 트랜잭션을 관리하는 형태로, 세션빈이나 엔터티빈 모두에서 사용 가능합니다. 이때 Deployment descriptor 안에는 각 메소드에 대한 트랜잭션 속성들을 표시하게 됩니다. 트랜잭션 속성들이 뭐냐구요? 네, 성질도 참 급하십니다. 나옵니다. 나와요. 트랜잭션 속성으로는 NotSupported, Supported, Required, RequiresNew, Mandatory, Never의 6가지 인데요. 알고 보면 별거 아닙니다. NotSupported은 트랜잭션 지원 안할래, Supported는 지원은 할래, Required 난 트랜잭션이 필요해, RequiresNew 항상 나만의 트랜잭션을 가질래, Mandatory 트랜잭션 없는 놈이 날 부르면 Exception, Never 트랜잭션 있는 놈이 날 부르면 Exception. 즉, [Not]Supported는 트랜잭션 안에서 수행될 것인가를 정하는 속성이고, Requi*는 무조건 트랜잭션 내부에서 실행되지만, 별도의 트랜잭션이 필요한가를 정하는 속성이고, Mandatory와 Never는 부르는 쪽의 트랜잭션 지원 여부에 따라 수행 또는 Exception을 발생시키는 속성입니다. descriptor의 내부를 잠깐 들여다 볼까요?

```
<ejb-jar>
    …
    <transaction-type> Container </transaction-type>
    …
    <Container-transaction>
            <trans-attribute> Required </trans-attribute>
            …
</ejb-jar>
```

그런데, CMT에서 트랜잭션과 관련된 함수를 쓸 수도 있습니다. EJBContext의 getRollbackOnly와 setRollbackOnly 함수들이 그렇죠. 즉, 트랜잭션이 정상적이지 않을 경우, 강제로 rollback을 시키는 함수가 setRollbackOnly이고, 그 여부를 알아보는 함수가 getRollbackOnly 함수입니다. 기본적인 것들은 언급이 된 것같군요.

이젠, BMT입니다. 뭐, BenchMarking Test는 아닙니다. 사실, 트랜잭션 처리에 익숙하신 개발자들의 경우, 이 부분은 오히려 대충 보고 넘어가셔도 될 것같군요. 직접 모든 것을 제어하는 것이 익숙한 경우... (트랜잭션 말고 , 데이터베이스 애플리케이션 개발 경험이 많으신 분들은 Entity Bean을 만들면서도, BMP에 DAO를 만들어 사용하는 경우는 감이 팍팍오는데, CMP의 경우는 뜬 구름 같아서 필(feel)이 잘 안온다고 하시더군요...)
앞에서 CMT의 장점을 잠깐 보았는데, BMT의 장점은 무얼까요? 딱, 하나 있습니다. "트랜잭션을 내 맘대로 제어할 수 있다." 그게 전부입니다. 따라서 잘 생각하셔서 쓰셔야합니다. 더구나, 엔터티빈은 BMT 할 수 없습니다. 해보았다구요? 네, EJB1.0에서는 그랬습니다. 스펙 1.1에서는 달라졌습니다.^^ 페이지 165지를 보시면, The enterprise bean with bean-managed transaction demarcation must be a session bean. 특히, Stateless bean의 경우에는 함수 종료 전에 반드시 커밋을 해야한다고 규정하고 있습니다. A stateless session bean instance must commit a transaction before a business method returns. EJB 2.0 스펙 337 페이지에는 조금 더 추가된 내용이 있군요.^^ Session bean뿐만 아니라, Message-driven bean도 BMT가 가능하다고 나와있군요. Message-driven bean은 자바 메시지 서비스를 지원하기 위한 빈으로, JMS message를 처리하도록 설계되었습니다. 왜, MQSeries나 MSMQ 있잖아요. 그 일환이죠. 제 개인적인 생각인데, 당분간은 EJB 스펙이 계속 변경/추가될 것 같군요.^^
우선 트랜잭션의 상태를 나타내는 값들을 소개하도록 하겠습니다.

```
STATUS_ACTIVE                STATUS_COMMITTED,
STATUS_COMMITTING,           STATUS_MARKED_ROLLBACK,
STATUS_NO_TRANSACTION,  STATUS_PREPARED,
STATUS_PREPARING,            STATUS_ROLLEDBACK,
STATUS_ROLLING_BACK,         STATUS_UNKNOWN
```

이놈들은 javax.transaction.Status라는 인터페이스에 정의된 상수들입니다. 가끔, 인터페

이스를 이와 같은 용도로 쓰기도 한답니다. 시스템 전반에 걸친 상수들의 정의에 편하지요. UserTransaction의 getStatus함수가 바로 이들 값을 리턴합니다. STATUS_ACTIVE는 트랜잭션이 시작되서 트랜잭션 매니저가 Two-Phase 커밋을 시작하기 전인 상태일 때 리턴됩니다. STATUS_COMMITTED는 트랜잭션이 커밋된 것을 의미하겠죠? STATUS_COMMITTING는 커밋 진행중, STATUS_MARKED_ ROLLBACK는 setRollbackOnly를 호출한 경우이고요, STATUS_NO_TRANSACTION은 트랜잭션 이전 또는 완료 이후겠구요, STATUS_PREPARED는 Two-Phase 커밋에서 첫번째 상태가 완료된 것을 의미합니다. STATUS_PREPARING는 첫번째가 진행중이겠죠? STATUS_ROLLEDBACK 롤백되었군요. STATUS_ROLLING_BACK는 롤백이 진행중이군요. STATUS_UNKNOWN는 그외 나머지의 경우입니다. 재미 엄청없죠? 사실 BMT는 실제 코딩으로 제어해보시면, 그게 답니다. 여태까지 한 것들은 단지 참고적으로 알아둘 필요가 있는 사항을 열거한 것에 불과하죠. 이것으로 BMT를 마치도록 하겠습니다.

CMT, BMT, CMP, BMP 그리고 Session beans, Entity beans. 게다가 각종 속성들. 만만치 않죠? 프로젝트를 시작하기 전에 전체적으로 모든 것을 파악할 필요는 없다고 생각됩니다. 모든 것을 알기에는 분량이 많아지고, 알아도 자꾸 변하고, 벤더들 마다 조금씩 다르고, 따라서 프로젝트가 시작되고 구체적인 환경이 정해지면, 그 주어진 범주 내에서 보다 자세한 사항을 파악하시는 것이 바람직하다고 생각됩니다. 따라서 여기에서 제시한 것들은 전체를 설명한 것이라기 보다는 향후, 어떤 사항들을 점검해야 할 것인가에 대한 감을 갖는데 약간의 도움이라도 드리기 위한 것이었습니다. 도움이 되셨기를... 최소한, "난 안할래.", "조금 어렵네." 또는 "별것 아니네."등의 판단은 세우셨죠? 임무 완성!

EJB Security

Security를 설명하기 전에 보통 Authentication(Identity), Authority, Integrity, Privacy, Auditability를 언급하고 지나갑니다. Authentication는 인증된 사용자인지, Authority는 어떤 권한을 갖고 있는가에 대한 사항이죠. Integrity는 허용된 방법으로 데이터들이 변경되는 것을, Privacy는 권한 받은 곳에서 허락된 형태로만 데이터를 볼 수 있다는 것을, Auditability는 향후 분석을 위해 기록을 유지하는 것을 의미합니다.

EJB Security를 설명하기 전에는 다음과 같은 용어를 미리 설명하고 지나갑니다. User, Entity, Principal, Role에 대해서 말이죠. User는 사용잡니다. EJB 빈에 대한 사용자죠. 시스템이 될 수도 있고, IP 주소가 될 수도 있죠. Entity는 접근 권한을 가질 수 있는 것들을 의미합니다. 즉, An entity is something that can have access rights applied to it. Principal은 권한이 부여될 수 있는 entity를 의미합니다. Role은 권한들의 모임을 의미합니다. 자. 이렇게 설명하면 무슨 소린지 알아들을 사람이 없을 것은 불보듯 뻔합니다.

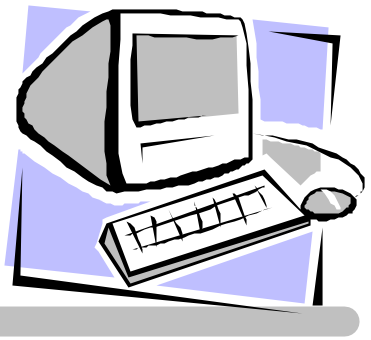
자, 그럼 EJB Security를 시작하겠습니다. 우선 사용자를 걸러야죠? 사용자는 다양한 방법으로 관리됩니다. *.properties 형태로 등록되어 애플리케이션 서버가 기동 되면서 사용자로 인증될 수도 있고, 데이터베이스에 저장되어 있을 수도 있고, Active Directory에 만들어져 있을 수도 있겠죠. 하여튼 다양합니다. 일단 로그인을 하면, 사용자를 관리하는 메카니즘에 의해 패스워드를 확인하겠죠? 패스워드가 맞다면, 다음은 무엇을 해야할

까요? 웹 환경이라면, 못먹는 쿠키(Cookies)를 사용해서 SessionID를 저장하고, 인증된 사용자가 동일한 브라우저를 사용해서 서비스를 요청하는 한, 동일 인물로 가정합니다. 그 후, URL을 통해 내부 화일을 직접 요청할 경우를 대비해서, 매 요청때마다 SessionID를 검사하고, 틀릴 경우 여지없이 "너 나쁜놈이지."라는 에러메시지를 보내줍니다. 이때 클라이언트와 서버 사이에 SSL(Secure Socket Layer)는 매우 잘 알려진 안전한 통신 방법이죠. 최근, TLS(Transport Layer Security) 개발되었는데, 아직은 많이 활용되는 단계는 아니군요. 다음은 사용하는 환경에 따라 많이 다를 수 있는데, 여기서는 WebLogic 5.1을 사용한 예입니다. 참고로, WebLogic ACL은 JavaSoft ACL standard(java.security.acl)을 사용해서 구현되었습니다. 사용자를 적절한 그룹에 할당하고, JDNI 컨텍스트에 대한 ACL 등을 정의합니다. 예를 잠깐 볼까요?

```
...
weblogic.password.grady=morning
...
weblogic.security.group.mighty=grady
...
weblogic.allow.lookup.weblogic.jndi.applications=mighty,...
weblogic.allow.execute.weblogic.servlet.*.jsp=mighty,...
...
```

다음은 ejb-jar.xml에 해당 role과 허용하는 빈의 method이름을 적습니다. 이제 남은 것은 딱 하납니다. 즉, getCallerPrincipal과 isCallerInRole아라는 함수를 사용하는 거죠. 권한이 필요한 메소드를 실행할 경우, principal을 얻어오고, 해당 role을 갖고있는지를 검사한 다음, 모든 것이 OK면 실행하고, 그렇지 않으면, 시스템 무단 사용에 대한 적절한 보복조치를 행하면 됩니다.

내용은 많지 않지만, 실전 경험이 가장 없으신 부분으로 알고 있습니다. 모쪼록 실전 프로젝트에서 좋은 EJB Security를 구현하셔서 탄탄한 시스템을 구축하시길 바랍니다.



Design Patterns in Application

우선 밝혀두고 싶은 것은, 디자인 패턴이라는 것은 제가 만든 것이 아닙니다. 어렵다던가 잘 이해가 안되면, 만든 사람의 잘못이라는 것을 먼저 말씀드립니다. 정말입니다. 제가 만들지 않았어요... 그래도 저는, 패턴을 어떻게 설명해야 피부에 와 닿을까 고민은 했습니다. 우선 먼저 느껴보시죠.

패턴이라는 것은 어디에나 나타나는 것입니다. 예를들어, "초이스"패턴이라는 것을 먼저 볼까요?"초이스"했을 때 커피가 연상되신다구요? 순진도 하셔라... 여기서 초이스란 카드게임(포카는 정식 명칭이 아니랍니다.)에서 처음 3장의 카드를 받아들고 이들 중 오픈할 카드를 한 장 결정하는 것을 말합니다. 이때 돈따기 위한 심리적 작태를 초이스패턴이라 합니다. 즉, 처음 손에 든 3장 중 같은 무늬가 두장이 있으면, 이거 절대 안냅니다. 따라서 오픈한(바닥에 깔아 논) 1장의 무늬는 손에든 무늬와는 관계가 없거나, 드물게 3장 모두 같은 무늬라는 거죠. 그런데, 저희들(꾼들) 끼리만 통하는, "뽀뿔(Four Flush)은 노인정"이라는 말이 있습니다. 즉, 액면(널려진 카드) 네장이 모두 같은 무늬라면 오히려 플러시로 인정을 안한다는 거지요. 그 이유는 초이스 패턴에 입각해서, 손에 처음 든 것도 3장 같은 무늬, 그 후 연달아 3장 같은 무늬가 떨어질 확률은? 무쟈게 적죠. 그래서 액면에 같은 무늬가 4장이나 깔렸지만 물패로 본다는 겁니다. (참고로 저의 학부 때 전공은 계산통계학이었습니다. 응용확률론 A+였습니다. 게다가 마장에서 통계학 실습 엄청했습니다. 그런데 왜 하필 "노인정"이냐구요? 아. 그건 "No 인정."입니다.) 물론 이경우 딸린 예외가 몇가지 있죠. 상대방이 빵빵한(보통 7이상) 원페어일 경우, 무늬와는 상관없이 초이스할 수도 있다. 오디가 아닌 하이로 또는 애니 게임에서는 변형된 패턴이 있다... 이런거는 돈주고 배우세요. 더이상은 절대 안가르쳐줍니다.

자. 앞에서 우리는 초이스 패턴을 배웠습니다. 패턴을 알면 좋습니다. 돈땁니다. 이와 같은 패턴을 실전에 응용해보세요. 돈됩니다. 다른데서도 - 고스톱이나 당구, 슬럿 머신... - 찾아보세요. 정말 많습니다. 패턴을 습득하신 분들은 게임 후에 웃으면서 나오실 확률이 매우 높지요. ("인생은 게임이다."라는 말도 있던데...) 딱. 하나만 더 해볼까요? Gamma에 의하면 디자인 패턴에는 Creational Patterns, Structural Patterns, Behavioral Patterns의 3가지 부류가 있지만, 당구의 세계에는 알다마 패턴, 쿠션 패턴, 포켓볼 패턴의 3가지 형태가 있습니다. (물론 예술구 패턴도 추가해볼 수는 있지만, 제 영역이 아니라서.) 그 중 하꼬(짧은 쿠션 돌리기) 패턴을 예를 들어볼까요? 짧은 하꼬를 살살치면 또 나온다. 이게 힘

없어서 안맞아도 최소한 공은 잡는다. 응용1. 우라 패턴과의 연결. 나미로 약간 세게 치면 우라를 만들 수 있다. 우라를 치면 또 우라를 만들 수 있다. 응용2. 알다마 패턴과의 연계. 반절 두께로 적절히 힘조절하면 반대편 구석에 공들을 모을 수 있다. 등등. (참고로 저는, D사 시절 500이었습니다. 하지만 지금은 물입니다. 400도 힘겹습니다.) 마찬가지입니다. 디자인 패턴들끼리도 연계 또는 복합적 구성이 가능하기 때문에, 잘만하면 환상의 패턴 복식조가 탄생할 수도 있습니다. 알다마와 쿠션을 적절히 몰고 다니듯, 디자인 패턴을 자유로이 구사할 수만 있다면? 무지 좋겠네... 그럼 오늘부터 당구장에서 패턴 연습을...@@;

죄송합니다. 다시, 제정신 차려 계속하겠습니다. 정신 차려. (열중 쉬어...) 디자인 패턴이라는 것은 소프트웨어 개발 단계에서 도메인에 대한 파악이 대충(?) 끝나고, 기본적인 컨셉추얼 모델이 만들어진 상황에서, 프로그래머를 위한 구체적인 모델을 만들 때 적용될 수 있는 "패턴"입니다. 이거 많이 알고 계시면, 프로그래머와 웃으면서 대화할 수 있습니다. 관리자와 개발자간의 팽팽한 심리전이 와해되는 순간이죠.

99년 6월부터 8월까지, 한솔PCS에서 작업했을 때, C4O라는 그 당시 저의 개인적 방법론을 적용했었습니다. 그당시 제가 만들었던 최종 산출물로는 프로젝트 개요, SW 개발 방법론 개요, 유즈케이스 기술서, 클래스 다이어그램들, 시퀀스 다이어그램들, 컴포넌트 다이어그램, 배치 다이어그램, 테스트 시나리오, 테스트 문서 등이 있었습니다. 이 때 ITPlus에서 저보다 연배가 높으신 프로그래밍 경력 "빵빵" 베테랑 Developer 두분과 한솔PCS에서 데이터베이스와 Java, 그리고 OT에 특히 관심이 많으신 두 분과 함께 작업했었습니다. 그 당시만해도, 프레임웍 구축이 아닌 애플리케이션 개발에서는 디자인 패턴이 큰 의미를 갖지않는다는 조금 떨어진(?) 마인드를 갖고 있던 제게, 일부 "디자인 패턴의 적용"은 한솔PCS 측에서 강제로 요구한 사항이었습니다. 에구구. 저보다 더 잘 아시더라구요...

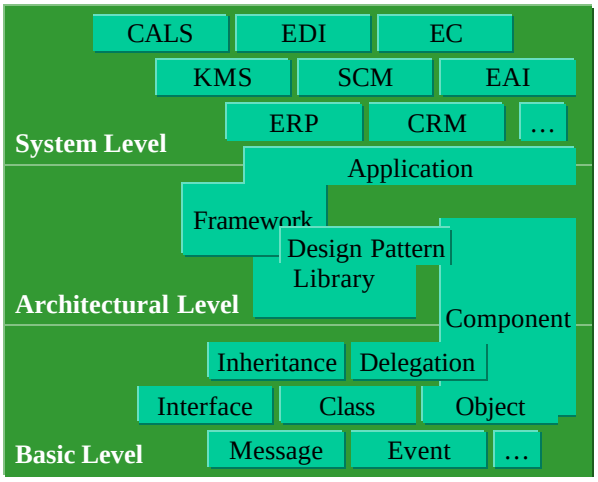
다음은 그 때 적용했던 패턴들 입니다. 열거해보면, Factory Method Pattern, Reflection Pattern(Multi-Lang, Meta-Env.), MVC Pattern (Presentation-Abstraction-Control Pattern), Broker Pattern, 그 외 Interface-Implementation, Whole-Part, Delegation, 그리고 Inheritance 등을 여러 형태로 적용했었습니다.

99년 9월부터 2000년 11월까지 O1 Korea에서 했던, 프로젝트도 디자인 패턴 측면에서 보았을 때 크게 다를 바는 없습니다. 오히려 더 아껴서 썼습니다. TopLink라는 ORMapper를 사용했기 때문에 EntityBean을 사용하지 않았었고, 이 때문에 패턴이 결부될 여지가 더욱 줄어들었었었었죠. 참고로, 두 프로젝트 모두 Rational Rose, WebLogic, Java, Oracle을 사용했었었었었습니다.

더 자세한 사항은 해당 기업의 입장을 감안해 못보여드림을 죄송하게 생각합니다. 아, 게다가 기억이 안나네요... (안타깝게도, 제게 청문회 증후군이 가끔 있습니다.)

그럼. 디자인 패턴을 본격적으로 이해해보기 전에 디자인 패턴의 위치를 먼저 파악해볼까요? 그럼, 전체적인 시스템요소를 나누는 3가지 레벨을 먼저 제시하겠습니다. 1. Basic Level, 2. Architectural Level, 3. System Level. 전체적인 시스템 요소들을 이렇게 3가지 부류로 나누어보자는 거지요. 물론 이들 레벨은, 철저하게 저의 개인적인 시각에 의해 나

눈 것입니다. 다른 책들에서는 더욱 방대하고 거창한 체계를 보여주기는 하지만, 여기에서는 제가 아는 한도 내의 개념 정리 하에서 설명을 드리도록 하겠습니다.



먼저, **Basic Level**에는 Class, Object, Event, Message, Inheritance, Delegation, Interface 등이 있습니다. 대부분 아시는 것이라구요? 다행입니다. 어? 잘 모르시겠다구요? 제가 94년도에 만든 이후 수십차례 써먹어온 노래방 메타포어를 한번 보셔야되는데... 별거 아닙니다. 압축버전으로 해볼까요? 노래방에 놀러가면, 우리 모두와 노래방 기계들은 객체이구요, 노래를 부르는게 기계에 대한 메시지입니다. 점수는 응답이구요. 주인아저씨가, 끝났다고 나가라고 하면, 이벤트구요. 노래부르기를 두려워하는 사람들이 하는 행태가 델리게이션입니다. 사람들의 취향은 폴리모피즘 때문이구요. 어? 그럼 클래스는? "용피리 노래방"은 객체이구요, "노래방"은 클래스예요. 좀, 유식하게 말하면, 플라톤이 말하던 "이데아"가 클래스구요, 그 밑에 있는 실세계 것들이 객체랍니다. 인터페이스는 기본적으로 "뒤에서 호박씨 까는"걸 의미하지만 그 이상의 의미를 가진, 가장 대단한 놈입니다. 즉, 준비되어 있지도 않으면서, 일단 할 수 있다고 말로(IDL) 저질러 놓고, 나중에 별별 희한한 주먹구구식 땜빵(Implementation)을 내세울 수 있는 놈이죠. 조심하셔야 합니다. 하지만, 무림(분산 컴포넌트 환경)의 세계에서 살아남기 위한 가장 기본적인 초식에 해당합니다. OMG산하 무당파에 속하든지, 소림사(Sun)에 속하든지, 사파(MS)에 속하든지 암튼 아셔야합니다. 다시 말해볼까요? "거하게 낸다."는 Interface구요, "노래방에서 논다."는 Implementation입니다. "단란주점에서 논다."는 또 다른 Implementation입니다. 심한 경우는 "붕어빵으로 해결한다."도 있을 수 있습니다. 노래방이나 단란주점이나 돈드는데(유흥업소)는 매한가지죠? 이때 유흥업소와 노래방/단란주점과의 관계를 Inheritance라고 합니다. 특히 간판에 "XX유흥업소"라는 것이 없죠? 따라서 유흥업소와 같은 것들을 Abstract Class라고도 부르지요. 노래방은 대부분 콘크리트 건물이죠? 그래서 노래방 같은 것들을 Concrete Class라고 합니다.

Architectural Level에 가면 좀 더 복잡한 놈들이 있습니다. Library, Design Pattern, Component, Framework 등이 있지요. 도서관에 가면 대충 빼껴서 레포트를 낼 수 있는 소

스들(책)이 많지요? 요놈들이 모두 Library에 해당합니다. (진짜 라이브러리네?) 주위를 잘 찾아보면, 논문이나 레포트 "삐끼"들이 있는데 돈만 조금 주면, 미리 만들어진 다양한 레포트나 논문을 통째로 살 수도 있습니다. 큰 논문들은 매우 비싸죠... 이런 것들을 컴포넌트라고 하지요. 선거때에는 조사를 빙자한 일부 후보 홍보도 해주고, 길거리에서 설문지도 돌리고, 각종 통계치도 뽑아주고, 적군들에 대한 정보도 염탐해오고, 흑색을 포함한 온갖 잡색들이 동원되는 선전을 하죠? 이처럼 다양한, 하지만 일관된 목적의 서비스를 제공하면, 프레임웍이 됩니다. 디자인 패턴은 말씀드렸죠?

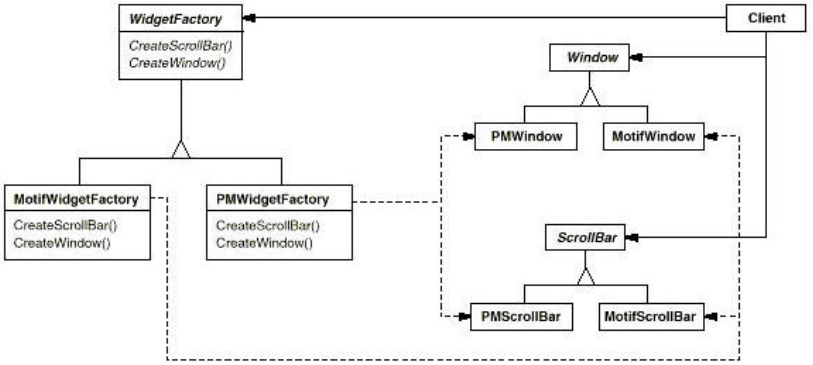
System Level에는 다양한 애플리케이션들이 들어있습니다. 또한 ERP/XRP, DW, CRM/e-CRM, SCM, KMS, EAI, ... 처럼 쪼잔한 놈들도 있고, EDI, EC, CALS와 같이 뜬 구름 잡는(요즘 많이 잡아가고 있죠?) 놈들도 있습니다. 이 부분은 제가 잘 모른다는 사소한 이유 말고도, 현재 이글에서 언급하고자 하는 부분과 조금 거리가 있다는 중요한 이유 때문에 언급을 회피하겠습니다.

디자인 패턴이 어디쯤에서 오골오골 있는지 아시겠지요? 이제 본론으로 들어가겠습니다. 본론에서 디자인 패턴에 대해 하나 하나씩 자세히 설명할 줄 알았죠? 읽는데만 3박 4일 걸립니다. 쓰는데는? 3박 400일정도 걸립니다. (397일밤 새야합니다.) 그래서 자세한 디자인 패턴을 알고싶으신 분들은 많은 참고서적을 보아주시기 바랍니다. 디자인 패턴은 그동안 많은 논문이나, 컨퍼런스에서 취급해왔던 오래된 이슈입니다. 패턴 관련한 많은 책들도 나와있습니다. 그 중 딱 하나를 고르라면, Gamma의 "Design Pattern"이죠. 말밥입니다. 저는 잘 모르지만, 매우 좋은 책이라고들 합니다... 왜, 그거 있잖아요. 이해하기 어려운 것은 좋은 책이라고 치부하는... 그럼 안되는데...자 그럼. 보시죠...
그럼. 다 보셨죠? ^^ 아직 다 못 보신 분들은, 이글을 다 읽고, 시간을 내서 마저 보기로 하죠.

설명과 부합되지는 않지만 일반적으로 애플리케이션이라면 위의 그림과 비슷한 구조를 일부 포함하고 있을겁니다. 이런 전체적인 그림을 머리에 넣으시고 다음을 계속봐주세요.

제 생각에 J2EE 기반하의 프로젝트에, 쉽게(?) 적용할 수 있는 디자인 패턴에는, WebTier 에 Facade Pattern, Factory Pattern, Singleton Pattern, Reflection Pattern, MVC Pattern, Observer Pattern, Mediator Pattern 등이 있겠구요, EJBTier에는 Facade Pattern, Factory Pattern, Singleton Pattern, Reflection Pattern 등이 있겠네요.

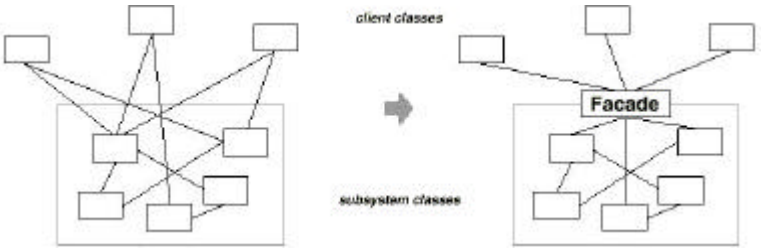
Factory Pattern



애플리케이션 구축에서 항상 어디에서든지 쓸 수 있는, 아니 써야 하는 패턴이 Factory 패턴입니다. 일단, 만들어야 무언가 할 수 있겠죠? 하지만, 복잡한 형태의 Factory Pattern 은 무언가 잘못되었거나, 지나친 디자인의 결과라고 생각됩니다. 단순한 형태의 Abstract Factory Pattern 정도가 무난할 것으로 보입니다. 다양한 환경에서의 불특정 다수가 쓰기 위한 프레임웍이라면, 당연히 다양함을 표현해야하지만, 대부분의 애플리케이션 구축에 있어 컴퓨팅 환경이 거의 고정적이기 때문입니다. Abstract Factory Pattern 의 자세한 부분은 Gamma 의 책, 87페이지에 있습니다. ^^

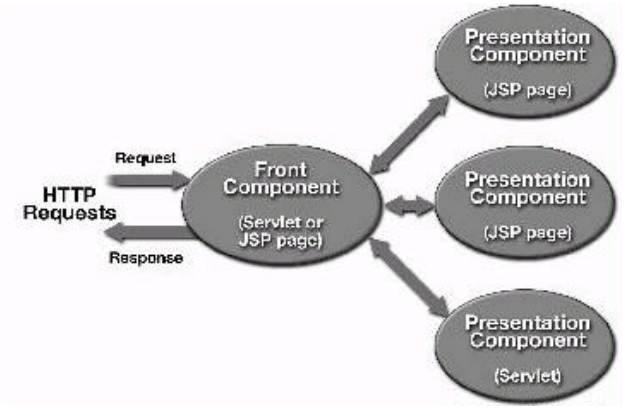
위의 그림을 보고 아무생각이 없으시면 빨리 잊어버리시고 다음으로 넘어가세요. 후다닥.

원리는 위에서 잠깐 언급한 유흥업소 있죠? 즉, 언제든지 놀고싶다는 생각이 들면 그 곳에 정통한 친구에게 물어봅니다. "어디 갈까? 나 돈 얼마 있는데." 그럼 그 친구는 주머니 형편에 따라 적절하게 가는 장소를 말해주는 패턴입니다. 직접 고생을 안해도, 새로운 장소 또는 물 좋은 곳에 대한 정보를 항상 알 수 있지요. 물론 그런 똑똑 친구를 두어야 삶이 행복하겠지요? 그럼, Abstract Pattern을 만드세요. 클라이언트의 상황에 필요한 서비스를 제공할, 적절한 객체/컴포넌트를 얻을 수 있을 겁니다. 참고로 EJB Home 객체한테 JNDI 이름(스트링)을 주면, 원하는 서비스를 전달해 줄, Remote Object의 레퍼런스를 얻을 수 있지요? 이때 사용된 패턴은 Factory Method Pattern과 Remote Proxy Pattern이랍니다. 즉, Home 객체의 create 함수를 통해 리모트 객체의 레퍼런스를 얻어오는 것은 Factory Method. 실제 객체가 아닌 Proxy 가 넘어오지요? 이것이 Remote Proxy Pattern이지요. 이거 잊어버리세요. 기억하려고 애쓰시면 건강에 부담됩니다.(여기에서는 단순히 Remote 환경에서의 Proxy 라는 것에만 초점을 두었습니다. Remote Object가 smart reference, transaction, security, instance pooling 등과도 연계되어있으니 만큼, Remote Proxy Pattern 은 부적절한 용어라는 지적이 있었습니다.)



Facade Pattern

그 다음은 위에서 보는 Facade Pattern인데요. 발음은 "퍼싸드" 정도 입니다. 요놈은 WebTier와 EJBTier 모두에 중요합니다. 그리고 제 생각에 무조건 써야합니다. WebTier에 하나, EJBTier에는 업무 별(또는 워크플로우 별) 하나씩 써야합니다. WebTier에 쓰이는 Facade는 Dispatcher라는 놈입니다. (Blueprint에는 FrontComponent로 표현 되어있습니다.) 클라이언트 입장에서는 내부의 다양한 Servlet, JSP를 구지 부르지 않아도 됩니다. 이 Dispatcher 하나만 알면 됩니다. Dispatcher는 Request Parsing만을 할 수도 있지만, 유효한 User인지 식별하고, Session을 관리하는 역할을 포함할 수도 있습니다. 잘 안와닫는다구요? 특정한 초기 화면 없이 서비스만을 위한 모듈들만 있다고 가정하면, index.html, default.html가 될 수 있는데 그 안에는 뒤에 있는 *.jsp를 부르는 로직이 있는, 그런 파일입니다. ...있잖아요. 돈만 조금 주면, 알아서 부킹 다해주는 웨이터 역할입니다. 다음 그림을 참조하세요.

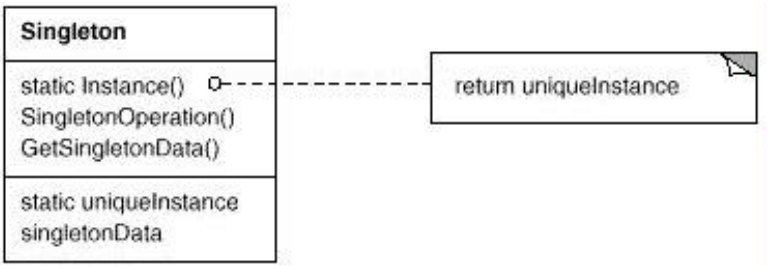


EJB 티어에는 Session Bean이 Facade 역할을 합니다. MS의 MTS(Microsoft Transaction Server)를 쓰는 경우, 어쩔 수 없이 Stateless이지만, EJB를 쓸 경우, 선택의 폭이 다양해집니다. 즉, Stateless, Stateful의 두가지 중에 하나를 선택할 수 있지요. 이런 부분은 대개의 경우 논란의 여지가 있습니다. 그래서 누가 똥고집이냐에 따라 선택하면 된다고 보면 틀림없습니다. 단지, 저는 권고합니다. 특별한 이유 없으면, Stateless를 쓰세요. 무지하게 효과적입니다. 퍼포먼스가 중요합니까? 이거 쓰세요. 하지만, 워크플로우가 생명인 업무는 Stateful을 쓰셔야합니다. 클라이언트의 특정 상황을 목숨걸고 지켜야합니까?

Stateful을 쓰셔야합니다. 흥분해서, 약간 말이 빗나갔는데요. Facade의 역할이란, Session Bean 내부에서 또 다른 Session Bean을 부르거나, Entity Bean 들을 부르는 것으로 결론 납니다. 즉, WebTier의 입장에서는 Facade에 해당하는 Session Bean 하나만 알면, 내부적인 상황은 더이상 고려하지 않아도 되는 Simplicity를 제공되는 것이지요. 하지만, 이것보다 정확히 48.123배 더 중요한 이유가 따로 있습니다. 클라이언트/서버 환경에서의 퍼포먼스 병목은 Local Call에 있지 않습니다. Remote Call의 수를 줄이는 방법이야말로, 최상의 비급이며, 더 이상의 초식이 존재할 수 없습니다. WebTier에서 서버단의 여러 Session Bean이나 Entity Bean을 건드릴 수는 있지만, 그렇게 하시면, 안됩니다. 즉, Local에 매우 많은 Layer 또는 역할을 수행하는 놈들을 만들어 서로 상호작용시켜도, 퍼포먼스를 잡아먹지는 않습니다. 하지만, 생각지도 않았던, Remote Call 하나가 엄청난 퍼포먼스의 저하를 초래합니다. Session Bean을 서버단의 Facade로 쓰세요. 참, Facade Pattern이 무엇이냐고요? Gamma의 책 185페이지입니다.

Singleton Pattern

Singleton Pattern은 사실 무소부재, 만방깔림입니다. 많이 쓰인다는 말이지요. 자기자신을 가리키는 Static 변수를 리턴하는 Static 함수를 하나 갖고 있어서 여러개의 객체를 만들지 않고 항상 같은 놈이 모든 서비스를 제공하는 유형입니다. 자세히 보고싶은 분은 Gamma의 책 129페이지입니다.



Singleton은 Fatory Pattern과 함께하는 특성이 있습니다. 참, Static이 나와서 말인데요. JDK에서 System, Runtime이라는 놈들이 있지요? 그런데 이런 놈들은 Constructor가 Private입니다. (무슨 말인지 모르겠습니까? 일반 사교 클럽이 아닌, 회원만 받는 ****입니다.) 그럼 어떻게 이런 놈들을 활용하나요? 크게 두가지가 있지요. 이런 놈들은 대개 Static 함수들을 갖고 있습니다. 따라서 객체를 만들지도 않고 그냥 쓰시면 됩니다. System.out.println(...) 그냥 허락없이 마구 쓰고 계시죠? 눈치 안보고 마구 사용하셔도 괜찮습니다. Static은 별도의 메모리에 있기 때문에, 객체하고는 도통 관계가 없다는 사실. 객체를 만들지 않아도 이 클래스가 참조될 때 VM에 로드되고, 실행되는거죠. 두 번째 방법은 Static함수로 된 Factory 함수를 쓰는 겁니다.(??) 예를 들면, Runtime rt = Runtime.getRuntime() 이런식이죠. 결국, 아는 사람들은 무사통과입니다. Singleton의 레퍼런스를 반환하는 함수들도 이와 같은 Static 함수입니다.
그럼 딱 2개의 인스턴스(또는 n개)만 만들고 싶을 때는? 상황에 따라 여러 형태의 인스턴

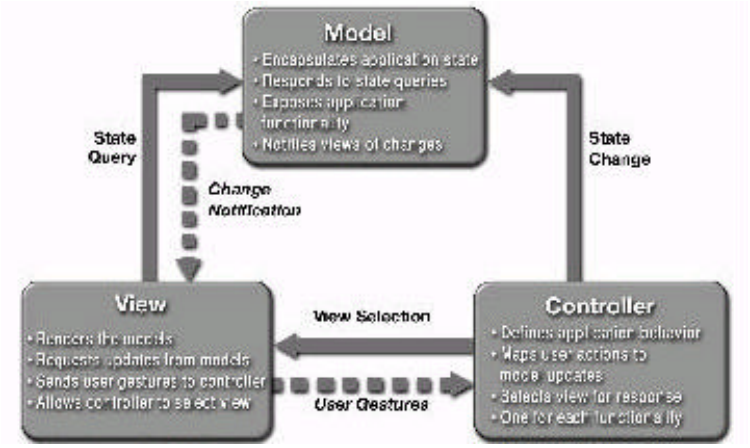
스가 필요할 경우는? Gamma의 책이 필요한 순간입니다. ^^

Reflection Pattern

다음은 Reflection Pattern입니다. Java언어에 클래스에 대한 메타정보를 보기도하는 Reflection있죠? 그것도 여기에 해당되겠네요. 이 것은 개발자들 보다는 많은 벤더들이 애용하는 패턴입니다. 예를 들면 *.properties 파일 많이 보시지요? 이 것이 일종의 Reflection Pattern입니다. 자세히 보시려면, 부시맨의 책 193페이지를 보세요. (Frank Buschmann과 그 일당이 쓴 POSA: Pattern-Oriented Software Architecture. 저는 아직 안보았는데요. 최근 나온 POSA2는 죽인답니다. 예술이래요.) 요즘, 서버, 컨테이너, 엔진, 컴포넌트 등을 위한 프로퍼티 설정으로 *.xml을 쓰지요? 이 것도 같지는 않지만, 비슷한 부류로 생각하셔도 됩니다. 중요한 것은 "매우 좋다."라는 것입니다. 한솔PCS에서도 멀티랭귀지 지원과 멀티 GUI 프레임웍을 위해 .properties를 활용했습니다. 형태는 다르지만, COM+ 컴포넌트들도 마찬가지 입니다. 개발과 특성 지정의 분리. 이거 중요합니다. 컴포넌트 졸업하시려면, 이거 학점 따야합니다. 하여튼 이 패턴의 핵심은 다이나믹하게, 컴포넌트의 속성을 제어하고, 메타정보를 준다는 것이지요.

MVC Pattern

에구 지겨워. MVC는 그냥 그림만 보고 넘어갈까요? 말이 필요 없습니다. (글은 필요하겠죠?)

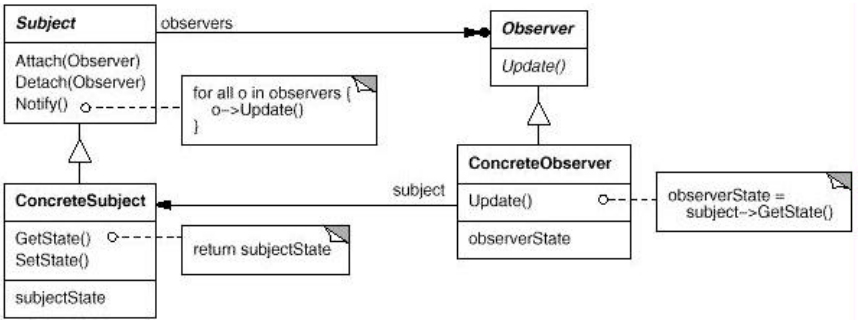


이건 Smalltalk 시절부터 사용되었던, 구석기시대적 유물 패턴이지만, 결코 시대 착오적이지 않은, 매우 중요한 패턴입니다. 물론 작금의 웹기반 분산환경에서는 확장된 MVC 패턴의 개념이 중요하지만, 기본적인 패턴을 먼저 이해하고 넘어가야겠지요? Smalltalk은 내부적으로 프레임웍을 갖고 있으며, Persistence를 언어 차원에서 제공하는 신기한 놈입니다. 원리는 사실 간단하죠. 내부적으로 메모리 덤프를 갖고 있습니다. 요 안에 몽땅 저장하죠...

어쨌든 위의 그림에서 보는 것처럼, 정보를 갖고 있는 Model이 있고, 보여주기 위한 View가 있고, 중간에 그들을 조정하는 Controller가 있지요. 사실 이 MVC 패턴은, 이후에 나온 많은 프레임웍의 아키텍처에 큰 영향을 주었습니다. 그리고, 개발자분들이라면, 여태까지 해왔던 그모습, 그대로이죠?

초기 웹환경처럼 기본적인 html 페이지들이 있고, CGI를 통해 DB를 연동하는 경우 거의 완벽한 MVC 패턴입니다. 즉, html 페이지들은 Views, CGI 프로그램들은 Controllers, DB 내의 테이블 들은 Models이죠. 하지만, 현재의 분산 환경에서의 모델들은 뷰와 매우 멀리 떨어져 있습니다. 따라서 대부분 메모리에 캐쉬된 VO와 같은 놈들이 모델의 역할을 하게 됩니다. Web tier의 경우, 여러 핸들러들은 모두 Controllers이구요, JSP 또는 Servlet들은 동적으로 View를 만들어내는 Controller의 역할을 하게 됩니다. 물론 Servlet은 순수한 Controller만의 역할을 하기도 하지요. 따라서 요즘에는 MVC 패턴이 다소 확장된 형태로 적용되고 있습니다.

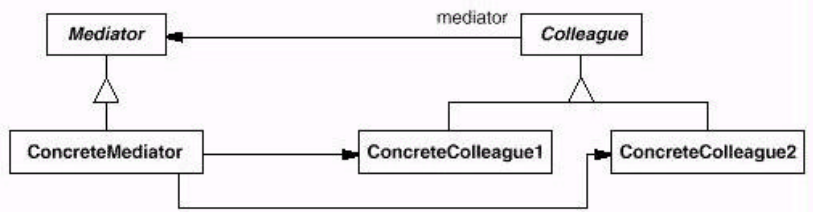
Observer Pattern



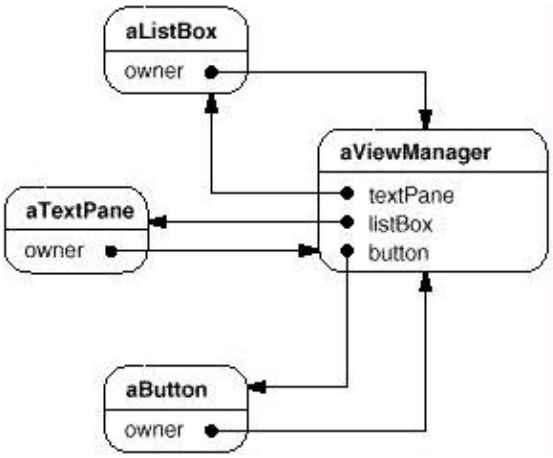
다음은 옵저버 패턴이군요.

옵저버 패턴은 개념만은 쉽습니다. 개념만 알고 넘어가자구요. 무언가 새롭게 정보가 바뀔 때마나 알려주어야 할 곳이 많다면, 일일이 돌아가면서 알려줄 것이 아니고, 알고싶은 놈이 알아서 자기를 서비스에 등록하고, 서비스를 받으라는 패턴입니다. 끝. 위의 그림 중에 왼쪽에 Subject 보이시죠? 거기의 Attatch와 Detatch를 통해 Observer 들이 등록되고 삭제됩니다. 요놈이 데이터를 갖고 있다가 또는 참조하고 있다가 내용이 변경되면, Notify 함수 보이시죠? 이놈이 등록된 Observer들을 불러 차례차례 Update시켜줍니다. 물론 이벤트에 대해서도 마찬가지로 동작합니다. 예를 들면, Applet 구현하실 때 ActionListener implementation하면서, addActionListener(this)하시죠? 이게 등록하는 거구요. actionPerformed(ActionEvente) 구현하시죠? 이게 Update 함수에 해당합니다. 이벤트가 생기면 리스너가 알아서 actionPerformed를 실행시켜줍니다.

이 옵저버 패턴이 이해되지 않으시는 분들은 3.7 건너뛰시기 바랍니다.



Mediator Pattern



다음은 위에서 보시는 Mediator Pattern입니다. 그림상으로는 쉬워보이시죠? 얘는 앞에서 설명한 Facade와 Observer의 짬뽕(우아한 표현으로는 칵테일)이라고 생각하시면 편합니다.

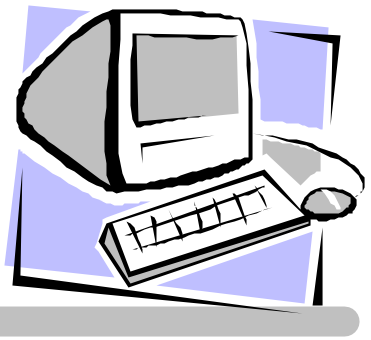
예를 들면, 화면에 버튼들이 있고, 콤보 박스와 리스트 박스가 있다고 생각해보세요. 요놈들이 내용에 따라, 또는 버튼 클릭에 따라, 다른 버튼이 disable되거나 콤보, 리스트의 내용에 변화가 있을 수 있겠지요. 즉, 여러 구성요소들이 있고, 이들간의 긴밀한 상호작용이 필요한 상황입니다. 그런데, 어떤 한 놈에게 어떤 이벤트가 발생할 때마다 다른 구성 요소들이 이 것에 대해 일사분란하게 움직여주려면, 모두가 서로를 알아야하는 상황에 이릅니다. 으악이죠. 구성요소 추가, 기능 추가 때마다 비명 나옵니다. 쨘. 이때 Mediator를 쓰세요. 각각의 컴포넌트는 Mediator에 등록하게 되고, Mediator는 특정 컴포넌트에 어떤 일이 발생하면, 다른 컴포넌트에게 적절한 메시지를 보내게 됩니다. 즉, 모든 구성 요소들에 대해 Mediator는 Facade입니다. 또한 Mediator는 등록된 놈들에게 적절한 메시지를 보내게 되지요. 옵저버 패턴에서의 Subject입니다.

화면에 여러 UI 컴포넌트가 있고, 이들이 혼연일체된 모습을 보여줘야 한다면. 이게 정답입니다. 하지만 이거 꼭 UI 아니어도 되겠죠?

사실, 처음에 Observer pattern과 Mediator Pattern은 거론하지도 않으려고 했었습니다. 하지만 맛보기로 패턴이 무엇인지를 설명하기 좋은 놈들을 골라 넣은 불순한 의도도 있습니다. 제, 개인적인 의견인데요, 아직 Design pattern을 적용해보신적이 없다면, 우선

Facade, Factory, MVC의 3 패턴을 우선 적용해보세요. 요놈들은 애플리케이션 개발에 있어서도 필숩니다. 천리길도 한걸음부터, 우쌰, 우쌰.

디자인 패턴이 무언지 본론도 뺐고, 그 많은 패턴들 중 지극히 일부만을 했을 뿐인데도, 넘 길어지네요. 더 이상의 내용은 다음을 기약하겠습니다. 이런 약속을 기약 없는 약속이라하죠. (기약했는데...) 아무튼 시간이 허락하는 대로 내용을 더 넣겠습니다. 아쉽죠? 하나도 아쉬울 것 없다구요? 쩝. - -; 호응 없으면, 절대루 안할래. 빳데루 당해도 안할래.



웹/컴포넌트 기반 소프트웨어 개발 방법론 정립에 관한 사족

개인적으로 말씀드리고 싶은 두가지 사족이 있습니다.

많은 사람들이 소프트웨어 개발을 설명하면서, 소프트웨어를 건축에 비유하고 있습니다. 훌륭한 메타포어입니다. 하지만 개인적으로 이 메타포어는 이미 공룡이 나돌아다니던 수백만년전의 유적에서 발견된 것이 아닌가 싶습니다. 제가 아는 한 전문가가 "이제는 적절한 메타포어가 아니다."라고 지적했던 것처럼, 건축이란 철저한 워터폴 모델이기 때문입니다. 제게는, "반복적/점증적 소프트웨어 개발이 필요하다."는 말이 너무나 자연스럽게 들립니다. 하지만 "XX 빌딩을 반복적으로 개발했다."는 등의 말은 아직 들어본 적이 없습니다. 월드컵 경기장 같은 커다란 건물을 지으면서도, 설계와 구축을 반복/점증적으로 하지는 않지요. (물론, 우리나라에는 이런 반복/점증 설계 전문가들이 많은 것으로 압니다. 실제는 이런 일들이 빈번하게 일어나서 엄청난 예산을 말아먹고 있다는 이야기를 듣고는 하지요. 고속철, 인천 어디 도로, 삼풍도 iteration 중에 변을 당했죠?...) 건축에 iteration을 적용하면 망합니다. 따라서 소프트웨어 개발은 단순한 건축, 그 이상의 개념이 필요합니다. 이 말을 하는 이유는 여기에서 보이고자 했던 것들 또한, 전체적인 건물의 아키텍처에서 일부 패턴에 대한 묘사에 지나지 않는, 그만큼 단편적인 것이라는 것을 강조하고 싶어서입니다. 그리고 요즘처럼 변화무쌍한 Biz World, IT World 속에서 적절한 메타포어의 선정은 우리들 모두의 숙제라고 생각합니다. 예를 들면 XP에서는 "...Driving is about constantly paying attention, making a little correction this way, a little correction that way." This is the paradigm for XP. ... Everything in software changes, The requirements change. The design changes, The business changes. The technology changes. The team changes ... 라고 말하고 있더라구요. 100점짜리 메타포어는 아니라는 생각이 듭니다. 우리나라 대부분의 청소년 여러분들, 운전면허 없으시죠? XP 이해하려고, 무면허 운전하시면 안됩니다... 꿈나무 프로그래머들도 쉽게 이해할 수 있는 메타포어가 필요합니다. 스노보드타기라든가...@@. 꿈나무 아닌사람들은 이해 못한다구요? 에구, 쉽지 않네. 뽀빠이 도와주세요.

또 다른 사족 하나는 방법론에 관한 것으로, "누구를 위한, 정말 써먹을 수 있는 방법론을 만들고자 하는 것인가?"에 대한 문제입니다. STEP2000 객체지향 방법론 개발에도 직접 참여했었고, 많은 기업들의 방법론들, 예를 들면, 현대 정보기술의 HSDM과 OOHSDM, ETRI의 마르미 II, 오라클의 인터넷 개발 방법론인 CDMi, HSDM의 모태였던 언스트&영의 네비게이터, 동부산업의 SDLC, PTC(Parametric Tech. Corp.)의 소프트웨어 개발 방법론, LG-EDS의 PDM 수행방법론과 객체지향 방법론 Concept&Guideline, Rational의 Objectory와 RUP(Rational Unified Process), SLC(System Life Cycle), XP(Extreme Programming) 등 다양한 방법론에 대한 검토, 또는 대충 목차라도 읽어본 경험이 있습니다. 또한 여러 OT, CBD에 대한 Guru들(Meyer, Odell, Lorenz, Graham, Booch...)의 방법론들도 훑어본 적이 있었습니다. (혀로 훑지는 않았습니다. 정말입니다!) 하지만, 파일럿 프로젝트가 아닌 기업에 필요한 소프트웨어를 개발할 때는 단 한가지의 방법론도 제대로 적용해본 적이 없습니다. 개인적으로, 물론 팀원들과 함께, 프로젝트 시작 전에 기본적인 이터레이션과 마일스톤들, 프로세스와 산출물 등을 정하고, 이전 프로젝트에서 그때 그때 만들어 사용했던 여러 Guide 들을 수정해서 지침, 템플릿, 가이드라인 등을 만들어 활용했습니다. 물론, 방법론을 몰라서 적용을 못한 것일 수도 있지요. 그러나 어떤 프로젝트이든지 당장 시행해야하는 프로젝트에 적합한 방법론은 단 하나도 없을 것(이라고 이 여린 소년 굳게 굳게 외칩니다.)입니다.

Jacobson도 "책 또는 방법론만을 보고 실제 프로젝트를 하려고 하는가?"하고 반문하고 있습니다. 물론 그가 능력이 없기 때문에, 단지 책만 보고도 훌륭하게 프로젝트를 감당해내는 이 땅의 많은 전문가들을 이해하지 못할 수도 있겠지요. 하지만 도메인과 프로젝트의 성격/규모 등과 상관없는 방법론은, 쓰기 힘든 방법론이라 생각합니다. (읽기만 쉬우면 된다구요?) 즉, 도메인과 프로젝트의 성격, 규모에 맞는 방법론을 적용하는 것이지, 방법론에 프로젝트를 맞추어서는 안된다고 생각합니다. Martin 은 "Business는 llnesses 요, 방법론은 Medicine이다. 다양한 병에 대해 다양한 처방이 필요하다."라고 했죠. 결국, 도메인에 정통한 많은 Biz 컨설턴트들이 전문 업무 영역을 정형화하고, 컴퓨팅 환경과 프로그래밍에 익숙한 Senior Developer들이 개발에 필요한 가이드라인과 템플릿들을 만들어야 그나마 적절하게 써먹을 수 있는 방법론이 만들어질 수 있지 않을까요?

매번 개발 프로젝트마다, 적절한 커스터마이징이 필요하다고 생각됩니다. 이는 그 조직이 방법론을 활용하기 앞서 그동안 어떤 프로젝트를 어떻게 수행해왔는지에 많은 영향을 받지요. 즉, 그 동안 그 조직에서 만들어왔던 산출물들과, 경험있는 도구를 활용하지 않으면 체계적인 방법론의 적용이란 매우 어려운 일이 될 것이라는 것이지요.

(사실, 지금 말씀드리는 부분은 매우 민감한 부분입니다. 부족한 저의 개인적 생각일 뿐이죠. 하지만, 이렇게 여러분들 앞에 표현했으니까, 틀린 부분에 대해서는, 고쳐주실 분이 많이 계시겠죠?)

자! 그렇다면, 누가 고양이 목에 방울을 달 것입니까? 웹 기반 엔터프라이즈 분산 환경에 적합한 메타포어의 설정과, 그때 그때 조직의 경험을 통한 방법론 커스터마이징을 기본으로 하는 방법론을 어떻게 만들 수 있을까요? 변화하는 IT를 포용하지 못하는 방법론은 단지 또 하나의 "국민교육헌장"을 만들 뿐 아닐까요? 누가 소프트웨어 개발을 시작하기 전에 국민교육헌장을 읽고, 가슴 깊이 새기고 싶어할까요? 물론 여태까지의 방법론이

그 이상의 의미라는 것은 알고 있습니다. 또한 기존의 방법론이 필요없다는 말도 아닙니다. 다만, 방법론은 Biz 컨설턴트들과 개발 현장에 있는 Senior Developer들의 경험과 지식이 "그 중심" 또는 최소한 "중요한 한 부분"을 차지하고 있어야 할 것이라는 점입니다. 그럼 결론을, Biz 컨설턴트들이나 개발 전문가들의 경험과 지식을 "컴포넌트 기반 소프트웨어 개발 방법론"에 넣는 것으로 해도 되겠지요? 결론 끝.

(참. 아주 사소한 문제가 하나 있기는 있네요. 많은 Biz 또는 Dev. 전문가들이 돈벌레라는 사실. 돈되는 일은 밤을 새워서라도 하지만. 돈 없이 쓰라고 하면 한 줄도 못쓰던데...
참. 아주 사소한 문제가 하나 더 있네요. 컨설턴트들과 개발자들의 경험과 지식들을 반영한다해도, Business가 계속 변한다던데. 더구나 비즈니스는 자고나면 변하지만, IT는 자기전에도 변하고 자고나서도 변하고, 안자고 버텨도 변하고... ((참고로 J2EE 1.3 버전의 내부에는 JDBC 2.0, EJB 2.0, Servlets 2.3, JSP 1.2, JMS 1.0, JTA 1.0, JavaMail 1.2, JAF 1.0, JAXP 1.1, Connector 1.0, JAAS 1.0 등이 있구요. 그외 JDK, BDK, JSWDK,...들도 버젼이 있고. 많은 관련 벤더 제품들도 계속 버젼이 바뀌고,... 으악.))그럼. 애써 만들어 놓은 방법론을 오늘 다시 만들어 놓으면, 내일 다시 바꾸어야 하고. 결국 지속적으로 갱신해야겠네요...
그렇다면, 결국, 방법론과 컨설팅 영역이 병행/분리되어야 한다는 이산가족의 아픔이...)

참. 덧붙이고 싶은 말이 또 있습니다. 그 것은 "깨달음과 아는 것과의 차이는, 아는 것과 모르는 것과의 차이보다 더 크고, 깊고 무궁무진하다는 것입니다." 저도 여태까지 조금 아는 것으로 마구마구 떠들어댔었습니다. 아직은 깨달음의 경지에 이르지는 못했습니다. 하지만, 이제는 깨닫고 싶습니다. 그래서, MCSE 듣고 있습니다. 네트웍이 넘 재미있어요. 개인으로 듣기 때문에 노동부 환급도 안됩니다.ㅠㅠ 그나마 야간이라 300만원밖에 (?) 안합니다. 고등학교 1학년생도 있더라구요. 무서운 세상입니다. 제가 아는 Study 조직이 있는데, RUP 합니다. 거기도 고등학생 있습니다. 이런 학생이 대학교에 들어가서 SE수업 들으면, "저기 교수님, 그거 제가 고등학교 때 해본건데요..." 회사에 들어가면 "선배님, 그거 고 1때 배운건데요..." 그래서 한달짜리 사이버 강좌 두개 더 신청했습니다. "Javascript"하고 "XML" 입니다. 이 것도 돈듭니다. 그리고 다음달에 "C", "D" 듣고, 그 다음달에 "E",... "Z"를 들을까 합니다. 계속 돈듭니다. 컴포넌트와는 상관 없다구요? 여기서 빳데루... 아니, 이거 들어야 컴포넌트 합니다. 요놈들이 한마리 길 잃은 양의 팔과 허벅지 입니다. 아직도 들어야 할 것들이 많습니다. 여러분 함께 공부합시다. 물론 프로젝트도 열심히 하구요. 또한 기본에 충실합니다. 컴포넌트만 중요합니까? 여기서 컴포넌트를 두글자로 줄여보겠습니다. "허울" 3글자로 해볼까요? "껍데기" 결국, 컴포넌트 무상입니다. 붕어빵에 붕어 없습니다. 우리 동네는 돌고래빵이더군요. 물어보았습니다. 웃지도 않더군요. 컴포넌트 방법론도 결국, 그 안에는 사람, 네트웍, DB, UML... 등이 있을 뿐이지요. 여기서 한가지, "컴포넌트가 중요하지 않다는 뜻이 아니고, 다른 모든 것도 중요하다는 의미입니다. 중요도는 프로젝트마다 차이가 있기 마련이구요."

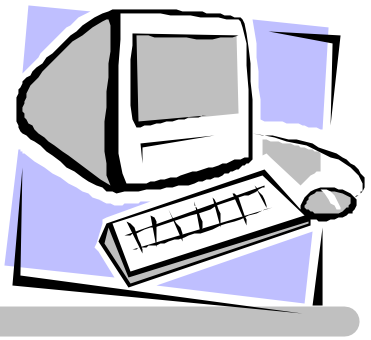
그리고 저는 슬슬 EJB를 조만간에 졸업하려 합니다. 더 정확한 표현은 수료 또는 중퇴가 맞겠지요? 짤린거는 아닙니다. 저기, 휘슬러라는 잔디위에 COM+라는 또 다른 허울이

C#이라는 사투리를 지껄여대는군요. DNS와 ActiveDirectory의 조화에 이은, Kerberos와 Public Key의 절묘한 어우러짐 - 뽕!, ASP+, DTC(Distributed Transaction Coordinator), ADO+, NLB(Network Load Balancing)와 CLB(Component Load Balancing), 그리고 무지하게 많은 Built-In COM+ 컴포넌트들... 지금 저의 비젼은 이미 그 쪽으로 옮아가고 있습니다. (사실은 병행입니다.)

마지막으로, 다소 과격한 표현이나 저질스런 언행, 무분별한 사투리, 틀린 철자, 외래어, 영어의 혼용 등에 대해 감히 용서를 구하고자 합니다. 쉬운 전달을 위해 사용했던 저의 순진무구하고 깨끗한 마음을 이해해주시리라 생각합니다. 사실, 저 담배 끊은지도 10년이 넘었구요, 술은 어쩌다 한두잔 정도만 합니다. 솥단지로? 아니요. 드럼통이요... 농담입니다. 술도 거의 안해요. 아침 일찍 일어나는것 좋아하죠. 최근 나이트 가본 것이 95년도 11월입니다. 참, 절대로! 이 글을 19세 이하의 미성년자에게 배포하지 말아주실 것을 간곡히 당부드립니다. 또 한가지 더. 이 글은 저의 자산입니다. 저의 피와 땀의 결정체입니다. 아시죠? 글 쓰는게 힘들다는 것을. 따라서 이 글의 일부 또는 전부를 아무런 사전 허락없이 카피 또는 인용을 하셔도 아무런 하자가 없슴을 알려드립니다. 많이 퍼뜨려야 합니다. 만약, 글쓴이를 저라고 밝혀주시면? - 제가 저녁 사겠습니다. 연락주세요. 019-279-0302, hpark@kebi.com 민토라고 대학로에 있는 좋은데 알고 있습니다. 그 맛있는 삼양컵라면이 꽁짜에요. 꽁짜.
참, 품위는 없지만 나름대로 격식을 차린 글을 읽고싶으신 분들은(저 박"격식"입니다) "객체지향 분석 설계 VC++ 프로그래밍"을 읽어보세요. 비앤씨 출판이구요. 1999년 3월에 냈지요. 사보실 필요는 없구요. 필요하신 사항만 눈여겨 보셨다가, 틈틈이 서점에 갈때마다 읽어보세요. 객체기술 기반이었구요, 소프트웨어 개발 전반을 다루었기 때문에 그리 어렵지는 않습니다. 내용이 얇다는 뜻이지요. 특히, 객체지향 분석/설계를 시작하는 분들에게만 권하고 싶구요, 뒷부분 프로그래밍(VC6.0, ATL3.0 사용) 단계에서는 COM을 기반으로 한 컴포넌트 설명을 했었지요. MS가 게으른 덕분에, 2년이나 지났는데도 현재 버전이네요...

참. 왜 Grady Report냐구요? 제 영어 이름이 Grady거든요. Grady Park! 왜 Grady냐구요? 저의 정신적인 "쓰부"가 Grady입니다. 그래서 뽄땃지요. 닮아보고 싶어서. 제 기억에 아직도 생생합니다. 92년도 현대전자 하계 인턴으로 중대형OS 부서에 갔었습니다. 저녁때마다 기숙사 2층 침대에 누워서 Grady 책 "OOD"를 보았답니다. 이해도 못하면서... 그 후로 오로지 이쪽만 했습니다. OT, Component만을요... 91년부터 94년까지 C++ 프로그래밍, 94년도부터 97년도까지 OOAD, OOM, 97년도부터 2000년도까지 COM/DCOM, EJB기반 Project 개발/관리/컨설팅을 해왔습니다.

방법론 만드는 일이 무쟈게 어렵게만 느껴지는 겁 많은 여린 소년.
^^ 2001년 올해는 뱜의 해. 저 뱜띱니다.(몇살이게요?) 1월 4일, 박현철이었습니다.
행복한 새해! 만세!

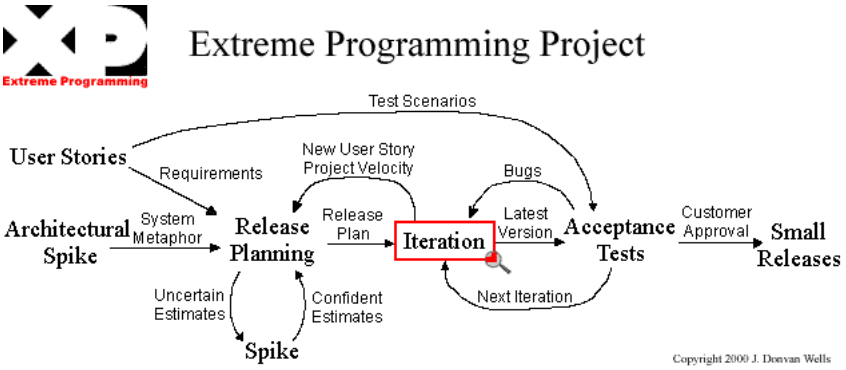


부록

차세대 방법론: XP(eXtreme Programming)

여러분들의 열화와 같은 성화가 있을 것에 힘입어 간단하나마 부록을 하나 준비했습니다.
시작하기 전에 두가지 퀴즈를 내겠습니다. 구조적 방법론의 대가는? 정보공학 방법론의 대가는? 참고로 저는 아닙니다... 이견이 있을 수는 있지만, 제가 알기로는 Yourdon과 Martin입니다. 그런데 이 두 대가께서 각각 OOA/OOD, OOIE를 내놓으면서 객체지향 문단에 입문한 것이 90년대 초라는 것도 알고 계시죠? 대가의 체면과 명예와 지위를 버리고 왜 당적을 바꾸었을까요? 비슷한 일이 있습니다. Gamma와 같은 디자인패턴의 대가가 왜 XP 진영으로 들어갔을까요? Fowler와 같은, Guru의 위치에 오르고도 두어끼 정도의 식량이 남을 것 같은, 사람이 왜 Beck이 만들어놓은 XP 쪽에 합류했을까요? 달마가 동쪽으로 간 것과 관계가 있을까요?
본론입니다. 오늘의 부록은 바로 XP입니다.
결론입니다. XP는 정말 좋습니다.

XP는 Extreme Programming이 줄어 들은 말입니다. 성질이 급하다고 생각되시는 분은 지금 바로 "http://www.extremeprogramming.org/"를 들어가보세요. 단 10분만 투자하시면 XP의 전반적인 것을 아실 수 있습니다. 혹시, 파악이 안되시면, 10분만 더 투자하세요. 그래도 안되면, ...될 때까지 계속 10분만 투자하세요. 그럼, 반드시 됩니다. 제 명예를 걸겠습니다.



일단 들어가시면, 위의 그림을 만나게 되실 겁니다. 마구 클릭해보세요. 연관된 사항들에 대한 내용들이 계속 나오게 됩니다. 환상입니다. (연결 고리가 많다는 뜻이지요. 보세요. 첫 화면만 봐도, 돌고 돌고...)

사실, 어느 정도 XP를 아시는 분에게는 죄송한 표현이기는 하지만, XP는 막가파식 방법론입니다. 앞단에는 Communication, Simplicity, Feedback, Courage를 내세우고, 뒷 단에서는 좋다는 놈은 무조건 갖고와서 끝을 볼 때까지 족칩니다. 즉, "코드 리뷰가 좋아?" 끝까지 가! "테스트가 좋아?" 끝까지 가! 설계? 간단명료? 아키텍처? 통합테스트가 좋아? 끝까지 가! 이터레이션(반복)이 좋아? 끝까지 가! 까짓것 초/분/시간 단위로 쪼개! 정말 막가파죠? (잠시 원문을 보시면, If short iterations are good, we'll make the iterations really, really short-seconds and minutes and hours, not weeks and months and years (the Planning Game))

정말 좋은 개념이 많구요. e-World 구축에 반드시 필요한 정신적/육체적(?) 철학이 담겨 있습니다. (육체적 철학이라 함은 몸으로 때우는 것을 의미합니다.) 기존 방법론의 틀을 깨고 싶으신 분들은 반드시 보셔야 합니다. 막가파식으로 설명을 조금 드려볼까요? CRC card? 써! Pair programming? 한 컴퓨터에서 두 사람이 번갈아가면서 프로그래밍 해! Refactoring? 코드로 디자인해봐! Test programming? 코딩하기 전에 테스트 프로그램 먼저 만들어! Task: 1-3 days? 작은 것 갖고 한달 붙들을 생각일랑 말어! Collective Code Ownership? 누구라도! (Anybody can change any line of code to add functionality, fix bugs, or refactor.) Move people around? 사람 계속 바꿔! 마구 퇴사해도 문제없어! Integrating and releasing code into the code repository every few hours! (우리나라에서는 불가능하지만) 담배 한대 필 때마다 코드 통합해서 저장해! Developers will go to the customer and receive a detailed description of the requirements face to face! 개발자들이 직접 고객과 맞장 떠!... 아직도 많은데,...

얼핏 보면, 엽기적이기까지 합니다. 그런데, 사실은 제가 그동안 프로젝트 하면서 많은 부분 해왔던 일들일 뿐입니다. 이렇게 하지 않으면 안되는 일이 넘 많걸랑요. 즉, 실제적인 방법론이자 현실적 방법론이라는 겁니다. 개발자들에게 "잘 만든 코드" 이상의 좋은 산출물은 존재할 수 없습니다. (돈 위에 그린 다이어그램이라면?...)

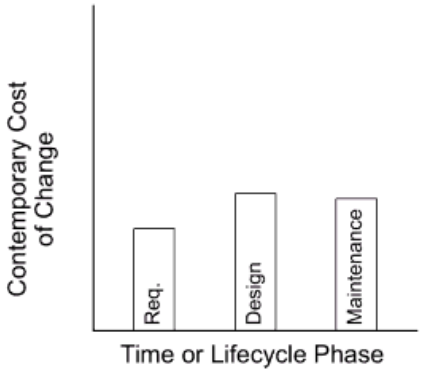
그럼, 두목님 얼굴 잠깐 보시구요. 다시 시작하도록 하겠습니다.

Beck이예요. 살벌하죠? 자 그럼 다시 시작해볼까요? 아. 잠깐. 네. 저기에 있는 말총머리 한 학생. 질문이 있나요? "네, 저기... 리팩토링이 뭐예요? 공장을 다시 짓는건가요? 막노동은 싫은데..." 아. 그거요. "긁적, 긁적..." 정의를 먼저 볼까요? Improving the design of existing code, without changing its observable behavior. 외적기능 변화없는 코드개선. 쉽죠? 이걸로 무얼하는 거냐면요. 저는 잘 모르겠는데, 코딩을 많이 하다보면, 코드에서 냄새가 나나봐요. (청결한 환경을 준수합시다!) 이 때 이 냄새들을 어떻게 하면, 효과적으로 제거해 나가나 하는 것을 다룹니다. 간단하게 말하면, 탈취 방법이죠. 안타까운 것은 탈취제는 없고, 여러분들이 직접 냄새를 제거해나가셔야 합니다. 이런 설명을 "공감각적 설명"이라고 하지요.

구체적으로 다음과 같은 증상에 대한 치료법을 다룹니다. Duplicate code. 중복 없애야 됩니다. Long Method. 긴건 나쁘죠. Large class. 큰 것도 나빠요. Long Parameter List 파라메터가 길면, 어렵죠. 다음부터는 제가 잘 몰라요. 직접 해석해보세요. Divergent Change, Shotgun Surgery, Feature Envy, Data Clumps, Primitive Obsession, Switch Statements, Parallel Inheritance Hierarchies, Lazy Class, Speculative Generality, Temporary Field, Message Chains, Middle Man, Inappropriate Intimacy, Alternative Classes with Different, Interfaces, Incomplete Library Class, Data Class, Refused Bequest, Comments 등이 있습니다. 자세한 것은 Fowler가 지은 "Refactoring"을 참고하세요.

이런, 삼천포로 빠졌군요. 빠진김에... 디자인패턴과 리팩토링의 관계는 어떻게 될까요? 결론부터 말씀 드리면, 리팩토링의 궁극적 목표는 디자인패턴입니다. 이런걸 "이율배반적 아리송"이라고 하지요. 구체적인 사항을 알고싶으신 분들은 XP로 들어오세요.

다시 제자리로 돌아와서. 저의 집에는 "eXtreme Programming eXplained", "Planning eXtreme Programming", "eXtreme Programming Installed"의 3권의 책이 있습니다. 안읽어봤습니다. 내용은 묻지 마세요. 읽어보시고 연락주세요. 아마존에서 샀습니다. 무지 비싸요. 물론, XP는 공개적입니다. 돈내고 살 필요가 없습니다. 물론, 살 것도 없지만. 그래도 책은 사셔서 읽어보세요. 이 책들은, 아니 XP는 기존의 소프트웨어 공학, 방법론, 여러 벤더 들이 그렇게도 목메어 주장했던, "Cost of Change"를 저녁상 엎어버리듯이 뒤집어 버리고 있습니다. 즉, 뒤에 가서 요구사항이 바뀌면 비용이 하늘 높은 줄 모르고 치솟는 모델을 많이 보셨죠? "뒤로 가면 몇 백배 됩니다. 앞 단계에 돈 투자하세요."하고 외치는 구세군을 가장한 컨설턴트들 보셨죠? 저도, 그중 하나였습니다. 그런데 XP 진영에서는 다음과 같은 그림을 내놓고 있습니다. "니 맘대로 바꾸세요!"라고 외치면서.



고객의 입장에서는 환상 아니 환각입니다. 말 조금 바꾸려면 계약서 들이대고, "돈 많이 있으세요? 돈 더 내셔야겠군요. 화면에 필드 하나 추가하시려면 개당 100$ 이름 바꾸시려면 개당 50$입니다..." 이제, 그 시절은 갔습니다. 정말 고객 중심의 시대로 가야 합니다. 고객은 왕입니다. 왜냐하면 주머니에 돈이 있거든요. 따라서 고객님들 마음 껏 외칩시다. 오늘은 이렇게 바꾸었다가. 내일은 그저께 걸로 다시 돌려도 보고,...

물론, 난 절대로 XP를 들고 프로젝트 하지는 않을 겁니다. XP로 프로젝트하는 쪽에 달랑 달려가, 고객들에게 저녁 얻어먹으면서 말할겁니다. "바꾸세요. 맘껏 바꾸세요. 변화가 중요합니다. 정체되면 안됩니다. 매일마다 새로운 것을 생각하세요. XP 좋습니다." 왜 노래도 있잖아요. 바꿔 바꿔... 음... 또 삼천포로 빠졌군요...

저는 여기서 XP를 소개하지는 않겠습니다. 단지, XP가 무엇인지 궁금증만을 유발시켜서 그 스트레스로 하여금 XP를 만천하에 고하고 싶을 따름입니다. 그리고 많은 사람들과 함께 XP의 철학을 공유하고 싶을 따름입니다.

참고로 저의 개인적 전략은 기존의 방법론과 XP를 적절하게 배분해서 저 나름대로의 개인적인 방법론을 만들려고 생각하고 있습니다. 물론, XP 위주로. (사실은 방법론을 만드는 방법까지만요. 물론, 시간 안나면 절대 안합니다.) 어쨌든, 이거 개인적으로 해야합니다. 여러명 모이면 절대 안될 것 같아요. 여러명이 한다 했을 때 "구조적, 정보공학적, C/S, Package, OOM, CBD 방법론과 같은 기존의 방법론들과 XP 모두 엄청 중요하다."는 인식이 기본적으로 깔려 있어야 합니다. 그리고 철학은 하나. 그게 기본적 전제 조건입니다. 하실 분 없으시죠? 다행입니다...

지상 컨설팅: 돈은없죠, 신기술과 방법론의 적용은 필요하죠, 우짜면 좋을까요?

돈이 많다면 컨설팅 또는 기술 자문을 받던지, 좋은 툴들을 사용하던지, 사람들을 더 고용하던지... 해법이 다양하겠죠? 그렇다고 돈을 벌어올 수도, 훔쳐올 수도 없죠. 하지만 돈이 없다고 주저앉아있을 수는 없습니다. 그래서 가상 프로젝트를 설정하고, 제 맘대로, 절대 책임 안지는 지상 컨설팅을 해볼까 생각합니다. 현재 구상하고 계시는 프로젝트와 비슷하지 않다고 해도, 최소한 약간의 힌트 정도는 얻을 수 있지 않을까 해섭니다. 한가지 가정은 기존의 기업 인프라를 신기술 기반으로 옮긴다는 것입니다. 즉, 웹과 컴포넌트, 체계적인 방법론의 적용, 최신 도구, 최신 컴퓨팅 환경 등이 부분적으로 또는 전부 필요한 프로젝트를 가정하겠습니다.

예를 들면, "저희들 인원은 4명이구요. 2명은 개발 경력은 미천하지만 SCJP 자격증이 있는 사람, 1명은 VC++, VB를 이용한 프로젝트 경험이 5년인 사람, 마지막 한명은 가칭 "능구렁이"로 개발은 손을 놓았지만, IT 업계에 10년의 경험을 갖고 있으며 현재 팀 리더로 있어요. 접니다. 앞으로 진행할 프로젝트는 고객관리인데 기존의 2 tier C/S 버전을 Web 버전으로 옮기면서, 컴포넌트 기반의 프로젝트를 진행하고 싶군요. 당연히 웹과 컴포넌트에 관한 경험이 없어 이 부분에 많은 고민을 하고 있습니다. 특이한 사항은, 이 프로젝트가 규모는 작지만, 향후 계정계와 정보계의 업그레이드에 대한 IT와 방법론의 선정에 영향을 줄 수 있을겁니다. 잘 만들면 제가 그 쪽에서 한자리 할 수 있걸랑요... 그래서 작은 프로젝트이지만, 나름대로 방법론의 적용을 통해 체계적인 개발을 경험하고 싶은데 어떻게 안될까요? 현재 저희가 생각하고 있는 개발 환경은..."

이때 제가 짠하고 등장해서 제 얄팍한 지식을 조금이나마 나눈다면, 얼마나 좋을까요? 혹시, 이런 부분(공개 무료 지상 컨설팅)에 관심이 계시다면, Feedback 주세요. 없으면 안합니다.

꼬릿말

먼저, 이글에 대해 리뷰를 해주신 몇몇분들께 감사의 말씀을 드립니다. 여러 격려의 말씀을 주셨고, 기본적인 오타의 수정부터, 잘못된 내용에 대한 지적, 과장된 내용, 전반적인 글의 문맥 및 의도에 관한 일침까지 아끼지 않으셨습니다. 고맙습니다.

참고로 이 글을 쓰게 된 계기는 2000년을 마무리하면서 제 개인적인 생각을 정리하고 싶어서였고, (좀 잡다하죠?) 리포트 형태로 배포하게 된 의도는, 많은 사람과의 기술적 공유를 통해, 저의 미진한 부분이나 잘못 생각하고 있는 부분을 다듬고 싶어서였습니다. 따라서 이 글의 대상은 불특정다수입니다. 따라서 특정 부위(?)에 따라 객관성이 부족할 수 있으며, 이 부분에 대해서는 사과의 말씀을 드리고싶군요. 피드백을 주시면 계속 고쳐나가도록 하겠습니다.

우선, 기본적인 오타나 내용상의 오류는 정정했습니다. (아직도 있겠지요?) 하지만, "JSP Model 1, 2 Architecture", 기본적인 여러 Pattern들 추가, "Deployment Descriptor도 별도의 한 부분으로" 등 몇가지 추가 요청 사항은 추가하지 못했습니다. 다음 버전에 추가하도록 하겠습니다.^^ 또한 컴포넌트가 대단히 중요한 것처럼 설명을 하다가 나중에 허울이라는 표현을 사용한 것은 너무 심하다고 지적해주셨지요? 맞습니다. 사실, 극적인 반전(?)을 꾀하려는 어줍잖은 의도가 "조금 먼저 알고있는 자의 만행"이 되었습니다. 이부분은 사과를 드립니다. 분명한 것은 컴포넌트는 좋은 것이고 EJB는 많은 사람들이 활용해야 할 대단한 놈입니다. 물론, 기본도 충실해야하구요.

또한 가장 많은 지적 부분은 Funtional/Non-functional, Layer/Tier, System/Software Architecture 등 눈에 잘 보이지않는 부분들이었습니다. 좀더 구체적인 설명이 필요하다든지... 그래서 일부러 레퍼런스도 달고, 미리 양해(?)의 문구도 넣었던 부분입니다. 일단, 구체적인 것들은 레퍼런스를 참조해주시길 바랍니다. 이 부분은 책들마다, 사람들마다 어느 정도는 견해 차이가 존재하는 부분이기 때문에 더욱 조심하게 되는데요, 다음 버전에는 좀 더 신중을 기해 내용을 변경/추가 해보도록 하겠습니다.

향후의 계획은, 우선 모의 프로젝트에 대한 코드를 첨가할까 합니다. 즉 규모는 작지만, JSP를 포함해서 여러 JavaBean과 Session Bean, Entity Bean들에 대한 전반적인 틀을 소스레벨에서 이해할 수 있도록 하겠습니다. 이쪽으로 초행길이신 개발자분들을 위한 것이죠. 물론, 전반적인 글의 이해도를 더욱 높이기 위함도 있구요.

부족한 글, 읽어주셔서 대단히 고맙습니다.