ESCUELA SUPERIOR POLITÉCNICA DE CHIMBORAZO

FACULTAD DE INFORMÁTICA Y ELECTRÓNICA

ESCUELA DE INGENIERÍA EN SISTEMAS

"ESTUDIO DE LA ARQUITECTURA DE LOS COMPONENTES EJB (ENTERPRISE JAVABEANS), CASO APLICATIVO EN EL SISTEMA DE EVALUACIÓN DOCENTE DE LA ESPOCH"

TESIS DE GRADO

Previa a la obtención del Título de

INGENIERO EN SISTEMAS INFORMÁTICOS

Presentado por:

Oswaldo Villacrés Cáceres

Riobamba – Ecuador

2011

AGRADECIMIENTO

Agradezco a mí madre, Myriam Patricia, a mi padre, Juan Enrique, por guiar micamino, ser el pilar fundamental en mivida, por todo el esfuerzo y dedicación que me brindaron día a día, a mis hermanos, quienes me apoyaron e impulsaron en mis momentos más difíciles, al Ing. Jorge Huilca Palacios, por la colaboración en el desarrollo del presente trabajo y la confianza en mí depositada, y a todos aquellos que pusieron un granito de arena a lo largo de mi vida estudiantil, mil gracias, mil gracias a Dios por permitirme vivir este momento.

DEDICATORIA

Dedico este trabajo a mis padres que sin su apoyo no hubiese sido posible alcanzar el tan anhelado sueño de ser un profesional, a mi hijo, Ián Santiago, por su alegría y compañía, porque gracias a él soy lo que soy hoy en día.

NOMBRE	FIRMA	FECHA
Ing. Iván Menes DECANO DE LA FIE	_____	_____
Ing. Raúl Rosero DIRECTOR DE LA EIS	_____	_____
Ing. Jorge Huilca DIRECTOR DE TESIS	_____	_____
Ing. Raúl Rosero MIEMBRO DEL TRIBUNAL	_____	_____
Tlgo. Carlos Rodríguez DIRECTOR DEL CENTRO DE DOCUMENTACIÓN	_____	_____

NOTA DE LA TESIS _____

“Yo, Oswaldo Villacrés Cáceres, soy el responsable de las ideas, doctrinas y resultados expuestos en esta Tesis, y el patrimonio intelectual de la misma pertenecen a la Escuela Superior Politécnica de Chimborazo.”

_____

Oswaldo Villacrés Cáceres

ÍNDICE GENERAL

CAPÍTULO I
MARCO PROPOSITIVO

1.1. Antecedentes……..18
1.1.1. Definición……19
1.1.2. Descripción……19
1.1.3. Problema a Solucionar……19
1.2. Justificación del Proyecto de Tesis……20
1.2.1. Justificación Teórica……20
1.2.2. Justificación Práctica……21
1.3. Objetivos……22
1.3.1. Objetivo General……22
1.3.2. Objetivos Específicos……22
1.4. Alcance del Estudio……22
1.5. Hipótesis……23

CAPÍTULO II
MARCO TEÓRICO

2.1. Historia de los Servidores de Aplicación……25
2.2. Arquitectura Java EE……26
2.2.1. Seguridad……27
2.2.2. Componentes de Java EE……28
2.2.3. Clientes Java EE……28
2.2.3.1. Clientes WEB……28
2.2.3.2. Applets……29
2.2.3.3. Aplicaciones Clientes……29
2.2.3.4. La arquitectura del componente JavaBeansTM……30
2.2.3.5. Comunicaciones del servidor Java EE……30
2.2.4. Componentes WEB……31
2.2.5. Componentes de Negocio……32
2.2.6. Capa del Sistema de Información Empresarial……32
2.3. Contenedores Java EE……32
2.3.1. Servicios del Contenedor……33
2.3.2. Tipos de Contenedor……34
2.4. Ensamblaje y desplieque de una Aplicacion Java EE……35
2.4.1. Empaquetado de Aplicaciones……35
2.4.2. Empaquetado de EJB simplificado……37
2.5. API de Java EE 6……38
2.5.1. Tecnología de JavaBeans Empresariales……40
2.5.2. Tecnología Java Servlet……41
2.5.3. Tecnología JavaServer Faces……41
2.5.4. Tecnología de Java Server Pages……42
2.5.5. Biblioteca de Etiquetas estándar Java Server Pages……43
2.5.6. API Java de Servicio de Mensajes (JMS) ……43

2.5.7. API Java de Transacciones……..43
2.5.8. API JavaMail……..44
2.5.9. Framework de Activación de JavaBeans……..44
2.5.10. API Java para procesamiento XML……44
2.5.11. API Java para Servicios Web XML (JAX-WS) ……44
2.5.12. API Java para Servicios Web REST……..45
2.5.13. Gestión de Beans……..45
2.5.14. Contextos y la inyección de dependencias para la plataforma Java EE……46
2.5.15. Inyección de Dependencias para Java (JSR 330) ……46
2.5.16. Validación de Beans……..46
2.5.17. API de SOAP con adjuntos para Java……46
2.5.18. API Java para registros XML……47
2.5.19. Arquitectura del conector JEE……..47
2.5.20. Contrato de autorización de Java para contenedores ……48
2.5.21. Java Authentication Service Provider Interface para contenedores……..48
2.5.22. API Java de conectividad a Base de Datos……..48
2.5.23. API Java de Persistencia (JPA)……..49
2.5.24. Interfaces de Nombres de Directoria de Java (JNDI)……49
2.5.25. Servicio de autenticación y autorización Java……..50
2.5.26. Integración de sistemas simplificada……..50
2.6. Enterprise JavaBeans……51
2.6.1. Generalidades……51
2.6.1.1. ¿Qué es un Enterprise Bean? ……..51
2.6.1.2. Beneficios de los Enterprise JavaBean……51
2.6.1.3. Cuándo utilizar Enterprise Java Beans……53
2.6.1.4. Roles y Responsabilidades EJB……..53
2.6.1.4.1. Enterprise Bean Provider……..55
2.6.1.4.2. Application Assembler……55
2.6.1.4.3. Deployer……...56
2.6.1.4.4. EJB Server Provider……57
2.6.1.4.5. EJB Container Provider……..57
2.6.1.4.6. Persistence Provider……...57
2.6.1.4.7. System Administrator……..58
2.6.1.5. Operaciones con Beans no permitidas……59
2.6.1.6. Anotaciones……..61
2.6.1.6.1. Uso……..61
2.6.1.6.2. Programación……...63
2.6.1.7. Generics……67
2.6.2. Tipos de EJB……...68
2.6.2.1. Stateless Session Beans……69
2.6.2.1.1. Uso……69
2.6.2.1.2. Estructura……..70
2.6.2.1.2.1. Clase Bean……70
2.6.2.1.2.2. Interfaz remota……70
2.6.2.1.2.3. Interfaz local……...71
2.6.2.1.2.4. Relación de interfaz alternativa……..72
2.6.2.1.2.5. Constructor estándar……72
2.6.2.1.2.6. General……...73

2.6.2.1.3. Metodología de trabajo……...73
2.6.2.1.4. Acceso al entorno……75
2.6.2.1.5. EJB Context……..77
2.6.2.1.6. Session Context……...78
2.6.2.1.7. Ciclo de vida de un Stateles Session Bean……80
2.6.2.1.7.1. Does Not Exist……...81
2.6.2.1.7.2. Método Ready Pool……..82
2.6.2.1.8. XML Deployment Descriptor……..84
2.6.2.1.9. Acceso mediante un programa Cliente……86
2.6.2.2. Stateful Session Beans……..89
2.6.2.2.1. Uso……89
2.6.2.2.2. Estructura……..91
2.6.2.2.2.1. La clase bean……91
2.6.2.2.2.2. Interfaz Remote……92
2.6.2.2.2.3. Interfaz Local……92
2.6.2.2.2.4. Interfaz RemoteHome y LocalHome……..93
2.6.2.2.2.5. Referencia a interfaz alternativa……94
2.6.2.2.2.6. Constructor estándar……94
2.6.2.2.2.7. Métodos de inicialización……95
2.6.2.2.2.8. Liberar la instancia de bean……95
2.6.2.2.3. Metodología de trabajo……...96
2.6.2.2.4. Llamadas de métodos locales o remotos……98
2.6.2.2.5. EJBContext y SessionContext……100
2.6.2.2.6. Ciclo de vida de un Stateful Session Bean……...100
2.6.2.2.6.1. Does not exist……101
2.6.2.2.6.2. Method-Ready……102
2.6.2.2.6.3. Passive……106
2.6.2.2.7. XML Desployment Descriptor……..108
2.6.2.2.8. Acceso a través del programa cliente……110
2.6.2.3. Singleton Session Beans……115
2.6.2.3.1. Uso……115
2.6.2.3.2. Inicialización……...116
2.6.2.3.3. Destrucción……119
2.6.2.3.4. Semántica de transacciones de inicialización y destrucción……..119
2.6.2.3.5. Ciclo de vida de un Singleton Session Bean……120
2.6.2.3.6. Control de errores en un Singleton……121
2.6.2.3.7. Concurrencia Singleton……121
2.6.2.3.8. Concurrencia gestionada por Contenedor……122
2.6.2.3.8.1. Comportamiento de bloqueo Reentrante……123
2.6.2.3.9. Concurrencia Gestionada por Bean……...124
2.6.2.3.10. Especificación de un tipo de simultaneidad de Gestión……..124
2.6.2.3.11. Especificación del Contenedor administrador de concurrencia de metadatos para los métodos de un Singleton bean……125
2.6.2.3.12. Especificación de atributos de concurrencia de bloqueo con anotaciones….126
2.6.2.4. Message-Driven Bean……..128
2.6.2.4.1. Uso……..128
2.6.2.4.2. Estructura……130
2.6.2.4.2.1. La clase bean……..130

2.6.2.4.2.2. Interfaz MessageListener……..133
2.6.2.4.2.3. Constructor estándar……..133
2.6.2.4.3. Metodología de trabajo……133
2.6.2.4.4. Contexto dirigido por mensajes……..135
2.6.2.4.5. Ciclo de vida de un Message-Driven Bean……...136
2.6.2.4.5.1. Does not exist……..137
2.6.2.4.5.2. Method-Ready Pool……138
2.6.2.4.6. XML Deployment Descriptor……139
2.6.2.4.7. Enviar mensajes desde un cliente……..140
2.6.2.4.8. Recibir mensajes con un cliente……142
2.6.2.4.9. Enviar mensajes desde un Bean……143
2.7. Timer Service……146
2.7.1.1.1. Uso……..146
2.7.1.1.2. Estructura……147
2.7.1.1.2.1. Interfaz TimedObject……..147
2.7.1.1.2.2. Interfaz TimerService……148
2.7.1.1.2.3. Interfaz Timer……..150
2.7.1.1.3. Metodología de trabajo……151
2.7.1.1.3.1. Diferentes tipos de Timers……151
2.7.1.1.3.2. Inicio del temporizador……...152
2.7.1.1.4. Stateless Session Bean Timer……154
2.7.1.1.5. Message-Driven Bean Timer……...154
2.7.1.1.6. Temporizador automático……154
2.8. Persistencia (JPA)……155
2.8.1. Arquitectura……...155
2.8.2. Entity Manager……157
2.8.2.1. Panorama general……157
2.8.2.2. Persistence Unit (Unidad de Persistencia)……158
2.8.2.3. Persistence Context……..160
2.8.2.3.1. Persistence Context orientados a transacciones……160
2.8.2.3.2. Persistence Context ampliado……161
2.8.2.3.3. Stateful Session Beans anidados……...162
2.8.2.4. EntityManager Factory……163
2.8.2.5. Interfaz EntityManager……165
2.8.2.6. Administración y Desadministración de Entidades……..173
2.8.3. Entity Beans……...174
2.8.3.1. Uso……..174
2.8.3.2. Estructura……176
2.8.3.3. Metodología de trabajo……182
2.8.3.4. Métodos del Ciclo de vida……186
2.8.3.5. XML Desployment Descriptor……..189
2.8.4. Consultas y EJB QL……190
2.8.4.1. Panorama general……190
2.8.4.2. Interfaz Query……191
2.8.4.2.1. getResultList()……192
2.8.4.2.2. getSingleResult()……...195
2.8.4.2.3. executeUpdate()……195
2.8.4.2.4. setMaxResults() y setFirstResult()……196

2.8.4.2.5. setHint()……196
2.8.4.2.6. setParameter()……197
2.8.4.2.7. setFlushMode()……198
2.8.4.3. EJB QL……199
2.8.4.3.1. Sintaxis BNF……201
2.8.4.3.2. Tipos de instrucciones……202
2.8.4.3.3. FROM……203
2.8.4.3.4. WHERE……206
2.8.4.3.5. GROUP BY……207
2.8.4.3.6. HAVING……208
2.8.4.3.7. SELECT……208
2.8.4.3.8. ORDER BY……212
2.8.4.3.9. UPDATE……213
2.8.4.3.10. DELETE……213
2.9. Transacciones……214
2.9.1. Panorama general……214
2.9.2. Container Managed Transaction……216
2.9.2.1. @TransactionAttribute……216
2.9.2.2. NOT_SUPPORTED……219
2.9.2.3. SUPPORTS……220
2.9.2.4. REQUIRE……220
2.9.2.5. REQUIRES NEW……220
2.9.2.6. MANDATORY……221
2.9.2.7. NEVER……221
2.9.3. Administrador de Bean de Transacciones……221
2.9.4. EJBs sin transacción……224
2.9.5. Accesos competitivos……224
2.9.5.1. Trabajar sin bloqueo de conjuntos……227
2.9.5.2. Versionar automáticamente……228
2.9.5.3. Trabajar con bloqueo de conjuntos……231
2.9.6. Rollback en EJBContext……233
2.9.7. Transaccion y excepciones……233
2.9.8. Session Beans transaccionales……237
2.10. Interceptores y Entity Listener……238
2.10.1. Panorama general……238
2.10.2. Interceptor……239
2.10.2.1. Anotaciones……239
2.10.2.2. Descriptor de Despliegue……242
2.10.3. Entity Listener……244
2.10.3.1. Anotaciones……244
2.10.3.2. Descriptor de Despliegue ……246
2.11. Seguridad……247
2.11.1. Asegurar llamadas de métodos……247
2.11.1.1. @RolesAllowed……247
2.11.1.2. @PermitAll……249
2.11.1.3. @DenyAll……251
2.11.1.4. @RunAs……252
2.11.2. Comprobación técnica de derechos……254

2.11.2.1. @DeclareRoles……..255

CAPITULO III
ANÁLISIS, DISEÑO E IMPLEMENTACIÓN

3.1. Metodología de Desarrollo de Software……...268
3.1.1. Procesos de Desarrollo……...268
3.1.2. RUP (Rational Unified Process)…..……269
3.1.2.1. Características del RUP……...270
3.1.2.1.1. Desarrollo Iterativo……270
3.1.2.1.2. Administración de Requerimientos……271
3.1.2.1.3. Arquitecturas basadas en Componentes……..271
3.1.2.1.4. Modelamiento Visual……271
3.1.2.1.5. Verificación de la calidad de software……271
3.1.2.1.6. Control de cambios……...271
3.1.2.2. Fases del RUP……...272
3.1.2.2.1. Fase de Concepción (Inicio) ……...272
3.1.2.2.2. Fase de Elaboración……272
3.1.2.2.3. Fase de Construcción……...272
3.1.2.2.4. Fase de Transición……272
3.1.3. UML (Unified Modeling Language)……...273
3.1.3.1. Objetivos del UML……273
3.1.3.2. Arquitectura de UML……274
3.1.3.3. Áreas conceptuales de UML……274
3.1.3.3.1. Estructura estática……274
3.1.3.3.2. Comportamiento dinámico……...274
3.1.3.3.3. Construcciones de implementación……275
3.1.3.3.4. Organización del modelo……275
3.1.3.4. Diagramas de UML……...275
3.1.3.4.1. Diagramas de Objetos……...276
3.1.3.4.2. Diagramas de Clases……277
3.1.3.4.3. Diagramas de Caso de Uso……278
3.1.3.4.4. Diagramas de Actividades……...279
3.1.3.4.5. Diagramas de Estado……...280
3.1.3.4.6. Diagramas de Interacción……280
3.1.3.4.7. Diagramas de Secuencia……281
3.1.3.4.8. Diagramas de Colaboración……281
3.1.3.4.9. Diagramas de Componentes……...282
3.1.3.4.10. Diagramas de Despliegue……283
3.1.3.4.11. Diagrama de Paquetes……283
3.2. Requerimientos……284
3.3. Análisis y Diseño……..285
3.4. Implementación……285

CAPITULO IV
ESTUDIO DE RESULTADOS

4.1. Modificaciones entre formas de desarrollo……..287

4.2. Establecimiento de variables……289
4.3. Definición de las variables……289
4.3.1. Total LDC del Programa……290
4.3.2. Tiempo……290
4.3.3. LDC al Programar……290
4.4. Generación de resultado de las aplicaciones……292
4.4.1. Variable “Total LDC del programa”……292
4.4.2. Variable “Tiempo”……293
4.4.3. Variable “LDC al Programar”……293
4.5. Estudio comparativo de los resultados de ambas aplicaciones……298
4.6. Resultado final……300
4.7. Demostración de la Hipótesis……301

CONCLUSIONES……305

RECOMENDACIONES……306

RESUMEN……307

SUMMARY……308

GLOSARIO……309

ANEXOS……310

BIBLIOGRAFÍA

ÍNDICE DE TABLAS

Tabla II.I: Tipos de EJB……68

Tabla III.II: Diagramas de UML……..276

Tabla IV.III: Ponderación de Variables……289

Tabla IV.IV: Coeficientes de confianza……291

Tabla IV.V: Resumen de resultados obtenidos “Practiline Source Code Line Counter”..293

Tabla IV.VI: Resultados LDC al Programar en la Clase CARRERA……293

Tabla IV.VII: Resultados LDC al Programar en la Clase DIRECTIVODOCENTE…….294

Tabla IV.VIII: Resultados LDC al Programar en la Clase DIRECTIVO……...294

Tabla IV.IX: Resultados LDC al Programar en la Clase DOCENTEDOCENTE……294

Tabla IV.X: Resultados LDC al Programar en la Clase DOCENTE……..294

Tabla IV.XI: Resultados LDC al Programar en la Clase ESCUELA……295

Tabla IV.XII: Resultados LDC al Programar en la Clase ESTADISTICAGENERAL…….295

Tabla IV.XIII: Resultados LDC al Programar en la Clase ESTANDAR……...295

Tabla IV.XIV: Resultados LDC al Programar en la Clase ESTUDIANTEDOCENTE……295

Tabla IV.XV: Resultados LDC al Programar en la Clase ESTUDIANTE……296

Tabla IV.XVI: Resultados LDC al Programar en la Clase INDICADOR……..296

Tabla IV.XVII: Resultados LDC al Programar en la Clase MATERIA……296

Tabla IV.XVIII: Resultados LDC al Programar en la Clase OPCIONES……296

Tabla IV.XIX: Resultados LDC al Programar en la Clase PREGUNTA……..297

Tabla IV.XX: Resultados LDC al Programar en la Clase PROCESOEVALUACION……297

Tabla IV.XXI: Resultados LDC al Programar en la Clase PROCESO……297

Tabla IV.XXII: Resultados LDC al Programar en la Clase USUARIO……297

Tabla IV.XXIII: Ponderación de la mejora final……301

ÍNDICE DE FIGURAS

Figura I.: Aplicaciones de múltiples capas…….27

Figura II: Comunicación del servidor……..30

Figura III: Capa WEB y Aplicaciones Java EE……..31

Figura IV: Capa de Negocio y Capa EIS……32

Figura V: Servidor y Contenedores Java EE…….34

Figura VI: Estructura de un fichero EAR……36

Figura VII: Estructura de un paquete EAR…….37

Figura VIII: Empaquetado EJB simplificado……..38

Figura IX: Contenedores Java EE y sus relaciones…….38

Figura X: API de Java EE en el contenedor Web…….39

Figura XI: API de Java EE en el contenedor EJB……..39

Figura XII: API de Java EE en el contenedor de Aplicaciones Cliente……40

Figura XIII: Una anotación transforma un simple POJO en un EJB……..52

Figura XIV: Ciclo de vida de un Bean de Sesión sin estado……..81

Figura XV: Ciclo de vida de un Stateful Session Bean……..101

Figura XVI: Interfaz de la cesta de la compra…….115

Figura XVII: Ciclo de Vida de un Singleton Session Bean……120

Figura XVIII: Ciclo de vida de un Message-Driven Bean……..136

Figura XIX: Arquitectura JPA…….156

Figura XX: Modelo de datos……..200

Figura XXI: Modelo de datos ampliado……201

Figura XXII: Vista general de RUP…….269

Figura XXIII: Flujos de Trabajo de RUP……...270

Figura XXIV: Diagrama de Objetos……..276

Figura XXV: Diagrama de Clases……277

Figura XXVI: Diagrama de Casos de Uso……...278

Figura XXVII: Diagrama de Actividades……...279

Figura XXVIII: Diagrama de Estado……..280

Figura XXIX: Diagrama de Interacción……280

Figura XXX: Diagrama de Secuencia……...281

Figura XXXI: Diagrama de Colaboración……282

Figura XXXII: Diagrama de Componentes……..282

Figura XXXIII: Diagramas de Despliegue……283

Figura XXXIV: Diagrama de Paquetes……284

Figura XXXV: Capas de Aplicación sin tecnología EJB……287

Figura XXXVI: Módulo EJB y sus Capas de Persistencia y Lógica de Negocios…….288

Figura XXXVII: Módulo Web con Servlets y JSP……289

Figura XXXVIII: Resultados “Total LDC del programa” sin EJB……..292

Figura XXXIX: Resultados “Total LDC del programa” con EJB……...292

Figura XL: Gráfico de Demostración de la Hipótesis……303

INTRODUCCIÓN

En esta tesis se presentan los resultados de la investigación realizada para determinar si la tecnología EJB permite optimizar el tiempo de desarrollo en el sistema de evaluación docente de la ESPOCH.

El objetivo principal de esta tesis es realizar un estudio de la arquitectura de los EJB (Enterprise JavaBeans) aplicándolos en la implementación de componentes para el Sistema de Evaluación Docente de la Escuela Superior Politécnica de Chimborazo.

En el primer capítulo se tratará sobre el marco referencial, en el cual se encuentra de manera general la justificación del proyecto de tesis además de los objetivos a alcanzar con el desarrollo de la misma.

En el segundo capítulo, se ha previsto estudiar todo lo referente a la tecnología EJB.

En el tercer capítulo, se ha previsto detallar la metodología de desarrollo utilizada junto con el lenguaje de modelado que esta utiliza y además de algunas de las disciplinas que está metodología propone.

Para finalizar con el capítulo cuatro, en donde se plantea el estudio de resultados. En donde se detallarán los parámetros utilizados para la comprobación de la hipótesis planteada.

Adicional a los capítulos, puede encontrar en los anexos la documentación mucho más detalla del análisis estadístico y de la documentación técnica generada del sistema de evaluación docente de la ESPOCH.

CAPÍTULO I

MARCO PROPOSITIVO

1.6. Antecedentes

En los últimos tiempos Java se ha convertido en un lenguaje popular para la creación de aplicaciones distribuidas. De hecho, Sun definió una plataforma específica para el desarrollo de aplicaciones distribuidas, J2EE (Java 2 Enterprise Edition) ahora llamada JEE, que proporciona todo lo necesario para la creación de aplicaciones de gran complejidad destinadas al ámbito corporativo.

Las aplicaciones distribuidas suelen tener una estructura en la que se pueden distinguir esencialmente tres capas:

- El interface de usuario.
- La lógica de negocios.
- La capa de almacenamiento de datos.

Una de las capas más trascendentales dentro del desarrollo de aplicaciones distribuidas es la capa de lógica de negocios, que es en donde estará el comportamiento o funcionalidad en sí de la aplicación; y a su vez uno de los principales componentes con el que cuenta JEE enfocado a este ámbito, es la tecnología de Enterprise JavaBeans, que proporciona un estándar para el desarrollo de las clases que encapsulan la funcionalidad y reglas del negocio.

1.6.1. Definición

Los EJB son componentes del contexto de servidor que cubren la necesidad de intermediar entre la capa de interfaz de usuario y diversos sistemas empresariales o la capa de acceso a datos.

Un EJB es un conjunto de clases y un archivo XML que describe el despliegue de dichas clases en un contenedor de EJB. Un EJB se ejecuta en un contenedor de EJB y ofrece al desarrollador del Bean, servicios que no necesita programar (persistencia, seguridad, transacciones, etc.).

1.6.2. Descripción

Los EJB nacen para encapsular la lógica de negocio de una forma integrada, no quedando dispersos su representación en un grupo de sistemas empresariales. Los EJB están especialmente pensados para integrar la lógica de la empresa que se encuentra en sistemas distribuidos, de tal forma que el desarrollador no tenga que preocuparse por la programación a nivel de sistema (como control de transacciones, seguridad, etc.), sino que se centre en la representación de entidades y reglas de negocio. El hecho de estar basado en componentes permite que éstos sean flexibles y sobre todo reutilizables.

El programador de EJB debe centrarse en los problemas de representación de entidades y relaciones de negocio, tratando de que los detalles más concretos, más dependientes de la plataforma o del sistema queden para el contenedor EJB.

No hay que confundir los Enterprise JavaBeans con los JavaBeans. Los JavaBeans también son un modelo de componentes creado por Oracle - Sun Microsystems para la construcción de aplicaciones, pero no pueden utilizarse en entornos de objetos distribuidos al no soportar nativamente la invocación remota (RMI).

1.6.3. Problema a Solucionar

En la actualidad la Escuela Superior Politécnica de Chimborazo cuenta con un Sistema de Evaluación Docente, el mismo que se ve limitado en cuanto a conectividad de usuarios (un máximo de 150 conexiones concurrentes) por motivo de licencias, existiendo ocasiones en las que el mismo ha colapsado, razón por la cual se ha visto la necesidad de implementar una aplicación que se ejecute bajo un sistema operativo de licencia libre, o independiente de la plataforma y mismo que estará constituido por componentes Enterprise JavaBeans.

1.7. Justificación del Proyecto de Tesis

Se lo realizará en función a una justificación Teórica y Práctica.

1.7.1. Justificación Teórica

La tecnología Java, es una tecnología madura, extremadamente eficaz y sorprendentemente versátil, se ha convertido en un recurso inestimable y posee buenas perspectivas de futuro para el desarrollo de Aplicaciones.

Motivo por el cual se ha decido desarrollar el presente tema de tesis con la utilizando de componentes EJB que forman parte de la arquitectura JEE, y que proporcionan al desarrollador, servicios que no necesita programar como persistencia, seguridad, control de transacciones, etc., además de permitir agrupar la lógica de negocios de una manera integrada.

Se considera un tema de real importancia debido a que será de vital ayuda para el conocimiento de todos aquellos estudiantes y profesionales que desean incursionar en el ámbito del desarrollo utilizando componentes Enterprise JavaBeans.
Entre las ventajas que brindan los Enterprise JavaBeans a un desarrollador de aplicaciones se pueden mencionar:

- **Simplicidad**.- Debido a que el contenedor de aplicaciones libera al programador de realizar las tareas del nivel del sistema, El desarrollador no tiene que preocuparse de temas de nivel de sistema como la seguridad, transacciones, multi-threading o la programación distribuida.

- **Portabilidad de la aplicación**.- Una aplicación EJB puede ser desplegada en cualquier servidor de aplicaciones que soporte J2EE.

- **Reusabilidad de componentes**.- Una aplicación EJB está formada por componentes Enterprise Beans. Cada Enterprise Bean es un bloque de construcción reusable.

- **Posibilidad de construcción de aplicaciones complejas**.- La arquitectura EJB simplifica la construcción de aplicaciones complejas. Al estar basada en componentes y en un conjunto claro y bien establecido de interfaces, se facilita el desarrollo en equipo de la aplicación.

- **Separación de la lógica de presentación de la lógica de negocio.-** Un Enterprise Bean encapsula típicamente un proceso o una entidad de negocio (un objeto que representa datos del negocio), haciéndolo independiente de la lógica de presentación.

- Despliegue en muchos entornos operativos.

- Despliegue distribuido

- Interoperabilidad entre aplicaciones

- Integración con sistemas no-Java

Entre las ventajas que aporta esta arquitectura al cliente final, se pueden destacar:

- La posibilidad de elección del servidor.
- La mejora en la gestión de las aplicaciones.
- La integración con las aplicaciones y datos ya existentes.
- La seguridad.

1.7.2. Justificación Práctica

Se tomará como parte práctica: el análisis del potencial del Servidor de Aplicaciones con la implantación de un sistema que incorpore componentes que forman parte de la arquitectura JEE, el estudio del potencial que tiene un servidor CentOS 5 para soporte de concurrencia, la implementación de componentes que se acoplen al Sistema de Evaluación Docente haciendo uso de los Enterprise JavaBeans y el análisis de los componentes EJB como mecanismo para el incremento en el número de conexiones de usuarios.

1.8. Objetivos

1.8.1. Objetivo General

- Estudiar la arquitectura de los EJB (Enterprise JavaBeans) aplicándolos en la implementación de componentes para el Sistema de Evaluación Docente de la Escuela Superior Politécnica de Chimborazo.

1.8.2. Objetivos Específicos

- Analizar conceptos, definiciones, teorías, elementos y arquitectura relacionados con la tecnología de los EJB para conocer como están constituidos y como implementarlos.
- Hacer uso de los Enterprise JavaBeans en la implementación de componentes para vincularlos al Sistema de Evaluación Docente de la ESPOCH y mejorar el tiempo de respuesta.
- Evaluar el rendimiento del Sistema de Evaluación Docente con la incorporación de componentes EJB para establecer el incremento en el número de conexiones permitidas y el tiempo dedicado a las evaluaciones en la institución.

1.9. Alcance del Estudio

En el desarrollo del presente tema se pretende estudiar a los componentes EJB (Enterprise JavaBeans) y analizar las características que estos tienen, para luego implementar componentes que se acoplen al Sistema de Evaluación Docente de la ESPOCH, haciendo uso de la tecnología EJB.

1.10. Hipotésis

La implementación de componentes EJB que se acoplen al Sistema de Evaluación Docente de la ESPOCH permitirá optimizar el tiempo de desarrollo destinado en la lógica de negocios del mismo.

CAPÍTULO II

MARCO TEÓRICO

Hoy en día los desarrolladores pueden construir aplicaciones distribuidas, transaccionales, seguras y confiables basadas en la tecnología de servidores de aplicación (server-side tecnology). Cada vez, las aplicaciones deben ser diseñadas, construidas y producidas por menos dinero, a más velocidad y utilizando menor cantidad de recursos y profesionales.

Para reducir costos y acelerar el proceso de desarrollo, Java Platform Enterprise Edition o Java EE (anteriormente conocido como Java 2 Platform, Enterprise Edition o J2EE hasta la versión 1.4), provee un enfoque basado en componentes para el diseño, ensamblaje e instalación de aplicaciones. La plataforma Java EE provee un modelo de aplicación multicapa, componentes reutilizables, un modelo de seguridad unificado, control flexible de transacciones, soporte de Web Services a través de intercambio de datos integrados por estándares y protocolos abiertos XML.

Java EE incluye varias especificaciones de API, tales como JDBC, RMI, e-mail, JMS, Servicios Web, XML, etc. y define cómo coordinarlos. Java EE también configura algunas especificaciones únicas para Java EE para componentes. Estas incluyen Enterprise JavaBeans (EJB), servlets, portlets, JavaServer Pages (JSP) y varias tecnologías de servicios web. Esto permite al desarrollador crear una Aplicación de Empresa portable entre plataformas y escalable, a la vez que integrable con tecnologías anteriores. Otros beneficios añadidos son, por ejemplo, que el servidor de aplicaciones puede manejar transacciones, la seguridad, escalabilidad, concurrencia y gestión de los componentes desplegados, significando que los desarrolladores pueden concentrarse más en la lógica de negocio de los componentes en lugar de en tareas de mantenimiento de bajo nivel.

Las soluciones basadas en componentes de la plataforma Java EE son desarrolladas en forma independiente de la implementación, por lo tanto no se encuentran asociadas a los productos particulares y a las APIs de los proveedores.

2.12. Historia de los Servidores de Aplicación

En el pasado los desarrolladores al construir una aplicación, no solamente se enfrentaban a los problemas de negocios a resolver, sino que siempre debían lidiar con los mismos problemas de la capa de middleware (middleware tier), transacciones, persistencia, distribución de objetos, administración de conexiones, escalabilidad, redundancia, tolerancia a fallos, seguridad, etc. Para resolver estos problemas comunes a todos los desarrollos, las empresas construían sus propios entornos de trabajo (frameworks). Estos entornos de trabajo solían ser muy complejos de construir y mantener, y requerían conocimiento experto, que muchas veces poco tienen que ver con el ***negocio*** central de la compañía.

Otra opción era comprar las soluciones de middleware y construir la aplicación sobre el entorno de trabajo del proveedor. El problema con las soluciones propietarias pasaba porque la aplicación quedaba atada a la implementación y a las APIs del proveedor.

El servidor de aplicación (Aplication Server) nace para resolver estos problemas. Provee servicios comunes de middleware como persistencia, transacciones, distribución de objetos, administración de conexiones, etc. Los proveedores implementan estas interfaces estándares para integrar proveer los servicios a sus productos mediante el servidor de aplicaciones. Por lo tanto los desarrolladores ahora pueden centrarse en la problemática central de negocio, o los requerimientos funcionales, que debe resolver la aplicación utilizando APIs estándares y para invocar al middleware y decidiendo luego que implementaciones específicas utilizar para resolver mejor los requerimientos de tipo no funcionales.

Basándose en los servidores de aplicación, las soluciones son desarrolladas independientemente de la plataforma, pueden correr en cualquier servidor de aplicación y pueden ser integradas en forma transparente e independiente con los productos de los proveedores que brinden los servicios del servidor de aplicaciones.

Un ejemplo concreto puede ser el típico caso en que la aplicación debe guardar información en una base de datos. Los desarrolladores pueden utilizar distintas estrategias estándares dentro del conjunto de especificaciones JEE para persistir sus datos, como por ejemplo: JDBC, EJB CMP o BMP, etc. Luego es necesario configurar el servidor de aplicación para indicarle que implementación debe utilizar para conectarse con la base de datos. De esta manera, la aplicación conoce el servicio estándar del servidor de aplicaciones para utilizar la base de datos, y el servidor de aplicaciones conoce el nombre de la base de datos, donde se encuentra y cuál es el proveedor. Esta aplicación puede correr en cualquier servidor de aplicaciones que implemente JEE. Además de la independencia que provee esta separación de capas permite a los administradores de la aplicación cambiar el nombre de la base de datos donde se encuentra o directamente cambiar de proveedor de base de datos sin necesidad de modificar otra cosa que la configuración del servidor de aplicación. Por lo tanto la configuración de los servicios también recaen en el servidor de aplicación, evitando a los desarrolladores construir soluciones ad-hoc tales como el uso de archivos para las configuraciones.

2.13. Arquitectura Java EE

La plataforma Java EE utiliza un modelo de aplicación de múltiples capas para aplicaciones empresariales. La lógica de la aplicación es dividida en componentes de acuerdo con su función y los diferentes componentes de aplicación que hacen una aplicación Java EE son instalados en diferentes máquinas dependiendo de la capa en el ambiente de múltiples capas de Java EE para el cual el componente de aplicación pertenece.

La Figura I, muestra dos aplicaciones Java EE de capa múltiple dividida en sus capas descriptas en la siguiente lista. Las partes de aplicación Java EE mostradas en la Figura I están presentes en Componentes Java EE.

- Los componentes de la capa de cliente se ejecutan en la máquina del cliente.
- Los componentes de la capa Web se ejecutan en el servidor Java EE.
- Los componentes de la capa de negocios se ejecutan en el servidor Java EE.
- La capa de Sistemas de información empresariales (EIS - Enterprise Information System) se ejecutan en el servidor EIS.

A pesar que una aplicación Java EE puede consistir en las tres o cuatro capas mostradas en la Figura I, las ampliaciones de capas múltiples de Java EE son consideradas generalmente aplicaciones de tres capas porque son distribuidas sobre tres ubicaciones: máquinas clientes, la máquina servidor Java EE y la base de datos o las máquinas legadas. Las aplicaciones que se ejecutan de esta forma extienden el modelo estándar cliente servidor de dos capas colocando un servidor de aplicación entre la aplicación del cliente y el almacenamiento.

Como se puede ver un concepto clave de la arquitectura es el de **contenedor**, que dicho de forma genérica no es más que un entorno de ejecución estandarizado que ofrece unos servicios por medio de componentes. Los componentes externos al contenedor tienen una forma estándar de acceder a los servicios de dicho contenedor, con **independencia del fabricante.**

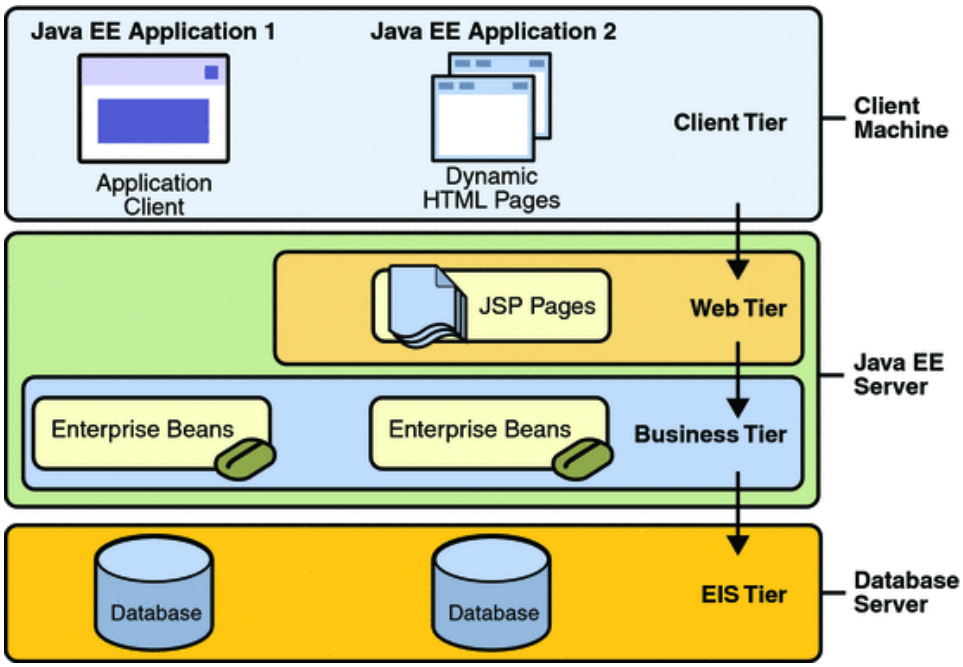


Figura I: Aplicaciones de múltiples capas

2.13.1. Seguridad

Mientras otros modelos de ampliación empresarial requieren medidas de seguridad específicas de la plataforma en cada aplicación, el ambiente de seguridad de Java EE habilita que las restricciones de seguridad sean definidas en tiempo de desarrollo. La plataforma Java EE hace que las aplicaciones sean portátiles a una gran variedad de implementaciones de seguridad protegiendo a los desarrolladores de la aplicación de la complejidad de implementar características de seguridad.

La plataforma Java EE proporciona reglas declarativas estándar de control de acceso que son definidas por el desarrollador e implementadas cuando la aplicación es desplegada en el servidor. Java EE también proporciona mecanismos de acceso estándares para que los desarrolladores de la aplicación no tengan que implementarlos en sus aplicaciones. La misma aplicación trabaja en una gran variedad de ambientes de desarrollo sin cambiar el código fuente.

2.13.2. Componentes de Java EE

Las aplicaciones Java EE están hechas con componentes. Un componente es una unidad de software auto contenida funcional que es ensamblada en una aplicación Java EE con sus clases relacionadas y ficheros y que se comunica con otros componentes.

La especificación de Java define los siguientes componentes Java EE:

- Las aplicaciones clientes y Applets son componentes que se ejecutan en el cliente.

- Los componentes Servlets, JavaServer Faces, y tecnología JavaServer PagesTM (JSPTM) son componentes web que se ejecutan en el servidor.
- Los componentes JavaBeansTM (EJBTM) empresariales (beans empresariales) son componentes de negocios que se ejecutan en el servidor.

Los componentes Java EE son escritos en el lenguaje de programación Java y son compilados de la misma forma como cualquier programa en el lenguaje. La diferencia entre los componentes Java EE y las clases Java estándar es que los componentes Java EE son ensamblados dentro de una aplicación Java EE, además son verificados para estar seguros de que están bien formados, conformes con la especificación Java EE y son desplegados en producción donde son ejecutados y manejados por el servidor Java EE.

2.13.3. Clientes Java EE

Un cliente Java EE puede ser un cliente Web o una aplicación cliente.

2.13.3.1. **Clientes WEB**

Un **cliente web** consiste en dos partes: (1) una página web dinámica que contiene varios tipos de lenguajes de marcas (HTML, XML y demás), que son generados por componentes web ejecutándose en la capa web y (2) un navegador web que despliega las páginas recibidas del servidor.

Un cliente web, llamado a veces terminal (thin client, cliente ligero, NetPC). Los terminales usualmente no consultan bases de datos, ejecutan reglas de negocio complejas o conectan a aplicaciones legadas. Cuando se utiliza un terminal, estas operaciones pesadas son delegadas a los beans empresariales que se ejecutan en el servidor Java EE que pueden soportar la seguridad, velocidad, servicios y ofrecer el nivel de confianza de la tecnología Java EE del lado del servidor.

2.13.3.2. **Applets**

Una página web recibida de una capa web puede incluir un Applet incrustado.

Un **Applet** es una pequeña aplicación cliente escrita en el lenguaje de programación Java que se ejecuta en la máquina virtual de Java en el navegador web. Sin embargo, los sistemas clientes pueden necesitar la necesidad de un plug-in y posiblemente un fichero con políticas de seguridad para que el Applet se ejecute con éxito en el navegador web.

Los componentes son la API preferida para crear un programa para cliente web ya que no se necesitan plug-ins o políticas de seguridad en los sistemas clientes. Los componentes web además tienen un diseño más limpio y más modular porque estos proporcionan una forma de separar la programación del diseño de la página web de la aplicación. El personal

involucrado en el diseño de la página no necesita entender la sintaxis del lenguaje de programación Java para hacer su trabajo.

2.13.3.3. **Aplicaciones Clientes**

Una **aplicación cliente** se ejecuta en una máquina cliente y proporciona a los usuarios una forma de manejar las tareas que requieren una interfaz de usuario rica que puede ser proporcionada por un lenguaje de formato. Normalmente este tiene una interfaz gráfica de usuario (GUI) creada con la API de Swing o AWT, aunque una interfaz de línea de comandos ciertamente es posible.

Las aplicaciones cliente acceden directamente a beans empresariales que se ejecutan en la capa de negocio. Sin embargo, si los requerimientos de una aplicación lo exigen, un cliente de aplicación puede abrir una conexión HTTP para establecer una comunicación con un Servlet que se ejecuta en la capa web. Los clientes de aplicación escritos en lenguajes diferentes a Java pueden interactuar con servidores Java EE 5, habilitando la plataforma Java EE 5 a interactuar con sistemas legados, clientes y lenguajes que no son Java.

2.13.3.4. **La arquitectura del componente JavaBeansTM**

Las capas del servidor y del cliente pueden también incluir componentes basados en la arquitectura del componente JavaBeans (componentes JavaBeans) para manejar los datos que fluyen desde una aplicación cliente o Applet y componentes que se ejecutan en el servidor Java EE, o entre los componentes del servidor y la base de datos. Los componentes JavaBeans no son considerados componentes Java EE por la especificación Java EE.

Los componentes JavaBeans tienen propiedades y tienen métodos get y set para acceder a sus propiedades. Los componentes JavaBeans utilizados de esta forma normalmente simples en diseño e implementación pero suelen ajustarse a las convenciones de nombres y diseño perfiladas en la arquitectura de componentes JavaBeans.

2.13.3.5. **Comunicaciones del servidor Java EE**

La Figura II, muestra los elementos que pueden formar la capa cliente. La comunicación del cliente con la capa de negocio que se ejecuta en el servidor Java EE se realiza directamente o en el caso de un cliente ejecutándose en un navegador lo realiza a través de una página JSP o un Servlet que se ejecuta en la capa web.

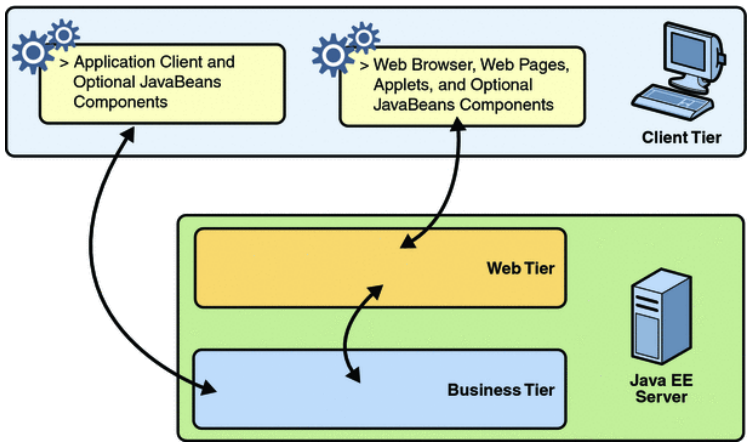


Figura II: Comunicación del servidor

Su aplicación Java EE utiliza un navegador o una aplicación cliente pesada. Al decidir cual utilizar se debe de tener cuidado en llegar a un equilibrio entre mantener la funcionalidad en el cliente y acercarse a el usuario (aplicación cliente pesada) y cargarle toda la funcionalidad posible al servidor (terminal). Cargar la mayor funcionalidad posible al servidor facilita la distribución, despliegue y manejo de la aplicación, sin embargo, mantener más funcionalidad en el cliente puede hacer que se perciba una mejor experiencia de usuario.

2.13.4. Componentes WEB

Los componentes web de Java EE son Servlet o páginas creadas utilizando tecnología JSP (páginas JSP) y/o tecnología JavaServer Faces. Los **Servlets** son clases programadas en lenguaje Java que dinámicamente procesan consultas y construyen respuestas. Las **páginas JSP** son documentos basados en texto que se ejecutan como Servlet pero permite un enfoque más natural para crear contenido estático. La tecnología **JavaServer Faces** construida sobre tecnología Servlets y JSP, proporcionan un marco de trabajo para aplicaciones web.

Las páginas HTML estáticas y los Applets son entregados con los componentes web durante el ensamblaje de la aplicación pero no son considerados componentes web por la especificación Java EE. Las clases de utilidad del lado del servidor pueden ser empaquetadas con los componentes web y al igual que las páginas HTML no son consideradas componentes web.

Como se muestra en la Figura III, la capa web, como la capa de cliente, pueden incluir componentes JavaBeans para manejar la entrada del usuario y enviar esa entrada a beans empresariales que se ejecutan en la capa de negocio para procesamiento.

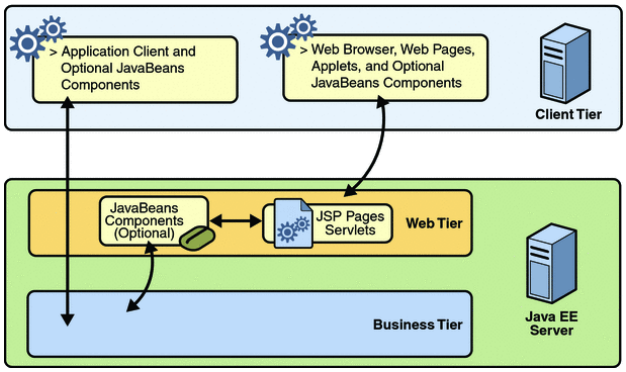


Figura III: Capa WEB y Aplicaciones Java EE

2.13.5. Componentes de Negocio

El código de negocio es la lógica que soluciona o cumple con las necesidades de un dominio de negocio en particular como lo es un banco, ventas o finanzas, el cual es manejado por beans empresariales que se ejecutan en la capa de negocio. La Figura IV, muestra como un bean empresarial recibe datos de un programa cliente, lo procesa (si es necesario) y lo envía a la capa de sistema de información empresarial para almacenar. Un bean empresarial también recupera datos del sistema de almacenamiento, lo procesa (si es necesario) y lo envía de regreso al programa cliente.

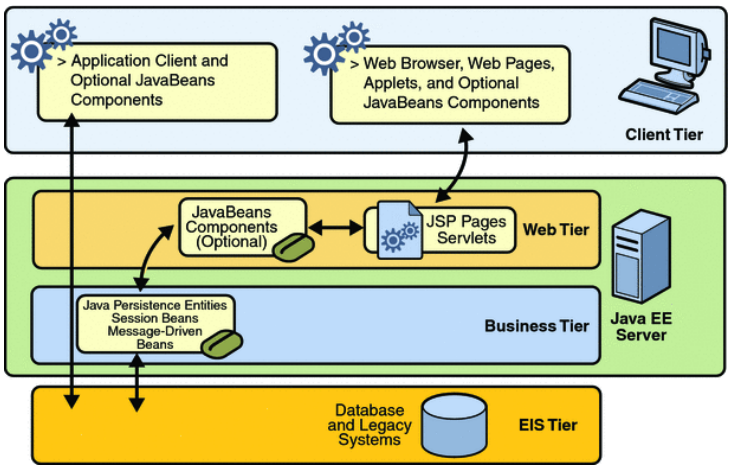


Figura IV: Capa de Negocio y Capa EIS

2.13.6. Capa del Sistema de Información Empresarial

La capa del sistema de información empresarial maneja el software EIS e incluye sistemas que son infraestructura como planeadores de recursos empresariales (ERPs), procesamiento de transacciones mainframe, sistemas de bases de datos y otros sistemas legados. Como ejemplo, los componentes de aplicaciones Java EE pueden necesitar acceder a sistemas de información empresariales para conectividad con la base de datos.

2.14. Contenedores Java EE

Normalmente las aplicaciones cliente para terminales son difíciles de escribir porque involucran muchas líneas de código intrincado para manejar transacciones y administración de estados, hilado múltiple, recursos puestos en común y otros detalles complejos de bajo nivel. La arquitectura basada en componentes e independiente de la plataforma de la arquitectura Java EE hace que las aplicaciones Java EE sean fáciles de escribir porque la lógica de negocio está organizada en componentes reutilizables. Además el servidor Java EE proporciona servicios de capas bajas en la forma de contenedores para cada tipo de componente. Dado que no se tienen que desarrollar estos servicios, se es libre de concentrarse en solucionar los problemas de negocio a mano.

2.14.1. Servicios del Contenedor

Los **contenedores** son la interfaces entre los componentes y la funcionalidad específica de la plataforma a bajo nivel que soporta el componente. Antes que un componente web, bean empresarial o cliente de aplicación pueda ser ejecutado, debe ser ensamblado en un módulo Java EE y desplegado dentro de su contenedor.

El proceso de ensamblado involucra configuración específica de contenedor para cada componente en la aplicación Java EE y para la aplicación Java EE misma. La configuración de los componentes define el soporte de capas bajas proporcionadas por el servidor EE, incluyendo los servicios como seguridad, manejo de transacciones, búsquedas en Java Naming y Directory InterfaseTM (JNDI) y conectividad remota. Algunos de los principales son:

- El modelo de seguridad de Java EE permite configurar un componente web o bean empresarial para que los recursos del sistema sean accedidos solo por usuarios autorizados.

- El modelo de transacciones de Java EE nos permite especificar las relaciones entre métodos que conformen una sola transacción de tal forma que todos los métodos en una transacción sean tratados como una sola unidad.

- Los servicios de búsqueda JNDI proporcionan una interfaz unificada para muchos servicios de nombres y directorio en la empresa de tal manera que los componentes de la aplicación puedan acceder a estos servicios.

- El modelo de conectividad remota de Java EE maneja las comunicaciones a bajo nivel entre clientes y beans empresariales. Luego de que un bean empresarial es creado un cliente invoca los métodos en este como si estuviera en la misma máquina virtual.

Debido a que la arquitectura Java EE proporciona servicios configurables, los componentes de aplicación dentro de la misma aplicación Java EE pueden comportarse de forma diferente basado en donde se hayan desplegado. Por ejemplo, un bean empresarial puede tener configuraciones de seguridad que le permitan un cierto nivel de acceso a los datos de una base de datos en un ambiente de producción y otro nivel de acceso a base de datos en otro ambiente de producción.

El contenedor también maneja servicios que no son configurables como bean empresariales y ciclos de vida de Servlets, agrupación de recursos de conexiones a bases de datos, persistencia y acceso a las APIs de la plataforma Java EE. (Vea API de Java EE 5).

2.14.2. Tipos de Contenedor

El proceso de despliegue instala componentes de aplicación Java EE en los contenedores como se ilustra en la Figura V.

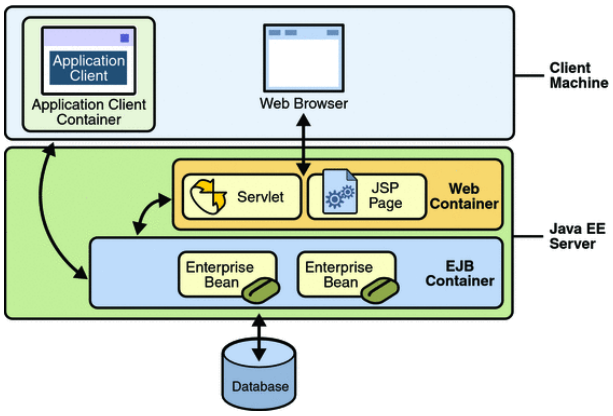


Figura V: Servidor y Contenedores Java EE

- **Servidor Java EE**: La porción en tiempo de ejecución de un producto Java EE. Un servidor Java proporciona EJB y contenedores Web.
- **Contenedor de JavaBeans empresariales (EJB)**: Maneja la ejecución de beans empresariales para aplicaciones Java EE. Los bean empresariales y su contenedor se ejecutan en el servidor Java EE.
- **Contenedor Web**: Maneja la ejecución de páginas JSP y componentes Servlets para aplicaciones Java EE. Los componentes web y sus contenedores se ejecutan en el servidor Java EE.
- **Contenedor de aplicación cliente**: Maneja la ejecución de los componentes de una aplicación cliente. La aplicación cliente y sus contenedores se ejecutan en el cliente.
- **Contendedor de applet**: Maneja la ejecución de Applets. Consiste en un navegador web y un plug-in que se ejecutan juntos en el cliente.

2.15. Ensamblaje y desplieque de una Aplicacion Java EE

Una aplicación Java EE es empaquetada en una o más unidades estándar para despliegue en cualquier sistema compatible con la plataforma Java EE. Cada unidad contiene:

- Un componente o componentes funcionales (como un bean empresarial, página JSP, servlet o Applet).
- Un descriptor de despliegue que describe su contenido.

Una vez que una unidad Java EE ha sido producida, está lista para ser desplegada. El despliegue típicamente involucra el uso de una herramienta de despliegue para especificar la información de ubicación específica, como una lista de usuarios locales que pueden acceder a esta y el nombre de la base de datos local. Una vez desplegado en una plataforma local, la aplicación esta lista para ejecutarse.

2.15.1. Empaquetado de Aplicaciones

Una aplicación Java EE es distribuida en un fichero Archivo Empresarial (EAR) que es un Archivo Java estándar (JAR) con una extensión .ear. El uso de archivos EAR y módulos hace posible ensamblar una gran cantidad de aplicaciones Java EE utilizando alguno de los mismos componentes. No se necesita codificación extra; es solo un tema de ensamble (o empaquetado) de varios módulos Java EE en u fichero EAR de Java EE.

Un fichero EAR contiene, como muestra la Figura VI, módulos Java EE y descriptores de despliegue. Un **descriptor de despliegue** es un documento XML con una extensión .xml que describe la configuración de despliegue de una aplicación, un módulo o un componente. Dado que la información en el descriptor de despliegue es declarativa, esta puede ser cambiada sin la necesidad de modificar el código fuente. En tiempo de ejecución, el servidor Java EE lee el descriptor de despliegue y actúa sobre la aplicación, módulo o componente como corresponde.

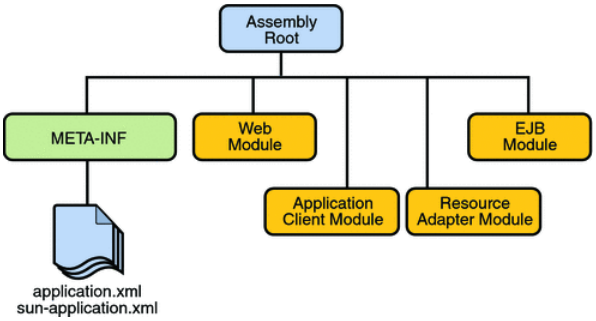


Figura VI: Estructura de un fichero EAR

Hay dos tipos de descriptor de despliegue: Java EE y en tiempo de ejecución. Un **descriptor de despliegue Java EE** está definido por una especificación Java EE y puede ser utilizado para configurar el despliegue de una implementación compatible con Java EE. Un **descriptor de despliegue en tiempo de ejecución** es utilizado para configurar parámetros específicos de una implementación Java EE. Por ejemplo, el descriptor de

despliegue en tiempo de ejecución de Sun Java System Application Server Platform Edition 9 contiene información como el contexto raíz de la aplicación web, la relación entre nombres potables de los recursos de una aplicación a los recursos del servidor y los parámetros específicos de implementación del servidor de aplicación, como las directivas de caché. Los descriptores de despliegue en tiempo de ejecución del servidor de aplicación son nombrados sun- *tipoDeModulo*.xml y están localizados en el mismo directorio META-INF que el descriptor de despliegue de Java EE.

Un **módulo Java EE** consiste en uno o más componentes Java EE para el mismo tipo de contenedor y un descriptor de despliegue de componente para ese tipo. Un descriptor de despliegue de módulo de bean empresarial, por ejemplo, declara los atributos de la transacción y las autorizaciones de seguridad para un bean empresarial. Un módulo Java EE sin un descriptor de despliegue de aplicación puede ser desplegado como un módulo independiente.
Los cuatro tipos de módulos de Java EE son los siguientes:

- Módulos EJB, que contienen los ficheros con clases para beans empresariales y un descriptor de despliegue EJB. Los módulos EJB son empaquetados como ficheros JAR con una extensión .jar.

- Los módulos web, que contienen ficheros con servlets, ficheros JSP, fichero de soporte de clases, ficheros GIF y HTML y un descriptor de despliegue de aplicación web. Los módulos web son empaquetados como ficheros JAR con una extensión .war (Web Archive).

- Los módulos de aplicaciones de cliente que contienen los ficheros con las clases y un descriptor de aplicación de cliente. Los módulos de aplicación clientes son empaquetados como ficheros JAR con una extensión .jar.

- Los módulos adaptadores de recursos, que contienen todas las interfaces Java, clases, librerías nativas y otra documentación junto con su descriptor de despliegue de adaptador de recurso. Juntos, estos implementan la arquitectura Conector (Ver Arquitectura del conector JEE) para un EIS en particular. Los módulos adaptadores son empaquetados como ficheros JAR con una extensión .rar (archivo adaptador de recurso).

2.15.2. Empaquetado de EJB simplificado

Antes de EJB 3.1 todos los beans tenían que estar empaquetados en estos archivos. Como una buena parte de todas las aplicaciones Java EE contienen un front-end web y un back-end con EJB, esto significa que debe crearse un *ear* que contenga a la aplicación con dos módulos: un ***war***y un ***ejb-jar****.*

Esto es una buena práctica en el sentido que se crea una separación estructural clara entre el front-end y el back-end. Pero para las aplicaciones simples resulta demasiado.

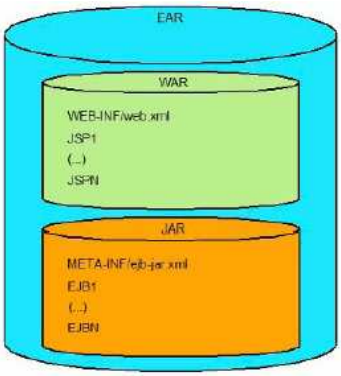


Figura VII: Estructura del un paquete EAR

EJB 3.1 permite empaquetar EJBs dentro de un archivo **war**. Las clases pueden incluirse en el directorio **WEB-INF/classes**o en un archivo jar dentro de **WEB-INF/lib**.

El **war** puede contener como máximo un archivo **ejb-jar.xml**, el cual puede estar ubicado en **WEB-INF/ejb-jar.xml** o en el directorio **META-INF/ejb-jar.xml** de un archivo **jar**.

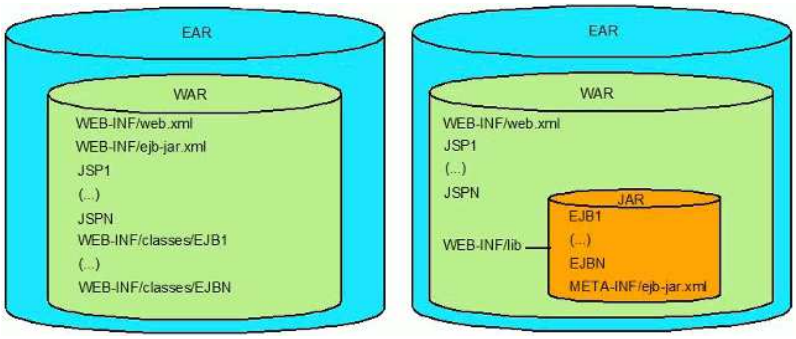


Figura VIII: Empaquetado EJB simplificado

Cabe destacar que este empaquetado simplificado sólo debería usarse en aplicaciones simples. El resto de las aplicaciones se verá beneficiada con el empaquetado actual separado.

2.16. API de Java EE 6

La Figura IX, ilustra las relaciones entre los contenedores de Java EE, para cada tipo de contenedor Java EE.

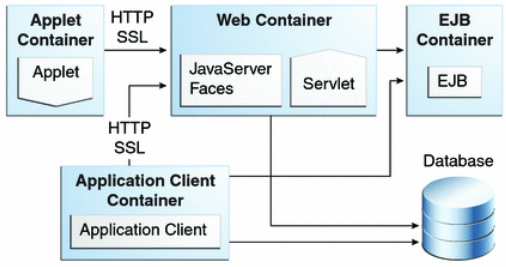


Figura IX: Contenedores Java EE y sus relaciones

En la siguiente Figura, se ilustran las API's de Java EE en el contenedor Web.

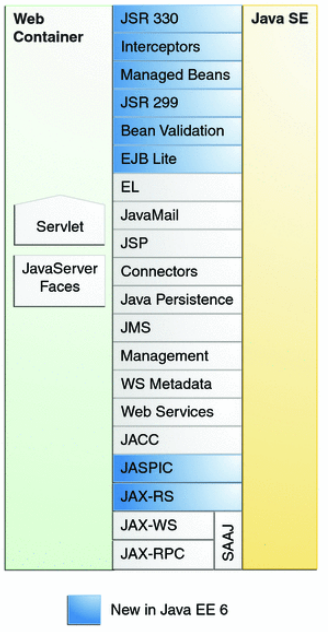


Figura X: API de Java EE en el contenedor Web

La Figura XI muestra las distintas API's de Java EE 6 en el contenedor EJB.

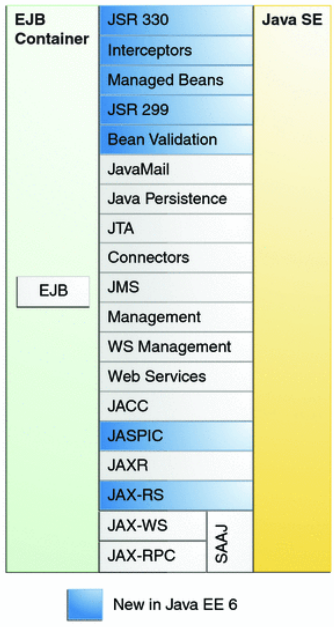


Figura XI: API de Java EE en el contenedor EJB

Las API's de Java EE 6 disponibles en el contenedor de Aplicaciones Cliente, se pueden visualizar en la Figura XII.

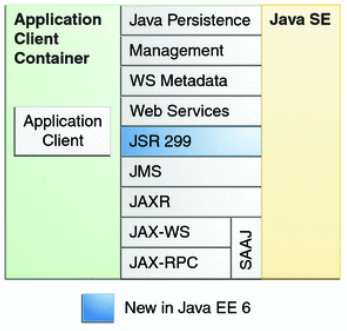


Figura XII: API de Java EE en el contenedor de Aplicaciones Cliente

En las secciones siguientes se presentará un resumen breve de las tecnologías requeridas por la plataforma Java EE y la API utilizada en las aplicaciones Java EE.

2.16.1. Tecnología de JavaBeans Empresariales

Un componente JavaBeans™ empresarial (EJB), o bean empresarial es un cuerpo de código que tiene campos y métodos para implementar módulos de lógica de negocio. Se puede pensar en un bean empresarial como un ladrillo que puede ser utilizado solo con otros beans empresariales para ejecutar lógica de negocio en el servidor Java EE.

Hay tres tipos de beans empresariales: los beans de sesión, los beans manejadores de mensajes y los singleton beans. Un **SessionBean** (bean de sesión) representa una conversación transitoria con un cliente. Cuando el cliente termina de ejecutarse, el bean de

sesión y sus datos desaparecen. Un **Message-Driven Bean** (bean manejador de mensajes) combina las características de un bean de sesión y un escucha (listener) de mensajes permitiendo a un componente de negocio recibir mensajes de forma asincrónica. Comúnmente, estos son mensajes JMS (Java Message Service).

En la versión 2.x de EJB existía un tercer tipo: **Entity Beans**, que eran objetos distribuidos para la persistencia de datos en una Base De Datos. Estos objetos fueron reemplazados en la versión 3.0 por la Java Persistence API (JPA).

En la plataforma Java EE 6, las nuevas características de los Enterprise JavaBeans son los siguientes:

- La capacidad de paquetes locales Enterprise JavaBeans en un archivo WAR.
- Singleton beans de sesión, que proporcionan un fácil acceso al estado compartido.
- Un subconjunto de funcionalidad Enterprise JavaBeans Ligero (EJB Lite) que pueden ser prestados en perfiles Java EE, tales como el perfil Java EE Web.

La especificación de interceptores, que forma parte de la especificación EJB 3.1, hace más generalizada la instalación de interceptores y se definió originalmente como parte de la especificación EJB 3.0.

2.16.2. Tecnología Java Servlet

La tecnología de servlet de Java permite definir una clase servlet específica para HTTP. Una clase servlet extiende las habilidades de los servidores que hospedan aplicaciones que son accedidas por de la forma del modelo de programación solicitud respuesta. A pesar que los servlets pueden responder a cualquier tipo de solicitud, comúnmente son utilizados para extender las aplicaciones hospedadas por servidores web.

En la plataforma Java EE 6, Java Servlet presenta nuevas tecnologías cuyas características son las siguientes:

- Anotación de apoyo.
- Asincrónica apoyo.
- Facilidad de configuración.
- Mejoras en las API’s existentes.
- Enchufabilidad (Plugins)

2.16.3. Tecnología JavaServer Faces

La tecnología JavaServer Faces es un marco de trabajo de interfaz de usuario para la creación de aplicaciones web. Los principales componentes de la tecnología JavaServer Faces son los siguientes:

- Un framework de trabajo para componentes GUI.

- Un modelo flexible para la prestación de los componentes en diferentes tipos de HTML o lenguajes de marcas y tecnologías diferentes. Un objeto Renderer genera el marcado para representar el componente y convierte los datos almacenados en un modelo de objetos a los tipos que pueden representarse en una vista.

- Un RenderKit estándar para la generación de mascas HTML/4.01.

Las siguientes características soportan los componentes GUI:

- Validación de entrada.
- Manejo de eventos.
- Conversión de datos entre los objetos del modelo y los componentes.
- Gestión del modelo de creación de objetos.
- Página de configuración de navegación.
- Expression Language (EL).

Toda esta funcionalidad está disponible utilizando las API's estándar de Java y archivos de configuración basado en XML.

En la plataforma Java EE 6, las nuevas características de JavaServer Faces son las siguientes:

- La capacidad de usar anotaciones en lugar de un archivo de configuración para especificar la gestión de beans.
- Facelets, una tecnología de visualización que suprime JavaServer Pages (JSP) usando archivos XHTML.
- Ajax de apoyo.
- Compuesto de componentes.
- Implícita de navegación.

2.16.4. Tecnología de Java Server Pages

La tecnología de JavaServer Pages ™ (JSP) nos permite colocar partes del código servlet directamente en un documento de texto. Una página JSP es un documento de texto que contiene dos tipos de texto:

- Datos estáticos, que pueden ser expresados en cualquiera de los formatos basados en texto como HTML, WML y XML.

- Elementos JSP, que determinan como la página construye el contenido dinámico.

2.16.5. Biblioteca de Etiquetas estándar JavaServer Pages

Las Bibliotecas de Etiquetas Estándar JavaServer Pages (JSTL por sus siglas en inglés) encapsula la funcionalidad principal común a muchas aplicaciones JSP. En lugar de etiquetas mezcladas de varios vendedores en sus aplicaciones JSP, se emplea un solo, grupo estándar de etiquetas. Esta estandarización permite, desplegar sus aplicaciones en cualquier contenedor que soporte JSTL y lo hace más deseable que la implementación de etiquetas optimizada.

La JSTL tiene etiquetas para crear iteradores y condiciones para manejar flujo, etiquetas para manejar documentos XML, etiquetas de internacionalización, etiquetas para acceder a bases de datos utilizando SQL y las funciones mas comúnmente utilizadas.

2.16.6. API Java de Servicio de Mensajes (JMS)

La API del servicio de mensajes de Java (JMS) es un estándar de mensajería que permite a componentes de aplicación Java EE crear, enviar, recibir y leer mensajes. Para habilitar comunicación distribuida con bajo acoplamiento, confiable y asincrónica.

2.16.7. API Java de Transacciones

La API Java de transacciones (JTA) proporciona una interfaz estándar para distinguir transacciones. La arquitectura Java EE proporciona un **commit automático** para manejar commit y rollbacks. Un auto commit significa que otras aplicaciones que están viendo estos datos verán los datos actualizados luego de que cada base de datos realice la operación de lectura o escritura. Sin embargo, si su aplicación realiza dos operaciones de acceso a base de datos que dependen una de otra se deseará utilizar la API JTA para distinguir donde la transacción completa incluyendo ambas operaciones, comienza, se regresa o se completa.

2.16.8. API JavaMail

Las aplicaciones Java EE utilizan la API JavaMail para enviar notificaciones por correo electrónico. La API de JavaMail tiende dos partes:

- Una interfaz a nivel de aplicación utilizada por los componentes de la aplicación para enviar correo.
- Una interfaz de proveedor de servicio.

La plataforma Java EE incluye JavaMail con un proveedor de servicio que permite a los componentes de aplicación enviar correo por la Internet.

2.16.9. Framework de Activación de JavaBeans

El Framework de Activación JavaBeans (JAF) está incluido porque JavaMail lo utiliza. JAF proporciona servicios estándares para determinar el tipo de un trozo arbitrario de datos, acceso encapsulado al, descubrimiento de operaciones disponibles y creación de los componentes JavaBeans apropiados para realizar estas operaciones.

2.16.10. API Java para procesamiento XML

La API de Java para procesamiento XML (JASP), parte de la plataforma Java SE, soporta el procesamiento de documentos SML utilizando el Modelo de objetos del documento (DOM), API simple para SML (SAX) y Transformaciones del Lenguaje Extensible de Hojas de estilo (XSLT). JAXP habilita a las aplicaciones a analizar la sintaxis y transformar documentos SML independientemente de una implantación de procesado XML particular.

JAXP también proporciona soporte para espacios de nombres, lo que permite trabajar con esquemas que puedan, de otra forma tener nombres con conflicto. Diseñado para ser flexible, JAXP nos permite utilizar cualquier analizador sintáctico que cumpla con procesadores XML o XSL desde su aplicación y soporte el esquema W3C.

2.16.11. API Java para Servicios Web XML (JAX-WS)

La especificación JAX-WS proporciona soporte para servicios web que utiliza la API JAXB para vincular datos XML en objetos Java. La especificación JAX-WS define APIs clientes para acceder a servicios web así como técnicas para implementar puntos finales de servicios web. Los servicios web para la especificación J2EE describen el despliegue de servicios basados en clientes JAX-WS. La especificación EJB y servlet también describe aspectos de ese desarrollo. Esto debe posibilitar el despliegue de aplicaciones basadas en JAX-WS utilizando cualquiera de estos modelos de despliegue.

La especificación JAX-WS describe el soporte para los manejadores de mensajes que pueden procesar mensajes de solicitud y respuesta. En general, estos manejadores de mensajes se ejecutan en el mismo contenedor y con los mismo privilegios y contextos de ejecución que el JAX-WS cliente o en el componente punto final con el que está asociado. Estos manejadores de mensajes tienen acceso a el mismo espacio de nombres JNDI java:comp/env como su componente asociado. Serializadores y deserializadores a la medida, si son soportados, son tratados de la misma forma que los manejadores de mensaje.

2.16.12. API Java para Servicios Web REST

El API Java para Servicios Web REST (JAX-RS) define las API para el desarrollo de servicios web construidos de acuerdo al estilo arquitectónico de la Representational State

Transfer (REST). A-RS aplicación JAX es una aplicación web que consta de clases que se empaquetan como un servlet en un archivo WAR, junto con las bibliotecas necesarias.

La API de JAX-RS es nueva en la plataforma Java EE 6.

2.16.13. Gestión de Beans

Gestión de Beans, objetos ligeros de contenedor (POJOs) administrados con los requisitos mínimos, el apoyo a un pequeño conjunto de servicios básicos, tales como la inyección de recursos, las devoluciones de llamada del ciclo de vida, y los interceptores. La Gestión de Beans representa una generalización de los beans de gestión especificados por la tecnología JavaServer Faces y se puede utilizar en cualquier lugar de una aplicación Java EE, y no sólo en los módulos web.

La especificación de Gestión de Beans es parte de la especificación EE 6 de la plataforma Java (JSR 316).

La Gestión de Beans es nueva en la plataforma Java EE 6.

2.16.14. Contextos y la inyección de dependencias para la plataforma Java EE (JSR 299)

Los Contextos y la inyección de dependencias (CDI) para la plataforma Java EE define un conjunto de servicios contextuales, suministrado por contenedores de Java EE, que facilitan a los desarrolladores utilizar beans de empresa, junto con la tecnología JavaServer Faces en aplicaciones web. Diseñado para usar con los objetos con estado, la CDI también tiene muchos usos más amplios, lo que permite a los desarrolladores una gran flexibilidad para integrar diferentes tipos de componentes en una junta, pero con seguridad de tipos.

CDI es nuevo en la plataforma Java EE 6.

2.16.15. Inyección de Dependencias para Java (JSR 330)

La Inyección de Dependencias para Java define un conjunto estándar de anotaciones (y una interfaz) para su uso en las clases de inyectables.

En la plataforma Java EE, CDI proporciona soporte para la inyección de dependencias. En concreto, puede usar la inyección de puntos DI sólo en una aplicación compatible CDI.

2.16.16. Validación de Beans

La especificación de Validación de Beans define un modelo de metadatos y API para la validación de datos de componentes JavaBeans. En lugar de distribuir la validación de datos a través de varias capas, como el navegador y el servidor, puede definir las restricciones de validación en un solo lugar y compartirlos a través de las diferentes capas.

2.16.17. API de SOAP con adjuntos para Java

La API de SOAP con adjuntos para Java (SAAJ) es una API de bajo nivel de la cual dependen JAX-WS y JAXR. SAAJ habilita la producción y el consumo de mensajes que cumplen con la especificación de SOAP1.1 y notas SOAP con adjuntos. Muchos desarrolladores no utilizan la API SAAJ, en lugar de esto utilizan la API de alto nivel JAX-WS.

2.16.18. API Java para registros XML

La API de Java para registros XML (JAXR) nos permite acceder a registros de negocio y de propósito general a través de la red. JAXR soporta los estándares ebXML para registros y repositorios y la especificación UDDI emergente. Utilizando JAXR, los desarrolladores pueden aprender una sola API y logran acceso a estas dos importantes tecnologías de registro.

Además, los negocios pueden enviar material para que sea compartido y buscar material que otros envíen. Grupos de estándares han desarrollado esquemas para tipos de documentos XML particulares; dos negocios pueden, por ejemplo, estar de acuerdo a utilizar el esquema pare la orden de compra estándar de su empresa. Dado que el esquema es almacenado en un registro de estándar de negocio, ambas partes pueden utilizar JAXR para acceder a él.

2.16.19. Arquitectura del conector JEE

La arquitectura del conector JEE es utilizada por los vendedores de herramientas e integradores de sistemas para crear adaptadores que soporten el acceso a sistemas de información empresarial que puedan ser conectados en cualquier producto Java EE. Un **adaptador de recurso** es un componente de software que permite a los componentes de aplicación Java EE acceder e interactuar con el manejador de recursos de capas bajas del EIS. Dado que un adaptador de recurso es específico de su manejador de recursos, típicamente hay un adaptador de recursos diferente para cada tipo de base de datos o sistema de información empresarial.

La arquitectura del conector JEE también proporciona una integración transaccional orientada a rendimiento, seguro, escalable y basada en mensajes de servicios web basados en Java EE con EIS existentes que pueden ser sincrónicos o asincrónicos. Aplicaciones existentes y EIS integrados a través de la arquitectura del conector J2EE dentro de la plataforma Java EE pueden exponer servicios web basados en XML utilizando JAX-WS y modelos de componentes Java EE. De esta manera JAX-WS y la arquitectura del conector JEE son tecnologías complementarias para integraciones de aplicaciones empresariales (EAI) e integraciones de negocios de un extremo al otro.

2.16.20. Contrato de autorización de Java para contenedores

El Contrato de autorización de Java para contenedores (JACC) especificación define un contrato entre un servidor de aplicaciones Java EE y una política de proveedor de autorización. Todos los contenedores de Java EE apoyar este contrato.

La especificación JACC define clases java.security.Permission que satisfagan los modelos EE de autorización de Java. La especificación define la unión de las decisiones de acceso al contenedor a las operaciones en instancias de estas clases de permisos. Se define la semántica de los proveedores de la política que utilizan las clases de permisos nuevos para hacer frente a los requisitos de autorización de la plataforma Java EE, incluida la definición y el uso de papeles.

2.16.21. Java Authentication Service Provider Interface para contenedores

La especificación de Servicio de autenticación de Java de la interfaz del proveedor para contenedores (JASPIC) define una interfaz de proveedor de servicios (SPI) por el cual los proveedores de autenticación que implementan mecanismos de autenticación de mensaje puede ser integrado en el servidor de procesamiento de mensajes o contenedores cliente o tiempos de ejecución.

Los proveedores de autenticación integrada a través de esta interfaz para operar en los mensajes de la red que les proporcionó su contenedor de llamada. Los proveedores de autenticación de transformar los mensajes salientes a fin de que la fuente del mensaje puede ser autenticado por el recipiente que recibe, y el destinatario del mensaje puede ser autenticado por el mensaje del remitente. Los proveedores de autenticación autentican los mensajes entrantes y regresan a su contenedor llamando a la identidad establecida como resultado de la autenticación de mensajes.

JASPIC es nuevo en la plataforma Java EE 6.

2.16.22. API Java de conectividad a Base de Datos

La API Java de conectividad a base de datos (JDBC) permite invocar comandos SQL desde métodos del lenguaje de programación Java. Se utiliza la API JDBC en un bean empresarial cuando se tiene un bean de sesión accediendo a la base de datos. Se puede utilizar también la API JDBC desde un servlet o una página JSP para acceder a la base de datos directamente sin pasar a través de un bean empresarial.

La API JDBC tiene dos partes: una interfaz a nivel de aplicación usado por los componentes de aplicación para acceder a la base de datos y una interfaz de proveedor de servicio para anexar un controlador JDBC a la plataforma Java EE.

2.16.23. API Java de Persistencia (JPA)

La API Java de Persistencia es una solución Java basada en estándares para persistencia. La persistencia utiliza una estrategia de "mapeo" objeto relacional para unir la brecha entre el modelo orientado a objetos y la base de datos relacional. La persistencia de Java consiste en tres áreas:

- La API Java de persistencia
- El lenguaje de consultas
- Los datos de alto nivel con el mapeo objeto relacional

Los Entity Beans forman parte de JPA el mismo que forma parte del estándar EJB 3.1.

2.16.24. Interfaces de Nombres de Directoria de Java (JNDI)

La Java Naming and Directory Interface TM (JNDI) proporciona la funcionalidad de nombres y directorios, habilitando las aplicaciones a acceder a múltiples servicios de nombres y directorios, incluyendo los servicios de nombres y directorios existentes como LDAP, NDS, DNS, y NIS. Esta le proporciona a las aplicaciones métodos para realizar operaciones de directorio estándares, como lo es asociar atributos con objetos y la búsqueda de objetos utilizando sus atributos. Usando JNDI una aplicación Java EE puede almacenar y recuperar cualquier tipo de objeto nombrado, permitiendo a las aplicaciones Java EE coexistir con muchas aplicaciones y sistemas legados.

Los servicios de nombres de Java EE proporcionan a las aplicaciones clientes, beans empresariales y componentes web con el acceso al ambiente JNDI de nombres. Un **ambiente de nombres** permite a un componente ser construido en base a especificaciones sin la necesidad de acceder o cambiar el código fuente. Un contenedor implementa el ambiente del componente y se lo proporciona al componente como un **contexto de nombres** JNDI.

Un componente Java EE puede localizar su contexto de ambiente de nombres utilizando interfaces JNDI. Un componente puede crear un objeto javax.naming.InitialContext y buscar el contexto de nombres de ambiente en InitialContext bajo el nombre de java:comp/env. Un ambiente de nombres de componentes es almacenado directamente en el contexto de ambiente de nombres o en cualquiera de sus subcontextos directamente o indirectamente.

Un componente Java EE puede acceder a objetos nombrados provistos por el sistema y definidos por el usuario. Los nombres de los objetos provistos por el sistema, como objetos JTA UserTransaction, son almacenados en el contexto de ambiente de nombres, java:comp/env. La plataforma Java EE permite a un componente nombrar objetos definidos por el usuario como bean empresariales, entradas de ambiente, objetos DataSource de JDBC, y conexiones de mensajes. Un objeto puede ser nombrado dentro de un subcontexto del ambiente de nombres de acuerdo al tipo de objeto. Por ejemplo, los beans

empresariales son nombrados dentro del subcontexto java:comp/env/ejb, y las referencias DataSource JDBC en el subcontexto java:comp/env/jdbc.

2.16.25. Servicio de autenticación y autorización Java

El servicio de autenticación y autorización Java (JAAS) proporciona una forma de que una aplicación Java EE se autentifique y autorice a un usuario o grupo de usuario a ejecutarla.

JAAS es una versión del lenguaje de programación Java del marco de trabajo PAM (Pluggable Authentication Module), que extiende la arquitectura de seguridad de la plataforma Java para soportar autorización basada en usuario.

2.16.26. Integración de sistemas simplificada

La plataforma Java EE es una solución de integración completa de sistemas independiente de la plataforma que crea un mercado abierto en donde cada vendedor puede vender a cualquier cliente. Como un mercado anima a los vendedores a completar, no tratando de cerrar a los clientes a su tecnología sino que trata de mejorar cada uno en productos y servicios que beneficie a los clientes, como mejor rendimiento, mejores herramientas o mejor soporte de cliente.

Las APIs de Java EE habilitan la integración de sistemas y aplicaciones a través de lo siguiente:

- Unificando el modelo de aplicación a través de las capas con beans empresariales.
- Simplificando el mecanismo de solicitud y respuesta con páginas JSP y Servlets.
- Un modelo de seguridad fiable con JAAS.
- Integración con JAXP, SAAJ y JAX-WS mediante intercambio de datos basado en XML.
- Interoperabilidad simplificada con la arquitectura de conector JEE.
- Fácil conectividad con bases de datos con la API JDBC.
- Integración de aplicaciones empresariales con beans manejadores de mensajes y JMS, JTA y JNDI.

2.17. Enterprise JavaBeans

Son los componentes de Java EE que aplican y se ejecutan en el contenedor EJB, un entorno de ejecución. Aunque transparente para el desarrollador de la aplicación, el contenedor EJB proporciona servicios de nivel de sistema, tales como las transacciones y la seguridad. Estos servicios le permiten crear y desplegar rápidamente beans de empresa, que forman el núcleo de las transacciones de aplicaciones Java EE.

2.17.1. Generalidades

2.17.1.1. ¿Qué es un Enterprise Bean?

Escrito en el lenguaje de programación Java, un Enterprise Bean es un componente del lado del servidor que encapsula la lógica de negocio de una aplicación. La lógica empresarial es el código que cumple la función de la aplicación.

Enterprise Java Beans (EJB) es una plataforma para construir aplicaciones de negocio portables, reusables y escalables usando el lenguaje de programación Java. Desde el punto de vista del desarrollador, un EJB es una porción de código que se ejecuta en un contenedor EJB, que no es más que un ambiente especializado (*runtime*) que povee determinados componentes de servicio.

Los EJBs pueden ser vistos como **componentes**, desde el punto de vista que encapsulan comportamiento y permite reutilizar porciones de código, pero también pueden ser vistos como un **framework**, ya que, desplegados en un contenedor, proveen servicios para el desarrollo de aplicaciones enterprise, servicios que son invisibles para el programador y no ensucian la lógica de negocio con funcionalidades trasversales al modelo de dominio (a menudo *requerimientos no funcionales* o *aspectos*). A partir de la especificación 3.0, los EJB no son más que POJOs (clases planas comunes y corrientes de Java) con algunos poderes especiales implícitos (anotaciones), que se activan en *runtime* cuando son ejecutados en un contenedor de EJBs.

Figura XIII:Una anotación transforma un simple POJO en un EJB

Los servicios que debe proveer el contenedor de EJBs deben ser especificados por el programador a través de *metadata* de configuración que puede escribirse como:

- Anotaciones de Java intercaladas en el código de las clases.
- Descriptores XML (archivos XML separados).

A partir de EJB 3 se puede usar cualquiera de estas dos técnicas. Las técnicas no son exclusivas, pueden coexistir anotaciones con descriptores XML y, en el caso de

superponerse la *metadata*, los XML tendrán prioridad y podrán sobreescribir las anotaciones.

2.17.1.2. **Beneficios de los Enterprise JavaBean**

Por varias razones, los Enterprise JavaBean simplifican el desarrollo de aplicaciones grandes y distribuidas. En primer lugar, porque el contenedor EJB proporciona un nivel de servicios del sistema a los beans de la empresa, el desarrollador de bean se puede concentrar en la solución de problemas empresariales. El contenedor EJB, más que el desarrollador del bean, es responsable por el nivel de los servicios del sistema, tales como gestión de transacciones y la autorización de seguridad.

En segundo lugar, porque los beans en lugar de los clientes contienen la aplicación lógica de negocio, el desarrollador del cliente puede centrarse en la presentación del cliente. El desarrollador del cliente no tiene que codificar las rutinas que implementan las reglas empresariales o bases de datos de acceso. Como resultado, los clientes son más livianos, una ventaja que es especialmente importante para los clientes que se ejecutan en dispositivos pequeños.

En tercer lugar, porque los Enterprise JavaBeans son componentes portátiles, el ensamblador de la aplicación puede construir nuevas aplicaciones de los beans existentes. Siempre que utilicen las API estándar, estas aplicaciones pueden ejecutarse en cualquier servidor compatible con Java EE.

2.17.1.3. **Cuándo utilizar Enterprise Java Beans**

Se debe considerar el uso de EJB si la aplicación tiene alguno de los siguientes requisitos:

- **La solicitud debe ser escalable.** Para dar cabida a un número creciente de usuarios, puede ser necesario distribuir los componentes de una aplicación a través de múltiples máquinas. No sólo los EJB de una aplicación serán ejecutados en máquinas diferentes, sino también su situación seguirá siendo transparente para los clientes.

- **Las transacciones deben garantizar la integridad de datos.** Enterprise JavaBeans de Transacciones son de apoyo a los mecanismos que gestionan el acceso concurrente de objetos compartidos.

- **La aplicación tendrá una gran variedad de clientes.** Con sólo unas pocas líneas de código, los clientes remotos pueden localizar fácilmente los EJB. Estos clientes pueden ser delgados, diversos y numerosos.

2.17.1.4. **Roles y Responsabilidades EJB**

La arquitectura EJB define 7 roles diferentes en el ciclo de vida de desarrollo y despliegue de las aplicaciones. Cada rol EJB puede ser llevado a cabo por un grupo diferente. La arquitectura EJB especifica los contratos que aseguran que el productos de cada rol EJB es compatible con el producto de los otros roles EJB.

El estándar EJB prevé junto a la definición de los detalles técnicos también un esquema de roles de la programación hasta el uso de la aplicación, que pueden adoptarse dentro de una empresa por una o mas personas o departamentos. De hecho es bastante razonable y en grandes proyectos incluso imprescindible ocuparse de esta concepción y nombrar personas o departamentos concretos dentro de la organización.

En una pequeña empresa, en la que un departamento conste de tres personas que se ocupan ellas mismas del negocio, cada una de estas desempeñará también todos los roles. Si en cambio se trata de una organización mayor o en la que se trabaja con socios en el exterior, la cosa es bastante diferente. La mejor manera de explicar el concepto es partiendo de una situación en la que hay fabricante de un software estándar y los desarrolladores de la misma casa programan clases y beans adicionales para ampliar esta aplicación o para adaptarla a circunstancias especificas. En este caso ya se sobreentiende que desplegar la aplicación no puede ser incumbencia del fabricante. ¿Cómo podría hacerlo? Por una parte debería tener acceso desde el exterior al servidor y por otra disponer del código fuente de las aplicaciones de cada cliente. Algo parecido seria el caso cuando se desarrolla un software para diferentes departamentos dentro de una empresa y cada ámbito de desarrollo controla una parte. También aquí será necesaria una sección de coordinación central que monte e instale todas las partes.

Por esta razón al definir el estándar de EJB se ha reflexionado acerca de cómo abordar este tipo de desarrollo de software y de funcionamiento en una empresa. Depende en gran medida del tamaño de la empresa y de las responsabilidades internas si los roles que describiremos a continuación deben asumirse por personas totalmente diferentes o por el contrario siempre por la misma. Si se empiezan a programar aplicaciones Java en la empresa misma, es en todo caso razonable fijar que roles asume cada uno y comprobar si estas personas disponen también del conocimiento Know-how requerido.

En los siguientes apartados se definen los 7 roles EJB:

2.17.1.4.1. Enterprise Bean Provider

Este rol es el que se asigna a los desarrolladores del software, es decir a aquellas personas que programan los beans. Para ello necesitan un entorno de desarrollo, como Eclipse, NetBeans, etc.

También necesitan un servidor de aplicaciones para poder testear su programa. En el caso ideal cada desarrollador dispone de una instalación local de este servidor porque es muy

usual que durante la programación se produzcan errores graves y resulta muy contraproducente que los desarrolladores se perjudiquen entre si.

El tercer requisito es que todo aquel implicado en el desarrollo disponga por lo menos de un esquema de base de datos propio, sobre el que tenga plenos derechos, por ejemplo fuera necesario ampliar una tabla constituyente, será competencia de una persona del equipo adaptar el entity bean y la lógica de aplicación correspondiente. Si todos utilizan el mismo esquema de bases de datos, los colegas mientras tanto no podrían trabajar.

Igual que en el desarrollo de aplicaciones clásico, debe haber un concepto de desarrollo de varios niveles. Junto al entorno de trabajo de cada desarrollador abra de todos los demás, debe haber un entorno de trabajo de prueba común, al que solo lo ven aquellos módulos presuntamente programados y al que puedan acceder los programadores para poder probar sus funciones.

Según la especificación el desarrollador ofrece las clases bean completas y todos los deployment descriptors necesarios.

2.17.1.4.2. Application Assembler

El Application Assembler se ocupa de ensamblar todas las clases beans y deployment descriptors en unidades mayores. Así se crean archivos JAR o WAR que contienen todo lo necesario. Estos archivos se diferencian entre aquello que se transfiere posteriormente al servidor de aplicaciones y aquellos que necesitan un Cliente para poder registrarse en la aplicación.

También un fabricante externo o un departamento de desarrollo interno realiza los roles de Application Assembler para sus beans. Es habitual entregar archivos JAR o WAR en lugar de clases Java sueltas. Lo que pertenece en todo caso crear son de nuevo los Deployment Descriptor, que bajo según qué circunstancias deben ser ampliados. En EJB 2.1 aún era necesario inscribir referencias locales del session bean a los entity beans y otros session beans. Esto dejo de ser así, gracias a las anotaciones el servidor de aplicaciones reconoce él mismo estas interdependencias.

Cada departamento de desarrollo, el fabricante y el programador de los clientes, montan juntos hasta llegar a este punto su parte de la aplicación y trasmiten el resultado después al Deployer, responsable del siguiente paso. Como se puede leer en la especificación todavía es competencia del Application Assembler disponer aquellos componentes que no constan de beans, incluso aquellos que puede que ni siguiera estén programados en Java. A menudo es necesario poner a disposición sistemas externos, que realizan ellos mismos accesos a bases de datos, con total protección de transacción. Para esto último debe programarse un Resource Adapter según el Java Connector Architecture, que se compone aparte de las clases Java de nuevo también de un deployment descriptor. El fabricante de este sistema o el departamento correspondiente son los responsables de que el Deployer reciba un módulo completo que pueda vincular con el otro.

2.17.1.4.3. Deployer

Al Deployer se le encomienda la tarea de vincular cada archivo JAR y WAR del Application Assembler con una aplicación común. Normalmente aquí se crea el archivo EAR que lo resume todo y representa una Persistente Unit independiente. A menudo también es necesario integrar bibliotecas adicionales que requieren componentes separados para su trabajo.

El Deployer crea también el Deployment Descriptor application.xml, en el que por ejemplo fija individualmente a través de qué URL puede hacerse alusión a cada archivo WAR.

Este trabajo ya no puede realizarse por un proveedor externo, a no ser que realmente no haya ninguna ampliación individual en el sistema. Si se tiene que ver con varios departamentos de desarrollo en la propia empresa, se necesitará un punto independiente, que como Deployer se responsabilice de recoger cada una de las partes individuales y las vincule si es necesario según las instrucciones.

También pertenece a sus tareas, desplegar realmente la aplicación, es decir darla a conocer al servidor de aplicaciones deseado. Este debe saber si debe llevar la aplicación a un sistema de prueba o a la producción, y es necesario que este disponga de las herramientas necesarias del fabricante del servidor de aplicaciones.

También conecta el Persistente Unit con un Data Source concreto, a través del cual se define la conexión a la base de datos. Para ello debe depositar el UserId y la Contraseña bajo los que se registrará el servidor de aplicaciones en la base de datos. Eventualmente estas conexiones pueden crearse por otros como los administradores de la base de datos y entonces el Deployer simplemente hará referencia a estas.

2.17.1.4.4. EJB Server Provider

Este pone a disposición las funciones básicas para el uso de un servidor de aplicaciones y se ocupa de que los diferentes EJB-Containers dispongan de un entorno en tiempo de ejecución estable. También garantiza al administrador de transacción la capacidad de cluster del servidor.

Según la especificación se tratará de la misma persona o de la misma organización en el EJB Server Provider y en el EJB Container Provider.

2.17.1.4.5. EJB Container Provider

El fabricante del EJB-Container se ocupa de que los beans programados según la especificación sean ejecutables y que disponen de todos los servicios prescritos también por la especificación. Pone a disposición herramientas mediante las cuales el Deployer puede dar a conocer una aplicación.

El Container Provider se basa en la intersección del Server Provider y ofrece además al Persistente Provider la posibilidad de sincronizar todas las modificaciones del entity beans con un medio de guardado externo, como una base de datos.

2.17.1.4.6. Persistence Provider

El Persistence Provider debe posibilitar a los Entity Beans el hacer sus datos persistentes, es decir guardarlos permanentemente. Pone a disposición las herramientas que se encargan del mapeo entre clases Java y normalmente tablas de bases de datos y que importante sobretodo ocultar ante el container todos los específicos de la capa de persistencia subyacente. Debe ser posible sustituir en cualquier momento esta capa por otra, sin que se produzca ningún cambio en el container o en los beans desplegados.

La especificación EJB no prevé ninguna intersección entre el Server Provider, el Container Provider y el Persistence Provider. Más bien se hace referencia explícitamente que los tres roles mencionados se desempeñen por la misma organización. Solo en concreto del fabricante del servidor de aplicaciones que se quiera utilizar.

No es tan fácil decir qué servidor de qué creador es el más adecuado. Dado que mucho de nuestro grado de exigencia del software, el entorno en el que se ejecute es del soporte, que puede proporcionar el fabricante. También el número de instancia afines y las posibilidades de integración en entornos de desarrollo ya existentes son criterios importantes. Una vez se ha optado por un fabricante tampoco es recomendado vincularse a este perpetuamente incorporando por ejemplo muchas características dependientes del mismo fabricante en la aplicación. De mantenerse siempre flexibilidad, para poder cambiar si es necesario el servidor en cualquier momento.

2.17.1.4.7. System Administrator

El System Administrator es responsable de la configuración y administración de la infraestructura de computación y red (incluye el perfecto funcionamiento del servidor y contenedor EJB) de la empresa. Es también el responsable de revisar el inicio correcto del despliegue de los beans de la aplicación en ejecución.

Además, el servidor o servidores de aplicación que funcionan en una empresa deben ser controlados y evaluados. El funcionamiento de una aplicación EJB no se puede mantener gratis. Según la dimensión, la frecuencia de modificaciones de la aplicación y la cuestión de cuántos usuarios trabajan de forma paralela con el sistema, puede ser necesario prever a una persona o a todo un departamento para la administración.

El Administrador debe disponer de las herramientas adecuadas para poder controlar el transcurso del funcionamiento, debe poder reconocer a tiempo si hay espacio disponible suficiente en la memoria principal y si se está en situación de poder reaccionar rápidamente en caso de que caiga un servidor o todo un cluster. En el caso ideal puede analizar si la potencia de las fuentes disponibles es suficiente o si por el contrario sería necesario aportar al cluster otro ordenador. En momentos de rendimiento máximo también

es necesario saber si el servidor o la base de datos están saturados o si ambos se cargan, porque la red ya no puede resistir tantas exigencias.

2.17.1.5. **Operaciones con Beans no permitidas**

Hay varias cosas que un desarrollador de aplicaciones no debe realizar al programar Enterprise Java Beans. Esto se debe simplemente a que se encuentra en una arquitectura de servidor y los controles de todos los recursos no los tiene él solo. El servidor de aplicaciones es parte de cada llamada de métodos y otorga el control temporalmente a todos los métodos de negocio y se debe recordar que al finalizar el método se vuelve a ramificar en el servidor de aplicaciones. Así por ejemplo iniciar y finalizar una transacción es tarea del servidor, a no ser que se programen los beans correspondientes, que desempeñen ellos mismos esta acción. El acceso general a las bases de datos, darse de alta en ellas, la administración de los EntityBeans creados y muchos más lo lleva a cabo el entorno del servidor en el que está anidado cada bean.

- Un bean no debe utilizar atributos estáticos. Si aún así se programan, no ocurre nada especial al principio. No hay ningún mensaje de error ni se produce de inmediato ningún problema. Esta limitación por lo tanto no es técnica, si no que se debe simplemente a cuestiones profesionales. El contenido de un atributo estático sirva para todas las instancias de la clase de beans y debe tener en cuenta que posteriormente trabajarán varios usuarios a la vez con el sistema. Por lo tanto no tiene ningún sentido guardar cualquier tipo de información en un atributo de este tipo, porque entonces solo podrá trabajar una persona con el sistema. Si se quisiera leer cualquier tabla de datos una sola vez desde la base de datos y ponerla a disposición de todos los lectores, entonces se podría realizar esto con atributos estáticos, pero no se debe olvidar que estos están limitados dentro de un Cluster de una Java VM. Por lo tanto cada VM dentro de un cluster debe preocuparse de su propio caching.

- En un bean no se debe utilizar nada de un AWT o del Swing Package. Tan poco tiene sentido querer mostrar superficies gráficas en un servidor. ¿Quién aparte del administrador del sistema, podría verlas? Pero incluso esto es prohibido. No puede ser que el servidor de aplicaciones tenga que esperar hasta que alguien pulse la tecla de Aceptar.

- Están prohibidas todas las clases de java.io. esto significa que un bean nunca debe leer o escribir un archivo o un directorio clásicos. Pensemos de nuevo en el funcionamiento del Cluster. Aquí puede ocurrir que se ejecute la primera llamada del método en el ordenador A y la segunda en el ordenador B, que técnicamente pueden estar totalmente separados el uno del otro. Solo el servidor de aplicaciones sabe algo sobre estos. ¿Qué se pretende hacer con el archivo situado en el ordenador A si uno se encuentra de repente en el ordenador B? aún más grave puede ser el hecho que sesenta usuarios intenten a la vez mediante su clave de bean escribir algo en el

mismo archivo. De ahí no surgiría nunca nada razonable. Thread-Synchronisation por cierto también está prohibida.

- Tampoco se permite la creación y administración de conexiones Socket propias. El problema es que también aquí se debe acceder al threadmanagement del servidor y todos los recursos, y también las conexiones de red, pertenecen al servidor.

- No se debe utilizar ningún cargador de clases propio, porque el único que controla qué clases se ejecutan es el servidor de aplicaciones.

- No se debe iniciar o parar ningún thread propio. Si esto ocurre, no se sabe lo que podría provocar.

- Tampoco está permitido cargar bibliotecas nativas. El servidor de aplicaciones no lo evitará, pero no obstante este tipo de bibliotecas suponen a menudo inestabilidad del servidor. Si en la empresa ubiera un ordenador central programado en C y situado en Windows como DLL, como desarrollador se va a poder evitar utilizarlo. El objetivo no puede ser programar de nuevo toda la matemática de Java. No obstante el ordenador central debe asegurar en todo caso que puede ser utilizado por varios usuarios. Java y especialmente un servidor de aplicaciones pone todos los servicios a disposición de muchos clientes simultáneamente. Por lo tanto no puede ser que estos deban esperar su turno en el ordenador central. No está permitida, como ya hemos dicho, la Thread-Synchronisation. Sería especialmente grave que el ordenador central quisiera acceder a una base de datos e incluso en el peor de los casos, incluso escribir en ella. Si ocurriera, debe vincularse mediante J2EE Connector Architecture como Ressource Adapter, porque entonces se podrá crear una protección de transacción común.

- La referencia **this** es bastante crítica. Cada bean se ejecuta únicamente bajo los controles del servidor de aplicaciones y estos no deben evitarse bajo ningún concepto. Por eso no se debe proponer **this** ni como método ni como resultado de un método propio. Para cada Session Bean hay un Session Handle, que puede incluso guardarse en una tabla de base de datos. Este Handle sustituye la referencia **this**.

Como ya hemos dicho, muchas de estas limitaciones se deben a la arquitectura. Pero aún así es necesario ocuparse de ellas u pensar siempre durante el trabajo cotidiano en que habrá varios usuarios trabajando simultáneamente con el sistema y en que un servidor de aplicaciones puede extenderse a través de varios ordenadores físicos en un cluster. Estos dos aspectos deben tenerse en cuenta si se pretende programar una funcionalidad particular, que no forma parte del Estándar.

2.17.1.6. **Anotaciones**

2.17.1.6.1. Uso

Las anotaciones son observaciones que se pueden añadir a una clase para describirlas con más precisión. Las anotaciones no contienen lógica ninguna, sino que marcan una clase o solo un atributo o un método de esta. Por ejemplo la anotación @Stateless marca un clase Java como bean sin estado. Debido a está anotación el compilador Java escribe información correspondiente en la clase Java traducida, que podrá ser seleccionada después con la ayuda del API Java Reflection. El servidor de aplicaciones examina todas las clases hechas conocidas por él y pregunta a cada una de ellas si lleva una anotación como @Stateless. Si es así, la clase se manejará de manera concreta, este caso como un bean sin estado.

En el Código 2.1 se representa la sintaxis de la anotación @Stateless. Como ve, se trata de un tipo de interfaz. Se reconoce que es un anotación por el símbolo @ delante de interface.

```
@Target(TYPE) @Retention(RUNTIME)
Public @interface Stateless
{
String name() default “”;
String mappedName() default “”;
String description() default “”;
}
```

Código 2.1: Anotación @Stateless

Una anotación también puede contener anotaciones. Se llaman metaanotaciones. En el Código 2.1 estas son @Target y @Retention. Con la metaanotación @Target se fija, dónde se debe colocar la anotación. Posible valores son TYPE, o sea en nivel de clase, FIELD para atributos o METHOD para métodos. Aún hay otras posibilidades pero se utilizan más bien poco. La anotación presentada @Stateless solo se puede utilizar por lo tanto en niveles de clases, si se escribe antes de un atributo o un método se producirá un error de traducción. Es totalmente posible que una anotación conste una vez de varias entradas de tipo. Entonces se puede usar tanto en un nivel como en otro. Si la anotación @Target no conlleva ningún parámetro, esta anotación no se podrá colocar nunca en una clase Java, sino solo utilizar como metaanotaciones en otras anotaciones.

La metaanotación @Retention aparece en el ejemplo con el valor RUNTIME. Así se puede valorar la observación de la duración. Si aquí se fijara el valor CLASS, que es el predeterminado, se producirá una entrada en la clase Java traducida, pero no podría consultarse. Una anotación con @Retention(SOURCE) se ignorará completamente por el compilador. Entonces tan solo se dota de código fuente con una anotación.

Cada anotación puede tener tantos parámetros como se quiera. El tipo de estos parámetros no está limitado. Puede ser clases Java normales o utilizar también tablas de estas. También se permite una anotación como tipo. En el ejemplo, los parámetros son las tres cadenas de símbolos **name**, **mappedName** y **description**. Las tres constan por definición de un valor por defecto. Si al utilizar la anotación @Stateless no se fija ninguno

de estos valores, se obtendrá cadenas de símbolos vacías. Si falta la entrada por defecto en uno de estos parámetros, se debe fijar sin falta al utilizarse.

Si se marca una clase Java con la anotación @Stateless y no se realizan más entradas se recurrirá al nombre de clase de la clase Java para el nombre del bean. Si se quiere definir otro nombre entonces se describe por ejemplo @Stateless(name=”NombrePropio”). Así el parámetro ya está abastecido.

```
@Target(METHOD) @Retention(RUNTIME)
Public @interface Timeout { }
```

Código 2.2: Anotación @Timeout

En el Código 2.2 aparece otro ejemplo de anotación. Esta solo puede encontrarse en un nivel de método y caracteriza en un bean aquellos métodos que deben ser llamados por el Timer Service del servidor de aplicaciones cuando llegue el momento. También aquí se ve muy claramente cómo deben utilizarse este tipo de anotaciones. Toda la información necesaria que se desprende de la simple lógica de aplicación, se puede entregar de esta manera a las clases Java.

Antes era necesario compartir con el servidor de aplicaciones mediante un archivo adicional en XML, de qué tipo de beans se trataba en cada clase. Estos Deployment Descriptors deben mantenerse, lo que se ha hecho o bien a mano o con ayuda de herramientas como XDoclet. Rápidamente crecieron y se hicieron inabarcables. La utilización de anotaciones representa aquí una enorme simplificación para el desarrollador. Hoy en día se puede prescindir casi totalmente de los Deployment Descriptors. Apenas contienen ya información.

Las anotaciones también pueden aparecer anidadas. Un ejemplo de ello es el Código 2.3. A través de una anotación externa @NamedQueries (atención con el plural) se puede distribuir una lista completa de anotaciones @NamedQuery. El ejemplo sirve para la definición de la búsqueda de una base de datos. Su nombre deber ser Banco.BuscarPorCB, contiene el parámetro **query** y una expresión válida EJB-QL.

```
@NamedQueries
({
@NamedQuery(name = “Banco.BuscarPorCB”,
query = “SELECT b FROM Banco b”
+ “WHERE b.cb = :cb ORDER BY b.banco”)
})
```

Código 2.3: Ejemplo de anotaciones anidadas

Se tienen toda una serie de anotaciones con las que se pueden describir clase. Por ejemplo la anotación @Entity, que afirma que se trata de un EntityBean.

2.17.1.6.2. Programación

Se pueden programar anotaciones propias, para designar así sus clases. El servidor de aplicaciones no se interesará por estas, pero quizás si un cliente escrito por el mismo. Si por ejemplo se quiere utilizar la clase bean de identidad ya presentada también para el transporte de datos al cliente y ofrecerle información adicional para visualización de datos, puede ser útil definir para este objetivo anotaciones propias. El ejemplo correspondiente se muestra en el Código 2.4.

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface CampoDeDatos
{
String definicion() default “”;
TipoDeDatos tipodedatos() default TipoDeDatos.TEXTO;
}
```

Código 2.4: Anotación propia

La anotación @CampoDeDatos se puede utilizar en el nivel de los atributos y con los dos parámetros, ambos prescindibles. El parámetro **definicion** indica bajo qué nombre se tiene que presentar el campo correspondiente sobre la superficie del cliente, si falta esta entrada se utiliza simplemente el nombre del atributo mismo. El parámetro **tipodedatos** quiere influenciar una posible entrada de campo. El valor se ocupa del TEXTO, pero se puede sobreescribir al utilizar una anotación. Para una mayor compresión se encuentra la enumeración **TipoDeDatos** en el Código 2.5. Todos los parámetros se programan en forma de métodos, el valor estándar se indica con la palabra clave default.

```
public enum TipoDeDatos
{
TEXTO,
NUMERO,
FECHA
}
```

Código 2.5: Enumeración TipoDeDatos

Ahora se puede otorgar cada atributo del con la anotación correspondiente. Qué aspecto tiene esto se representa brevemente en el Código 2.6. Solo atributos numeroConj, cb y banco tiene la anotación @CampoDeDatos, lo que significa que solo se mostrarán estos sobre una superficie. En Banco no se otorga ni definición ni tipodedatos por lo que se fijan los valores por defecto.

```
@Entity
public class Banco implements java.io.Serializable
{
@CampoDeDatos( definicion = “NumeroDatos”,
tipodedatos = TipoDeDatos.NUMERO )
```

```
private int numconj;
@CampoDeDatos( definicion = “ClaveBancaria”,
tipodedatos = TipoDeDatos.NUMERO )
private int cb;
@CampoDeDatos
private String banco;
private String cifracomprob;
}
```

Código 2.6: EntityBean con anotaciones propias.

La mejor indicación sin embargo no servirá de nada si no hay nadie que la valore. Para ello se amplían en Java 5 las clases **Class**, **Method**, **Constructor** y **Field** con el método **getAnnotation**(), que espera como parámetro la clase de la anotación buscada y la ofrece como **object** en el resultado. Si el objeto consultado no contiene la anotación, se ofrece **null**. Por lo tanto se debe preguntar intencionadamente por la observación. Como alternativa se puede dar una tabla con todas las anotaciones asignadas con la llamada **getAnnotation**().

En el Código 2.7 se representa un cliente con el nombre AnotacionCliente, dado de alta en el servidor de aplicaciones y que recoge desde allí una instancia de Banco. Después investiga los atributos de esta clase con la anotación @CampoDeDatos y reproduce aquello que encuentra.

```
import java.lang.reflect.Field;
import java.util.Properties;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
public class AnotacionCliente
{
public static void main(String[ ] args) throws Exception
{
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY, “org.jnp.interfaces.NamingContextFactory”);
p.put(Context.URL_PKG_PREFIXES, “org.jboss.naming:org.jnp.interfaces”);
p.put(Context.PROVIDER_URL, “jnp://localhost:1099”);
Context ctx = new InitialContext(p);
Object ref = ctx.lookup(“BancoBean/remote”);
BancoRemote br = (BancoRemote) PortableRemoteObject.narrow (ref, BancoRemote.class);
Banco banco = br.getBanco();
Field[] fields = banco.gelClass().getDeclaredFields();
for( Field field: fields)
{
CampoDeDatos Campodedatos = field.getAnnotation(CampoDeDatos.class);
if( Campodedatos != null )
{
System.out.println(“Encontrado campo con anotación: ” + field.getName());
If( Campodedatos.definicion().length() != 0 )
{
System.out.println(“\t Campodedatos.definicion = ” + Campodedatos.definicion());
}
System.out.println(“\t Campodedatos.tipodedatos = ” + campodedatos.tipodedatos());
}
```

```
}
}
}
```

Código 2.7: Cliente para la valoración de anotaciones.

Al iniciar el cliente, este ofrece el siguiente resultado, representado en el Código 2.8.

```
Encontrado campo con anotación: numconj
Campodedatos.definicion = Numeroconjuntodedatos
Campodedatos.tipodedatos = NUMERO
Encontrado campo con anotación: cb
Campodedatos.definicion = Clave bancaria
Campodedatos.tipodedatos = NUMERO
Encontrado campo con anotación: banco
Campodedatos.tipodedatos = TEXTO
```

Código 2.8: Emisión del Cliente

Si nos ocupamos más a fondo del ejemplo, llama la atención el hecho que se construya una instancia de la clase Banco del servidor pero que aún así se examine después la clase. Esto se especifica en la entrada:

```
Field[] fields = banco.gelClass().getDeclaredFields();
```

Se aplica el método **getClass()** a la variable de la instancia y para este el método **getDeclaredFields()**. El método **getFields()** ofrecería los atributos de acceso libre y todos los campos de la clase Banco son prívate, por eso **getDeclaredFields()**. La instancia se serializa al transmitirse del servidor al Cliente y se vuelve a construir con una clase correcta en el cliente. Para ello el cliente debe tener acceso a la clase correspondiente. Así pues es posible ahorrarse toda la declaración en el servidor y retomar una instancia viva. Así se muestra que las anotaciones ya se interpretan junto con sus parámetros durante el tiempo de compilación, se valoran y se escriben en el archivo de clase traducido. Y aquí se encuentran también los límites de la anotación. Si se quieren fijar estas o sus parámetros dinámicamente, no hay ninguna oportunidad. Por esa razón el estándar EJB prevé todavía hoy la utilización del Deploymente Descriptors (archivos XML externos para la descripción de uno solo o de todos los beans). Si se utilizan se pueden sobrescribir todas las anotaciones fijadas en las clases. Esto no es sin embargo ninguna funcionalidad estándar de Java, sino que se programa así explícitamente por el servidor de aplicaciones. Si se encontrara una definición en un Deployment Descriptor, entonces esta sobrescribiría o completaría las anotaciones.

2.17.1.7. **Generics**

Los llamados Generics se introducen por primera vez en este lenguaje de programación con Java 5. Al contrario que las anotaciones, los Generics no se utilizan directamente en el

estándar EJB. Aún así contribuyen enormemente a la mejora del código fuente escrito porque procuran específicamente una mayor tipicidad. Para entenderlos basta con comprender las formas de escritura de las clases genéricas, no es necesario programarlos uno mismo.

Para ello, un simple ejemplo. La clase **java.util.vector** existe desde el JDK 1.0, o sea desde el principio. Implementa una tabla dinámicamente creciente de cualquier tipo de objetos. Para poder admitir un nuevo elemento la clase prevé el método **add()**, que espera cualquier instancia **object**. Como ya se sabe, todas las clases se derivan de **object**, por lo tanto se pueden reunir diferentes tipos en un único vector. Si después se quiere acceder a un elemento concreto, se llama al método **get()**, que espera un índice y ofrece como resultado una instancia de **Object**, es más, el elemento exacto que está guardado en el índice nombrado. De qué se trata, es algo que debe saber el programador. Si se vincular el resultado al tipo incorrecto –da por hecho que allí se ha grabado un String, pero no recibe ningún Integer- esto provocará durante el desarrollo un error muy aparatoso, que se puede evitar con los Generics.

El secreto consiste simplemente en inicializar el vector junto con un tipo concreto. Cuando se escribe:

```
vector v = new vector();
```

No se trata entonces de un tipo genérico, sino de un vector como los que ya conocemos y es descrito como tal. Si por el contrario solo se administran cadenas de símbolos en el vector, entonces se da esto al llamar el constructor correspondiente.

```
vector<String> v = new vector <String> ();
```

El tipo deseado se escribe inmediatamente después del nombre de clase entre paréntesis angulares. En uno de los vectores creados de este modo solo se puede guardar instancias de String o, si las hubiera, clases derivadas de String. El intento de otorgar en el método **add()** de este contenedor un Integer, conlleva un error de traducción y el del método **get()** devuelve para este vector siempre String, por lo que no es necesario una transformación del tipo de resultado. Las siguientes líneas de código se pueden aplicar sin problemas al vector genérico.

```
v.add("Es un String");
String erg = v.get(0);
```

2.17.2. Tipos de EJB

Actualmente, en la especificación EJB, existen 2 tipos principales de EJB. El Código 2.1 resume los tres tipos de Enterprise JavaBeans. En las secciones siguientes se describirán cada tipo con más detalle:

	Propósito

Tipos de EJB	
Session Bean	Realiza una tarea para un cliente, opcionalmente, puede llevar a cabo un servicio web. Existen tres tipo de Sessions Beans: Stateless Session Bean, Stateful Session Bean y los Singleton Session Bean.
Message-Driven Bean	Actúa como un detector para un tipo particular de mensajería, tales como Java Message Service API. Los Message-Driven Bean consumen JMS. Es decir que la comunicación entre un cliente y un Message-Driven Bean se produce mediante JMS.

Tabla II.I. Tipos de EJB

En la versión 2.x de los EJB existía un tercer tipo: **Entity Beans**, que eran objetos distribuidos para la persistencia de datos en una Base De Datos. Estos objetos fueron reemplazados en la versión 3.0 por la Java Persistence API (JPA), es decir los Entity Beans forman parte del estándar EJB para la persistencia de datos JPA.

2.17.2.1. **Stateless Session Beans**

2.17.2.1.1. Uso

Hay dos tipos de Session Beans: con estado y sin estado. Los Session Beans sin estado son los Stateless Session Beans y los más sencillos respecto a su creación y administración. Normalmente se utilizan para ofrecer servicios concretos, que no es necesario anidar en ningún otro contexto profesional. Con ello nos referimos a servicios como la comprobación del número de una tarjeta de crédito o la investigación acerca de la cantidad de existencias de un artículo. Todas las demás informaciones necesarias se pueden transferir al llamar el servicio y no es necesario controlar además información extra en el servidor. Si se quiere, se puede afirmar que en lo que se refiere a los métodos de un Stateless Bean se trata de funciones simple, capaces de trabajar tan solo con sus parámetros.

Especialmente este último aspecto es importante para la diferencia entre Stateless Beans y Stateful Beans. Estos últimos son capaces de guardar información más alla de varias llamadas de métodos y se hablará sobre ellos en el apartado sobre los Stateful Session Beans.

En una aplicación EJB bien diseñada se aplicarán ambos tipos de Session Beans y se debería decidir entre un tipo o el otro según los requisitos concretos del proyecto. La práctica muestra que no tienen ningún sentido convertir a los Stateless Beans de alguna manera en Statefull Beans mediante todo tipo de trucos, tan solo porque se cree que estos últimos son tan costosos de administrar que más vale evitarlos a toda costa y programarlo todo con Stateless Session Beans.

De la misma manera, trabajar solo con Stateless Session Beans tampoco es práctico. Sobre todo cuando se trata de administrar breves consultas de una cantidad de usuarios

simultáneamente, los Stateless Session Beans son siempre la primera elección. Como cada instancia de bean está básicamente a disposición de cada usuario al mismo tiempo es especialmente sencillo para el servidor de aplicaciones llevar a cabo consultas masivas con una actuación óptima. Con Stateful Beans esto sería algo más complicado.

2.17.2.1.2. Estructura

2.17.2.1.2.1. Clase Bean

La creación de un bean de sesión sin estado es sencilla, se trata básicamente de una clase Java normal. En el Código 2.9 se representa una clase de este tipo. No es realmente necesario que aparezca la denominación Bean en el nombre de clase **ServidorTemporalBean**, pero se ha establecido prácticamente como un estándar. Sise observa detalladamente el código fuente, llama la atención la entrada **@Stateless** inmediatamente por encima de la definición de clase. Se trata de una anotación **javax.ejb.Stateless**, que se puede introducir o bien con la forma abreviada aquí representada o también mediante **@javax.ejb.Stateless**. Anteriormente ya se mencionó sobre las anotaciones en general. Si se escoge la forma abreviada es necesario incluir javax.ejb.Stateless o el Package entero **javax.ejb.*** con ayuda de la indicación **import**.

```
import java.util.Date;
import javax.ejb.Stateless;

@Stateless
public class ServidorTemporalBean implements ServidorTemporalRemote
{
public long getTime()
{
Return new Date.getTime();
}
}
```

Código 2.9: Un Stateless Session Bean

Es la anotación @Stateles la que le dice al servidor de aplicaciones que es esta clase se trata de un Stateless Bean. Si se activa una clase con esta entrada en el servidor de aplicaciones, este tomará todo aquello necesario para que la clase funcione como un bean.

2.17.2.1.2.2. Interfaz remota

Como ya se sabe un bean se trata de un componente del servidor. Esto significa que debe haber algún cliente que utilizará sus servicios. Con ello ya se describe el concepto cliente. No desempeña en principio ningún papel donde se encuentra un cliente de este tipo, es decir si está dentro o fuera del servidor de aplicaciones. Para ambos casos se debe definir qué métodos del bean deben ser accesibles para el cliente. Estos métodos se llaman los métodos business del bean. La manera más fácil de conseguirlo con una interfaz como se representan en el Código 2.10, implementada por la clase de bean.

```
import javax.ejb.Remote;
```

```
@Remote
public interface ServidorTemporalRemote
{
public long getTime();
}
```

Código 2.10: Interfaz remota con un método business.

Lo llamativo en la interfaz **ServidorTemporalRemote** es la anotación **@javax.ejb.Remote**. Determina que los métodos aquí representados estén a disposición de un Cliente que se encuentra fuera del servidor de aplicaciones. Lo llamará con ayuda de Java RMI (Remote Method Interface). En el nombre de la interfaz no debe aparecer obligatoriamente la palabra Remote, pero también aquí es casi algo estándar.

2.17.2.1.2.3. Interfaz local

Como alternativa se puede definir también una interfaz local. Llevará por lo tanto la anotación **@Local** y se implementará de la misma manera por la clase bean. Incluir la palabra Local en el nombre de la interfaz, hace la definición más elocuente.

```
import javax.ejb.Local;

@Local
public interface ServidorTemporalLocal
{
public long getTime();
}
```

Código 2.11: Interfaz local con el método business

Las clases de bean con interfaz local son accesibles para los clientes que se encuentran dentro del servidor de aplicaciones. Por lo general se trata de otros beans que quieren utilizar estos servicios.

No hay nada en contra de que un Stateless Session Bean disponga a la vez de dos interfaces. Entonces se podrá utilizar tanto local como remotamente. El listado de los métodos a disposición puede ser idéntico o diferente.

Un cliente obtendrá mediante el servidor de aplicaciones el acceso a una instancia de una clase que ha implementado la correspondiente interfaz. Como funciona esto exactamente se explicará más adelante en un apartado de este mismo capítulo.

Un cliente situado fuera del servidor, trabaja por lo tanto con una instancia de **ServidorTemporalRemote**. Fuera están por cierto también todas las clases que funcionan dentro de un servidor Web, porque el contenedor Web se tiene que presentar separado del contenedor EJB.

2.17.2.1.2.4. Relación de interfaz alternativa

Para comprobar se hará referencia también a otra posibilidad para determinar una clase de bean sea su interfaz remota y/o local. Se debe aplicar por lo tanto siempre que la clase bean no quiera o no deba implementar la interfaz correspondiente. Una razón realmente convincente para ello no es fácil de encontrar. La clase bean en este caso debe disponer además de la anotación @Local o @Remote, a la que se le debe asignar la referencia de clase de la interfaz correspondiente. Se representa en el Código 2.12.

```
import java.util.Date;
import javax.ejb.*;

@Stateless
@Remote(ServidorTemporalRemote.class)
@Local(ServidorTemporalLocal.class)
public class ServidorTemporalBean
{
public long getTime()
{
return new Date().getTime();
}
}
```

Código 2.12: Referencia a interfaz alternativa

2.17.2.1.2.5. Constructor estándar

Es imprescindible que cada Stateless Session Bean disponga de un constructor estándar. Este siempre es el caso cuando no se autoprograma ningún constructor. Si por el contrario es necesario tener que programar un constructor con parámetros, no se evita un constructor adicional sin parámetros, pues el servidor de aplicaciones solo puede crear una instancia de la clase de bean a través de este.

2.17.2.1.2.6. General

Para un proyecto concreto se deberían fijar estándares concretos, a los que todos los desarrolladores deben atenerse. A esto se debe que la clase bean deba incluir en nombre la palabra clave bean. Análogamente las interfaces local y remota se caracterizan con **Local** y **Remote**. Además la clase bean debe implementar sin falta su interfaz y al no comprobar, justo después de la llamada, que no existe ninguno de los métodos llamados con este registro. Sin embargo, la norma según la cual un Stateless Session Bean siempre debe ofrecer las dos interfaces no es muy razonable. Deberían ser dependientes de las necesidades reales. La causa para ello es que se establece realmente una diferencia respecto a los parámetros asignados si se trabaja con la interfaz remota o local. En una llamada remota las instancias se entregan como copia. Si se modifican por el método llamado, estos cambios no afectaran al cliente. Es diferente en una llamada local. Aquí se mantiene el comportamiento típico de Java.

2.17.2.1.3. Metodología de trabajo

Se ha afirmado que una instancia de un Stateless Session Bean está a disposición de todos los clientes al mismo tiempo. Realmente debe imaginar algo así para poder programar correctamente un Stateless Bean. Una vez un cliente llama un método de un bean de este tipo, el servidor de aplicaciones debe encargarse de que el método pueda ejecutarse. Para ello crea una instancia de la clase bean y accede con esta al método. Después ya no se necesitara mas la instancia acabada de crear y se podrá desbloquear. Esto sin embargo no es muy eficiente puesto que se tienen que crear una instancia para cada consulta. Un servidor de aplicaciones bien programado guardaría estas instancias para cada consulta. Un servidor de aplicaciones bien programado guardaría estas instancias en un pool propio para poder utilizarlas de inmediato para el siguiente acceso a un método. O bien utiliza siempre la misma instancia en el correspondiente nuevo Thread creado. Es indiferente que estrategia decide seguir el servidor puesto que no está asegurado, ni que un cliente que ejecuta varios accesos a métodos consecutivos siempre quiere la misma instancia de bean, ni que la misma instancia de bean sea utilizada realmente de manera simultánea por diferentes clientes.

Esto tiene consecuencias directas en la programación del bean. Así pues no tiene ningún sentido que este posea atributos, que se completaran por un método u otro y volverán a ser consultados por otros métodos. En el Código 2.13 se reproduce una de estas clases programadas erróneamente.

```
public class ObjetoAlmacen
{
private Object obj;

public void setObj(Object obj)
{
this.obj = obj;
}

public Object getObj()
{
return obj;
}
}
```

Código 2.13: Clase bean inadecuada

Supongamos que un cliente llama primero el método **setObj()** y transmite una instancia cualquiera con la esperanza de recibirla de nuevo al llamar **getObj()**. Justo aquí reside el problema. El cliente no opera en realidad directamente con la instancia de bean sino solo con una referencia local o remota y es tarea del servidor de aplicaciones con que instancia de bean ejecuta las llamadas. Lo peligroso aquí es que cuando se prueba todo con un solo cliente esto puede incluso llegar a funcionar. ¡Pero cuidado si después entra un segundo o un tercer usuario en juego!

También aquí se recomiendan normas exactas para el equipo desarrollador. Los Stateless Session Bean no tienen ningún atributo, a no ser que deban estar todos a disposición de los clientes simultáneamente. En un caso así también se debería declarara **static**, y así lo serían tal y como se desprende de su significado. Aparece sin embargo asi de inmediato el problema que en un Cluster de varios servidores todos los atributos estáticos existen independientemente los unos de los otros. La practica ha demostrado que con una

programación así se debe procurar, que una aplicación no sea de repente apta para un cluster y se desaproveche así una ventaja fundamental de cualquier aplicación EJB.

Resumiendo, se podría afirmar que una instancia de un Stateless Session Bean no está nunca reservada para un cliente concreto sino para todas las consultas entrantes, para las que esté disponible. Esto subraya de nuevo la afirmación de que en los métodos de un Stateless Bean se trata de funciones que se pueden llamar sin un contexto técnico y que son efectivas tan solo con los parámetros transmitidos.

2.17.2.1.4. Acceso al entorno

Como todos los beans los Stateless Session Beans también se ejecutan por el servidor de aplicaciones dentro un entorno definido. Dentro de este entorno se pueden definir todo tipo de recursos, a los que el bean tiene acceso. Estos recursos pueden tratarse de, por ejemplo, una instancia de la clase **javax.sql.DataSource** a través de la cual se puede crear una conexión con una base de datos, o también de otros metadatos en forma de Strings normales, a través de los que se puede definir el comportamiento de la clase bean.

Como ejemplo para una entrada de entorno así se ampliara la clase bean **ServidorTemporalBean** con un método **getTimeString()**, que muestra la fecha y la hora actual como una cadena de símbolos. Todo se lleva a cabo con ayuda de la clase **SimpleDateFormat**, a la que se le transmite el formato de **String** correspondiente. Como se puede ver en el fragmento del Código 2.14, es posible obtener acceso a los recursos del entorno a través de la anotación **@Resource**. Además la anotación debe ser programada justo antes del atributo en el que se va a inyectar el recurso.

```
@Resource(name=”FormatString”) String formatString;

public String getTimeString()
{
SimpleDateFormat sdf = null;

if( formatString != null )
sdf = new SimpleDateFormat(formatString);
else
sdf = new SimpleDateFormat();

return sdf.format(new Date());
}
```

Código 2.14: Acceso al entorno

La variable **formatString** del ejemplo permanece en el valor **null**, si no puede encontrar ninguna entrado con el nombre **FormatString** en el entorno. Por eso la consulta diferente de **null** es absolutamente necesaria si no se quiere correr el riesgo de obtener una **NullPointerException**. Ahora solo queda pendiente la cuestión de cómo llevar a término una entrada así en el entorno del bean. se trata de un archivo XML con el nombre **ejb-jar.xml**, que se crea de una norma precisa y que debe ser copiado en el directorio **META-INF** del archivo JAR. Al contrario que en el estándar EJB previo este Deployment Descriptor es esencialmente más breve. El archivo externo XML, tiene la ventaja que se

puede adaptar antes del Deployment, sin que se tengan que traducir nuevamente las clases Java contenidas.

```
<ejb-jar
xmlns=”http://java.sun.com/xml/ns/javaee”
xmlsn:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:SchemaLocation=”http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd”
versión=”3.1”>
<enterprise-beans>
<session>
<ejb-name>ServidorTemporalBean</ejb-name>
<env-entry>
<env-entry-name>FormatString</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>E, dd.MM.aaaa HH:mm:ss</env-entry-value>
</env-entry>
</session>
</enterprise-beans>
</ejb-jar>
```

Código 2.15: Deployment Descriptor ejb-jar.xml

Cree en su proyecto actual una carpeta de nombre **META-INF** y en él un archivo con el nombre **ejb-jar.xml** con el contenido del Código 2.15. Abra el **JAR Packager** para el archivo **jardesc** del proyecto e intente que la carpeta **META-INF** también se vincule. Para que se cree de nuevo el correspondiente archivo JAR basta con ejecutar la opción **CREATE JAR** también con ayuda del botón derecho del ratón. Si no se encontrara el archivo JAR automáticamente en el directorio correspondiente, debe ser copiado allí. En el Código 2.16 se reproduce la clase bean completa.

```
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.annotation.Resource;

import javax.ejb.Stateless;

@Stateless
public class ServidorTemporalBean implements ServidorTemporalRemote
{
@Resource(name=”FormatString”) String formatString;
public long getTime()
{
return new Date().getTime();
}
public String getTimeString()
{
SimpleDateFormat sdf = null;

if( formatString != null )
sdf = new SimpleDateFormat(formatString);
else
sdf = new SimpleDateFormat();

return sdf.format(new Date());
}
}
```

Código 2.16: Stateless Session Bean completo de ServidorTemporalBean

2.17.2.1.5. EJB Context

Cada bean de sesión sin estado es parte del contenedor EJB durante su ejecución y allí tiene la posibilidad de acceder a este mediante una instancia de la clase **javax.ejb.EJBContext**. Los métodos de esta interfaz se detallan en el Código 2.17 por orden alfabético.

```
import java.security.*;
import java.util.Properties;
import javax.ejb.*;
import javax.transaction.UserTransaction;
public interface EJBContext
{
public Identity getCallerIdentity();
public Principal getCallerPrincipal();
public EJBHome getEJBHome();
public EJBLocalHome getEJBLocalHome();
public Properties getEnvironment();
public boolean getRollbackonly()
throws IllegalStateException;
public TimerService getTimeService()
throws IllegalStateException;
public UserTransaction getUserTransaction()
throws IllegalStateException;
public boolean isCallerInRole(Identity role);
public boolean isCallerInRole(String roleName);
public Object lookup(String name);
public void SetRollbackonly throws IllegalStateException;
}
```

Código 2.17: La interfaz EJBContext

Esta interfaz ya existía en EJB 2.1, por eso contiene métodos que hoy ya no se puede utilizar. Se enumeran en el Código 2.18 y pueden conducir a una excepción si se intenta utilizarlos.

```
public Identity getCallerIdentity();
public boolean isCallerInRole(Identity role);
public Properties getEnvironment();
public EJBHome getEJBHome();
public EJBLocalHome getEJBLocalHome();
```

Código 2.18: Listado de los métodos anticuados.

Los otros métodos se pueden utilizar sin problemas. Así se consigue acceso al resto del entorno y sus recursos con ayuda del método **EJBContext.lookup()**.

El método **EJBContext.getTimerService()** permite el acceso a una instancia de la clase **TimerService**, que se administra por un servidor de aplicaciones. Así una sesión sin estado está en situación de encargar al servidor que lo active en un momento concreto o después de cierto intervalo de tiempo y que ejecute un método determinado del bean.

Mediante el método **EJBContext.getCallerPrincipal()** se puede transmitir el nombre del cliente que ha llamado el método actual. Con **EJBContext.isCallerInRole()** y **EJBContext.setRollbackonly()** el bean puede llegar a influenciar el control de la

transacción. Si se encontrara por ejemplo un error que provocara que se anulara la transacción entera, el método llama simplemente **setRollbackonly()**, el resto lo dispone el servidor de aplicaciones. En el apartado sobre Transacciones se tratará más a fondo el tema.

Un bean concreto no operara nunca directamente con una instancia de **EJBContext** sino siempre con los métodos de la interfaz **SessionContext** derivada de **EJBContext**, por eso en el siguiente apartado se describe como se puede acceder a su entorno.

2.17.2.1.6. Session Context

El contexto en el que se encuentra un Stateless Session Bean se expresa mediante una instancia de la interfaz **javax.ejb.SessionContext**, que se deriva de **javax.ejb.EJBContext**. Añadimos cinco métodos mas a los ya descritos, de los cuales dos son ya obsoletos y ya no se deben utilizar. Se trata de los métodos **SessionContext.getEJBObject()** y **SessionContext.getEJBLocalObject()**. Si se llamara alguno de estos métodos se produciría de inmediato una excepción.

```
import javax.ejb.*;
import javax.xml.rpc.handler.MessageContext;

public interface SessionContext extends EJBContext
{
public Object getBusinessObject(Class businessInterface)
throws IllegalStateException;
public EJBLocalObject getEJBLocalObject()
throws IllegalStateException;
public EJBObject getEJBObject()
throws IllegalStateException;
public Object getInvokedBusinessInterface()
throws IllegalStateException;
public MessageContext getMessageContext()
throws IllegalStateException;
}
```

Código 2.19: La interfaz SessionContext

```
public EJBLocalObject getEJBLocalObject();
public EJBObject getEJBObject();
```

Código 2.20: Listado de métodos obsoletos

El método **SessionContext.getBusinessObject()** ofrece una referencia al bean actual que se transmite después a otros clientes. Esta referencia sustituye **this**, lo que no se debe transmitir nunca a otros. Para entender esto, se debe tener presente que todos los clientes, indiferentemente de si son locales o remotos, siempre se comunican con un bean a través de una interfaz local o remota, nunca directamente con un bean. Por este motivo un bean no está autorizado a revelar una referencia por sí mismo, lo que un cliente, que se encuentra fuera del servidor de aplicaciones tampoco debería hacer nunca. Más bien se debe determinar una referencia a la interfaz correspondiente, a partir de la cual después

será posible un acceso. Para que el método **getBusinessObject()** sepa que referencia de interfaz debe generar, se le debe transmitir la interfaz como clase. Un ejemplo de ello es el Código 2.21. En la práctica tiene de hecho poco sentido implementar un método así, pues para su llamada ya es necesaria una referencia a la interfaz correspondiente. Sucede a menudo que una instancia de bean transmite el resultado de **getBusinessObject()** a otro método o incluso que lo escribe en la base de datos. Así otros beans tienen siempre la posibilidad de llamar métodos en la instancia de bean concreta para, por ejemplo, modificar el comportamiento del logging del bean.

```
@Stateless
public class ServidorTemporalBean implements ServidorTemporalRemote
{
@Resource private SessionContext ctx;

public Object getRemoteReference()
{
return ctx.getBusinessObject(ServidorTemporalRemote.class);
}
}
```

Código 2.21: Referencia a interfaz remota

Si un bean quisiera saber si se le ha llamado a través de su interfaz local o remota el método **SessionContext.getInvokedBusinessInterface()** le informará sobre ello. Da la interfaz correspondiente como referencia de clase.

En el Código 2.21 se ve como un bean obtiene acceso a su **SessionContext** derivado de **EJBContext**. Se permite inyectar de la manera mas fácil la instancia correspondiente mediante la anotación **@Resource** en el atributo definido para ello del tipo **SessionContext**. Como alternativa es posible programar un método que espere como único parámetro un **SessionContext**. Si se provee entonces este método de la anotación **@Resource** se llamara por el servidor de aplicaciones y se transmitirá el recurso requerido. Un ejemplo lo encontramos en el Código 2.22.

```
@Stateless
public class ServidorTemporalBean implements ServidorTemporalRemote
{
private SessionContext ctx;

@Resource
public void setSessionContext(SessionContext ctx)
{
this.ctx = ctx;
}
}
```

Código 2.22: Anotacion @Resource a nivel de método.

2.17.2.1.7. Ciclo de vida de un Stateles Session Bean

Un bean de sesión sin estado, como el resto de tipos de beans, puede adoptar diferentes estados durante el trabajo con el. En el caso de los Stateless Beans, solo son posibles como máximo dos estados. Se representan en la Figura XIV. Todos los beans que se

encuentran en Method-Ready Pool, no se utilizarán por el momento. Se trata de un simple pool de instancias de una clase de beans determinada, cuyo tamaño puede limitarse a menudo con los parámetros del servidor de aplicaciones. Un pool de este tipo solo existe para los Stateless Session Beans. Es esto lo que diferencia esencialmente los Stateless Session Beans. A continuación se describe el peculiar método de trabajo de los Stateless Beans.

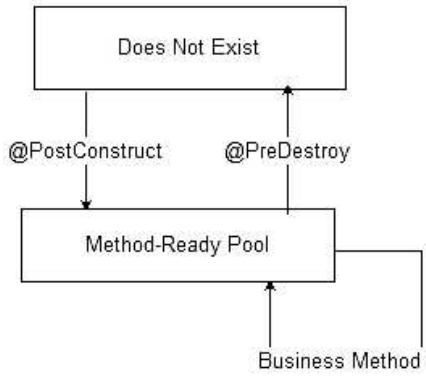


Figura XIV: Ciclo de vida de un Bean de Sesión sin estado

2.17.2.1.7.1. Does Not Exist

Este estado no significa que no haya todavía ninguna instancia del bean. El servidor de aplicaciones no ha creado aun ni una sola instancia de bean.

PostConstruct

Cuando un bean del estado **Does not exist** cambia al estado **Method-Ready Pool**, sucede en tres pasos. Primero se ejecuta el método **Class.newInstance()** de la clase bean para crear la instancia técnica. Por eso es imprescindible que todos los Stateless Session Beans dispongan de un constructor sin parámetros. Después se inyectan todos los recursos al bean con los que se relacionara la anotación correspondiente. En el ejemplo de la clase **ServidorTemporalBean** este era el recurso **FormatString**. Por último el servidor llama finalmente un método de la instancia del bean que consta de la anotación **@PostConstruct** o que ha sido definido en el Deployment Descriptor como método **Post-Construct**. El nombre de este método opcional puede ser cualquiera. No contiene ningún parámetro, siempre devuelve **void** y no provoca ninguna excepción. Los ejemplos correspondientes se encuentran en los segmentos de Código 2.23 y 2.53. Durante el tiempo de vida entero del bean se llamara este método como máximo una vez. No esta prescrito que un bean deba disponer de una método de este tipo. Durante la llamada, el SessionContext sin referencia al cliente y el entorno completo JDNI están a disposición del método.

```
@Stateless
public class ServidorTemporalBean implements ServidorTemporalRemote
{
@PostConstruct
public void init()
{
```

```
…
}
```

Código 2.23: Ejemplo de la anotación @PostConstruct

```
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>ServidorTemporalBean</ejb-name>
<post-construct>
<lifecycle-callback-method>
init
</lifecycle-callback-method>
</post-construct>
</session>
</enterprise-beans>
</ejb-jar>
```

Código 2.24: Definicion XML de un método Post-Construct

Todas las referencias construidas dentro de un método de inicialización de este tipo están a disposición del bean hasta que vuelve al estado **Does not exist**. Los Stateless Beans no se pasivizan, por eso no es necesario serializar sin falta estas referencias.

2.17.2.1.7.2. Método Ready Pool

En este estado existe una instancia de bean concreta, pero actualmente no se utiliza por ningún cliente. El servidor de aplicaciones podría desbloquear igual de bien la instancia, pero antes debería crear una nueva instancia para la siguiente llamada de método. De ahí que sea mas potente desactivar las instancias de beans innecesarias para la siguiente subida.

Método Business

Si un cliente llama un método de un Stateless Session Bean se tomara una instancia disponible del pool con el SessionContext necesario. El servidor de aplicaciones se ocupara de si utiliza varia instancias de beans en llamadas simultaneas o siempre la misma. Sin embargo debe asegurar que cada llamada ocurra dentro del contexto correcto. Además cada llamada toma parte de una transacción específica de cada cliente. Para el momento de la llamada del método también se establece un tipo de referencia con el cliente para el bean. Si el método llega a su fin, termina a su vez esta relación.

Como ya se ha descrito, cada cliente necesita una referencia con la interfaz local o remota del Stateless Session Bean. En el momento en que un cliente proporciona una referencia así, no debe haber aun ninguna instancia de la clase de bean en el servidor. El método Ready Pool aún puede estar vacío, hasta que se lleva a cado la primera llamada de método real.

PreDestroy

Una vez el servidor de aplicaciones deja de necesitar la instancia concreta de un bean de sesión sin estado y tampoco quiera desactivarla en su pool interno para una posible llamada futura, debe liberarla. Antes llama el método de bean con la anotación **@PreDestroy** o se define en el Deployment Descriptor como método **Pre-Destroy**. No debe esperar ningún parámetro, devuelve **void** y no ejecuta ningún tipo de excepción. Durante el tiempo de vida de la instancia este método se llamara como máximo una vez, porque después la instancia deja de existir. Durante la llamada están a disposición del método el SessionContext sin referencia al cliente y el entorno completo JNDI. Sin embargo no es obligatorio que el bean disponga de estos métodos.

```
@Stateless
public class ServidorTemporalBean implements ServidorTemporalRemote
{
@PreDestroy
public void clean()
{   …
}
```

Código 2.25: Ejempo de anotación @PreDestroy

```
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>ServidorTemporalBean</ejb-name>
<pre-destroy>
<lifecycle-callback-method>
clean
</lifecycle-callback-method>
</pre-destroy>
</session>
</enterprise-beans>
</ejb-jar>
```

Código 2.26: Definicion XML de un método PreDestroy

La tarea principal de un método así es la devolución de todos los recursos que se han construido durante el método Post-Construct.

2.17.2.1.8. XML Deployment Descriptor

Los metadatos de una aplicación Java EE se pueden dividir en dos grupos. Unos son aquellos relevantes para el servidor de aplicaciones y los otros conciernen a la aplicación misma. Las anotaciones **@Local**, **@Remote** o también **@Stateless** pertenecen claramente a os metadatos importantes para el servidor. Gracias a estas entradas está en situación de reconocer qué clases e interfaces están pensadas para él. Estos metadatos son relativamente fijos pues prácticamente describen la arquitectura de la aplicación. Todo lo necesario para su descripción se puede nombrar mediante las anotaciones, por lo que

no hay ningún motivo para un Deployment Descriptor. Esto es diferente en los metadatos relacionados con aplicaciones. Se vinculan con la anotación **@Resource** y por lo tanto puede ser practico que se fijen en uno u otro valor para una instalación determinada. Como ejemplo para un recurso de este tipo sirve la cadena de símbolos **formatString** en la clase **ServidorTemporalBean**. Mediante estos parámetros se puede determinar desde fuera cómo preparar la fecha y la hora sin tener que traducir de nuevo la clase Java que desempeña esta tarea. Para poder dar metadatos dinámicos a una aplicación de este tipo, se debe dar un archivo adicional descriptivo que contenga esta información. Aquí se trata de un Deployment Descriptor en el archivo **ejb-jar.xml**. Tampoco en el nuevo estándar EJB nos las arreglamos del todo sin este archivo, pero se ha reducido bastante su extensión.

Si se quiere prescindir del todo de la entrada de anotaciones, se tiene que ampliar el Deployment Descriptor de tal manera que el servidor sepa de nuevo de qué clase se trata en cada tipo de bean. En el Código 2.26 se representa el contenido del archivo **ejb-jar.xml** para la clase bean **ServidorTemporalBean**.

```
<ejb-jar
xmlns=”http://java.sun.com/xml/ns/javaee”
xmlsn:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:SchemaLocation=”http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd”
versión=”3.1”>
<enterprise-beans>
<session>
<ejb-name>ServidorTemporalBean</ejb-name>
<remote>server.ServidorTemporalRemote</remote>
<ejb-class>server.ServidorTemporalBean</ejb-class>
<session-type>Stateless</session-type>
<env-entry>
<env-entry-name>FormatString</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>E, dd.MM.aaaa HH:mm:ss</env-entry-value>
<injection-target>
<injection-target-class>
server.ServidorTemporalBean
</injection-target-class>
<injection-target-name>
formatString
</injection-target-name>
</injection-target>
</env-entry>
</session>
</enterprise-beans>
</ejb-jar>
```

Código 2.27: Deployment Descriptor completo

Como esta todo descrito en el Deployment Descriptor ya no se encuentran ningún tipo de anotaciones en las clases Java. Como ejemplo de ello se encuentran en el Código 2.28 la clase **ServidorTemporalBean**.

```
import java.text.SimpleDateFormat;
import java.util.Date;
public class ServidorTemporalBean implements ServidorTemporalRemote
{
private String formatString;
```

```
public long getTime()
{
Return new Date().getTime;
}

public String getTimeString()
{
SimpleDateFormat sdf = null;

if( formatString != null )
sdf = new SimpleDateFormat(formatString);
else
sdf = new SimpleDateFormat();

return sdf.format(new Date());
}
}
```

Código 2.28: Clase bean sin anotaciones

Si observamos detenidamente el método **getTimeString()** del Código 2.28 llama la atención que el atributo **formatString** se compara con **null**. En ningún sitio sin embargo esta la posibilidad de fijar el valor de **formatString**. Tampoco la clase misma se ocupa de la documentación de estas variables. De ahí surge la cuestión sobre de donde debe recibir sus valores. La respuesta se encuentra en el Deployment Descriptor. La descripción de la entrada del entorno **FormatString** se ha ampliado con la etiqueta **<injection-target>**. Así se encarga al servidor de aplicaciones que inyecte el valor “E, dd.MM.aaaa HH:mm:ss” **(<env-entry-value>**) en el atributo **formatString(<injection-target-name>**) de la clase **server.ServidorTemporalBean**.

Si se quiere, como se ha descrito arriba, hacer útil la clase Java también fuera del servidor de aplicaciones se tiene que prescindir de este tipo de inserciones, pues allí no existe ningún contenedor EJB que se ocupe de la asignación de este tipo de metainformación.

Todavía falta aclarar que ocurre cuando se utilizan tanto anotaciones como también un Deployment Descriptor y los datos puede que lleguen a contradecirse. En un caso así es siempre válido lo que está en el Depoyment Descriptor. Sus datos sobrescriben todas las anotaciones. Así se está en situación de utilizar para una instalación de clientes determinado, clases de beans y de interfaz divergentes, un procedimiento que puede provocar muy rápidamente la perdida de la visión general de la arquitectura de clientes amplios, se debe intentar siempre mantener igual la aplicación y hacerla ampliable en interfaces estándar individuales.

2.17.2.1.9. Acceso mediante un programa Cliente

Si un cliente quiere llamar métodos business de una clase bean, debe conseguir acceso a través del servidor de aplicaciones a la instancia de una clase, que ha implementado la interfaz correspondiente. Como ya se ha mencionado, no se trata nunca de una instancia de bean propia, sino siempre de un objeto representante, que recibe las llamadas y las transmite al servidor. Para llegar a esta instancia, el cliente necesita tener primero acceso al contexto técnico en el que se ha activado el bean.

Esto se posibilita mediante una instancia de la clase **javax.naming.Context**. Desafortunadamente no se puede llamar simplemente el constructor de la clase, sino que se debe procurar que haya informaciones a su disposición, como por ejemplo con ayuda de qué clase se tiene que construir el **NamingContext** o bajo qué dirección (URL) del servidor de aplicaciones se podrá encontrar. Estas entradas son diferentes en cada servidor. Se debe saber antes si el servidor al que nos dirigimos se trata de un servidor JBoss, WebLogic o WebSphere.

También la cantidad de las informaciones a introducir es diferente, por eso no hay ningún constructor que disponga de parámetros correspondientes. Más bien se tienen que transmitir los datos mediante propiedades. Para ello se dispone de dos medios. O bien se crea una instancia de la clase **java.util.Properties** y se llena este contenedor con los datos necesarios o se llama el programa cliente con las propiedades requeridas, que se reproducen en la line de comando con **–D**.

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
“org.jnp.interfaces.NamingContextFactory”);
p.put(Context.URL_PKG_PREFIXES, “org.jboss.naming:org.jnp.interfaces”);
p.put(Context.PROVIDER_URL, “jnp://localhost:1099”);
Context ctx = new InitialContext(p);
```

Código 2.29: Contexto sobre la instancia Properties

En el Código 2.29 tenemos un ejemplo de cómo se transmiten los datos mediante una instancia de la clase Properties al contructor de InitialContext. Se refieren a la instalación estándar de JBoss.

```
// -Djava.naming.factory.initial = org.jnp.interfaces.NamingContextFactory
// -Djava.naming.factory.url.pkgs = org.jboss.naming:org.jnp.interfaces
// -Djava.naming.provider.url = jnp://localhost:1099
Context ctx = new InitialContext();
```

Código 2.30: Contexto de los parámetros de la línea de comandos

La segunda posibilidad para fijar la información necesaria se muestra en el Código 2.30. Si los datos en la línea de comentarios se han fijado en la línea de comandos, basta con llamar el constructor sin parámetros de la clase **InitialContext**. Si se crea en Eclipse una configuración de inicio para su programa cliente se pueden hacer estas entradas en la ficha **ARGUMENTS** en el campo **VM ARGUMENTS**.
Si se tiene acceso al contexto se puede buscar desde allí el bean deseado. Para ello llama al método **lookup()** en la instancia de **InitialContext**. El método espera como parámetros una cadena de símbolos, compuesta del nombre del bean, barra y la palabra **remote** o **local**, según que interfaz se necesite. Para la interfaz remota de la clase **ServidorTemporalBean** esto significa que se tiene que transmitir a **lookup()** la cadena de símbolos “**ServidorTemporalBean/remote**”. Como resultado se obtiene o bien una referencia del tipo **Object** o una **NamingException**, en caso de que no se pueda encontrar ningún bean bajo el nombre introducido. La llamada se reproduce en el Código 2.31.

```
Try
{
// -Djava.naming.factory.initial = org.jnp.interfaces.NamingContextFactory
// -Djava.naming.factory.url.pkgs = org.jboss.naming:org.jnp.interfaces
// -Djava.naming.provider.url = jnp://localhost:1099
Context ctx = new InitialContext();

Object ref = ctx.lookup("ServidorTemporalBean/remote");
}
catch( NamingException e)
{
e.printStackTrace();
}
```

Código 2.31: Ejecución de Lookup

En el caso de la interfaz remota se recibe una referencia transmitida, que se tiene que reconstruir en el cliente en una instancia de la clase deseada. Esto se consigue llamando el método **narrow()** de la clase **javax.rmi.PortableRemoteObject**, en la que la referencia contenida y la interfaz deseada se tienen que reproducir como clase, plasmado en el Código 2.32. Este método se introdujo en EJB 1.1 y es siempre necesario cuando las llamadas RMI no se llevan a cado directamente con el protocolo propio de Java sino mediante IIOP, que es compatible con CORBA. Para el caso en el que el cliente se refiere a la interfaz local, el resultado del método **lookup()** puede inserirse directamente en esta interfaz.

```
ServidorTemporalRemote szr = (ServidorTemporalRemote)
PortableRemoteObject.narrow(ref, ServidorTemporalRemote.class);
```

Código 2.32: Reformar la referencia remota

En el Código 2.33 se representa el aspecto completo de un cliente para la clase **ServidorTemporalBean**. Se parte de la base que todas las propiedades necesarias para el contexto están fijadas en la línea de comandos. Se indicaran de nuevo en el código fuente mediante las líneas de comentarios.

```
import javax.naming.*;
import server.ServidorTemporalRemote;

public class ServidorTemporalCliente
{
public static void main(String[ ] args)
{
try
{
// -Djava.naming.factory.initial = org.jnp.interfaces.NamingContextFactory
// -Djava.naming.factory.url.pkgs = org.jboss.naming:org.jnp.interfaces
// -Djava.naming.provider.url = jnp://localhost:1099
Context ctx = new InitialContext();

Object ref = ctx.lookup("ServidorTemporalBean/remote");
ServidorTemporalRemote str = (ServidorTemporalRemote)
PortableRemoteObject.narrow(ref, ServidorTemporalRemote.class);
System.out.println("ServidorTemporal (long): " + szr.getTime());
System.out.println("ServidorTempora (String): " + szr.getTimeString());
}
```

```
catch( NamingException e )
{
e.printStackTrace();
}
}
}
```

Código 2.33: Cliente complete para la clase ServidorTemporalBean

2.17.2.2. **Stateful Session Beans**

2.17.2.2.1. Uso

Se tratarán los Stateful Beans. Tanto los Stateless Session Beans como los Stateful Session Beans se parecen mucho, así que su construcción es prácticamente idéntica. También la manera en la que un cliente accede al bean es la misma. Por lo tanto es bastante conveniente leerse primero el apartado sobre los Stateless Session Bean, pues se hará constantemente referencia a él. La diferencia fundamental es que una instancia de un Stateful Session Bean estará siempre firmemente vinculada a un cliente y solo está a su disposición. Ningún otro puede llamar métodos en la instancia de bean concreta, a no ser que se le transmita explícitamente una referencia a este bean.

Esto conlleva algunas consecuencias. Así pues es más que razonable que un Stateful Session Bean disponga de atributos en los que se recuerdan las llamadas de métodos. Es también posible iniciar la instancia del bean individualmente debido a la firme conexión con el cliente. Por otro lado también es necesario que el cliente comunique al servidor cuando deja de requerir la instancia de Bean para que así el servidor pueda desbloquearla. Este a su vez debe tomar las precauciones necesarias para poder administrar tantas instancias de un Stateful Session Bean como quiera. Se ilustraran estos puntos más detalladamente al hablar sobre el método de trabajo.

Los Stateful Session Beans siempre entran en acción cuando un cliente ejecuta una conversación con el servidor durante varias llamadas de método, mientras este debe guardar el estado constantemente cambiante y la gran cantidad de información transmitida. Los métodos individuales de un bean con estado no son por lo tanto servicios simples, que pueden llamarse el uno detrás del otro sin orden ni concierto, carentes de ningún tipo de relación entre ellos, sino mas bien métodos business que se construyen uno a partir del otro y que contribuyen pieza a pieza a ejecutar una tarea mas o menos compleja.

Una tarea de este tipo, que se extiende a lo largo de varias llamadas de métodos, podría ser la recepción de un pedido, para el que el usuario primero debe registrarse y después introducir qué cantidad desea recibir de cada artículo, hasta que finalmente da por aceptado el pedido completo o bien lo rechaza. En cada una de estas acciones la instancia bean del servidor debe controlar con exactitud bajo que nombre se ha registrado cada cliente y que artículos ha escogido. Debe asegurarse de que dos usuarios diferentes que deseen llevar a cabo un pedido al mismo tiempo no se entorpezcan el uno al otro.

Como ya se ha descrito, en una aplicación concreta habrá tanto Stateful como Stateless Session Beans. La idea es que un bean pueda utilizar servicios de otro y funcione

entonces como cliente del otro bean. En este caso son posibles todas las combinaciones aunque lo mas probable es que un bean con estado utilice los servicios de un bean sin estado. En el caso contrario el bean sin estado debería resolver el trabajo completo dentro de solo un de sus métodos.

El servidor de aplicaciones debe mantener una instancia propia de un Stateful Session Bean para cada cliente. Ante esta idea, desarrolladores experimentados, que ya han trabajado previamente con tecnologías basadas en servidores como CICS o similar, temen que el servidor justo en pleno funcionamiento pueda no estar en situación de administrar tantas instancias. Podría fallar la memoria principal o algo parecido. De hecho debe comentarse que es principalmente este círculo de persona el que intenta programarlo siempre todo con Stateless Session Beans porque estos no exigen tantos recursos. Pues bien, los diseñadores de JavaEE Standard son plenamente conscientes del reto del pleno rendimiento. Un servidor de aplicaciones tiene medios suficientes para enfrentarse a esto sin problemas. Por un lado, hoy en día cualquier servidor decente está dotado de por lo menos 1 o 2 Gigabytes de memoria principal, por otro lado puede llevar a un estado pasivo las instancias de beans no requeridas en el momento, es decir, se almacenan y se guardan en el disco duro. Por este motivo para un servidor de aplicaciones es importante disponer de discos duros lo más rápidos posibles. Si aun así el hardware no fuera suficiente, se puede dividir el servidor en varias maquinas físicas, creando un cluster. Al programador de una aplicación JavaEE esto no le afecta en absoluto. En ningun punto se verá obligado a escribir un código concreto para capacitar este cluster, a no ser que utilice tecnologías que funcionen en uno.

2.17.2.2.2. Estructura

2.17.2.2.2.1. La clase bean

Un Stateful Session Bean es una clase java normal. La denominación Bean en el nombre de la clase **CestaCompraBean** hace referencia a su uso como tal. La clase implementa una interfaz con el nombre **CestaCompraRemote**, en la que se llevan a cabo los métodos business del bean, que en este caso se puede utilizar por un cliente remoto.

```
package server.sf;
import java.util.Vector;
import javax.ejb.Stateful;
@Stateful
public class CestaCompraBean implements CestaCompraRemote
{
protected Vector<Pedido> pedidos = new Vector<Pedido>( );
public void addPedido(Pedido best)
{
best.setCantidadEntrega(best.getCantidad( ));
pedidos.add(best);
}

public Vector<Pedido> getPedidos( )
{
return pedidos;
}
}
```

Código 2.34: Stateful Session Bean

La anotacion @Stateful del package javax.ejb es la que le dice al servidor de aplicaciones que en esta clase se trata de un bean con estado. Durante la activación el servidor de aplicaciones realizara todo lo necesario para que la clase funcione como bean.

2.17.2.2.2.2. Interfaz Remote

La estructura y el significado de una interfaz remota ya se a descrito en el apartado sobre los Stateless Session Beans. Representa todos los métodos, que puede utilizar un cliente que se encuentra fuera del servidor de aplicaciones. La anotación **@javax.ejb.Remote** hace referencia a esta peculiaridad.

```
package server.sf;

import java.util.Vector;
import javax.ejb.Remote;

@Remote
public interface CestaCompraRemote
{
public void addPedido( Pedido best );
public Vector<Pedido> getPedidos( );
}
```

Código 2.35: Interfaz remota con métodos business

2.17.2.2.2.3. Interfaz Local

Si el cliente del Stateful Session Bean se encuentra igualmente en el servidor (por ejemplo otro bean) entonces este puede operar de manera mucho más directa con la instancia del bean, cuando esta ofrece una interfaz local. Sigue recibiendo una instancia de un objeto representativo, pero esta transmite los parámetros directamente, sin tener que comprimirlos para un largo trayecto. Se tratara con más profundidad este aspecto en el apartado sobre las llamadas de métodos locales y remotos.

El objeto representativo es necesario porque al llamar un método, también cuando presuntamente se realiza localmente, se debe contemplar la gestión de la transacción en su totalidad. Se dedica todo un subcapítulo entero al tema de las transacciones.

```
package server.sf;
import java.util.Vector;
import javax.ejb.Local;

@Local
public interface CestaCompraLocal
{
public void addPedido( Pedido best );
```

```
public Vector<Pedido> getPedidos( );
}
```

Código 2.36: Interfaz local con métodos business

En este punto recordamos de nuevo que un bean puede implementa tanto una interfaz local como remota y puede ser utilizado por lo tanto por todo tipo de clientes. La cantidad de los métodos ofrecidos puede ser idéntica pero también diferente.

2.17.2.2.2.4. Interfaz RemoteHome y LocalHome

Junto a las anotaciones **@Remote** y **@Local** están también **@RemoteHome** y **@LocalHome** que en la descripción del nuevo estándar EJB se tratan de manera muy superficial. La razón de ello es que ya no interesa mantenerlos. En el estándar EJB 2.1 se han definido en las llamadas interfaces Home métodos **create()**, con la ayuda de los cuales se creaba la instancia de bean. Estos métodos servían como sustitutos para un constructor. Un cliente se procuraba primero acceso a la interfaz home, llamaba después uno de los métodos **create()** creados y recibía entonces la referencia a la interfaz remota o local. Este procedimiento resultaba demasiado complicado para los creadores del nuevo estándar. Además no se podía garantizar la protección de la transacción durante la ejecución de un método **create()**. Por eso se sustituían en su significado por métodos business normales de la interfaz remota o local.

Si se quiere de todas maneras seguir con el antiguo procedimiento, se tiene que escribir una interfaz RemoteHome o LocalHome, que se refiera a la clase bean perteneciente en el atributo **value** de la anotación correspondiente. La interfaz puede disponer de tantos métodos **create()** como se quiera, que deben diferenciarse en su tipo y cantidad de parámetros. La clase bean a su vez debe implementar todos estos métodos con las signatura correctas, siendo irrelevantes los nombres de los métodos de implementación. Todos deben ser reconocibles por la anotación **@Init**.

2.17.2.2.2.5. Referencia a interfaz alternativa

Para comprobar se hará referencia también a otra posibilidad para determinar una clase de bean sea su interfaz remota y/o local. Se debe aplicar por lo tanto siempre que la clase bean no quiera o no deba implementar la interfaz correspondiente. Una razón realmente convincente para ello no es fácil de encontrar. La clase bean en este caso debe disponer además de la anotación @Local o @Remote, a la que se le debe asignar la referencia de clase de la interfaz correspondiente. Se representa en el Código 2.37. Pero como se ha dicho esta variante es más bien dudosa.

```
import java.util.Date;
import javax.ejb.*;

@Stateful
@Remote(CestaCompraRemote.class)
@Local(CestaCompraLocal.class)
```

```
public class CestaCompraBean
{
public void addPedido( Pedido best )
{
…
```

Código 2.37: Referencia a interfaz alternativa

2.17.2.2.2.6. Constructor estándar

Cada Stateful Session Bean debe disponer obligatoriamente de un constructor estándar. Como es sabido este no tienen porqué esperar ningún parámetro. Si no se programa ningún constructor, la clase tiene automáticamente un constructor estándar. Pero si se programa un solo constructor parametrizado se debe incluir además un constructor estándar pues sino no se puede utilizar la clase como clase bean.

Las causas son que el servidor de aplicaciones crea durante el lookup a partir de un Stateful Session Bean la instancia de bean correspondiente, y lo hace con ayuda del método **newInstance()** en la referencia de clase del bean. Como no se pueden transmitir más parámetros al método **lookup()** aparte del nombre JNDI el servidor de aplicaciones tampoco dispone de ninguno mas para un constructor parametrizado.

2.17.2.2.2.7. Métodos de inicialización

Con ayuda de un Stateful Session Bean se puede llevar a cabo una conversación con un cliente más complejo y de duración más larga. Por eso puede ser muy expeditivo que existan ciertas condiciones técnicas para una conversación de este tipo. Por ejemplo el usuario debe registrarse. Solo cuando lo ha conseguido, puede seguir llamando métodos.

En una aplicación Java normal este registro se realiza con el constructor. Se le da a este un nombre y una contraseña y, si el registro no funciona de esta manera, se produce una excepción. Como ya se ha descrito, una clase bean no dispone de constructores con parámetros que se puedan utilizar desde un cliente. Las anteriores interfaces Home con sus métodos **create()** ya están obsoletos. Así pues tan solo queda programar un método business **login()** normal o un método comparable, que espera recibir los parámetros nombrados y que realiza el proceso login. Si el proceso fracase se produce una excepción del sistema, pues el método tiene como consecuencia que se libera de inmediato una instancia de bean. Unas excepción del sistema es toda excepción sin tratar, que no está marcada explícitamente con la anotación **@ApplicationException**. En caso de que el proceso tenga éxito se guarda una instancia de la clase UserId en un atributo. Esto significa en definitiva que todos los otros métodos business, para la llamada de los cuales debe asegurarse que el usuario se haya registrado, deben comprobar la existencia de la clase UseId y finalizar su trabajo si no es así con una excepción del sistema.

2.17.2.2.2.8. Liberar la instancia de bean

La instancia de un Stateful Session Bean se mantiene por el servidor de aplicaciones hasta que el cliente ya no la requiere o si este no se anuncia durante largo tiempo. En cada servidor de aplicaciones se puede configurar libremente con la Session Timeout la duración de este espacio de tiempo. En el JBoss es por defecto de 30 minutos.

Junto al Timeout el cliente debe disponer además de otro medio de comunicar al servidor que debe liberar la instancia. Esto se realiza con la llamada de un método business, provisto de la anotación **@Remove**. Al finalizar este método, el servidor devuelve la instancia de bean y cualquier otra llamada de método a través del cliente provocará inevitablemente una excepción. El nombre del método puede ser cualquiera. Puede esperar parámetros y ofrecer cualquier tipo de resultados.

```
import javax.ejb.Remove;

@Stateful
public class CestaCompraBean implements CestaCompraRemote
{
@Remove( retainIfException = true )
public void remove( )
{
}
}
```

Código 2.38: Fragmento de una clase bean con método Remove

En el Código 2.38 se define el correspondiente método con el nombre **remove()**, que también tiene sentido si no contiene ningún código. Después de todo es la única posibilidad de liberar intencionalmente un Stateful Session Bean. La adición de **retainIfException = true** indica que durante la ejecución del método **remove** se lleva a cabo una ApplicationException, pero no se libera el bean. El valor por defecto de **retainIfException** es **false**. La indicación aquí solo se ha hecho para presentarlo más complejo. Como el método del ejemplo de hecho no produce ninguna excepción, se puede eliminar este dato.

2.17.2.2.3. Metodología de trabajo

El método de trabajo básico de un Stateful Session Bean ya se ha descrito anteriormente. Esta exclusivamente a disposición de un cliente y por lo tanto puede guardar información para este individualmente. Vive desde el lookup en la interfaz remota o local y se libera de nuevo cuando viene de un Timeout o de un método business con la anotación **@Remove**. No existe por definición un pool de Stateful Session Bean.

Tener que administrar una instancia propia para cada cliente obliga al servidor de aplicaciones a crear tantas instancias como usuarios hay conectados en ese momento. Y normalmente estos son muchos más de los que están actualmente activos. Imaginémonos una aplicación de oficina para nuestros compañeros con un Timeout de sesión de 24 horas. Cada mañana se registran cientos de usuarios, incluso puede que mas de una vez y trabajan durante todo el día.

Para que la memoria principal no sea lenta sino segura, el servidor de aplicaciones tiene la posibilidad de almacenar instancias de bean individuales. Esto significa, que hace entrar la instancia con todos sus atributos y referencias a otras instancias Java a un estado pasivo y la guarda en el disco duro. Si se vuelve a registrar el usuario, el servidor carga la instancia de bean, la activa y la pone a disposición en la memoria principal. Para alcanzar en un cluster de varias máquinas físicas la mayor seguridad posible ante caídas, la mayoría de contenedores EJB participantes están construidos de manera que guardan todas sus instancias bean en un mismo banco de datos común. Si un ordenador fallara, el resto podrían continuar su trabajo.

Para que esto funcione técnicamente, es imprescindible que todos los atributos utilizados directa o también indirectamente por la instancia bean sean serializables, pero esto no tiene porqué coincidir obligatoriamente en todas las instancias que el vector lleva consigo. Para que el Stateful Session Bean se pueda pasar a un estado pasivo, este debe proveerse de tal modo que también se pueda escribir la clase Pedido en el disco duro. Además implementa la interfaz **java.io.Serializable** y utiliza el mismo solo aquellos atributos que por su parte son serializables, como se puede ver en el Código 2.39.

```
package server.sf;
import java.io.Serializable;
import java.math.BigDecimal;



public class Pedido implements Serializable
{
private static final long serialVersionUID = 1L;

protected Integer artNr;
protected Integer cantidad;
protected BigDecimal precio;
protected String denominacion;
protected Integer cantidadEntrega;

public pedido(Integer artNr, Integer cantidad,
BigDeciaml precio, String denominacion)
{
this.artNr = artNr;
this.precio = precio.setScale(2, BigDecimal.ROUND_HALF_UP);
this.denominacion = denominacion;
}

public Integer getArtNr( )
{
return artNr;
}
public String getDenominacion( )
{
return denominación;
}
public Integer getCantidad( )
{
return cantidad;
}
public BigDecimal getPrecio( )
{
return precio;
}
```

```
public Integer getCantidadEntrega( )
{
return cantidadEntrega;
}

public void setCantidadEntrega( Integer cantidadEntrega )
{
this.cantidadEntrega = cantidadEntrega;
}
}
```

Código 2.39: Clase Serializable Pedido

2.17.2.2.4. Llamadas de métodos locales o remotos

Como se ha mencionado de manera detallada sobre el hecho de que un bean puede disponer tanto de una interfaz remota como de una local. Así es accesible para clientes que se encuentran tanto dentro como fuera del servidor de aplicaciones. O en otras palabras accesible para programas que se desarrollan en otras maquinas virtuales Java o en la propia. Si se crea una conexión a través de una interfaz local se asegura así siempre que el cliente y la instancia de bean se ejecutan en la misma máquina virtual. Esto es especialmente así cuando un servidor se ha agrupado en diferentes máquinas físicas.

Así de sencillo. Un bean es un servicio a disposición de otros programas. Para alcanzar el mayor grado de flexibilidad se podría definir para un proyecto que todos los beans de sesión, indiferentemente de qué tipo, siempre estén preparados en la interfaz local o remota, para poder utilizar siempre y en todos sitios sus servicios. Si se toma esta decisión, también tiene que configurarse que se ofrezcan los mismos métodos en ambas interfaces, que una interfaz sea pues una copia de la otra o incluso mejor, que una derive de la otra. En la práctica se ha demostrado que hay pocos beans que se utilicen tanto remota como localmente, por eso se cuestiona si vale la pena tanto esfuerzo.

En este punto es mucho más interesante, sobre todo si esto provoca alguna diferencia, si se llaman los métodos a través de una u otra interfaz. Así pues, aquí existen dos diferencias. Primero una llamada local siempre será más funcional, pues la llamada del método se tiene que enviar sin comprimir y a través de una dirección de recorrido. El cliente local trabaja con un objeto representativo que conduce a una llamada de método directa al servidor. Por esta razón se puede insertar el valor de retorno del método **lookup()** directamente a la interfaz local esperada. El método **narrow()** no es necesario. Aun hay sin embargo otra diferencia y esta es mucho más importante. Si se entrega a un método Java como parámetro una instancia de una clase, entonces tan solo se transmitirá la referencia a la instancia. Si el método llamado modifica el contenido de la instancia dada, este cambio repercutirá en el método que llama. Contemplamos ahora pues el método **addPedido()** de la clase **CestaCompraBean**, representado de nuevo en el Código 2.40.

```
public void addPedido( Pedido best )
{
best.setCantidadEntrega( best.getCantidad() );
Pedidos.add( best );
}
```

Código 2.40: Método addPedido() del bean de sesión.

El método espera como parámetro una instancia de la clase **Pedido**. Antes de tomar esta en el Vector pedidos. Esta fija la cantidad de entrega prevista a la par que la cantidad pedida llamando el método **setCandidadEntrega()**. Así se ha modificado el contenido de la instancia transmitida. Para un cliente local esto significa que puede programar sin problemas la cantidad de entrega, después de haber llamado el método **addPedido()** del Session Bean. Para un cliente, que se encuentre fuera de la VM, todo es diferente. En su llamada de método se serializa la instancia de la clase Pedido se envía a la VM ajena, donde se convertirá de nuevo en una instancia Java. Naturalmente esta ya no tiene nada que ver con la instancia cliente. Si el método modifica desde el servidor la cantidad de entrega, será solo la copia de la instancia del servidor. Si la llamada de método regresa al cliente, no se habrá modificado nada en la instancia trasmitida. Si el cliente remoto pregunta la cantidad de entrega, recibirá **null**. Observe de nuevo el Código 2.39 con la clase Pedido. El atributo **cantidadEntrega**() permanece en **null**, excepto cuando se fija a través del **método setCantidadEntrega()**.

Java EE diferencia entre el llamado Remote-Client-View y el Local-Client-View. También cuando se ofrecen en ambos casos los mismos métodos, se puede diferenciar claramente el desarrollo de la aplicación completa de las bases anteriormente descritas. Al diseñar un Session Bean siempre se debe partir del hecho que será llamado remotamente y después tener en cuenta que ningún método modifique el contenido de los parámetros transmitidos como instancias. Para conseguir la mejor manera es programas clases, que sirvan para la entrega de parámetros inmutables, o sea invariables. Sus atributos se fijaran en el constructor y después solo se podrán modificar leyendo. Además se declaran todos los atributos como **private** y no hay ningún método **setter** o similar. Además hay toda una seria de clases Java que se construyen de este modo. Pertenecen a estas **String**, **Integer** o también **BigDecimal**. Una vez creadas ya no se pueden modificar. Si por ejemplo se añade un nuevo valor a un nuevo valor a un **BigDecimal** se obtiene como resultado una nueva instancia.

2.17.2.2.5. EJBContext y SessionContext

La SessionContext también está a disposición de los métodos de un Stateful Session Bean. Este no se diferencia apenas en nada de lo que ya se ha descrito en el apartado sobre los Stateless Session Bean. Los métodos, su significado y su método de trabajo son idénticos, por lo tanto no se los va a describir de nuevo.

2.17.2.2.6. Ciclo de vida de un Stateful Session Bean

Un Stateful Session Bean pasa por diferentes estados a largo de su vida. En general se diferencian aquí tres estados. Se presentan en la Figura XV. Si se contempla el comportamiento de transacción de un bean de este tipo tendríamos entonces otros estados más, pero en el dedicado a Transacciones se describirá más a fondo todo lo relacionado

con esto. Si se compara el ciclo de vida de un bean de sesión con estado con el de un bean de sesión sin estado, llama la atención que en los métodos descritos aquí no hay ningún método Ready Pool, sino que se encuentran solamente en el estado del método Ready. Ya al del tema se ha hecho referencia a ello, por este motivo un pool general de instancias de bean no tiene ningún sentido cuando se trata de Stateful Session Beans.

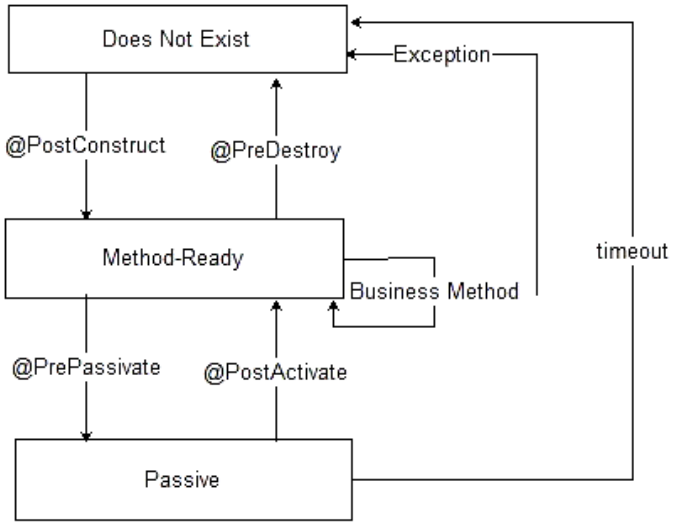


Figura XV: Ciclo de vida de un Stateful Session Bean

2.17.2.2.6.1. Does not exist

En este estado no existe todavía ninguna instancia del bean. El servidor de aplicaciones todavía no las ha creado.

PostContruct

Los procesos dentro de la fase **PostContruct** apenas se diferencian de los de un Stateless Session Bean. También aquí el servidor de aplicaciones utiliza primero el método **class.newIntance()** de la clase bean para llegar a una nueva instancia. Sea de nuevo mencionado que cada Stateful Session Bean debe disponer de un constructor estándar. A continuación inyecta todos los recursos deseados. Al contrario que los Stateless Beans la instancia del bean a partir de este momento está firmemente vinculada con un cliente. Por último se lama aun un método que dispone de la anotación **@PostContruct** o que se define mediante el Deployment Descriptor como método **PostContruct**, si es que hay alguno. No espera recibir ningún parámetro y tampoco lanza ninguna excepción. También en los Stateful Session Beans el método **Post-Contruct** opcional se llama tan solo una vez en su vida.

La especificación JavaEE dice que el servidor de aplicaciones debe transmitir un bean del estado Does Not Exist al estado Method-Ready, como muy tarde cuando el primer método Business es llamado por el cliente. Aún así, es también posible hacerlo después de un lookup.

Ejemplos para la anotación @postContruct o la definición de un método Post-Contruct en el Deployment Descriptor se encuentran en la sección sobre los Stateless Session Beans. De manera diferente a la de allí, es imprescindible que todas las instancias creadas dentro de este método y guardados en atributos del bean sean serializables ya que los Stateful Beans entran en estado pasivo. Esto es muy importante.

2.17.2.2.6.2. Method-Ready

En este estado existe una instancia de bean concreta en la memoria principal del servidor que está vinculada a un cliente concreto. Está a disposición para la ejecución de sus métodos Business. Siempre que no haya muchos usuarios conectados a la vez y tampoco demasiada confianza en la seguridad ante caídas del servidor en un cluster, es bastante probable que este tipo de instancias de bean se queden cierto tiempo en la memoria principal. Podría suceder incluso que nunca llegaran a entrar en un estado pasivo por el servidor, aunque tampoco debería darse por hecho.

Método Business

En esta fase se ejecuta un método del bean. Puede cambiar el estado interno del bean, reescribir por lo tanto la conversación y guarda todo lo necesario para ello en sus atributos en forma serializable. Naturalmente debe procurarse no acumular demasiada información. Sin embargo hoy en día tampoco es necesario tener tanto miedo como se tenía antes a que la memora principal pueda resultar insuficiente.

PreDestroy

Esta fase es de gran importancia en los Stateful Session Beans. Convierte la instancia del bean de **Method-Ready** a **Does Not Exist**. Esto significa que es liberada, es decir que el cliente ya no la requiere más. La iniciativa para ello tan solo puede venir del cliente. Si la sesión ha finalizado porque el usuario por ejemplo ha cerrado la sesión, entonces el cliente llama un método del bean provisto de la anotación **@Remove**. Esta es la señal para el servidor que indica que debe liberar la instancia después de la ejecución de este método. Puede haber métodos remove de este tipo a voluntad. Si se quieren devolver los recursos en un punto determinado, entonces es lógico programar un método mas en el bean, con la anotación **@PreDestroy** o declarada con el Deployment Descriptor como método **Pre-Destroy**. Ya se han mostrado ejemplos sobre esto en el apartado sobre los Stateless Session Beans. Si existe uno de estos métodos se ejecutara al finalizar el método **Remove**.

Aún existe una segunda posibilidad para llegar a la fase **PreDestroy**. Si el cliente no actúa durante un tiempo, el bean permanece entonces en la memoria principal y entra en **Timeout**, la instancia del bean pasará asimismo al estado **Does Not Exist** y se llamará antes el método opcional **Pre-Destroy**. El periodo de tiempo hasta que se entra en el **Timeout** se puede configurar. ¡Pero atención, si el bean entra en estado pasivo y en **Timeout**, ya no se producirá ninguna llamada más del método **Pre-Destroy**! Es decisión

del servidor de aplicaciones si cambia el estado de un bean a pasivo o no. Tampoco se puede confiar del todo en que el método **Pre-Destroy** entre en un **Timeout**, especialmente porque la probabilidad de pasivizarse irá creciendo a medida que aumente el tiempo en qué el cliente no se anuncie. En un sistema a pleno rendimiento las instancias de beans se encontrarán más bien en el estado **Passive** cuando empiece el **Timeout**.

Exception

Si durante la ejecución de un método Business apareciera una **RuntimeException**, esto significa para el servidor de aplicaciones que la instancia del bean ya no se encuentra en ningún estado interno que le permita liberar de inmediato la instancia del bean. Entonces cambia directamente al estado **Does Not Exist** sin que se llame un método **Pre-Destroy**. La transacción finalizará con un **Rollback**.

Si se produce una Exception normal, la consecuencia será simplemente que el método llamado finalizara antes de lo previsto, la instancia del bean sobrevive. La correspondiente excepción se transmitirá al cliente. Si no se hace ninguna indicación más entonces la transacción finalizará con un Commit, y los datos por lo tanto se escribirán en la base de datos.

Si se otorga una anotación **@ApplicactionException** a una **RuntimeException**, se manejará como una excepción normal. Provoca la finalización prematura de un método, pero aparte de esto no tendrá más consecuencias. En el Código 2.41 se puede ver como se puede programar una clase **Exception** de este tipo. En particular se representará también un **Commit**.

Es importante el estado **rollback = true**, que determine que una transacción activa se revoca al producirse esta excepción. El valor estándar para **rollback** es false. Por esta razón es también práctico dotar una **Exception** normal con la anotación **@ApplicationException**, puesto que en caso de error no se quiere escribir nada en la base de datos.

```
package server.sf;
import javax.ejb.ApplicationException;
@ApplicationException ( rollback = true)
public class CestaCompraException extends Exception
{
private static final long serialVersionUID = 1L;

public CestaCompraException( String msg )
{
super( msg );
}
}
```

Código 2.41: Definición de una ApplicationException

Dentro de un método Business se puede comprobar con tranquilidad un Stateful Session Bean, si por ejemplo todos los parámetros son validos o no. En caso de error se produce

una excepción declarada como **ApplicationException**, lo que no tiene que conducir inmediatamente a la aniquilación de la instancia. Un ejemplo de ello es el Código 2.42.

```
public void addPedido( Pedido best ) throws CestaCompraException
{
if( best == null )
throw new CestaCompraException(“Pedido es null”);
…
}
```

Código 2.42: Método Business con ApplicationException

Como en cualquier otra aplicación también el cliente debe ocuparse del tratamiento de la excepción. En el caso de una **ApplicationException**, sin embargo, se puede seguir trabajando con total normalidad. Después del ejemplo del Código 2.43 también se puede volver a intentar construir una instancia válida de Pedido y mediante la llamada de **addPedido()** transmitir el Session Bean.

```
try
{
Pedido local = new Pedido( artnr, cantidad, precio, den);
cdlc.addPedido( local );
}
catch( CestaCompraException ex)
{
out.print(“Error: ” + ex.getMessage());
}
```

Código 2.43: Cliente con Exceptionhandling

Si aparece de todos modos una excepción, el cliente aán está en situación de informar sobre ello al usuario para poder entonces finalizarse de la manera más ordenada posible. Cualquier otro intento de trabajar con la referencia al bean provocará irremediablemente más errores.

PrePassivate

Solo los Stateful Session Bean entran en estado pasivo. Como ya se ha descrito, esto significa que son escritos en un medio externo como un disco duro. Esto ocurre o bien en forma de un archivo normal o a través de una inserción en una base de datos. Esto último depende del servidor de aplicaciones.
No está fijado cuando un bean entra en estado pasivo. Lo único seguro es que nunca ocurre durante la ejecución de un método Business, independientemente de lo que este dure. Si el cliente no se anuncia durante cierto período de tiempo, al servidor le está permitido convertir la instancia del bean del estado **Method-Ready** al estado **Passive**. En la fase **PrePassivate** este llama inmediatamente, antes de guardarlo temporalmente, a un método bean con la anotación @**PrePassivate**. Uno de estos métodos no debe existir, no

espera ningún parámetro y tampoco produce ninguna excepción. Un pequeño ejemplo se ve en el Código 2.44.

```
import javax.ejb.PrePassivate;

@Stateful
public class CestaCompraBean implements CestaCompraRemote
{
@PrePassivate
public void prePasivizar( )
{
attribx = null;
}
}
```

Código 2.44: Fragmento de Bean con método Pre-Passivate

El nombre **PrePassivate** significa “antes de entrar en estado pasivo”. Mientras se ejecuta el método, este se encuentra normalmente aún en la memore principal. La tarea de este método es liberar todos los atributos de la clase bean que no son serializables.

En la vida de un Stateful Session Bean puede ocurrir a menudo que se pasivice. A medida que aumenta la edad, también la posibilidad de que esto suceda.

```
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>CestaCompraBean</ejb-name>
<pre-passivate>
<lifecycle-callback-method>
prePasivizar
</lifecycle-callback-method>
</pre-passivate>
</session>
</enterprise-beans>
</ejb-jar>
```

Código 2.45: Definición XML de un método Pre-Passivate

Como alternativa a la anotación @Pre-Passivate también es posible declarar un método bean mediante el Deployment Descriptor como método Pre-Passivate, como se puede ver en el Código 2.45.

2.17.2.2.6.3. Passive

En este estado la instancia de bean no puede ser utilizada. Se podría decir que está durmiendo. De este modo apenas supone una carga para los recursos del servidor de aplicaciones, este dispone de más memoria principal, pues se lo puede dar a los beans activados. Al principio de la sección ya se ha ilustrado qué medios utiliza un servidor de

aplicaciones moderno para solucionar el problema que surge por una parte del servir simultáneamente a muchos usuarios, y por otra al ofrecerles también la posibilidad de estar conectados con una instancia de bean.

PostActive

Si el cliente llama un método en un bean pasivo, este método se leerá en la fase **PostActive** por el medio externo y se proyectará de nuevo en la memoria principal. Si la clase bean ha provisto un método con la anotación **@PostActivate**, el servidor de aplicaciones lo llamará inmediatamente después del restablecimiento de la instancia incluso antes del mismo método business. Un método de este tipo no debe esperar ningún parámetro ni ejecutar ninguna excepción. Lo reproducimos en el Código 2.46.

```
import javax.ejb.PostActivate;

@Stateful
public class CestaCompraBean implements CestaCompraRemote
{
@PostActivate
public void postActivacion( )
{
attribx = new …;
}
}
```

Código 2.46: Esquema de un método Post-Activate

Su tarea consiste en restablecer las instancias liberadas en los métodos Pre-Passivate si estas son necesarias para continuar trabajando.

```
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>CestaCompraBean</ejb-name>
<post-activate>
<lifecycle-callback-method>
postActivacion
</lifecycle-callback-method>
</post-passivate>
</session>
</enterprise-beans>
</ejb-jar>
```

Código 2.47: Definicion XML de un método Post-Activate

También aquí es posible de nuevo, en lugar de la anotación, utilizar el Deployment Descripto.

Como en el caso del método Pre-Passivate, es más común en la vida de un Stateful Session Bean la llamada de su método Post-Activate, en caso de que esté programado.

Timeout

Esta es la fase más desagradable a la que puede llegar un Stateful Session Bean. Si entra en estado pasivo y es aparentemente olvidado por el cliente, la instancia del bean será directamente rechazada al actuar el **Timeout**, sin ser activada antes. Es todo en general desagradable porque el desarrollador de la clase bean ya no tiene oportunidad de liberar referencias propias o de informar a otros sobre el hecho de que ya no habrá una propia instancia.

Si se ha completado la mitad de la conversación con el cliente, es peligroso dar a conocer el estado alcanzado hasta ahora puesto que se realizan entradas en una tabla de base de datos. Si se hace esto se tienen que señalar como incompletas o provisionales. Esto puede realizarse mediante un campo de estado en la tabla. Si se llaga a **Timeout** aquí representado las líneas se quedan en la base de datos, pues la transacción en la que han sido escritas hace tiempo que ha pasado y ya no es posible intervenir para eliminarlas de nuevo. Por lo tanto debe haber un servicio que elimine estas líneas de tanto en tanto, por ejemplo todas las líneas que sean anteriores a una semana. En el EJB 3.1 esto podría ser un Timer Bean.

2.17.2.2.7. XML Deployment Descriptor

A pesar de las anotaciones puede resultar práctico dar a una aplicación JavaEE un Deployment Descriptor completo o por lo menos formulado en parte. Al fin y al cabo este permite parametrizar la aplicación desde fuera hasta su arquitectura. Quien renuncie del todo de las anotaciones o quiera sobrescribir sus indicaciones, encontrara en el Deployment Descriptor la herramienta perfecta. En este punto debemos hacer referencia de nuevo al apartado correspondiente en la sección sobre los Stateless Session Beans.

```
<ejb-jar
xmlns=”http://java.sun.com/xml/ns/javaee”
xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance
xsi:SchemaLocation=”http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3.1.xsd””
version=”3.1”>
<enterprise-beans>
<session>
<ejb-name>CestaCompraBean</ejb-name>
<remote>server.sf. CestaCompraRemote</remote>
<ejb-class>server.sf.CestaCompraBean</ejb-class>
<session-type>Stateful</session-type>
<remove-method>
<bean-method>
<method-name>remove</method-name>
</bean-method>
<retain-if-exception>true</retain-if-exception>
</remove-method>
<post-activate>
<lifecycle-callback-method>
postActivacion
</lifecycle-callback-method>
</post-passivate>
<pre-activate>
<lifecycle-callback-method>
prePasivacion
```

```
</lifecycle-callback-method>
</pre-passivate>
</session>
</enterprise-beans>
<assembly-descriptor>
<application-exception>
<exception-class>
server.sf.CestaCompraException
</exception-class>
<rollback>true</rollback>
</application-exception>
</assembly-descriptor>
</ejb-jar>
```

Código 2.48: Deployment Descriptor completo

En el proyecto JavaEE XML se cuida la utilización de las menores anotaciones posibles. A partir de este proyecto surge también el Deployment Descriptor, representado en el Código 2.48. Describe el bean **CestaCompra** con su interfaz remota y su clase bean. El **sessiontype** indica que se trata de un Stateful Session Bean. Además se definen un método **Remove**, un **PostActivate** y un **PrePassivate**. Preste atención a la entrada **retain-if-exception** en el método **Remove**. Indica que la instancia tras la ejecución de estos métodos no debe ser liberada si se produce una **ApplicationException** durante la ejecución.

En el Deployment Descriptor también podemos observar un ejemplo de cómo se pueden volver a describir clases Java normales que no funcionen como Bean. En este caso es la clase **CestaCompraException**, representada en el fragmento **assembly-descriptor**. Se define allí como **ApplicationException**, con cuya ejecución debe producirse un rollback de la transacción.

2.17.2.2.8. Acceso a través del programa cliente

Lo necesario técnicamente para acceder a un Session Bean desde un cliente ya se ha descrito detalladamente en el apartado con el mismo título sobre los Stateless Session Beans. Aquí se reproduce tan solo una JSP que el hasta ahora descrito **CestaCompraBean** utiliza como aplicación. Los fragmentos más importantes se explican a continuación.

```
CestaCompraRemote cdlc = (CestaCompraRemote)
sesión.getAttribute(“CestaCompraBean”);
if( cdlc == null )
{
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
“org.jnp.interfaces.NamingContextFactory”);
p.put(Context.URL_PKG_PREFIXES,
“org.jboss.naming:org.jnp.interfaces”);
p.put(Context.PROVIDER_URL,
”jnp://localhost:1099”);
Context ctx = new InitialContext(p);
Object ref = ctx.lookup(“CestaCompraBean/remote”);
cdlc = (CestaCompraRemote) PortableRemoteObject.narrow(ref, CestaCompraRemote.class);
sesión.setAttribute(“CestaCompraBean”, cdlc);
}
```

Código 2.49: Administración de la sesión.

Así el Código 2.49 muestra como el JSP administra dentro de su sesión la referencia bean que ha conseguido como cliente. Primero intenta leer la referencia desde los atributos de su sesión bajo el nombre CestaCompraBean. Si no la encuentra ejecuta un lookup como ya se ha descrito detalladamente en la sección sobre los Stateless Session Bean. Finalmente guarda la nueva referencia en sus atributos de sesión. Como se ha mencionado un Stateful Session Bean está firmemente vinculado a un cliente. De ahí se desprende que este deba realizar como máximo un lookup y que este deba considerar la referencia obtenida hasta el final de la conversación. Cada lookup posterior conducirá desde el servidor a siempre nuevas instancias de bean. Uun JSP tiene el problema de ser muy huidizo incluso dentro de su Contenedor Web mismo. Por este motivo lo mejor es anotar esta información en los atributos de su sesión.

```
if( request.getParameter(“pedir”) != null)
{
try
{
artnr = new Integer(request.getParameter(“artnr”));
}
catch( NumberFormatException e )
{
pedidoCorrecto = artNrCorrecto = false;
}

…

if( pedidoCorrecto )
{
try
{
Pedido local = new Pedido(artnr, cantidad, precio, den);
cdlc.addPedido(local);
if( local.getCantidadEntrega() != null )
out.print(“Cantidad entrega = ” + local.getCantidadEntrega());
}
catch( CestaCompraException ex )
{
out.print(“Error: ” + ex.getMessage());
}
}
}
```

Código 2.50: Esquema del procedimiento de pedido

A continuación sigue el proceso de pedido en sí. Si se pulsara el botón correspondiente con el nombre **Pedir**, la JSP intentaría seleccionar las informaciones individuales.

A estas corresponden junto al número del artículo también la cantidad, el precio y la denominación. En el Código 2.50 tan solo se ha representado la comprobación del número de artículo, el resto de parámetros se comprobarían de forma parecida. Si todo es correcto se crea una instancia del pedido y se transmite al servidor. Después se pregunta si se ha fijado la cantidad de entrega en la instancia de pedido local. El método del servidor **addPedido()** fija de hecho la cantidad de entrega pero como se trata de un cliente remoto,

no tiene consecuencias. Por lo tanto de esta manera no se llegara nunca a la emisión de la cantidad de entrega.

Después sigue la emisión de los pedidos realizados hasta ahora. Para ello se llama el método del servidor **getPedido()** y se emite el **Vector** obtenido de cada instancia de pedido. En estas está fijada el atributo **cantidadEntrega**, pues se trata de una copia de las instancias del servidor.

```
if( request.getParameter(“remove”) != null )
{
cdlc.remove();
sesión.removeAttribute(“CestaCompraBean”);
}
```

Código 2.51: Desbloqueo del Session Bean

Hacia el final del JSP vuelve a ser interesante, como muestra el Código 2.51. Si el usuario pulsa el botón con la inscripción **Finalizar Pedido**, el cliente libera el Stateful Session Bean llamando el método **remove()**. Como sabemos este posee por el lado de servidor la anotación **@Remove**. Además el JSP elimina su atributo con el nombre **CestaCompraBean**, pues ahora esta referencia ha dejado de ser válida.

```
<%@ page language=”java” contentType=”text/html; charset=ISO-8859-1”
pageEncoding=”ISO-8859-1” %>
<!DOCTYPE HTML PUBLIC “-//W3C//DTD HTML 4.01 Transitional//EN”>
<html>
<head>
<meta http-equiv=”Content-Type” contect=”text/html; charset=ISO-8859-1”>
<title>Cesta de la Compra</title>
</head>
<body>
<form action=”CestaCompra.jsp” method=”post”>
<h1 align=”center”>Cesta de la Compra</h1>
<table border=”1” align=”center”>
<tr>
<th>ArtNr</th>
<th>Cantidad</th>
<th>Precio</th>
<th>Denominación</th>
<th>Cantidad Entrega</th>
<th>Acción</th>
</tr>
<%@ page import = ”java.util.*”
import = “java.math.*”
import = “javax.naming.*”
import = “server.sf.*”
import = “javax.rmi.PortableRemoteObject”
%>
<%
CestaCompraRemote cdlc = (CestaCompraRemote)
session.getAttribute(“CestaCompraBean”);
if( cdlc == null )
{
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY, “org.jnp.interfaces.NamingContextFactory”);
p.put(Context.URL_PKG_PREFIXES, “org.jboss.naming:org.jnp.interfaces”);
p.put(Context.PROVIDER_URL, “jnp://localhost:1099”);
Context ctx = new InitialContext(p);
Object ref = ctx.lookup(“CestaCompraBean/remote”);
```

```
cdlc = (CestaCompraRemote) PortableRemoteObject.narrow(ref, CestaCompraRemote.class);
sesión.setAttibute(“CestaCompraBean”, cdlc);

boolean artNrCorrecto = true;
boolean cantidadCorrecto = true;
boolean precioCorrecto = true;
boolean denCorrecto = true;

Integer artnr = null;
Integer cantidad = null;
BigDecimal precio = null;
String den = null;

if( request.getParameter(“pedir”) != null )
{
try
{
artnr = new Integer(request.getParameter(“artnr”));
}
catch( NumberFormatException e )
{
pedidoCorrecto = artNrCorrecto = false;
}
try
{
cantidad = new Integer(request.getParameter(“cantidad”));
}
catch( NumberFormatExceptio e )
{
pedidoCorrecto = cantidadCorrecto = false;
}
try
{
precio = new BigDecimal(request.getParameter(“precio”).replace()’,’,’.’);
}
catch( NumberFormatException e )
{
pedidoCorrecto = precioCorrecto = false;
}

den = request.getParameter(“den”);
if( den == null  || den.length() == 0 )
{
pedidoCorrecto = denCorrecto = false;
}
If( pedidoCorrecto )
{
try
{
Pedido local = new Pedido(artnr, cantidad, precio, den);
cdlc.addPedido(local);
if( local.getCantidadEntrega() != null )
out.print(“Cantidad entrega = ” + local.getCantidadEntrega());
}
catch( CestaCompraException ex )
{
out.print(“Error: ” + ex.getMessage());
}
}
}

for( int i = 0; I < cdlc.getPedido().size; i++ )
{
Pedido best = (Pedido) cdlc.getPedido().get(i);
out.print(“<tr>”);
out.print(“<td>” + ped.getArtNr() + “</td>”);
out.print(“<td>” + ped.getCantidad() + “</td>”);
```

```
out.print("<td align='right'>" + ped.getPrecio() + "</td>");
out.print("<td>" + ped.getDenominacion() + "</td>");
out.print("<td>" + ped.getCantidadEntrega() + "</td>");
out.print("</tr>");
}
%>
<tr>
<td><input name="artnr" size="5" maxlength="5"
<%
if( artNrCorrecto == false )
out.print(" style='background-color:yellow'");
if( pedidoCorrecto == false )
out.print(" value='" + request.getParameter("artnr") + "'");
%>
></td>
<td><input name="cantidad" size="5" maxlength="5"
<%
if( cantidadCorrecto == false )
out.print(" style=''background-color:yellow'");
if( pedidoCorrecto == false )
out.print(" value='" + request.getParameter("cantidad") + "'");
%>
></td>
<td><input name="precio" size="8" maxlength="8"
<%
if( precioCorrecto == false )
out.print(" style=''background-color:yellow'");
if( pedidoCorrecto == false )
out.print(" value='" + request.getParameter("precio") + "'");
%>
></td>
<td><input name="den" size="30" maxlength="30"
<%
if( denCorrecto == false )
out.print(" style=''background-color:yellow'");
if( pedidoCorrecto == false )
out.print(" value='" + request.getParameter("den") + "'");
%>
></td>
<td></td>
<td><input type="submit" name="pedir" value="Pedir"></td>
</tr>
</table>
<p align="center">
<input type="submit" name="remove" value=" Finalizar pedido">
</p>
<%
if( request.getParameter("remove") != null )
{
cdlc.remove();
session.removeAttribute("CestaCompraBean");
}

out.print("<br>(Su session finalizará tras " + session.getMaxInactiveInterval() + " segundos / " +
sesión.getMaxInactiveInterval() / 60 + " minutos)");
%>
</form>
</body>
</html>
```

Código 2.52: JSP complete para la Cesta de Compra

En el Código 2.52 se reproduce el JSP completo. En cualquier caso, es conveniente trabajarlo y ponerlo en práctica. El dialogo que crea el JSP se representa en la Figura XVI.

Figura XVI: Interfaz de la cesta de la compra

2.17.2.3. **Singleton Session Beans**

2.17.2.3.1. Uso

Un Singleton Session Bean es un componente bean de sesión que es instanciado una vez por aplicación. En los casos en los que el contenedor es distribuido sobre muchas máquinas virtuales, cada aplicación tendrá una instancia de bean del Singleton para cada JVM.
Una vez instanciado, una instancia de Sigleton Session Bean vive durante el transcurso de la aplicación en la que se crea. Se mantiene el estado entre las invocaciones de cliente, pero este estado no es requerido para sobrevivir a paradas, caídas o accidentes del contenedor.

Un Singleton Session Bean está destinada a ser compartido y permite el acceso concurrente.

Un Singleton Session Bean no debe implementar la interfaz **javax.ejb.SessionSynchronization** o utilizar las anotaciones de sincronización de sesión.

2.17.2.3.2. Inicialización

De forma predeterminada, el contenedor es responsable de decidir cuándo iniciar una instancia del bean Singleton. Sin embargo, si el desarrollador del bean lo desea, puede configurar el Singleton para su inicialización. Si la anotación **@Startup** aparece en la clase bean Singleton o si el Singleton ha sido designado por el descriptor de despliegue que requieren inicialización, el contenedor debe inicializar la instancia del bean Singleton durante la secuencia de inicio de la aplicación. El contenedor debe inicializar todos los Singleton durante el tiempo de inicio antes que las solicitudes del cliente externo (es decir, las solicitudes de cliente se originan fuera de la aplicación) se entregen a los bean de componentes empresariales en la aplicación.

El siguiente ejemplo muestra un Singleton con la lógica de inicio que inicializa su estado compartido:

```
@Startup
@Singleton
public class SharedBean implements Shared
{
private SharedData state;
@PostConstruct
void init()
{
// initialize shared data
...
}
...
}
```

Código 2.53: Inicialización de un Singleton

En algunos casos, las dependencias de inicialización explícita pedidos existentes entre varios componentes de Singleton en una aplicación. La anotación **@DependsOn** se utiliza para expresar estas dependencias. Una dependencia **DependsOn** se utiliza en los casos en que un Singleton debe inicializar antes de que uno o más Singletons. El contenedor garantiza que todos los beans Singleton que tienen una relación **DependsOn** con otro Singleton se han iniciado antes de llamar al método **PostConstruct**.

Deberá tenerse en cuenta que si un Singleton sólo tiene que invocar a otro Singleton de su método **PostConstruct**, no se requiere metadatos explícito en el pedido. En ese caso, el primer Singleton Session Bean sólo utiliza una referencia EJB para invocar el recurso del Singleton. Allí, la adquisición de la referencia EJB (ya sea a través de inyección o de búsqueda) no implica necesariamente la creación real de la instancia del Singleton Bean correspondiente.

Los siguientes ejemplos ilustran el uso de la anotación **@DependsOn**:

```
@Singleton
public class B
{
...
}

@DependsOn("B")
@Singleton
public class A
{
...
}
```

Código 2.54: Uso de anotación @DependsOn

Esto le dice al contenedor que garantiza que el Singleton B se inicializa antes de Singleton A. El valor del atributo **DependsOn** tiene una o más cadenas, donde cada uno especifica

el nombre ejb de la clase de Singleton utilizando la misma sintaxis que el atributo **@EJB beanName()**.

```
@Singleton
public class B
{
...
}

@Singleton
public class C
{
...
}

@DependsOn({“B”, “C”})
@Singleton
public class A
{
...
}
```

Código 2.55: Anotacion @DependsOn con múltiples clases

Esto le dice al contenedor que garantice que el Singleton B y C se inicializanrán antes del Singleton A. En el caso de varios valores, el orden en el que los nombres ejb de objetivo figuran no se conserva en tiempo de ejecución. Por ejemplo, si el Singleton B tiene una dependencia al comprar en Singleton C, es responsabilidad del Singleton B para capturar explícitamente que en sus propios metadatos.

```
// dos componentes Singleton empaquetados
// diferentes ejb-jars dentro del mismo .ear

// empaquetado en b.jar
@Singleton
public class B
{
...
}

// empaquetado en a.jar
@DependsOn(“b.jar#B”)
@Singleton
public class A
{
...
}
```

Código 2.56: @DependsOn con Singleton empaquetado en módulo diferente

El Código 2.56 demuestra el uso de la sintaxis completa para hacer referencia a un componente Singleton empaquetado en un módulo diferente en la misma aplicación.

Dependencias circulares dentro de las anotaciones **@DependsOn** no están permitidas. Las dependencias circulares no están obligadas a ser detectadas por el contenedor, pero puede resultar en un error de implementación.

2.17.2.3.3. Destrucción

Cualquier instancia de un Singleton Session Bean que completa con éxito la inicialización es que se elimina explícitamente por el contenedor durante el cierre de la aplicación. En este momento el contenedor invoca el método **PreDestroy** del ciclo de vida, si las hubiere. El contenedor garantiza que todos los Singleton beans con la que un Singleton tiene una relación **DependsOn** todavía están disponibles durante el **PreDestroy**. Después de que el método PreDestroy se completa, el contenedor termina la vida de la instancia del Singleton Session Bean.

2.17.2.3.4. Semántica de transacciones de inicialización y destrucción

Los métodos **PostConstruct** y **PreDestroy** del Singleton con las transacciones gestionadas por contenedor son transaccionales. Desde el punto de vista del desarrollador del bean no hay ningún cliente de un método **PostConstruct** o **PreDestroy**.

Un método **PostConstruct** o **PreDestroy** de un Singleton con las transacciones gestionadas por contenedor tiene atributos de transacción **REQUIRED**, **REQUIRES_NEW** o **NOT_SUPPORTED** (Required, RequiresNew o NotSupported si el descriptor de despliegue se utiliza para especificar el atributo de transacción).

Hay que tener en cuenta que el contenedor debe iniciar una nueva transacción si el atributo **REQUIRED** (Required) de transacción se utiliza. Esto garantiza, por ejemplo, que el comportamiento transaccional del método **PostConstruct** es el mismo independientemente de si se ha inicializado con éxito a la hora de inicio del contenedor o como un efecto secundario de una invocación primaria del cliente en el Singleton. El valor del atributo de la transacción necesario se permite por lo que la especificación de un atributo de transacción para los métodos **PostConstruct** / **PreDestroy** del Singleton puede ser incumplidos.

2.17.2.3.5. Ciclo de vida de un Singleton Session Bean

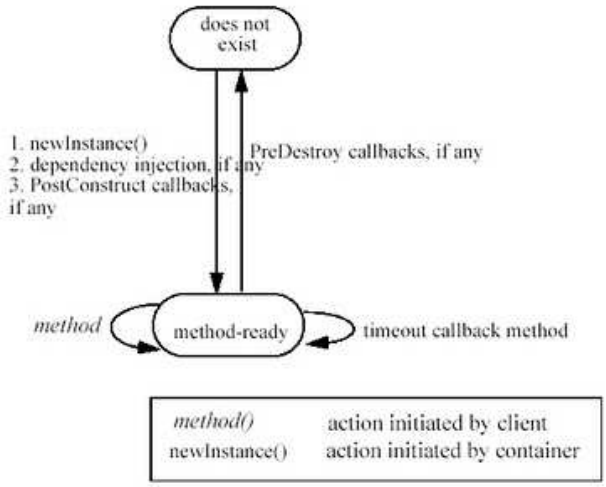


Figura XVII: Ciclo de Vidad de un Singleton Session Bean

La Figura XVII muestra el ciclo de vida de un Singleton Session Bean, a continuación se describe dicho ciclo de vida:

- La vida de una instancia de bean de sesión singleton se inicia cuando el contenedor llama al método **newInstance()**, que es inicializado por el contenedor, en la clase bean de sesión para crear la instancia del bean singleton. A continuación, el contenedor lleva a cabo cualquier inyección de dependencia según lo especificado por las anotaciones de metadatos en la clase del bean o por el descriptor de despliegue. El contenedor llama al método **PostConstruct** del ciclo de vida y los métodos interceptores para la devolución de la llamada del bean, en su caso.

- La instancia del bean Singleton ya está lista para ser delegada una llamada al método de negocio de cualquier cliente o una llamada desde el contenedor a un método de devolución de llamada de tiempo de espera.

- Cuando la aplicación se está cerrando, el contenedor invoca el método **PreDestroy** del ciclo de vida, si las hubiere. Esto termina la vida de la instancia del Singleton Session Bean.

2.17.2.3.6. Control de errores en un Singleton

Los errores que se producen durante la inicialización del Singleton se consideran mortales y debe resultar en el descarte de la instancia de Singleton. Posibles errores de inicialización incluyen la falta de inyección, una excepción del sistema lanzado desde un método **PostConstruct**, o el fracaso de una operación del método **PostConstruct** gestionada por el contenedor para ejecutar exitosamente.

La misma instancia del bean Singleton debe permanecer activo hasta que cierre de la aplicación. A diferencia de instancias de tipos de otro componente, los sistemas de excepciones lanzadas desde los métodos comerciales o devoluciones de llamada de un Singleton no dan lugar a la destrucción de la instancia de Singleton.

2.17.2.3.7. Concurrencia Singleton

Desde la perspectiva del cliente, un bena de Singleton siempre apoya el acceso concurrente. En general, un cliente Singleton no tiene que preocuparse por si otros clientes pueden acceder al Singleton, al mismo tiempo.

Desde la perspectiva del desarrollador de beans, hay dos enfoques para controlar el comportamiento de concurrencia Singleton:

- **Concurrencia gestionada por contenedor:** el contenedor de controles de acceso concurrente a la instancia de bean basado en el método a nivel de metadatos de bloqueo.

- **Concurrencia gestionada por bean:** el contenedor completo permite el acceso simultáneo ejemplo de bean y se remite la responsabilidad del estado de sincronización para el desarrollador de bean.

En el diseño de un bean de sesión Singleton, el desarrollador debe decidir si el bean se utiliza contenedores manejados o concurrencia de bean gestionados. Normalmente, los beans Singleton especificarán que la demarcación de contenedores de concurrencia gestionados. Este es el valor predeterminado si ningún tipo de gestión de la concurrencia se especifica. Los bean Singleton se pueden diseñar para utilizar concurrencia gestionada por contenedores o concurrencia de ban gestionada, pero no se puede utilizar los dos.

El ciclo de vida de las clases asociadas con un interceptor de Singleton tienen el mismo ciclo de vida y el comportamiento de la concurrencia como el Singleton. Cada clase interceptor se crea una instancia una vez por instancia del bean Singleton. Cualquier estado almacenados en una instancia de una clase interceptor asociado a un Singleton se debe considerar al elaborar del plan de concurrencia para el bean.

Es legal para almacenar objetos Java EE que no soportan el acceso concurrente (por ejemplo, administradores de entidades, las referencias Stateful Session Bean) dentro del estado de Singleton instancia del bean. Sin embargo, es responsabilidad del desarrollador de bean para asegurar tales objetos que no sean visitados por más de un subproceso a la vez.

Independiente del tipo de bean de gestión de concurrencia, el contenedor debe asegurarse de que no tienen acceso simultáneo a la instancia del bean Singleton, se produce hasta después de la instancia se ha completado con éxito la secuencia de inicialización, incluyendo cualquier método **@PostConstruct** de devolución de llamada del ciclo de vida

(s). El contenedor debe bloquear temporalmente cualquier intento de acceder al Singleton que llegan, mientras que el Singleton está siendo inicializado.

Los beans Singleton admiten llamadas de reentrada. Es decir, cuando una llamada saliente de un método da como resultado una llamada de bucle de retorno al Singleton en el mismo subproceso. El únicos reentrante debe ser programado y utilizado con precaución. La semántica especial de bloqueo se aplica a las llamadas de bucle invertido en Singleton con la concurrencia gestionada por contenedor (que se detallan a continuación).

2.17.2.3.8. Concurrencia gestionada por Contenedor

Con demarcación de Concurrencia Gestionada por Contenedor, el contenedor se encarga de controlar el acceso concurrente a la instancia de bean basado en los metadatos de bloqueo de nivel de método. Cada método de negocio o el método de tiempo de espera es asociada con un **Read** (compartido) de bloqueo o **Write** (exclusivo) de bloqueo.

Si el contenedor llama a un método asociado con un **Read** de bloqueo, cualquier número de otras invocaciones simultáneas sobre los métodos de lectura se les permite acceder a la instancia de bean al mismo tiempo.

Si el contenedor llama a un método asociado con un **Write** de bloqueo, no hay invocaciones simultáneas, otros podrán continuar hasta su procesamiento del método **Write** inicial completa.

Un intento de acceso concurrente que no se le permite continuar debido al bloqueo se bloquea hasta que se pueda hacer progresos. Los tiempos de espera se puede especificar a través de los metadatos para que una solicitud de bloqueo puede ser rechazada si un bloqueo no se adquiere en un período de tiempo determinado. Si una invocación Singleton es rechazada debido a bloqueo de tiempo de espera de la **ConcurrentAccessTimeoutException** se lanza al cliente.

Esta especificación sólo establece la semántica básica **Read/Write** descrita anteriormente. Hay muchas políticas de decisiones que un contenedor puede hacer para afectar el rendimiento del esquema de bloqueo para una aplicación dada. Por ejemplo:

- Determinar la concesión de los progresos hacia adelante a un método **Read** o un método **Write**, cuando los lectores y los escritores están a la espera, al mismo tiemo que un método de escritura completa.

- Determinar si se permitirá a los lectores adicionales, mientras que uno o más lectores es activo y un escritor está esperando.

El conjunto exacto adicional de bloqueo **Read/Write** para políticas de decisiones con el apoyo de un proveedor de contenedor y los requisitos de configuración de estas políticas están fuera del alcance de esta especificación.

2.17.2.3.8.1. Comportamiento de bloqueo Reentrante

La semántica especial de bloqueo se aplica a las llamadas de bucle invertido en los Singleton con la concurrencia gestionada por contenedores.

Si una llamada de bucle invertido se produce en un Singleton que ya tiene un bloqueo **Write** en el mismo hilo:

- Si el destino de la llamada de bucle invertido es un método de **Read**, el bloqueo **Read** siempre debe ser atendida inmediatamente, sin dejar de lado el bloqueo de **Write** original.

- Si el destino de la llamada de bucle invertido es un método de **Write**, la llamada debe proceder de inmediato, sin soltar el bloqueo **Write** original.

Si una llamada de bucle invertido se produce en un Singleton que mantiene un bloqueo **Read** en el mismo hilo (pero también no tienen un bloqueo de **Write** en el mismo hilo):

- Si el destino de la llamada de bucle invertido es un método de **Read**, la llamada debe proceder de inmediato, sin soltar el bloqueo de **Read** original.

- Si el destino de la llamada de bucle invertido es un método **Write**, un **javax.ejb.IllegalLoopbackException** debe ser arrojado al que lo llama.

2.17.2.3.9. Concurrencia Gestionada por Bean

Con la demarcación de Concurrencia Gestionada por Bean, el contenedor permite el pleno acceso concurrente a la instancia del bean Singleton. Es responsabilidad del desarrollador de beans, proteger su estado como sea necesario para contrar los errores de sincronización debido a un acceso concurrente. El desarrollador del bean está facultado para utilizar el lenguaje Java primitivas de nivel de sincronización como sincronizada y permanente para este fin.

2.17.2.3.10. Especificación de un tipo de simultaneidad de Gestión

De forma predeterminada, la demarcación de un bean singleton tiene concurrencia gestionada por contenedor, si el tipo de administración de concurrencia no se especifica. El Proveedor de Beans de un Bean Singleton puede utilizar la anotación **ConcurrencyManagement** en la clase de bean para declarar el tipo de gestión de concurrecnai del bean.

Por otra parte, el Proveedor de Bean puede utilizar el descriptor de despliegue (Deployment Descriptor) para especificar el tipo de gestión de concurrencia del bean. Si el descriptor de despliegue se utiliza, sólo es necesario especificar explícitamente el tipo de gestión de concurrencia del bean si la concurrencia gestionada por bean se utiliza.

El tipo de gestión de la concurrencia de un Singleton es determinado por el proveedor del bean. El ensamblador de la aplicación no está facultado para utilizar el descriptor de despliegue para reemplazar un tipo de gestión de concurrencia de bean, independientemente de si se ha especificado explícitamente o por defecto por el Proveedor de Bean.

2.17.2.3.11. Especificación del Contenedor administrador de concurrencia de metadatos para los métodos de un Singleton bean

El Proveedor de Beans de un Singleton Bean con demarcación de concurrencia gestionada por contenedor puede especificar los métodos de fijación de metadatos para el bean empresarial. De forma predeterminada, el valor de la cerradura asociado a un método de un bean a la demarcación de concurrencia administrada por contenedor es **Write** (exclusivo), y el atributo de bloqueo de simultaneidad no tiene por qué ser explícitamente especificado en este caso.

Un atributo del bloqueo de concurrencia es un valor asociado a cada uno de los siguientes métodos:

- Un método de interfaz de negocio de un bean.
- Un método de vista sin interfaz de un bean.
- Un método de devolución de llamada de tiempo de espera.
- Un servicio web de método de punto final.

El atributo de bloqueo de concurrencia específica la forma del contenedor debe administrar la concurrencia cuando un cliente llama al método.

Los atributos de concurrencia de bloqueo se especifican para los siguientes métodos:

- Para un bean escrito para la vista del cliente EJB 3.x API, la concurrencia de bloqueo se especifican los atributos de los métodos de la clase de bean que se corresponden con interfaz de negocio del bean, el superinterfaces directos e indirectos de la interfaz de negocio, los métodos expuestos por el no vista de la interfaz, y sobre las formas de devolución de llamada de tiempo de espera, en su caso.
- Para un bean que proporciona una vista de cliente, servicio web, los atributos de bloqueo de concurrencia se especifican para los métodos de la clase de bean que se corresponden con los métodos del bean extremo de servicio web, y sobre las formas de devolución de llamada de tiempo de espera, en su caso.

Por defecto, si una anotación de bloqueo de concurrencia no se especifica un método de un bean Singleton a la demarcación de concurrencia gestionada por contenedor, el valor del atributo de bloqueo de concurrencia para el método se define como **Write**.

El Proveedor de Bean puede utilizar el descriptor de despliegue como una alternativa a los metadatos de las anotaciones para especificar los atributos de bloqueo de concurrencia (o como un medio para complementar o reemplazar las anotaciones de metadatos para los atributos de bloqueo de concurrencia). Los atributos de bloqueo de concurrencia especificados en el descriptor de despliegue se supone que van a reemplazar o complementar la fijación de los atributos de concurrencia especificados en las anotaciones. Si un valor de bloqueo de concurrencia del atributo no se especifica en el descriptor de despliegue, se asume que el atributo de bloqueo de concurrencia especificado en las anotaciones se aplica, o –en el caso de que no exista ninguna anotación especificada– que el valor es **Write**

El ensamblador de la aplicación se le permite anular la concurrencia de bloqueo valores de los atributos utilizando el descriptor de despliegue de bean. El implementador también permite anular la concurrencia de bloqueo los valores de los atributos en tiempo de despliegue. Se debe tener precaución cuando se reemplaza los atributos de concurrencia de bloqueo de una aplicación, como la estructura de la concurrencia de una aplicación suele ser inherente a la semántica de la aplicación.

2.17.2.3.12. Especificación de atributos de concurrencia de bloqueo con anotaciones

La siguiente es la descripción de las reglas para la especificación de los atributos de concurrencia con anotaciones en Java.

Las anotaciónes de bloqueo (**READ**) y de bloqueo (**WRITE**) se utilizan para especificar los atributos de bloqueo de concurrencia.

Los atributos de bloqueo de concurrencia de los métodos de una clase de bean que se indique en la clase, los métodos de negocio de la clase, o ambos.

Especificación de la anotación de bloqueo en la clase bean significa que se aplica a todos los métodos de negocios aplicables de la clase. Si el atributo de concurrencia de bloqueo no se especifica, se supone que es de bloqueo (**WRITE**). La ausencia de una especificación de atributos de concurrencia en la clase de bean es equivalente a la especificación de bloqueo (**WRITE**) en la clase bean.

Una concurrencia de bloqueo para el atributo puede especificarse en un método de la clase del bean para anular el valor del atributo de bloqueo de concurrencia de manera explícita o implícitamente especificados en la clase del bean.

Si la clase de bean tiene superclases, se aplicarán las normas adicionales siguientes:

- Un atributo de bloqueo de concurrencia especificado en una superclase S appplies a los métodos de negocio definidas por S. Si un atributo de la concurrencia de nivel de clase no se especifica en S, es equivalente a la especificación de bloqueo (**WRITE**) en S.

- Un atributo de bloqueo de concurrencia puede ser especificado en un método M de negocio definidas por la clase S de presupuesto para el método M del valor del atributo de bloqueo de concurrencia de manera explícita o implícitamente especificado en la clase S.

- Si un método M de la clase S reemplaza un método de negocio definidas por una superclase de S, el atributo de bloqueo de concurrencia de M se determinada que las reglas anteriores se aplican a la clase S.

Un ejemplo de esto, se puede observar en el Código 2.57.

```
@Lock(READ)
public class SomeClass
{
public void aMethod ()
{
...
}

public void bMethod ()
{
...
}
...
}

@Singleton public class ABean extends SomeClass implements A
{
public void aMethod ()
{
...
}

@Lock(WRITE)
public void cMethod ()
{
...
}
...
}
```

Código 2.57: Bloqueo de concurrencia con Superclase

Suponiendo aMethod, bMethod, cMethod de bean ABean Singleton son métodos de la interfaz de negocio, sus atributos de bloqueo de concurrencia son de bloqueo (WRITE), bloqueo (READ), y bloqueo (WRITE), respectivamente.

2.17.2.4. **Message-Driven Bean**

2.17.2.4.1. Uso

No todas las tareas que debe realizar una aplicación deben llevarse a cabo de inmediato. A veces es suficiente con hacer llegar un mensaje a la aplicación al que esta reacciona en un

momento dado. En este caso lo más importante es que el emisor del mensaje no depende de si recibe de inmediato una respuesta o incluso de si llega a recibir alguna. Exactamente este escenario es el que crean los beans dirigidos por mensajes o Message-Driven Beans.

La comunicación entre un cliente y un Message-Driven Bean siempre se produce mediante JMS (Java Message Service). Se trata de un estándar que prescribe, con ayuda de algunos métodos, qué mensajes pueden enviarse de un emisor a un receptor o grupo de receptores. JMS no es ninguna implementación de un solo servicio. Hay toda una serie de fabricantes que ofrecen sistemas de mensajería basados en JSM. Los más conocidos son MQSeries de IBM, BEA de WebLogic Service, JBossMQ, Sun ONE MessageQueue de Sun MicroSystem o también SonicMQ de Sonic. Todos esos sistemas hacen en principio lo mismo, pero se diferencian entre ellos en cómo funcionan y en su variedad de funciones. En parte pueden más de lo que requiere el estándar.

Un Message-Driven Bean reacciona ante aquellos mensajes que se envían al servidor de aplicaciones. A diferencia de los Session Bean, el cliente no tiene relación directa o de ningún otro tipo con un Message-Driven Bean. Envía un mensaje y continúa trabajando con su lógica sin esperar una respuesta. En lugar de realizar un lookup en una interfaz remota para así llamar métodos Business, el cliente accede al JMS y transmite sus mensajes.

Al principio podría compararse un Message-Driven Bean con un Stateless Session Bean, porque tienen las mismas restricciones. El servidor de aplicaciones crea tantas instancias de beans como sean necesarias para poder trabajar todos los mensajes entrantes tan rápido como sea posible. No hay por lo tanto ninguna garantía de que todos los mensajes de un cliente sean trabajados por la misma instancia de bean, sino más bien al contrario. Por eso está prohibido en principio para un Message-Driven Bean grabar datos en atributos propios porque estos carecen totalmente de estado. La diferencia esencial con un Stateless Session Bean es que los Message-Driven Beans solo tienen un solo método Business que no devuelve ningún valor. Toda la comunicación se desarrolla asincrónicamente mientras que en las llamadas de métodos de los Stateless Session Beans se trata de llamadas sincrónicas.

JMS diferencia entre dos tipos diferentes de sistemas de mensajes: Point-to-Point (P_2P) y Publish-and-Subscribe (pub/sub). En las variantes P_2P participan un emisor y un receptor. En las otras variantes el mensaje se hace llegar a un grupo de receptores independientemente de cuántos de ellos estén registrados en ese momento. Así puede determinarse si se guardan los mensajes en caso de que ningún receptor se interese por ellos o si deben perderse.

Para una conversación en varios pasos, los Message-Driven Beans no serían los más adecuados. Incluso cuando dentro de la transacción debe asegurarse que el mensaje enviado también se trabaje siguiendo un orden, este tipo de bean no resulta útil. Emisor y receptor se ejecutan en transacciones diferentes. Si durante la elaboración del mensaje se produjera un error, el emisor no lo sabría. Es más, posiblemente este haya terminado su transacción hace ya tiempo. En esta sección se mostrará a partir del ejemplo de un ChatRoom cómo se puede trabajar con un Message-Driven Bean. Cada cliente participante del chat puede enviar mensajes que después se transmitirán al resto.

2.17.2.4.2. Estructura

2.17.2.4.2.1. La clase bean

Como ya se ha dicho, los Message-Driven Beans tienen una construcción muy simple. La clase bean consta exactamente de un método business, que debe llevar el nombre **onMessage()**. Mediante la anotación **@javax.ejb.MessageDriven** se determina que la clase Java debe ser un Message-Driven Bean. En el Código 2.58 se reproduce la correspondiente clase de ejemplo. Tampoco aquí es obligatorio dar el nombre de clase utilizado ChatBean, pero es recomendable para que quede claro que se trata de un Bean.

```
Package server.md;

import javax.ejb.*;
import javax.jms.*;

@MessageDriven(activationConfig={
@ActivationConfigProperty(
propertyName=”destinationType”,
propertyValue=”javax.jms.Queue”),
@ActivationConfigProperty(
propertyName=”destination”,
propertyValue=”queue/testQueue”),
@ActivationConfigProperty(
propertyName=”acknowledgeMode”,
propertyValue=”Auto-acknowledge”)})

public class ChatBean implements MessageListener
{
public void onMessage( Message message )
{…}
}
```

Código 2.58: Message-Driven Bean

La anotación utilizada aquí para la definición del tipo de bean se claramente más extensa que en el resto. De hecho determinarse más propiedades acerca de qué tipo de mensajes se trata y a qué canal de mensajería debe atender el bean. En el ejemplo reproducido se trata de una cola con el nombre **queue/textQueue**. Esta debe crearse por el servidor de aplicaciones y ser administrada. Por defecto, JBoss ya tiene la llamada Queue. Junto a las Queues también están a disposición los llamados Topics. Se hablará sobre la diferencia entre ambos en el apartado sobre el Método de Trabajo de los Message-Driven Beans.

Cada **Property** es del tipo @**ActivationConfigProperty**, que espera recibir tanto un **propertyName** como un **propertyValue**. Estos datos dependen de la implementación JMS utilizada concretamente. Desafortunadamente no existe ningún estándar auténtico. Las indicaciones se llevan a cabo con ayuda de cadenas de simbolos que se utilizan justo durante el tiempo de ejecución. Si faltaran ciertas propiedades no tiene por qué producirse

un mensaje de error. Si por ejemplo se eliminara la indicación de la destinación, JBoss activaría el Message-Driven Bean pero no sería nunca usado.

destinationType

Con ayuda de esta propiedad se indica a qué tipo de sistema de mensajería atiende este bean. Se puede elegir actualmente entre **javax.jms.Queue** y **javax.jms.Topic**. La principal diferencia es que, en una **queue**, cada mensaje se entregará como máximo a un receptor independientemente de cuántos de ellos estén registrados en el sistema de mensajería, mientras que en el otro caso todos los clientes dados de alta reciben el mensaje.

destination

Aquí se tiene que determinar el nombre de la **Queue** o del **Topic** al que debe atender el bean. Entonces siempre que se envíe un mensaje a través de este canal se producirá como consecuencia la llamada de cualquier instancia de este bean. Por defecto en JBoss está la Queue **queue/testQueue** y el Topic **topic/textTopic**.

acknowledgeMode

Este dato define cómo el servidor de aplicaciones debe acusar recibo de la recepción y transmisión de un mensaje frente al JMS-Provider, para que este sepa que se ha entregado el mensaje. Son posibles las indicaciones **Auto-acknowledge** y **Dups-ok-acknowledge**. La primera variante determina que el JMS-System debe acusar recibo de la recepción del mensaje inmediatamente después de trasmitirlo a un Message-Driven Bean. en la variante **Dups-ok-acknowledge** el servidor puede disponer de más tiempo. Aquí sin embargo puede ocurrir que el JMS-Provider crea que el mensaje no se ha entregado y lo envié de nuevo. El Message-Driven Bean debe estar por lo tanto concebido para recibir el mismo mensaje repetidamente. Autoacknowledge evita la recepción múltiple de un mensaje.

subscriptionDurability

Si se utiliza un **Topic** como canal de mensajería se debe determinar además si es estrictamente necesario que se entreguen todos los mensajes. Como opciones están disponibles **Durable** y **NonDurable**. En el caso de **Durable** el JMS-Provider tiene que guardar todos los mensajes que no puedan ser ya entregados, lo que realiza normalmente en el caso de un **Queue**. Si el servidor de aplicaciones no funciona en ese momento o tiene algún problema para recoger los mensajes, estos no se perderán. Una vez se pone en contacto de nuevo con el JMS-Provider recibe todos los mensajes. Si los mensajes no fueran muy importantes se selecciona **NonDurable**, lo que tiene como consecuencia que lo mensajes que no puedan ser entregados se pierdan realmente.

messageSelector

De manera opcional un bean puede determinar que no está interesado en todos los beans que circulan por el canal en el que se ha registrado. Con un MessageSelector se formula una condición que el mensaje debe cumplir para ser entregado.

```
@ActivationConfigProperty(
propertyName=”messageSelector”,
propertyValue=”MessageFormat = ‘ChatMessage’”)
```

Código 2.59: Ejemplo de MessageSelector

En el Código 2.59 la propiedad **MessageFormat** debe tener el contenido **ChatMessage** para que el bean reaccione. La condición propuesta debe ser exactamente igual de compleja que la condición **WHERE** de un SQL Statement. También las condiciones compuestas son posibles sin ningún tipo de problema. De la misma manera no existe tampoco ninguna prescripción o limitación en relación a las propiedades utilizadas. Simplemente deben ser configuradas por el emisor del mensaje mediante un método como **setStringProperty()**. Como alternativa podrían utilizarse los métodos setBooleanProperty(), setByteProperty(), setDoubleProperty(), setFloatProperty(),setIntProperty(), setLongProperty(), setObjectProperty() o setShortProperty.

2.17.2.4.2.2. Interfaz MessageListener

Un Message-Driven Bean no tiene una interfaz ni remota en la que se encuentren métodos business. No obstante el servidor de aplicaciones debe saber a través de que método entra el mensaje. Por esta razón un bean dirigido por mensajes implementa por ejemplo una interfaz **javax.jms.MessageListener**, reproducida en el Código 2.60. De aquí se deduce que la clase bean debe disponer obligatoriamente de un método con el nombre **onMessage()**. Para expresarlo más claramente: este método no será nunca llamado directamente por un cliente. Más bien este envía un mensaje a un JMS-Provider en el que también se ha dado de alta el servidor de aplicaciones. El proveedor transmite el mensaje al servidor y este a su vez a una instancia de bean que se encarga del mensaje enviado.

```
package javax.jms;
public interface MessageListener
{
public void onMessage( Message message );
}
```

Código 2.60: Interfaz MessageListener

2.17.2.4.2.3. Constructor estándar

Cada Message-Driven Bean debe disponer de un constructor estándar. Como en los Stateless Session Beans, es el servidor de aplicaciones el que crea la instancia, por eso

aquí no es posible ningún constructor especializado. Este tema ya se ha tratado con los Stateless Beans.

2.17.2.4.3. Metodología de trabajo

Como ya se ha ilustrado, el procesamiento de una demanda mediante un Message-Driven Bean siempre es asíncrono. También se dice que se produce un sistema paralelo. El emisor envía uno o más mensajes y con ello finaliza su trabajo. Cuando y por quién se procesará estos mensajes carece de importancia para el emisor.

La transacción en la que se ejecuta un cliente no tiene nada que ver con la transacción en la que se desarrolla el Message-Driven Bean. Si una de las transacciones fracasara, no afectaría a la otra. También todo el tema de la seguridad debe considerarse aparte. Un bean dirigido por mensajes no sabe normalmente absolutamente nada sobre la identificación de usuario o similar mediante propiedades extra. Esta es la única posibilidad para un bean de comprobar si el emisor del mensaje está autorizado a realizar esta demanda.

Los Session Stateless Beans tienen por lo menos cierto contacto con el cliente durante la llamada de un método business, los Message-Driven Bean por el contrario no. Por eso aún tiene menos sentido querer guardar cualquier tipo de información en los atributos de un bean de esta clase para la siguiente llamada. Si fuera necesario algo así se debería recurrir a un Stateful Session Bean.

Como ya se ha dicho JSM soporta dos tipos de sistemas de mensajería: Point-to-Point (P_2P) y Publish-and-Suscribe (pub/sub). La diferencia principal es de nuevo que en un sistema pub/sub el mensaje se transmite inmediatamente a varios receptores mientras que en P_2P solo recibe el mensaje un receptor.

Publish-and-Suscribe (Publicador y Suscriptor)

En un sistema de este tipo un cliente envía un mensaje a varios receptores con ayuda de un canal de mensajería denominado **Topic**. En estos canales pueden estar registrados varios emisores y varios receptores a la vez. Cada receptor recibe automáticamente todos los mensajes enviados sin tener que preguntar permanentemente al JMS-Provider si hay nuevos mensajes.

Un receptor determina si establece una conexión con el JMS-Provider **Durable** o **NonDurable**. En el caso de Durable debe cerrar sesión temporalmente y al volver a iniciar sesión recibirá todos los mensajes que se han producido mientras tanto. En un Message-Driven Bean es el servidor de aplicaciones el que inicia sesión en el JMS-Provider. Si se selecciona **NonDurable** como tipo de conexión, el receptor no sabrá nada acerca de los mensajes que se han perdido.

Point-to-Point (Punto a Punto)

El canal de mensajes de una conexión P2P se llama **Queue**. También aquí pueden estas registrados varios emisores y receptores a la vez. Normalmente los receptores consultan constantemente al JMS-Provider si existen mensajes para ellos. El primer receptor que lo hace recibe el mensaje, constando este como entregado. El siguiente receptor espera al siguiente mensaje. Si en el momento no hay ningún receptor registrado, JMS-Provider guarda todos los mensajes hasta que los puede entregar.

2.17.2.4.4. Contexto dirigido por mensajes

Como los Session Beans, un Message-Driven Bean también forma parte de un contenedor EJB y por lo tanto tiene la posibilidad de acceder a este mediante una instancia de la clase **javax.ejb.MessageDrivenContext**, derivada de un **EJBContext**. A través de una anotación @Resource se puede inyectar la referencia a la clase bean, como se representa en el Código 2.61.

```
import javax.annotation.Resource;
import javax.ejb.*;
import javax.jms.*;

@MessageDriven(…)
public class ChatBean implements MessageListener
{
@Resource
MessageDrivenContext context;
public void onMessage( Message message );
}
```

Código 2.61: Acceso al Contexto

```
package javax.ejb;

import java.security.*;
import java.util.Properties;
import javax.ejb.*;
import javax.transaction.UserTransaction;

public interface EJBContext
{
public Identity getCallerIdentity();
public Principal getCallerPrincipal();
public EJBHome getEJBHome();
public EJBLocalHome getEJBLocalHome();
public Properties getEnvironment();
public Boolean getRollbackonly() throws IllegalStateException;
public TimerService getTimerService() throws IllegalStateException;
public UserTransaction getUserTransaction() throws IllegalStateException;
public boolean isCallerInRole(Identity role);
public boolean isCallerInRole(String roleName);
public Object lookup(String name);
public void setRollbackonly() throws IllegalStateException;
}
```

Código 2.62: Interfaz EJBContext

Los métodos que la clase MessageDrivenContext herada del EJBContext se representan en el Código 2.62. No obstante no todos estos métodos pueden llamarse sin problemas.

Así los métodos **getEJBHome()** y **getEJBLocalHome()** ejecutan una RuntimeException porque un Message-Driven Bean no dispone de ninguna interfaz home. También los métodos **getCallerPrincipal()** e **isCallerInRole()** conducen a una **RuntimeException**, pues aquí no hay ninguna conexión con un cliente. Con una llamada asíncrona no hay ningún Securitycontext. El método **getEnvironment()** está obsoleto y no debería utilizarse. Por el contrario el acceso a un TimerService es del todo recomendable porque así los Message-Driven Bean puden registrase a la vez también como Timer Beans.

2.17.2.4.5. Ciclo de vida de un Message-Driven Bean

Un Message-Driven Bean pasa durante su procesamiento por diferentes estados. En principio se pueden comparar con los de un Stateless Session Bean, también aquí están disponibles los estados **Does Not Exists** y **Method-Ready**. Si el servidor de aplicaciones no necesita por el momento ninguna instancia de un Message-Driven Bean, la coloca en el Method-Ready Pool, por lo que entonces esta entrará en el estado con el mismo nombre. El tamaño de este pool depende del creador y se puede reconfigurar a menudo. En la Figura XVIII se representan los posibles estados.

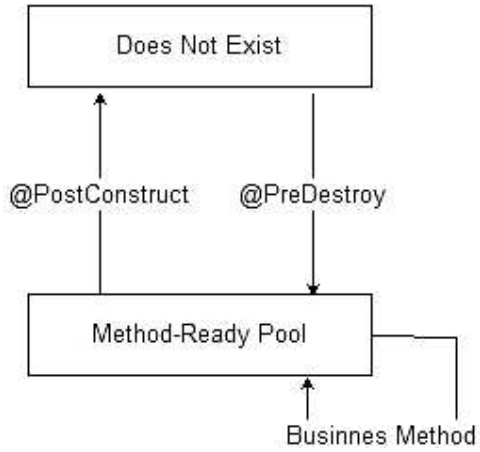


Figura XVIII: Ciclo de vida de un Message-Driven Bean

2.17.2.4.5.1. Does not exist

Este estado declara que aún no existe ninguna instancia del bean. Si un bean pasa del estado Method-Ready Pool a este estado significará que la instancia debe ser liberada por Java.

PostContruct

Cuando el servidor quiere crear la instancia de una Message-Driven Bean, lo hace en tres pasos. Primero utiliza el constructor estándar para crear técnicamente la instancia. Después complace todas las anotaciones de la clase y les inyecta todos los recursos deseados. Si la clase bean necesitara un método con la anotación **@PostConstruct**, la llamaría. El método no debería esperar ningún parámetro ni lanzar ninguna excepción.

```
@MessageDriven(…)
public class ChatBean implements MessageListener
{
@PostContruct
public void init()
{…}
}
```

Código 2.63: Ejemplo de anotación @PostContruct

```
<ejb-jar>
<enterprise-beans>
<message-driven>
<ejb-name>ChatBean</ejb-name>
<post-construct>
<lifecycle-callback-method>
init
</lifecycle-callback-method>
</post-construct>
</message-driven>
</enterprise-beans>
</ejb-jar>
```

Código 2.64: Definicion XML de un método Post-Construct

En el Código 2.63 se representa cómo se define un metod Post-Contruct con auda de una anotación en la clase bean. La definición alternativa dentro del Deployment Descriptor se encuentra en el Código 2.64.

2.17.2.4.5.2. Method-Ready Pool

Todos los beans en este estado no son por el momento necesarios. Si el servidor ha creado más instancias de las que necesita actualmente las coloca en este pool. Incluso si ha creado más instancias de las que quiere guardar simplemente las libera, las cambia por lo tanto al estado **Does Not Exist**. Cuándo y con qué instancia de bean trata no se puede predecir.

Método Business

Cuando llega un mensaje para el que se ha registrado un Message-Driven Bean el servidor toma una instancia correspondiente del Method-Ready Pool y transmite el mensaje. Esto conduce a la llamada del método **onMessage()**. Un bean se da de alta para un mensaje determinado, puesto que dice si este reacciona a un queue o a un topic y qué nombre debe

tener uno u otro. Además también puede delimitar mediante un MessageSelector qué mensaje quiere recibir. Si un Message-Driven Bean reacciona ante un Topic entonces se diría que un mensaje en cuestión en este canal se entregaría a todos los emisores. Esto no significa sin embargo que si el servidor tiene por ejemplo diez instancias del correspondiente tipo de bean en el pool, tenga que llamaralas a todas. Más bien es el servidor de aplicaciones el receptor del Topic y transmite cada mensaje a uno de sus beans.

PreDestroy

Antes de que el servidor de aplicaciones libere una instancia de bean traspasándolo así al estado Does Not Exist, llama primero un método con la anotación @PreDestroy. No está prescrito que cada bean dirigido por mensajes deba disponer de un método de este tipo, pero si es así entonces no debe esperar ningun paramtero ni causar ninguna excepción. Tampoco devuelve nada. Su única tarea consiste en liberar eventualmente determinados recursos. En el Código 2.65 se puede ver cómo se define un método Pre-Destroy mediante la anotación. La definición alternativa en el Deployment Descriptor se observa en el Código 2.66.

```
@MessageDriven(…)
public class ChatBean implements MessageListener
{
@PreDestroy
public void clean()
{
….
}
}
```

Código 2.65: Ejemplo de anotación @PreDestroy

```
<ejb-jar>
<enterprise-beans>
<message-driven>
<ejb-name>ChatBean</ejb-name>
<pre-destroy>
<lifecycle-callback-method>
clean
</lifecycle-callback-method>
</pre-destroy>
</message-driven>
</enterprise-beans>
</ejb-jar>
```

Código 2.66: Definicion XML de un método Pre-Destroy

2.17.2.4.6. XML Deployment Descriptor

Con cada tipo de bean se debe mostrar cómo definir de forma alternativa mediante el Deployment Descriptor. Este también entra en acción con los Message-Driven Beans cuando se quiere prescindir de las anotaciones o se tienen que sobrescribir las

indicaciones. En el Código 2.67 se encuentra la definición para el ChatBean del que se ha hablado hasta ahora en esta sección, que reacciona a un Queue con el nombre queue/testQueue y que consta de un MessageSelector.

```
<ejb-jar
xmlns=”http:///java.sun.com/xml/ns/javaee”
xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation=”http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd” versión=”3.1”>
<enterprise-beans>
<message-driven>
<ejb-name>ChatBean</ejb-name>
<ejb-class>server.md.ChatBean</ejb-class>
<messaging-type>
javax.jms.MessageListener
</messaging-type>
<message-destination-type>javax.jms.Queue
</message-destination-type>
<activation-config>
<activation-config-property>
<activation-config-property-name>
destination
</activation-config-property-name>
<activation-config-property-value>
queue/testQueue
</activation-config-property-value>
</activation-config-property>
<activation-config-property>
<activation-config-property-name>
messageSelector
</activation-config-property-name>
<activation-config-property-value>
MessageFormar = ‘ChatMessage’
</activation-config-property-value>
</activation-config-property>
<activation-config-property>
<activation-config-property-name>
acknowledgeMode
</activation-config-property-name>
<activation-config-property-value>
Auto-acknowledge
</activation-config-property-value>
</activation-config-property>
</activation-config>
</message-driven>
</enterprise-beans>
</ejb-jar>
```

Código 2.67: Deployment Descriptor

2.17.2.4.7. Enviar mensajes desde un cliente

Para que un cliente pueda enviar mensajes a un proveedor JMS antes debe establecer contacto con él. Además debe decir en qué **Queue** o en qué **Topic** quiere enviar mensajes.

Como el servidor de aplicaciones ha configurado el correspondiente canal de mensajería es necesario para el cliente dirigirse primero al servidor. Para ello crea una instancia de la

clase **javax.Naming.InitialContext** para la que son necesarias las propiedades usuales. Aquellas para JBoss se reproducen en el Código 2.68 de nuevo.

```
-Djava.naming.factory.initial = org.jnp.interfaces.NamingContextFactory
-Djava.naming.factory.url.pkgs = org.jboss.naming:org.jnp.interfaces
-Djava.naming.provider.url = jnp://localhost:1099
```

Código 2.68: Entradas de conexión para JBoss

Para establecer contacto con el proveedor JMS el cliente necesita una instancia de la clase **javax.jms.ConnectionFactory**. Esta se consigue a través del correpondiente lookup de un servidor de aplicaciones. El nombre JNDI para este lookup se llama en JBoss simplemente "ConnectionFactory", en otros servidores de aplicaciones no tiene porqué ser así. Además el cliente necesita una instancia del **Queue** deseado (**javax.jms.Queue**) o del **Topic** (**javax.jms.Topic**). Este también se puede conseguir mediante un lookup del servidor de aplicaciones. En el Código 2.69 se comunica el **Queue** a través de **queue/testQueue**.

```
InitialContext ctx = new InitialContext();
ConnectionFactory Factory = (ConnectionFactory) ctx.lookup("ConnectionFactory");
Javax.jms.Queue queue = (javax.jms.Queue) ctx.lookup("queue/testQueue");
```

Código 2.69: Acceso al ConnectionFactory y Queue

Finalmente establece una conexión llamando al método **CreateConnection()** en el **Factory**. Devuelve una instancia de la clase **javax.jms.Connection**. Si la llamada se ha efectuado correctamente, ya se habrá establecido contacto con el proveedor JMS. El siguiente paso consiste en crear una sesión a través de la cual el cliente pueda organizar el envie o la recepción de mensajes. Si un cliente quiere hacer ambas cosas, debe abrir dos sesiones, una que envie y otra que reciba. El método **CreateSession(booleantransacted, int acknowledgeMode)** espera recibir dos parámetros. Con el primero se determina si debe protegerse la transacción al enviar o recibir, el segundo parámetro influye sobre el comportamiento de la respuesta, como ya se ha descrito. Según la especifiacion EJB estos dos parámetros sin embargo se ignorarían, pues el servidor de aplicaciones determina ambos. El resultado del método **CreateSession()** es una instancia de la clase **javax.jms.Session**. Con esta instancia se configura o un emisor de mensajes o un receptor de mensajes. Si se quiere enviar mensajes, o sea crearlos, se llama el método **CreateProducer()** en la sesión. Se recibe una instancia de **javax.jms.MessageProducer**, con la que se pueden enviar tantos mensajes como se desee.

```
Connection connect = factory.createConnection();
Sesión sesión = connect.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer emisor = session.createProducer(queue);
```

Código 2.70: Crear desde MessageProducer

Para enviar un mensaje se crea con ayuda de la sesión una instancia de la clase de mensaje deseada y se transmite al **MessageProducer**. Como clases de mensajes están a disposición por ejemplo **TextMessage**, **MapMessage**, **ByteMessage** o **StringMessage**. Un **TextMessage** es un simple mensaje de texto. Mediante un **MapMessage** es posible, igual que con un **HashMap**, transmitir diferentes informaciones con ayuda de valores clave. Un **ObjectMessage** trasnporta instancias Java si se pueden serializar.

A cada mensaje se le puede asignar unas propiedades. Si por ejemplo se utilizara un MessageSelector, sería necesario configurar las propiedades requeridads por este.

```
TextMessage msg = sesión.createTextMessage();
msg.setText("Mensaje");
msg.setStringProperty("MessageFormat", "ChatMessage");
emisor.send(msg);
```

Código 2.71: Envio de un mensaje

Si el cliente no quisiera enviar más mensajes, debería detener la conexión. Para ello llama el método **close()** en **Connection**.

2.17.2.4.8. Recibir mensajes con un cliente

A parte de los Message-Driven Bean, también los clasicon clientes pueden recibir mensajes. Para ello deben direigrse al servidor de aplicaciones, igual que al enviar, para tener acceso al **ConnectionFactory** y recibir el **Queue** o el **Topic**. Todo lo que es necesario para ello ya se ha descrito. En el Código 2.72 se representa cómo se puede crear una sesión que trabaje con un **Topic**.

```
InitialContext ctx = new InitialContext();
ConnectionFactory factory = (ConnectionFactory) ctx.lookup("ConnectionFactory");
Javax.jms.Topic topic = (javax.jms.Topic) ctx.lookup(topic/testTopic);
Connection connect = factory.createConnection();
Session session = connect.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Código 2.72: Creacion de una sesión para un Topic

Para poder recibir mensajes, se tiene que instanciar un receptor de mensajes dede la sesión con ayuda del método **createConsumer()**. El método espera como parámetro la referencia al Queue o al topic básico. El resultado es una instancia de **javax.jms.MessageConsumer**, al que se le tiene que transmitir la instancia del receptor de mensajes mismo. Se trata aquí de una clase que implementa la interfaz **javax.jms.MessageListener**, la misma interfaz que entra en acción en los Message-Driven Beans. En el Código 2.73 se reproduce una clase receptora, que se registra en su constructor en JMS. Para iniciar la recepción de mensajes llama el metod **start()** en la **Connection**.

```
class Receptor implements javax.jms.MessageListener
{
public Receptor()
{
```

```
try
{
InitialContext ctx = new InitialContext();
ConnectionFactory factory = (ConnectionFactory)
ctx.lookup("ConnectionFactory");
Javax.jms.Topic topic = (javax.jms.Topic)
ctx.lookup(topic/testTopic);

Connection connect = factory.createConnection();
Session session = connect.createSession(false, Session.AUTO_ACKNOWLEDGE);

messageConsumer receiver = session.createConsumer (topic);
receiver.setMessageListener(this);
connect.start();
}
catch( Exception ex)
{
ex.printStackTrace();
}
}

public void onMessage(Message message)
{
}
}
```

Código 2.73: Receptor JMS clásico.

2.17.2.4.9. Enviar mensajes desde un Bean

Ya se ha descrito anteriormente como programas Message-Driven Beans para que reciban mensajes. Sin embargo cada bean está en situación de enviar mensajes él mismo. Para ello necesita también tener acceso al **ConnectionFactory** y al **Queue** o **Topic**, estos sin embargo pueden inyectarse fácilmente con anotaciones. El bean crea entonces una **Connection**, una **Session**, un **MessageProducer** y el mensaje en sí mismo, lo envía con ayuda del **Producer** y cierra de nuevo la **Connection**.

En el Código 2.74 se reproduce la clase ChatBean al completo, que espera en el **Queuequeue/testQueue** a los mensajes y los escribe después en el **Topictopic/testTopic**. Los mensajes pueden ser enviados del mismo modo también por Stateless Session Beans o por Stateful Session Beans. Para la recepción tan solo sirven los Message-Driven Beans.

```
package server.md;

import javax.annotation.Resource;
import javax.ejb.*;
import javax.jms.*;

@MessageDriven(activationConfig={
@ActivationConfigProperty(
propertyName="destinationType",
propertyValue="javax.jms.Queue"),
@ActivationConfigProperty(
propertyName="destination",
propertyValue="queue/testQueue"),
@ActivationConfigProperty(
```

```
propertyName=”messageSelector”,
propertyValue=”MessageFormat = ‘ChatMessage’”),
@ActivationConfigProperty(
propertyName=”acknowledgeMode”,
propertyValue=”Auto-acknowledge”)})

public class ChatBean implements MessageListener
{
@Resource( mappedName = ”ConnectionFactory” )
ConnectionFactory factory;

@Resource( mappedName = ‘‘topic/testTopic”)
Topic topic;

public void onMessage( Message message )
{
try
{
TextMessage txtMsg = (TextMessage) message;
String text = txtMsg.getText();
System.out.println( text );
if( factory != null && topic != null )
{
Connection connect = factory.createConnection();
Session session = connect.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer emisor = session.createProducer(topic);
TextMessage msg = session.createTextMessage();
msg.setText(text);
emisor.send(msg);
connect.close();
}
else
{
System.out.println(“No se encontró Factory o Topic”);
}
}
catch( JMSException ex )
{
ex.printStackTrace();
}
}
}
```

Código 2.74: Enviar mensajes con un Bean.

También se puede prescindir del todo de las anotaciones. En el Código 2.75 se reproduce el fragmento del Deployment Descriptor que se ocupa de que se inyecten los recursos factory y topic. El resto del Descriptor se ha reproducido ya en el Código 2.67.

```
<ejb-jar
xmlns=”http:///java.sun.com/xml/ns/javaee”
xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation=”http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd” versión=”3.1”>
<enterprise-beans>
<message-driven>
<ejb-name>ChatBean</ejb-name>
…
<resource-ref>
<res-ref-name>
ConnectionFactory
</res-ref-name>
<res-type>
javax.jms.ConnectionFactory
</res-type>
<mapped-name>
```

```
ConnectionFactory
</mapped-name>
<injection-target>
<injection-target-class>
server.md.ChatBean
</injection-target-class>
<injection-target-name>
factory
</injection-target-name>
</injection-target>
</resource-ref>
<resource-ref>
<res-ref-name>
Topic
</res-ref-name>
<res-type>
javax.jms.Topic
</res-type>
<mapped-name>
topic/testTopic
</mapped-name>
<injection-target>
<injection-target-class>
server.md.ChatBean
</injection-target-class>
<injection-target-name>
topic
</injection-target-name>
</injection-target>
</resource-ref>
</message-driven>
</enterprise-beans>
</ejb-jar>
```

Código 2.75: Deployment Descriptor de Message-Driven Bean

2.18. Timer Service

2.18.1.1.1. Uso

El Timer Service siempres entra en accion cuando se desea un proceso controlado temporalmente. En este caso se puede tratar respecto a la demanda, de la llamada de un método en un día y hora definidos con precisión o de llamada simplemente cada diez segundos.

La llamada del método puede sucederse tan solo una vez o cada cierto intervalo de tiempo. En cada caso sin tener ningun tipo de relación con el cliente. Tan solo el primer impulso debe ocurrir mediante la llamada explicita de un método business. Es prácticamente el mismo servidor de aplicaiones el que transmite en un determinado momento el control a la aplicación, que se reparte entre el resto de beans y timers.

Para implementar un Timer Service se utiliza normalmente un Stateless Session Bean o un Message-Driven Bean. Ambos posibilitan al servidor de aplicaciones poder tomar del método Ready-Pool cualquier instancia de bean para llamar el Timer-Methode si es necesario. En EJB2.1 las Entity Beans también pueden servir como Timer, siendo su tarea la de representar a los datos persistentes en la memoria principal y no agotar la lógica de aplicaciones. Los EJB 3.0 Entities ya no están disponibles como Timer. Los Stateful Session Beans tampoco pueden configurarse como Timer Service. Si se intentara registrar

un bean de este tipo como Timer se produciría una IllegalStateException. Esta es una limitación bastante importante. No esta permitido intervenir desde fuera en una conversación entre un bean con estado y su cliente.

Una tarea para un Timer Service podría ser por ejemplo el monitoraje de la aplicación. Quizá se quiera saber cada segundo cuánta memoria principal del servidor hay aún disponible, o cuántos bytes tiene reservados. También se puede emitir otro tipo de información específica sobre la aplicación, si esta la recoge. Un cliente en cuestión inicia el monitoraje estableciendo un Timer que cada cierto intervalo de tiempo o bien escribe información en un logfile o envía mensajes a trves de un Queue o Topic que el cliente recibe. En la sección sobre los Message-Driven Beans ya se ha descrito cómo enviar este tipo de mensajes desde un bean.

Alternativamente se podría usar un Timer Service para borrar líneas de un table de base de datos cada cierto periodo de tiempo. Todas las celdas caracterizadas con antigüedad superior a una semana podrían eliminarse automáticamente mediante el Service correspondiente. Este tipo de tablas se encuentran en casi todas las aplicaiones.

Sin embargo, no se debería confiar en qué el Service siempre empiece con puntualidad. Si se ha escogido un intervalo de un segundo y en ese momento hay mucha actividad en el servidor, entonces podría suceder que no ocurra nada durante dos o tres segundos y después se llame al servidor deo o tres veces seguidas.

Para un mismo Bean se pueden iniciar Timers que se ejecutan prácticamente de manera simultánea en diferenctes instancias de bean. Todos llaman siempre el mismo método. A cada Timer se le pueden otorgar todo tipo de informaciones en forma de objetos serializables, a través de las que se identifica cada servicio. Así se puede comprobar dentro de cada bean por qué Timer ha sido llamado.

Según la especificación los Timers son objetos persistentes, que deberían sobrevivir a una caída del servidor de aplicaciones. Si se reinicia el servido, este debe atender a todos los Timers ejecutados. Los Timers con un intervalo solo se llaman una vez u siempre una vez ya ha transcurrido el espacio de tiempo correspondiente.

Con EJB 3.1 la situación cambia para mejor. Hay dos formas de crear timers:

- **Programáticamente**, usando la interfaz TimerService ya existente. La interfaz fue mejorado desde la versión 2.1 para brindar más flexibilidad al crear timers.

- **Declarativamente**, usando anotaciones o en el descriptor de despliegue. De esta manera se puede definir un timer de manera estática para que sea creado automáticamente durante el inicio de la aplicación.

2.18.1.1.2. Estructura

2.18.1.1.2.1. Interfaz TimedObject

Para poder utilizar el Timer Service, un Stateless Session Bean o un Message-Driven Bean implementa la interfaz **javax.ejb.TimerObject**, representada en el Código 2.76. Consta tan solo del método **ejbTimeout()**, al que se le transmite una instancia de Timer. Siempre cuando se llega al momento justo, se llama exactamente este método y se transmite el Timer actual, que casi ha provocado la llamada.

```
package javax.ejb;

public interface TimedObject
{
public void ejbTimeout( Timer timer );
}
```

Código 2.76: Interfaz TimedObject

Aun existe la posibilidad de alternativoa, que es utilizar un bean como TimerService. En lugar de implementar la interfaz TimedObject, se dota a cualquier método de la anotación **@javax.ejb.Timeout**.
Este método debe esperar como parámetro una instancia de **javax.ejb.Timer**, no debe tener ningun valor de vuelta y no debe provocar excepción. Puede tener cualquier nombre. En el Código 2.77 se representa un Stateless Session Bean que dispone de un método de este tipo.

```
import javax.ejb.*;

@Stateless
public class MonitorBean implements MonitorRemote
{
@Timeout
public void timeout( Timer timer )
{
}
}
```

Código 2.77: Bean con anotacion @Timeout

Si el Timer se ha fijado con un interval muy corto, un milisegundo es el interval de tiempo mas corto possible, y el metodo llamado necesita más tiempo para llevar a cabo su trabajo, el Timer se llamará de nuevo lo más pronto posible después de la finalización del método. Esta es también una de las razones por las que el Timer Service no siempre es puntual.

2.18.1.1.2.2. Interfaz TimerService

Un bean se registra a sí mismo como Timer Service al llamar uno de los cuatro métodos **create** existentes en una instancia de la clase **javax.ejb**.TimerService. Más adelante se especificará el significado de cada método al describir el Método de Trabajo. Un bean tiene acceso a la instancia del TimerService o bien a través del método **getTimerService()** de la

clase EJBContext o bien permitieno la inyección de la referencia correspondiente con ayuda de la anotación @**Resource**. La interfaz se reproduce en el Código 2.78.

```
import java.io.Serializable;
import java.util;
import javax.ejb.*;

public interface TimerService
{
public Timer CreateTimer( long duration, Serializable info)
throws IllegalArgumentException, IllegalStateException, EJBException;

public Timer createTimer(Date expiration, Serializable inf)
throws IllegalArgumentException, IllegalStateException, EJBException;

public Timer createTimer(long initialDuration, long intervalDuration, Serializable)
throws IllegalArgumentException, IllegalStateException, EJBException

public Timer createTimer(Date initialExpiration,long intervalDuration, Serializable info)
throws IllegalArgumentException, IllegalStateException, EJBException;

public collection getTimer()
throws IllegalStateException,EJBException;

}
```

Código 2.78: Interfaz TimerService

Con el método **getTimers()** un bean obtiene acceso a todos los Timer activos conectados con sus clases bean. Así está en situación de parar cada Timer o de conocer el momento en el que el Service debe ser ejectuado de nuevo.

IllegalArgumentException

Si se transmite un argumento no valido a uno de los métodos create, este produce una IllegalArgumentException. Así por ejemplo está prohibido transmitir como intervalo un valor negativo. También el intento de introducir cero en lugar de un dato de referencia valido provocará este error.

IllegarlStateException

Esta excepción significa que un método TimerService ha sido llamado desde un sitio desde donde no esta permitido. El mejor ejemplo de este es el intento de configurar un bean de sesión sin estado como Timer Service. No esta permitido llamarlos desde uno de los métodos **setSessionContext()** o **setMessageDrivenContext()**.

EJBException

Esta excepción indica un fallo general del sistema.

2.18.1.1.2.3. Interfaz Timer

Al método Timeout llamado se le devuelve una instancia de la clase Timer que representa el Timer Service subyacente. Mediante esta instancia es posible detener el Timer o saber cuando se llevará a cabo de nuevo, es decir hasta ese momento cuanto tiempo durará. Además también se puede preguntar información general al Timer.

La interfaz se representa en el Código 2.79.

```
import java.io.Serializable;
import java.util.Date;
import javax.ejb.*;

public interface Timer
{
public void cancel() throws IllegalStateException, NoSuchObjectLocalException,
EJBException;

public TimerHandle getHandle() throws IllegalStateException, NoSuchObjectLocalException,
EJBException;

public Serializable getInfo() throws IllegalStateException, NoSuchObjectLocalException,
EJBException;

public long getTimedRemaining() throws IllegalStateException, NoSuchObjectLocalException,
EJBException;
}
```

Código 2.79: Interfaz Timer

Para detener un Timer se utilize el método **cancel()**. El método **getNextTimeout()** da la fecha y la hora en que se iniciará de nuevo el Timer. Por el contrario **getTimeRemaining()** proporciona cuantos milisegundos quedan hasta ese momento. Con **getInfo()** se puede recoger el objeto que se asocia al Timer al crearlo. Con ayuda del método **getHandle()** se puede obtener un TimerHandle para el Timer correspondiente, que se puede guardar en un archivo o en una base de datos para acceder a él posteriormente. El Handle naturalmente solo es útil mientras el Timer sea activo. Si se da el caso que otro Bean necesite acceder al Timer entonces puede leer el TimerHandle desde la base de datos y llamar métodos Timer.

NoSuchObjectLocalException

Esta excepción se produce cuando se llama un método en un Timer que no posee ningun intervalo y ya ha expirado o bien, cuando el Timer se ha detenido.

EJBException

Esta excepción indica un fallo general del sistema.

2.18.1.1.3. Metodología de trabajo

2.18.1.1.3.1. Diferentes tipos de Timers

Se pueden diferenciar dos tipos de Timer. Los llamados temporizadores de una sola acción (single-action timer) llaman una sola vez el método Timeout y entonces expiran. Un temporizador de intervalo ejecuta el método cada cierto intervalo de tiempo, una y otra vez hasta que este es detenido. Para crear un temporizador se le da al método **createTimer()** como primer parámetro o una instancia de la clase **Date** o una variable del tipo **long**. La instancia **Date** determina la fecha y la hora en la que se inicia por primera vez y eventualmente también por ultima vez el temporizador. Si en lugar de eso se le otorga el valor **long**, este indicará cuantos milisegundos faltan para que se inicie.

Como los dos tipos de temporizadores pueden crearse también de dos modos diferentes, se dispone en total de cuatro métodos **createTimer()** que ya se han mostrado en el Código 2.78. Si se necesita crear un temporizador de intervalo el segundo parámetro será el valor long, que precisa la duración del intervalo en milisegundos. El ultimo parámetro de todos los métodos **createTimer()** puede ser cualquier objeto serializable. Posteriormente se puede identificar un temporizador especial sobre este objeto. En el Código 2.80 se reproducen ejemplos de la creación de ambos tipos de temporizadores.

```
GregorianCalendar calender = new GregorianCalendar();
calender.add(GregorianCalendar.DATE, 1);
Date manana = calendar.getTime();

// Temporizador de intervalo para mañana
timerService.createTimer(mañana, “info”);
// Temporizador de una sola acción con inicio en 10 segundos.
timerService.createTimer(10*1000, “info”);
// Temporizador de intervalo a partir de mañana cada 10 segundos
timerService.createTimer(mañana, 10*1000, “info”);
//Temporizador de intervalo con inicio en un minuto y después cada 10 seg.
timerService.createTimer(60*1000, 10*1000, 1000, “info”);
```

Código 2.80: Creacion de diferentes temporizadores

2.18.1.1.3.2. Inicio del temporizador

Como ya se ha descrito, los temporizadores se pueden originar desde un Stateless Session Bean o un Message-Driven Bean. Para ello en el caso de un bean sin estado se implementa cualquier método business, que llama uno de los métodos **createTimer()**, configurándose de este modo a sí mimo como temporizador. Aquí es importante que el bean disponga de un método Timeout. Si se utiliza un Message-Driven Bean, la creación del temporizador se lleva a cado mediante el método **onMessage()**, ya que no se dispone de ningún otro. En el Código 2.81 se reproduce un Stateless Session Bean que dispone un temporizador que se inicia en un segundo y después es llamado de forma continua en un intervalo de un segundo cada vez.

```
import javax.annotation.Resource;
import javax.ejb.*;

@Stateless
public class MonitorBean implements MonitorRemote
{
@Resource TimerService timerService;
public static final String COMANDO = “HSP”;
public void startMonitor()
{
timerService.createTimer(1*1000, 1*1000, COMANDO);
}

@Timeout
public void timeout(Timer timer)
{
…
}
}
```

Código 2.81: Iniciar un temporizador

Dentro de un método Timeout se tiene acceso al temporizador a través del cual se llama al método. Se transmite como parámetro. Si se llama en esta referencia el método **cancel()** entonces se detiene el temporizador y no se vuelve a llamar.

Sin embargo también es posible conseguir a todos los temporizadores asociados a la propia clase bean. La clase **TimerService** prevé para ello el método **getTimers()**, que devuelve una Collection de instalacias timer.

Si solo se debe detener un temporizador en concreto, se puede consultar el objeto info otrogado al método **createTimer()**, para describirlo. Diferentes beans pueden registrar uno o más temporizadores. Todos conducen por tipo de bean siempre a la llamada del método timeout. Este tan solo debe saber reconocer de qué tipo especial de temporizador se tiene la posibilidad de otorgar un objeto identificador que después puede ser consultado. Cada bean recibe con la llamada de un **getTimers()** solo los temporizadores relevantes para su tipo de bean. En el Código 2.82 se reproduce un método business de un Stateless Sesión Bean a través del cual se detienen determinados temporizadores.

```
public static final String COMANDO = “HSP”;

public void stopMonitor()
{
for(object obj: timerService.getTimers(0))
{
Timer timer = (Timer) obj;
String rel = (String) timer.getInfo();
If( rel.equals(COMANDO) )
{
timer.cancel();
}
}
}
```

Código 2.82: Detener un temporizador en concreto

2.18.1.1.4. Stateless Session Bean Timer

La ventaja de utilizar como temporizador un Stateless Session Bean reside en que se pueden ofrecer diferentes métodos a través de la interfaz remota para iniciar u también detener diferentes temporizadores de forma individual. En cada caso es necesario un cliente para llamar uno de estos métodos y así poder controlar el temporizador. Para ello basta que un cliente se dé brevemente de alta, inicie el temporizador y después cierre de nuevo sesión. No es necesaria una conexión más duradera con el servidor puesto que la llamada del método **Timeout** ocurre sin ninguna relación con el cliente.

2.18.1.1.5. Message-Driven Bean Timer

A diferencia de un Sateless Session Bean en un Message-Driven Bean siempres se inicia o se detiene el temporizador mediante un mensaje. El cliente envía un mensaje a un queue y un topic, lo que conduce a la llamda del método **onMessage()** en el bean. Este debe analizar el mensaje entrante y reconocer si el timer se inicia o se detiene y qué información especial debe transmitirse al temporizador, con el que se identifica. Por lo demás no hay ninguna diferencia. El Message-Driven Bean también debe o bien implementar la interfaz **TimeObject** o proveer cualquier método con la anotación @**Timeout**.

2.18.1.1.6. Temporizador automático

La anotación **@Schedule** se utiliza para crear un timer de forma automática, y toma como parámetro el timeout correspondiente a la ejecución. Esta anotación se aplica al método que será utilizado como callback del timeout. En el ejemplo del Código 2.83 se definen dos timers, uno que expira cada lunes a la medianoche y el otro que expira el último día de cada mes.

```
@Stateless public class TimerEJB
{
@Schedule(dayOfWeek="Mon")
public void esLunes(Timer timer)
{…}

@Schedule(dayOfMonth="Last")
public void FindeMes(Timer timer)
{...}
}
```

Código 2.83: Ejemplo de Timer

Un método puede estar anotado con más de un timer, como se ve a continuación, en donde se definen dos timers para el método mealTime, uno de los cuales expira todos los días a la 1pm y el otro expira a las 8pm.

```
@Stateless
public class MealEJB
{
@Schedules( { @Schedule(hour="13"), @Schedule(hour="20") }
```

```
public void mealTime(Timer timer)
{
...
}
}
```

Código 2.84: Método con dos Timer

Tanto los timers automáticos como programáticos pueden ser persistentes (predeterminado) o no-persistentes. Los timers no-persistentes no sobreviven a un apagado de la aplicación o una caida del contenenedor. La persistencia puede definirse usando el atributo persistente de la anotación, o pasando la clase TimerConfig como parámetro del método createTimer en la interfaz TimerService.

2.19. Persistencia (JPA)

El Java Persistence API (JPA) es una especificación de Sun Microsystems para la persistencia de objetos Java a cualquier base de datos relacional. Esta API fue desarrollada para la plataforma JEE e incluida en el estándar de EJB 3.0, formando parte de la Java Specification Request JSR 220.

Para su utilización, JPA requiere de J2SE 1.5 (también conocida como Java 5) o superior, ya que hace uso intensivo de las nuevas características del lenguaje Java, como las anotaciones y los genéricos.

2.19.1. Arquitectura

El siguiente diagrama muestra la relación entre los componentes principales de la arquitectura de JPA:

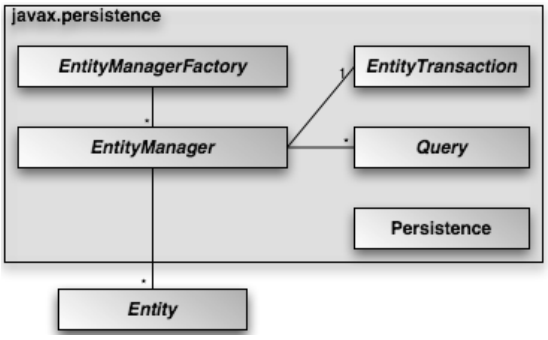


Figura XIX: Arquitectura JPA

Varias de las interfaces de la Figura XIX, son solo necesarias para su utilización fuera de un servidor de aplicaciones que soporte EJB 3, como es el caso del EntityManagerFactory que es ampliamente usado en desarrollo de aplicaciones de escritorio. En un servidor de aplicaciones, una instancia de EntityManager típicamente suele ser inyectada, haciendo así innecesario el uso de un EntityManagerFactory. Por otra parte, las transacciones dentro

de un servidor de aplicaciones se controlan mediante un mecanismo estándar de controles de, por lo tanto la interfaz EntityTransaction también no es utilizada en este ambiente.

- **Persistence:** La clase javax.persistence.Persistence contiene métodos estáticos de ayuda para obtener una instancia de EntityManagerFactory de una forma independiente al vendedor de la implementación de JPA.

- **EntityManagerFactory:** La clase javax.persistence.EntityManagerFactory nos ayuda a crear objetos de EntityManager utilizando el patrón de diseño del Factory (fábrica).

- **EntityManager:** La clase javax.persistence.EntityManager es la interfaz principal de JPA utilizada para la persistencia de las aplicaciones. Cada EntityManager puede realizar operaciones CRUD (Create, Read, Update, Delete) sobre un conjunto de objetos persistentes.

- **Entity:** La clase javax.persistence.Entity es una anotación Java que se coloca a nivel de clases Java serializables y que cada objeto de una de estas clases anotadas representa un registro de una base de datos.

- **EntityTransaction:** Cada instancia de EntityManager tiene una relación de uno a uno con una instancia de javax.persistence.EntityTransaction, permite operaciones sobre datos persistentes de manera que agrupados formen una unidad de trabajo transaccional, en el que todo el grupo sincroniza su estado de persistencia en la base de datos o todos fallan en el intento, en caso de fallo, la base de datos quedará con su estado original. Maneja el concepto de todos o ninguno para mantener la integridad de los datos.

- **Query:** La interface **javax.persistence.Query** está implementada por cada vendedor de JPA para encontrar objetos persistentes manejando cierto criterio de búsqueda. JPA estandariza el soporte para consultas utilizando Java Persistence Query Language (JPQL) y Structured Query Language (SQL). Podemos obtener una instancia de Query desde una instancia de un EntityManager

2.19.2. Entity Manager

2.19.2.1. **Panorama general**

Guardar o modificar los datos que contiene por ejemplo una base de datos es junto a toda la lógica de una aplicación compleja, de gran importancia. La mayor parte de la especificación de Java EE se ocupa de este tema. Anteriormente, en EJB 2.1 la definición de la capa de persistencia formaba parte de la especificación EJB. Desde la versión 3.0 esto es diferente, ya que desde entonces todas las instrucciones relevantes se guardan en una especificación propia con el nombre Java Persistence 1.0. De hecho en la especificación misma encontramos tan solo una referencia a este documento.

En este nuevo concepto, el Entity Manager desempeña un papel muy relevante. Actualmente los Entity Beans son las clases Java normales, enriquecidas con metainformacion, para que el Entity Manager se encuentre en posición de mantener sus atributos sincronizados con la base de datos. Esto ocurre a partir del momento en qué el Entity Bean se transmite al Entity Manager. A partir de ahí se gestionará el Bean, lo que significa que el Entity Manager supervisará cada modificación en los atributos y decidirá a lo largo de la transacción lo que se necesario para mantener la armonía entre estos cambios y el contenido de la base de datos. Todos los Entity Beans y todos los cambios que administra el Entity Manager constituyen el Persistence Context que normalmente, aunque no obligatoriamente, está estrechamente relacionado con una transacción. Si finaliza el Persistence Context, el Entity Manager deja de supervisar los Entity Beans, por lo que estos se convierten de nuevo en objetox Java normales.

El hecho de que exista una especificación propia para la capa de persistencia permite que este concepto también esté disponible desde fuera de un servidor de aplicaciones. La anteriormente estrecha relación con el estándar de Java EE se ha abandonado para poder desarrollar aplicaciones que también funcionan en el entorno clásico Java SE. Allí se tiene que renunciar a todas las ventajas de un servidor y ocuparse uno mismo por ejemplo de la administración de las transacciones y de la capacidad multiusuario, pero se garantiza como contrapartida una arquitectura de aplicación limpia, una aplicación de uso flexible que puede equipararse aun mejor para futuras exigencias.

2.19.2.2. **Persistence Unit (Unidad de Persistencia)**

Un Entity Manager se ocupa de una determinada cantidad de clases de Entity Beans que quiere mantener sincronizada con la base de datos. Esta cantidad se denomina Persistence Unit y representa el marco tecnológico con el que una aplicación puede actuar. Para poder trabajar con un Entity Manager es imprescindible definir el alcance de Persistence Unit. Esto ocurre mediante un Deployment Descriptor con el nombre **persistence.xml**, del que se debe disponer sin falta y que debe guardarse en el directorio **META-INF** de la aplicación. Al hablar de aplicaion nos referimos a un archivo JAR en el que se resumen las clases de la aplicación. Es posible definir varias Unidades de Persistencia dentro de un solo archivo **persistence.xml**, o construir una aplicación que conste de más de un archvio JAR, que describen cada uno sus correspondientes unidades de persistencia.

```
<? xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
version="2.0">
<persistence-unit name="JavaEE" transaction-type="JTA">
<jta-data-source>java:DefaultDS</jta-data-source>
<properties>
<property name="eclipselink.ddl-generation"
value="create-tables"/>
</properties>
</persistence-unit>
</persistence>
```

Código 2.85: persistence.xml

Bajo la etiqueta **<persistence>** se pueden definir a su vez una o más etiquetas **<persistence-unit>**. Cada **<persistence-unit>** tiene dos atributos: el nombre, que es indispensable, y el tipo de transacción **transaction-type**, que puede omitirse. El nombre nos servirá de referencia para la Persistence Unit posteriormente. El tipo de transacción determina si la Persistence Unit se administra a través del administrador de transacciones de un servidor de aplicaciones, se fija entonces el valor del atributo en **JTA** y la recepción estándar es en una aplicaion JavaEE, o bien si la aplicación misma la que se ocupa de administrar las transacciones. En este último caso se expresa mediante el valor de atributo **RESOURCE_LOCAL** y está en el entorno estancar JavaSE.

Si se utiliza el tipo de transacción JTA, debe definir el nombre del DataSource con la etiqueta **<jta-data-source>**. En el caso de **RESOURCE_LOCAL** esto se realiza con la etiqueta **<nom-jta-data-source>**. Si es necesario transmitir información especifica del fabricante, se dispone de la etiqueta opcional **<properties>**, que consta de tantas entradas **<property>** como se necesite.

Junto al Deployment Descriptor **persistence.xml** se puede guardar opcionalmente un archivo mapping con el nombre **orm.xml** en el directorio **META-INF** de la aplicación. Este archivo mapping contiene información acerca de a qué tabla de base de datos pertenece cada Entity Bean y sobre coómo deben representarse sus columnas en los atributos del bean. Como ya se ha dicho este archivo mapping es opcional ya que toda esta información se puede especificar con ayuda de anotaciones y entradas estándar. Si hubiera algún archivo mapping aparte de un **orm.xml**, este se puede dar a conocer en la Persistence Unit mediante la etiqueta **<mapping-file>**.

Todos los Entity Beans que se encuentran en el mismo archivo JAR como definición de la Persistence Unit pertenece a este grupo. Mediante la etiqueta **<jar-file>**, que puede introducirse repetidas veces, se pueden publicar más archivos JAR, los beans de los cuales entonces pertenecerían también al Unit. Si por el contratio solo pertenecen a la propia Unidad de Persistencia clases sueltas de otro archivo JAR, estas se pueden listar con ayuda de las etiquetas **<class>**, que también pueden repetirse a voluntad. Cada clase introducida aquí amplía la lista de los Entity Beans a administrar, que resulta de la suma de los propios beans. Si se quisieran vincular al Persistence Unit tan solo clases extrañas, se tiene que escribir la entrada **<exclude-unlisted-classes/>**.

La cantidad de clases que pertenecen a un Persistence Unit resulta de todos los Entity Beans del archivo JAR en el que se encuentra el archivo **persistence.xml**, a no ser que contenga la entrada **<exclude-unlisted-classe/>**. A estos se les suman todos los Entity Beans de otros archivos JAR nombrados con la etiqueta **<jar-file>**. Por ultimo se amplía el Unit con las clases que se han nombrado explícitamente con la entrada **<class>**. En este caso un Entity Bean es una clase Java que o bien consta de la anotación **@Entity** o bien se ha definido como **<entity>** en eun archivo mapping como por ejemplo **orm.xml**. A los archivos mapping de archivos JAR extraños se debe hacer también referencia mediante la etiqueta **<mapping-file>**.

2.19.2.3. **Persistence Context**

Un Persistence Context es la suma de todos los entity beans administrados por un Entity Manager. Dentro de una aplicación puede haber tantos contextos como sea necesario, que a menudo están estrechamente relacionados con una transacción en curso. Dos usuarios independientes entre si quieren modificar datos en un sistema. Para ello llama con ayuda de los métodos business de sus clientes al servidor, de manera que se lleva a cabo una transacción. Aquí carece totalmente de importancia si se trata ambas veces del mismo método o de diferentes. Para cada usuario se inicia, con la llamada de sus métodos, un Persistence Context diferente. Dependiendo de qué datos se trabajan se administrarán en cada contexto los mismos entity beans o diferentes. Cada usuario introduce datos diferentes, de manera que cada contexto se desarrollará de modo diferente. Al final de la transacción el Entity Manager intentará representar el estado actual de la base de datos, generando instrucciones SQL y realizándolas. Si no lo consiguiera, se producirá el correspondiente mensaje de error. Cada persistence Context concurre con todos los otros.

2.19.2.3.1. Persistence Context orientados a transacciones

Un contexto de este tipo vive tanto tiempo como una transacción y se cierra al finalizar. Esto tienen como consecuencia que antes el Entity Manager debe comparar todos los cambios con la base de datos, puesto que todos los beans administrados por él vuelven a convertirse en objetos Java normales. Lo que esto significa, se mostrará a partir del ejemplo del Código 2.86. En él encontrará un fragmento de un Stateful Session Bean con dos métodos, para los que se ha prescrito que deben disponer de protección en la transacción. La entrada del atributo de transacción **required** es la recepción estándar, pero aquí debe hacerse de nuevo referencia explícita al control de la transacción.

```
@PersistenceContext(unitName = “JavaEE”)
private EntityManager manager;
private Articulo articulo;
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public Articulo getArticulo(int artnr) throws ArticuloException(int artnr)
{
articulo = manager.find(Articulo.class, artnr);
if( articulo == null ) throws new ArticuloException(“Articulo no encontrado”);
articulo.setCantidad(articulo.getCantidad() - 1);
return articulo;
}
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void changeArticulo(Articulo nuevo) throws Articulo Exception
{
articulo.setDes(nuevo.getDes());
articulo.setCantidad(nuevo.getCantidad());
articulo.setPrecio(nuevo.getPrecio());
manager.merge(articulo);
}
```

Código 2.86: Persistence Context orientado a la transacción.

Una vez el usuario llama el método **getArcitulo()** el Entity Manager busca la línea correspondiente en la base de datos y en caso de éxito devuelve una instancia de la clase **Articulo**. Esta instancia se trata ahora de una Entity Bean que será administrado a través de un Entity Manager. En el ejemplo se reduce además la cantidad almacena al valor 1, lo que tendrá sus consecuencias en la base de datos al finalizar la transacción. Como el Entity Bean está bajo control del Entity Manager, este transmite la modificación y continúa escribiendo el Persistence Context. En este ejemplo es importante que la variable **articulo** se haya definido como atributo de la clase de manera que sobreviva al finalizar el método. Después de todo se trata de un Stateful Session Bean. Una vez finalice el método **getArticulo()** finaliza también el Persistence Context y el Entity Bean **articulo** aun administrado se convertirá en una instancia Java totalmente normal.

Los datos del artículo se transfieren al cliente que puede modificar como quiera. En cualquier momento llama al método **changeArticulo()** para guardar los cambios. Dentro de este método se utiliza la instancia de bean **articulo** a la que todavía se hace referencia, para poder recibir los nuevos datos. De momento de un objeto Java normal, que no es supervisado. Al Entity Manager todavía no se le ha comunicado ningún cambio. Solo después de llamar al método **merge()** se transmite el bean al Entity Manager encargándose a su vez que transfiera las modificaciones a la base de datos. Si el método **merge()** del Entity Manager no se utilizara, se producirían igualmente los cambios en los atributos de la instancia local articulo, pero no afectaría a la base de datos.

2.19.2.3.2. Persistence Context ampliado

Es posible definir un Persistence Context que sobreviva al final una transacción. De este modo su duración no depende de una breve transacción sino de un Stateful Session Bean en el que es definido. Durante el tiempo de ejecución puede haber incluso varios contextos ampliados, dependiendo de cuantos usuarios se registren simultáneamente. No solo se les permite a los Stateful Session Beans definir Persistence Context ampliado.

```
@PersistenceContext(unitName = “JavaEE”,  type = “PersistenceContextType.EXTENDED”)
private EntityManager manager;
private Articulo articulo;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public Articulo getArticulo(int artnr) throws ArticuloException(int artnr)
{
articulo = manager.find(Articulo.class, artnr);
if( articulo == null ) throws new ArticuloException(“Articulo no encontrado”);
articulo.setCantidad(articulo.getCantidad() - 1);
return articulo;
}

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void changeArticulo(Articulo nuevo) throws Articulo Exception
{
articulo.setDes(nuevo.getDes());
articulo.setCantidad(nuevo.getCantidad());
articulo.setPrecio(nuevo.getPrecio());
}
```

Código 2.87: Persistence Context ampliado

Para conseguir un Persistence Context se la añade a la anotación **@PersistenceContext** el tipo **PersistenceContextType.EXTENDED**. La llamada del método **getArticulo()** conduce en el ejemplo a que se provea a la referencia local **articulo** de un entity bean controlado por el Entity Manager. La modificación de la cantidad almacenada se sincroniza como es habitual al final de la transacción con la base de datos. Si el usuario llamara después el método **changeArticulo()**, bastaría con modificar los atributos de la instancia articulo, aún administrada por el Entity Manager para seguir también el contenido de la base de datos. La llamada explícita del método **merge()** en el Entity Manger no es necesaria.

2.19.2.3.3. Stateful Session Beans anidados

Se produce una situación peculiar cuando un Stateful Session Bean contiene una referencia a la interfaz local de otro Stateful Sesion Bean y este permite inyectarse mediante la anotación **@EJB**. Una vez se crea el Session Bean externo también se aplica el bean referido y se vincula con el externo. La duración de los beans a los que se hace referencia depende de los referentes. Si se elimina también el externo con **remove()**, también se libera el interno.
Si ambos Stateful Session Bean definen el mismo Persistence Context ampliado se crea un contexto común, que comparte ambos beans. Si ambos beans hacen referencia a un mismo artículo con un atributo local, se refieren a un Entity Bean idéntico y carece totalmente de importancia qué Session Bean modifica a qué información, los cambios repercuten inmediatamente en el resto de Session Bean.

```
@Stateful
public class ExternoBean implements ExternoRemoterInterface
{
@EJB
private InternoLocalInterface internoBean;
@PersistenceContext(unitName = “JavaEE”,
type = PersistenceContextType.EXTENDED)
private EntityManger manager;
…
}
@Stateful
public class InternoBean implements InternoLocalInterface
{
@PersistenceContext(unitName = “JavaEE”,
type = PersistenceContextType.EXTENDED)
private EntityManager manager;
…
}
```

Código 2.88: Stateful Session Beans anidados

Para que pueda crearse un contexto común es imprescindible que el bean externo haga refernica al bean interno. Si se refiriera por el contenido a la Remote-Interface de otro bean, los contextos estarán separados por principio.

2.19.2.4. **EntityManager Factory**

Para conseguir en un Entity Manager que se puede administrar un Persistence Context individual dentro de un Persistence Unit. En Java SE se utiliza para ello una instancia de la clase **javax.persistence.EntityManagerFactory**, lo que en Java EE es posible, pero no imprescindible.

```
public interface javax.persistence.EntityManagerFactory
{
public EntityManager createEntityManager();
public EntityManager createEntityManager(Map map);
public void close();
public Boolean isOpen();
}
```

Código 2.89: Interface EntityManagerFactory

Con uno de los métodos **create()** se crea el EntityManager. Si no se han introducido todos los requerimientos específicos del fabricante en el archivo **persistence.xml** se puede incluir opcionalmente un **Map** con la información que falta. Si Factory ya no es necesario se puede cerrar llamando al método **close()**, lo que ocurre automáticamente en un servidor de aplicaciones. Si se llama a **close()** en un Factory cerrado se producirá una **IllegalStateException**. Para llegar dentro de Java SE a una instancia de la clase **EntityManagerFactory**, se tiene que utilizar primero uno de los métodos de la clase **javax.persistence.Persistence**, en el que puede dar el nombre de la Persistence Unit.

```
package javax.persistence;

public class Persistence
{
public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName)
{
…
}

public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName, Map
properties)
{
…
}
}
```

Código 2.90: Fragmento de javax.persistence.Persistence

En un servidor de aplicaciones se llega al Factory fácilmente. Aquí se crea la anotación **@javax.persistence.PersistenceUnit**, en la que mediante el atributo unitName se introduce el nombre de la Persistence Unit.

```
package javax.persistence;

@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface PersistenceUnit
{
String name() default “”;
String unitName() default “”;
}
```

Código 2.91: Anotacion @PersistenceUnit

```
import javax.ejb.*;
import.javax.persistence;

@Stateful
public class SocioAdministracionBean implements SocioAdministracionRemote
{
@PersistenceUnit(unitName = “JavaEE”)
public EntityManagerFactory factory;
}
```

Código 2.92: PersistenceFactory inyectado

Como ya se descrito, en una aplicación Java EE no es necesario dar un rodeo por el EntityManagerFactory, sino que se puede inyectar directamente una referencia a EntityManager de un Persistence Unit determinado.

2.19.2.5. **Interfaz EntityManager**

La manera más sencilla de que un Session Bean tenga accesi al EntityManager es definir un atributo del tipo **javax.persistence.PersistenceContext** y completarlo mediante la anotación **@PersistenceContext**.

```
package javax.persistence;
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface PersistenceContext
{
String name() default “”;
String unitName() default “”;
PersistenceContextType type default TRANSACTION;
PersistenceProperty[] properties() default {};
}
public enum PersistenceContextType
{
TRANSACTION,
EXTENDED
}
@Target({})
@Retention(RUNTIME)
public @interface PersistenceProperty
{
String name();
String value();
}
```

Código 2.93: Anotación @PersistenceContext

Un atributo importante es la entrada de **unitName**, que hace referencia al **PersistenceUnit**, es decir que lleva su nombre.

```
import javax.ejb.*;
import javax.persistence.*;
@Stateful
public class ArticuloAdministracionBean implements ArticuloAdministracionRemote
{
@PersistenceContext(unitName = “JavaEE”)
private EntityManager manager;
}
```

Código 2.94: EntityManager inyectado

La interfaz **javax.persistence.EntityManager** ofrece una serie de métodos con los que el desarrollador puede administrar sus Entity Beans. Junto a métodos como **persist()** o **find()** que ofrecen la administración directa de los beans, también hay métodos qe se pueden controlar mediante el Persistence Context. A continuación se explican cada uno de los métodos.

```
package javax.persistence;

public interface EntityManager
{
public void persist(Object entity);
public <T> T merge(T entity);
public void remove(Object entity);
public <T> T find(Class<T> entityClass, Object primaryKey);
public <T> T getReference(Class<T> entityClass, Object primaryKey);
public void lock(Object entity, LockModeType lockMode);
public void refresh(Object entity);

public void clear();
public Boolean contains(Object entity);

public void joinTransaction();
public EntityTransaction getTransaction();
public void flush();
public void setFlushMode(FlushModeType flushMode);
public FlushModeType getFlushMode();

public Query createQuery(String qlString);
public Query createNamedQuery(String name);
public Query createNativeQuery(String sqlString);
public Query createNativeQuery(String sqlString, Class result-Class);
public Query createNativeQuery(String sqlString, String result-SetMapping);

public Object getDelegate();
public void close();
public Boolean isOpen();
}
```

Código 2.95: Interfaz EntityManager

persist()

Para escribir un nuevo Entity Bean en la base de datos se utiliza el método **persist()** que espera únicamente el parámetro de la instancia del bean. a partir de este momento el Entity Bean será supervisado por el EntityManager. Si este se escribe en la base de datos

de inmediato o al finalizar la transacción, depende del **FlushMode** que esté configurado en el momento, que se explicará con más detalle luego. Para obligar al EntityManager a hacer la inclusión en la base de datos de inmediato, se puede llamar inmediatamente después de **persist()** al método **flush()**, que sincroniza todo el Persistence Context con la base de dtaos. Si en el parámetro del método **persist()** no se trata de un Entity Bean, se produciría un **IllegalArgumentException**. Si la llamada para un Persistence Context vinculado a una transacción tiene lugar fuera de una, se producirá una **TransactionRequiredException**.

```
public void addArticulo(Articulo articulo) throws ArticuloException
{
if(manager.find(Articulo.class, articulo.getArtnr()) != null )
{
throw new ArticuloException("Artículo ya existente");
}
manager.persist(articulo);
}
```

Código 2.96: Ejemplo de persist()

find() getReference()

EntityManager ofrece dos métodos para encontrar un Entity Bean a partir de su clave primaria. Ambos esperan como parámetro la clase del Entity Bean deseado y una referencia a la clave misma. Su diferencia esencial consisten en que **find()** devuelve el valor null cuando no se ha podio encontrar la línea de la base de datos, mientras que **getReference()** ejecutra una **EntityNotFoundException()**. Además solo **find()** soporta el llamado **LazyLoading**, con el cual es posible cargar primero tan solo parte del un Entity Bean.

```
public Articulo getArticulo(int artnr) throws ArticuloException
{
Articulo erg = manager.find(Articulo.class, artnr);
if( erg == null ) throw new ArticuloException("Artículo no encontrado");
else return erg;
}
```

Código 2.97: Ejemplo de find()

Ambos métodos producen una IllegalArgumentException si en la clase introducida no es un Entity Bean.

createQuery(), createNamedQuery() y createNativeQuery()

Para encontrar un Entity Bean también según otros criterios, se dispone de diferentes métodos con la ayuda de los cuales se pueden formular consultas generales. El método **createQuery()** espera una expresión EJB-QL en forma de una cadena de simbolos. En la seción sobre Consulta y EJB QL encontrará ejemplos más detallados sobre el lenguaje de consulta EJB QL. Arquitectónicamente es mejor preformular determinadas consultas dentro de la clase Entity Bean y allí dotarlas de nombre. Esto se consigue mediante la anotación **@NamedQuery**. Para llevar a cabo este tipo de consulta se utiliza entonces el método

createNamedQuery() del EntityManager, que solo espera el nombre de la consulta como parámetro.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface NamedQuery
{
String name();
String query()
QueryHint[] hints() default {};
}

@Target({})
@Retention(RUNTIME)
public @interface QueryHint
{
String name();
String value();
}
```

Código 2.98: Anotacion @NamedQuery

Si con los medios que ofrece EJB QL no se consigue ir más lejos, porque por ejemplo se requiere preguntar también a las bases de datos, que no son administradas directamente por un Entity Bean, aun se dispone de los métodos **createNativeQuery()** que espera cualquier consulta SQL en forma de una cadena de simbolos. Sin embargo no todo lo que esposible es a su vez también bueno. Cuantas más consultas SQL se usen en una aplicación, pero será está. Si no hay otro remedio, se recomienda utilizar la anotación **@NamedNativeQuery** que también se puede utilizar de manera más limpia como **@NamedQuery** en el Entity Bean y después se puede utilizar simplemente a través del método **createNamedQuery()**.

```
<entity class = “server.articulo”>
…
<pre-persist method-name = “nueva entrada”/>
…
</entity>
```

Código 2.99: Definicion XML de un método Pre-Persist

La llamada de este método muestra que el Entity Bean acaba de incluirse en el banco de datos. Esto no es sin embargo ninguna garantía de que permanezca ahí. Si se produjera en lo que queda de transacción algún tipo de error, se produciría un Rollback y se eliminaría esta fila de la base de datos.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface NamedNativeQuery
{
String name();
String query();
QueryHint[] hints() default {};
Class resultClass() default void.classl
String resultSetMapping() default “”; // Nombre del SQLResultSetMapping
}

@Target({TYPE})
```

```
@Retention(RUNTIME)
public @interface SqlResultSetMapping
{
String name();
EntityResult[] entities default {};
ColumnResult[] columns() default {};
}

@Target({})
@Retention(RUNTIME)
public @interface EntityResult
{
Class entityClass();
FieldResult[] fields() default {};
}

@Target({})
@Retention(RUNTIME)
public @interface FieldResult
{
String name();
String column();
}
@Target({})
@Retention(RUNTIME)
public @interface ColumnResult
{
String name();
}
```

Código 2.100: Anotación @NamedNativeQuery

merge()

Si se llevan a cabo modificaciones en un Entity Bean actualmente bajo control del EntityManager, estas se trasladrán automáticamente a la base de datos, si por el contrario el Bean no pertenece en ese momento a ningún Persistence Context se pueden sincronizar las modificaciones transfiriendo el método **merge()** al EntityManager. Una vez un Bean cae de este modo bajo control del EntityManager, este comprueba si en la base de datos existe ya alguna entrada para la clave primaria del bean. Si es así, se sustituye todas las columnas de esta fila con los atributos del bean, en caso contrario, inserta una nueva fila con los atributos actuales del bean en la base de datos. Siempre y cuando las claves primarias sean provistas por la aplicación, garantizado de este modo su claridad, es suficiente con utilizar en insert el método **merge()** en lugar de **persist()**.

Se producirá una **IllegalArgumentException** si la instancia transmitida no es un Entity Bean y una TransactionRequiredException si era necesario proteger la transacion.

```
public void cambiarArticulo(Articulo nuevo) throws ArticuloException
{
manager.merge(nuevo);
}
```

Código 2.101: Ejemplo de merge()

remove()

si es necesario eliminar una instancia de bean esta se transmite con ayuda del método **remove()** al EntityManager. De manera similar a **persist()**, este también depende del FlushMode actual o de la llamada explicita del método **flush()**, cuando se elimina efectivamente la línea de la base de datos. Si no se introduce nada, basta con eliminarla al finalizar la transacción. Después de llamar el método **remove()**, la instancia de Bean transmitida ya no será controlada por el EntityManager. Si se modifican posteriormente atributos de este bean, estos cambios ya no surtirán efecto.

También el método **remove()** conduce a las exception ya descritas **IllegalArgumentException** y **TransactionRequiredException**.

```
public void borrarArticulo(int artnr) throws ArticuloException
{
Articulo anterior = manager.find(Articulo.class, artnr);
if( anterior == null ) throw new ArticuloException("Artículo no encontrado");
manager.remove(anterior);
}
```

Código 2.102: Ejemplo de remove()

lock()

Para bloquera un Entity Bean en la base de datos, EntityManger ofrece el método **lock()**. Como primer parámetro espera la instancia de Bean a bloquear y en el segundo el **LockModeType** deseado, que debe tener un valor **READ** o un valor **WRITE**. En ambos casos se abre una frase de bloqueo en la base de datos que prohíbe a los demás editar simultáneamente la misma fila. A menudo esto ocurre durante un **SELECT FOR UPDATE**. En el caso de **WRITE** aumenta además el nuemro de versión del Entity Bean, en caso que este disponga de uno gracias a la anotación **@Version**. Hasta ahora todavía no hemos hablado de Entity Beans versionados. Se utilizan para poder controlar en frases bloqueadas optimizadas si los datos de la base de datos e han modificado durante la restauración.

refresh()

Si se quisieran actualizar los atributos de un Entity Bean con los valores de la base de datos, se transmitern con el método **refresh().** Todos los cambios realizados mientras tanto en el bean se pierden. Si el Entity Bean tiene relaciones definidas con otros beans, estas también se llevarán al estado de la base de datos.

El método **refresh()** también puede provocar las excepciones **IllegalArgumentsException** y **TransactionRequieredException**.

contains() y clear()

Con ayuda del método **contains()** se puede comprobar si una instancia de bean se encuentra bajo control del Entity Manager. El método devuelve los correspondientes **true** o

false. Devolverá una **IllegalArgumentException** en caso que el parámetro no se trate de un Entity Bean.

Con el método **clear()** se debe tener cuidado. Se ocupa de que el Entity Manager pierda el control de todos sus beans y por lo tanto que no escriba ninguna modificación en la base de datos. Así pues, resulta útil antes de utilizar el método **flush()**.

flush() y setFlushMode()

Las llamadas de los métodos **persist()**, **merge()** y **remove()** no conducen siempre de inmediato a una acción en la base de datos. Más bien dejan constancia en el EntityManager y se ejecutan posteriormente, cuando la transacción actual finaliza o cuando se ejecuta un Query. Sin embargo una vez se llama al método **flush()**, el EntityManager lleva a cabo la sincronización necesaria con el banco de datos.

Si usar el método **flush()** es practico, depende en última instancia del **FlushModeType** actual, que puede adoptar el valor **AUTO** o **COMMIT** y que se fija con ayuda del método **setFlushMode()**. El valor por defecto es **AUTO** y significa que se cede al EntityManager cuando este accede al banco de datos. En este caso **flush()** no tiene mucha razón de ser, a no ser que se tenga previsto realizar modificaciones masivas en la base de datos mediante instrucciones **UPDATE** y **DELETE**. Si se escoge como **FlushModeTypeCOMMIT**, entonces significa que todos los cambios deben realizarse justo al finalizar la transacción, lo que aumenta considerablemente el rendimiento de una aplicación de base de datos. En este caso el desarrollador tiene mayores exigencias puesto que ahora debe llamar en el momento justo al método **flush()** y en caso de una aplicación demasiado frecuente puede estropear todo el proceso.

getDelegate()

A través de este método se puede dar acceso a la implementación concreta del EntityManager para llamar métodos dependientes del fabricante. No obstante en una aplicación Java EE no hay apenas nada que se puede utilizar que no esté asegurado mediante el estándar. Se pierde muy rápido la ventaja que se posee cuando se escriben aplicaciones que funcionen en diferentes servidores.

joinTransaction() y getTransaction()

Con ayuda del método **joinTransaction()** un EntityManager creado fuera del contexto de una transacción, puede llegar a formar parte de la transacción actual del servidor de aplicaciones que se ejecuta en el método actual. Esta situación es bastante infrecuente en una aplicación Java EE.

El método **getTransaction()** devuelve una instancia de la clase **EntityTransaction** a través de cuyos métodos se puede controlar transacciones a mano. En una Persistence Unit **RESOURCE_LOCAL** esto es bastante práctico.

```
public interface EntityTransaction
{
public void begin();
public void commit();
public void rollback();
public void setRollbackOnly();
public boolean getRollbackOnly();
public Boolean isActive();
}
```

Código 2.103: Interfaz EntityTransaction

close() y isOpen()

El método **close()** cierra un EntityManager, de modo que ya no podrá volver a utilizarse. Si el correspondiente Persistence Context se encuentra en ese momento dentro de una transacción este se conducirá hasta el final de la transacción. La llamada del método **close()** a un EntityManager administrado por un servidor de aplicaciones conduce a una **IllegalStateException**.

El método **isOpen()** devuelve **true**, cuando el EntityManager actual está abierto y está a disposición de futuras modificaciones.

2.19.2.6. **Administración y Desadministración de Entidades**

Los Entity Beans se diferencian entre si dependiendo de si se encuentran o no bajo control de un EntityManager. Un bean se denominará administrado cuando esta siendo supervisado y por lo tanto cada modificación puede conducir a una acción en el banco de datos. Si se crea una instancia de bean a través de la misma llamada del constructor, hablamos por el contrario de un bean desadministrado, porque ningún EntityManager se interesa por él. Para que una instancia de bean normal se convierta en bean administrado, debe transmitirse al EntityManager mediante un método como **persist()**. Si en el caso de un Persistence Context orientado a transacciones finaliza la transacción, todos los beans administrados se convertirán de nuevo en beans desadministrados, es decir serán de nuevo instancias Java normales. En el caso de un Persistence Context ampliado esto no es así. Los beans se mantienen bajo control del EntityManager. Incluso cuando se modifican fuera de una transacción, estos cambios se sincronizarán con la base de datos, una vez la siguiente transacción iniciada finalice. Esta situación se puede dar cuando un Stateful Session Bean disponga de un Persitence Context ampliado y ofrezce métodos que no soporten ninguna transacción, pero que modifiquen sin embargo los atributos de beans controlados.

2.19.3. Entity Beans

2.19.3.1. **Uso**

Los Entity Beans son el portal a la base de datos o mejor dicho, el paso intermedio al nivel de persistencia, que no siempre debe tratarse de una base de datos relacional. Todos los datos guardados a largo plazo deben subyace como atributos de este tipo de bean u no estar dispersos por toda la aplicación. Este hecho es una característica esencial en la arquitectura de las aplicaciones Java EE, que de este modo prevé una estricta separación entre el nivel de mantenimiento de datos y el nivel lógico.

En sistemas programados de modo clásico aún sucede bastante a menudo que no existe esta separación. Muchos desarrolladores de aplicaciones se concentran a veces demasiado en el problema a resolver y no prestan ninguna atención a cuestiones de arquitectura y diseño. Si se les encarga hacer un programa para la gestión de los datos maestros de productos en venta, en el caso ideal se crea una clase Java que contiene los métodos necesarios para ello como **CrearNuevoArticulo** o **ModificarArticulo**. Con una nueva creación se transmiten al método todos los datos relevantes del artículo y se comprueban. Si los datos son correctos, se inscriben simplemente en la base de datos con ayuda de una instrucción SQL. Y justo aquí reside el problema. Tanto el nombre como la estructura de la tabla de base de datos deben conocerse llegados a este punto. Se crea código fuente que solo se ocupa del proceso técnico de guardado, en una clase que debería representar tecnicidad. Si la tarea consiste entonces en modificar un artículo, se escribe otro método al que quizás debe dársele el número de artículo. Uno de los primeros pasos en este método será de nuevo un SQL-Satement, que esta vez leerá los datos desde la tabla. Nuevamente el desarrollador se refuere a una tabla concreta con nombres de columnas concretos y lo que es aún peor, programa la condición a partir de la cual el sistema de base de datos debe encontrar estos datos. Cuando se tiene ante los ojos este panorama en toda su complejidad, enseguida se constata que el nivel de acceso al completo se dispersa entre incontables clase, lo que hace prácticamente imposible cualquier modificación o consolidación. Con razón cualquier inclusión posterior de otra columna en una tabla de base de datos conlleve un esfuerzo inmenso, pues se tienen que comprobar todos los puntos que mantiene alguna relación con la tabla correspondiente.

Un bean de entidad es un clase Java normal, que representa con sus atributos las columnas de una tabla de un banco de datos. Un desarrollador EJB no necesita tener ningún contacto directo más con el banco de datos, sino simplemente manejar los datos mediante las instancias de Entity Beans. Si se tiene que archivar un nuevo artículo, se crea tan solo una nueva instancia de bean. Si quiere modificar datos llama el método de búsqueda correspondiente y recibe de vuelta un objeto Java normal, en el que se lleva a cabo su modificación. Tanto el hecho de que aquí debe ser algo guardado, como la manera en la que esto ha ocurrido técnicamente, lo determina el servidor de aplicaciones y de este modo se libera al desarrollador de la necesidad de tener en cuenta una gran cantidad de detalles.

Todas las informaciónes técnicas requeridas subyacen exclusivamente en la clase de los Entity Beans. Ahí y tan solo ahí se describe, por ejemplo, cómo se llama la tabla, qué nombres tienen cada una de las columnas y cómo se debe buscar información en el banco de datos en cada caso. Así todo el nivel de persistencia esta encapsulado en clases determinadas y es independiente de la aplicación misma. Si se añade posterirmente una nueva columna a un sistema de este tipo primero se amplia la lista de atributos del Entity Bean correspondiente, asegurando así que la nueva información estará disponible en

todos los puntos relevantes de la aplicación. Si el nuevo valor es relevante en pedidos existentes, estos se pueden modificar centralmente y esto repercutirá inmediatamente en toda la aplicación. Los desarrolladores se pueden concentrar en su propio trabajo, es decir en controlar que la nueva información se maneje técnicamente en forma conrrecta.

La aproximación existente entraña aún otra gran ventaja. En cada acceso a un Entity Bean se implica el servidor de aplicación. Él sabe, con ayuda de sun Entity Manager, dónde se utiliza cada bean y cómo se modifica. Conoce en cada caso ideal todas las rutas de acceso a los datos que existen dentro de la aplicación y así las puede desarrollar eficientemente. Todo servidor de aplicaciones razonable generará con el primer acceso a través de un SQL-Statement un **PreparedStatement**, con la ayuda del cual tienen lugar simultáneamente un análisis sintáctico de la indicación y una búsqueda de la ruta de acceso. Esto se puede utilizar siempre que sea necesario para realizar el acceso con los parámetros validos en esta ocasión. Después de poco tiempo de duración del servidor a este ya le correponde una reelaboración como en los sistemas estaticos SQL, que técnicamente son de los mas eficientes. La ventaja descrita aquí resulta más efectiva cuantas menos indicaciones SQL propias escriban los desarrolladores en sus programas. Cuanto más consecuentes sea la aproximación del Entity Bean, menos SQL deberán programarse.

Los Entity Beans existen tan solo desde EJB 3.0. en versiones previas del estándar EJB la separación entre la clase bean y sus metainformaciones como nombres de tablas o declaraciones de acceso era mucho más estricta que hoy y por ese motivo para muchos algo muy complicado. Todas estas informaciones se encontraban tan solo en el Deployment Descriptor. En este punto con el nuevo estándar se ha empeorado en lo que se refiere a la arquitectura, para favorecer un manejo más sencillo. Hoy en día el desarrollador EJB trabaja directamente con el EntityManager, para acceder a través de él a las instancias de beans. Desafortunadamente es posible transmitir al EntityManager una declaración SQL como cadena de simbolos para realizar una búsqueda o una actualización masiva. Un uso frecuente de esto puede conducir nuevamente a que el nivel técnico de acceso se disperse en clases. Para seguir trabajando de manera limpia es muy recomendable declarar instrucciones de codificación y controlar también estas. El estándar ofrece de hecho todo lo necesario para concetrar estos detalles técnicos en la clase Entity Bean. Es algo que tan solo depende de uno mismo.

2.19.3.2. **Estructura**

Los Entity Beans son POJOs. Esta abreviatura significa Plain Old Java Objects, es decir para clases Java normales, convencionales. A diferencia de los Session Beans no implementan ninguna interfaz especial y cuando se renuncia a anotación, nada hace referencia a su significado como Entity Bean. en consecuencia este tipo de instancias de bean pueden utilizarse en cualquier parte, siempre es posible incluso enviarlas al cliente, para que este pueda mostrar sus datos. A menudo los Entity Beans son objetos serializados, de modo que sus referencias pueden ser retenidas por Stateful Session Beans. Cada Entity Bean debe disponer de un constructor estándar porque, entre otros, es tarea del EntityManger crear instancias a partir de esto.

Solo después de asociar un Entity Bean con un EntityManager surte efecto su verdadero significado. El EntityManager lee la metainformacion o bien desde la clase bean o desde el Deployment Descriptor y se ocupa de que los datos sean comparados con los de la base de datos. Un EntityManager de este tipo también puede existir sin servidor de aplicaciones. De este modo es posible escribir aplicaciones que también funcionen en un entorno Java SE.

```
CREATE TABLE ARTICULO
{
ARTNR INTEGER NOT CERO PRIMARY KEY,
DES VARCHAR(256) NOT CERO,
PRECIO DECIMAL(7, 2) NOT CERO,
CANTIDAD INTEGER NOT NULL
}
```

Código 2.104: La tabla ARTICULO

En el Código 2.104 se reproduce la estructura de una tabla para la administración de datos de artículos. Junto al número de artículos se pueden administrar una descripción, el precio y la cantidad almacenada actual. Para poder trabajar estos datos con ayuda de un Entity Bean se programa una clase que se reproduce en el Código 2.105 La anotación **@javax.persistence.Entity** caracteriza la clase como Entity Bean. También es necesaria la utilización de **@javax.persistence.Id** para la definición del valor clave univoco. O bien se otorga el **Id** directamente al atributo o al método **getter**, como ocurre en el siguiente ejemplo.

```
import java.math.BigDecimal;
import javax.persistence.*;
@Entity
public class Articulo implements java.io.Serializable
{
private int artnr;
private String des;
private BigDecimal precio;
private int cantidad;

@Id
public int getArtnr()
{
return artnr;
}
public void setArtnr( int artnr )
{
this.artnr = artnr;
}
public String getDes()
{
return des;
}
public void setDes( String des )
{
this.des = des;
}
public BigDecimal getPrecio()
{
return precio;
}
public void setPrecio( BigDecimal precio )
```

```
{
this.precio = precio.setScale(2, BigDecimal.ROUND_HALF_UP);
}
public int getCantidad()
{
return cantidad;
}
public void setCantidad( int cantidad )
{
this.cantidad = cantidad;
}
}
```

Código 2.105: Entity Bean Articulo

Para el resto de información EntityManager hace suposiciones estándar. Así por ejemplo fija el nombre de la tabla igual que el nombre del Entity Bean. También en lo que se refiere a la denominación de las columnas, se atiene a los atributos ya existentes. En el Código 2.106 se repooduce la anotación **@Table**, con la ayuda de la cual se puede dar el correspondiente nombre de la tabla, si este debe diferir del nombre de la clase. Esto se consigue configurando el atributo **name** como corresponde. Con los otros atributos se puede determinar por ejemplo a qué corresponde la tabla. Estos datos se almacenan sin embargo más bien en el **DataSource** a través del cual se define la conexión concreta a un sistema de banco de datos. Este tema se verá en más profundidad en el apartado sobre el método de trabajo de los Entity Beans. Los UniqueConstraints definen las dependencias entre tablas y serán requeridos tan solo cuando el servidor de aplicaciones mismo deba guardarlas.

```
package javax.persistence;
@Target({TYPE})
@Retention(RUNTIME)
public @interfac Column
{
String name() default “”;
boolean unique() default false;
boolean nullable() default true;
boolean insertable() default true;
boolean updatable() default true;
String columnDefinition() default “”;
String table() default “”;
int length() default 256;
int precision() default 0;
}
```

Código 2.106: La anotación @Column

La mayoría de atributos de la anotación **@Column** se explican por sí solo, tan solo el atributo table puede resultar extraño. Siempre es necesario cuando una Entity Bean debe constar de dos o más tablas de base de datos físicas, que en realidad deben ir juntas. Mediante la anotación **@SecondaryTable** se define esta segunda tabla para la clase bean y se indica en las columnas correspondientes, que provienen de esta otra tabla.

Si la anotación **@SecondaryTable** se introduce dos veces, se hace referencia entonces a tres tablas.

```
CREATE TABLE ARTICULO
```

```
{
ARTNR INTEGER NOT CERO PRIMARY KEY,
DES VARCHAR(256) NOT CERO,
PRECIO DECIMAL(7, 2) NOT CERO,
CANTIDAD INTEGER NOT NULL
}

CREATE TABLE ARTICULO2
{
ARTNR2 INTEGER NOT CERO PRIMARY KEY,
ANADIR VARCHAR(256) NOT CERO
}
```

Código 2.107: Las tablas ARTITULO y ARTICULO2

Si por alguna razón se distribuyen datos relacionados entre si en diferentes tablas se debe definir una Entity Bean que vincule estas tablas. En el Código 2.107 se crea una segunda tabla de artículos que contiene información adicional. En el Código 2.108 se describe un Entity Bean adecuado.

```
@Entity
@Table(name = ”ARTICULO”)
@SecondaryTable( name = “ARTICULO2”,
pkJoinColumns = {@PrimaryKeyJoinColumn(name = “ARTNR2”)})
public class Articulo implements java.io.Serializable
{
private int artnr;
private String des;
private BigDecimal precio;
private int cantidad;
private String Suplemento;

@Column( name = “SUPLEMENTO”, table = “ARTICULO2”)
public String getSuplemento()
{
return suplemento;
}



public void setSuplemento(String suplemento)
{
this.suplemento = suplemento;
}
}
```

Código 2.108: Entity Bean para dos tablas

La anotación **@PrimaryKeyJoinColumn** define a través de qué columna de la tabla añadida se crea la conexión de los datos. Debe coincidir con el contenido del campo de código primario de la tabla de artículos.

Para poder entrar la anotación **@SecondaryTable** varias veces se necesita la anotación **@SecondaryTables** que consta simplemente de una lista de **@SecondaryTable**.

Con eso ya se ha descrito suficiente la estructura del Entity Bean para la EntityManager. Con la información existente se pueden guardar los datos y modificarlos. Es incluso posible cargarlos, puesto que ya existe toda la información necesaria para poder accer al código primario. Para describir el bean con coherencia arquitectónica y con un contenido completo todavía faltan las consultas a través de las cuales se buscaran los datos. Como ya se ha descrito, estas no deben encontrarse dentro de la clase bean aunque es recomendable concentrarlas allí. Se presenta a continuación la anotación @javax.persistence.NamedQuery, con cuya ayuda se puede proveer una consulta con un nombre. La relación con la clas se establece de hecho tan solo a partir de este nombre, sin tener que escribir ninguna instrucción SQL.

```
package javax.persistence;

@Target({TYPE})
@Retention(RUNTIME)
public @interface NamedQuery
{
String name();
String query();
QueryHint[] hints() default { };
}
```

Código 2.109: La anotación @NamedQuery

El nombre de una NamedQuery debe ser preciso y unívoco dentro de todo el sistema, por lo que se le debería anteponer el nombre dele bean. La consulta en sí consta de un EJB-QL Statement que es muy parecido a una instrucción SQL. En la sección Consultas y EJB QL se tratará el alcance lingüístico de EJB QL. En el Código 2.110 se reproduce el **EntityBeanArticulo** completo, que dispone de una consulta con el nombre **Articulo.buscarTodo**, mediante el cual se pueden leer todas las líneas de artículos.

```
package server.eb;

import java.math.BigDecimal;
import javax.persistence.*;

@Entity
@NamedQueries
({
@NamedQuery(
name = “Articulo.buscarTodo”,
query = “SELECT a FROM Articulo a ORDER BY a.artnr”)
})
@Table( name = “ARTICULO”)
public class Articulo implements java.io.Serializable
{
private int artnr;
private String des;
private BigDecimal precio;
private int cantidad;

@Id
@Column( name = “ARTNR” )
public int getArtnr()
{
return artnr;
}
```

```
public void setArtnr( int artnr )
{
this.artnr = artnr;
}

@Column( name = “DES” )
public String getDes()
{
return des;
}
public void setDes( String des )
{
this.des = des;
}

@Column( name = “PRECIO” )
public BigDecimal getPrecio()
{
return precio;
}
public void setPrecio( BigDecimal precio )
{
this.precio = precio.setScale(2, BigDecimal.ROUND_HALF_UP);
}

@Column( name = “CANTIDAD” )
public int getCantidad()
{
return cantidad;
}
public void setCantidad( int cantidad )
{
this.cantidad = cantidad;
}
}
```

Código 2.110: Entity Bean Articulo completo.

2.19.3.3. **Metodología de trabajo**

Un Entity Bean sin **EntityManager** es tan solo una clase Java normal sin ningun significado en particular. Solo al vincularse con un EntityManager se ocupará de la persistencia de los atributos bean. Todos los beans de una aplicación pertenecen a lo que se denomina **PersistenceUnit**, que debe ser descrita con la ayuda de su propio Deployment Descriptor. Este descriptor se trata de un archivo **persistence.xml** que debe encontrase en el **META-INF**. En él se adjudica el nombre **PersistenceUnit** y se determina a través de que **DataSource** se debe crear la asociación con el banco de datos. En el Código 2.111 se encuentra la definición del UnitJavaEE, asociado al **DataSource DefaultDS**, que trae el JBoss por defecto. Todos los Entity Beans que se encuentran con este Deployment Descriptor en el mismo archivo JAR pertenecen a la **PersistenceUnit**. Es posible con ayuda del descriptor, incluir más archivos JAR o también excluir clases, lo cual se verá más adelante.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
```

```
version="2.0">
<persistence-unit name="JavaEE" transaction-type="JTA">
<jta-data-source>java:DefaultDS</jta-data-source>
</persistence-unit>
</persistence>
```

Código 2.111: persistence.xml

Una vez definido la **PersistenceUnit** se puede inyectar a un Session Bean una referencia al **EntityManager** de esta Unit mediante la anotación **@javax.persistence.PersistenceContext**. De manera parecida a los Session Beans, que se encuentran en un **SessionContext** también los Entity Bean se encuentran en un **PersistenceContext**. En el Código 2.112 se reproduce un Stateful Session Bean, que se procura una referencia al **EntityManager** de la **PersistenceUnit** Java EE.

```
@Stateful
public class AdministracionArticuloBean
{
@PersistenceContext( unitName = “JavaEE” )
private EntityManager manager;
…
}
```

Código 2.112: Acceso al EntityManager

Para guardar un nuevo articulo en el banco de datos basta con crear una nueva instancia de la clase **Articulo** y transmitirla mediante el método **persist()** al **EntityManager**. A partir de este momento la instancia bean se etiquetará como EntityBean **administrada**, porque se encuentra bajo el control del **EntityManager**. Si se modifican atributos de la instancia del bean dentro de la transacción, se efectúan estas modificaciones sin que sea necesario ningún otro comando adicional. Cuando escribe realmente el **EntityManager** estos datos en la base de datos, no se sabe. Tan solo se debe confirmar que esto ocurre dentro de la transacción. Si por error se quisiera crear un nuevo articulo que ya existe en el banco de datos con el mismo código primario, no se obtendría ningun mensaje de erroe al llamar el método **persist()**. La aplicación parte inicialmente del hecho de que todo funciona. Es justo al finalizar la transacción cuando ua se ha procesado toda la lógica cuando se produce un erro SQL que provoca la excepción correspondiente. Por ello es aconsejable antes de crear un nuevo articulo si ya existe un articulo con el mismo numero. Para ello el **EntityManager** pone a disposición el método **find()**, que requiere dos parámetros. Primero se debe introducir la clase bean en la que se tiene que buscar. El segundo parámetro representa el código en concreto. El método devuelve cero, si no se ha encontrado la fila correspondiente en la base de datos. Si queremos ser precisos, esta no es una protección cien por cien segura para evitar los errores descritos. Si dos usuarios quisieran introducir a la vez el mismo articulo, el segundo recibiría una excepción al finalizar la transacción, aunque hubiera preguntado previamente. Este comportamiento es típico de toda aplicación que trabaje con una base de datos relacional. Un programa COBOL en un ordenador IBM tendría el mismo problema. Para tratarlos el sistema siempre da estos códigos univocos como número de artículo, garantizando así la precisión. En el Código 2.113 se muestra un ejemplo de un método que escribe un nuevo artículo en la base de datos, en la que no

había todavía ninguno bajo ese número. En caso de error se lanza una **ApplicationException**, que conducirá a un Rollback, pero no a una liberación del Session Bean.

```
public void addArticulo( Articulo articulo ) throws ArticuloException
{
if( manager.find(Articulo.class, articulo.getArtnr()) != cero )
{
throw new ArticuloException("Artículo ya existe");
}
manager.persist( articulo );
}
```

Código 2.113: Nueva inclusión

Para poder buscar todos los artículos debe encargársele al **EntityManager** que produzce la correspondiente consulta. Para ello está a disposición primero el método **createQuery()**, al que se le debe proporcionar una instrucción EJB-QL como cadena de signos. Sin embargo, el modo en el que se ha procedido hasta ahora es la peor opción a la hora de programar una aplicación. Es mucho mejor definir dentro del Entity Bean una **NamedQuery**, que contiene a su vez la instrucción EJB-QL. El método **createNamedQuery()** de la clase **EntityManager** espera después tan solo el nombre de la consulta para crearla. En cada clase se devuelve una instancia de la clase **Query**, a través de la cual se puede vincular el resultado. Para ello existe el método **getRetsultList()** que devuelve una instancia de **List** y se debe utilizar siempre que exista más de una línea. Si no se encontrara nada, se devuelve simplemente una lista vacía, no se produciría ninguna **Exception**. La alternativa se llama **getSingleResult()**, que tiene como resultado un **Object** y parte del hecho que la cantidad a resultar consta de tan solo una línea. Si tampoco aquí se encontrará nada, se produce la excepción **javax.persistence.EntityNotFoundException**. Si el resultado consta de más de una fila seguirá una **javax.persistence.NonUniqueResultException**. Las dos se tratan de una **RuntimeException**, por lo que aquí no se exigirá ningún bloque **try/catch**. Si no se produce ninguno de estos errores, el servidor de aplicaciones libera de inmediato el Session Bean y se interrumpe la comunicación con el usuario. En el Código 2.114 se transfieren todos los artículos encontrados a un vector, que después se puede transmitir al cliente. Hasta aquí no nos hemos introducido todavía en las transaccinoes, pero como aquí se trata tan solo de un método de lectura, es bastante razonalbe caracterizarlo con la anotación **@TransactionAttribute** como **NOT_SUPPORTED**. Esto conduce a que no exista ninguna protección en la transacción, tampoco es aquí necesario, por lo que se aligera bastante el trabajo al **EntityManager** y se contribuye así a mejorar su rendimiento.

```
@TransactionAttribute( TransactionAttributeType.NOT_SUPPORTED)
public Vector<Articulo> getArticulo()
{
List listas = manager.createNamedQuery("Articulo.buscarTodos").getResultList();
Vector<Articulo> erg = new Vector<Articulo>();
for( Object o : listas )
{
erg.add( (Articulo) o );
}
return erg;
}
```

Código 2.114: Buscar todos los artículos

Para modificar los datos de un Entity Bean existen dos posibilidades. O bien se procede del modo tradicional, buscando primero el bean y después modificando sus atributos con ayuda de métodos o se utiliza el método **merge()**. El primer caso asegura que el artículo se encuentra entre las existencias antes de modificarlo. En el caso del método **merge()** esto no es así. Espera una referencia a la instancia del bean y primero comprueba si ya existe un artículo con ese número. Si es así, se leen los datos y se sustituyen todos los atributos con lo nuevos valores. Si por el contrario no se encuentra nada en la base de datos, el método **merge()** produce un nuevo archivo. De este modo se podría solucionar superficialmente el problema de la doble nueva entrada, como se ha descrito anteriormente. Si dos usuarios intentaran simultáneamente guardar un artículo con el mismo número y se llama en los dos casos el método **merge()**, el primero introduciría el artículo, sin que se creara por ello un mensaje de error. En una aplicación profesional esto sería más que crítico que una excepción, mediante la cual el usuario por lo menos recibe noticia de que ha ocurrido un problema. De todos modos la probabilidad de que dos usuarios quieran guardar simultáneamente el mismo artículo es más bien escasa. Lo que más bien es probable que suceda es que dos artículos diferentes se quieran guardar y se les dé por error el mismo número de artículo. El método **merge()** por ello no es el más adecuado para una nueva entrada.

```
public void cambiarArticulo( Nuevo Articulo) throws ArticuloException
{
manager.merge( Nuevo );
}
```

Código 2.115: Modificar artículos

Por último, a veces puede ser necesario eliminar Entity Beans del banco de datos. Para ello el **EntityManager** provee el método **remove()**, que requiere como único parámetro una instancia de bean. La llamada de este método no tiene por qué conducir de inmediato a la eliminación en la base de datos. En este caso tampoco es necesario que la acción se ejecute siempre durante la transacción, sino en cualquier momento. Después de transmitir la instancia de bean a **remove()**, esta ya no se administrará más. Si se produjeran cambios posteriores en los atributos, ya no se efecturían. La única posibilidad de volver a crear el bean es volver a generarlo con **persist()**. En el Código 2.116 se reproduce un método para eliminar un artículo.

```
public void eliminarArticulo( int artnr ) throws ArticuloException
{
Articulo alt = manager.find( Articulo.class, artnr );
if( alt == cero ) throw new ArticuloException("Artículo no encontrado");
manager.remove( alt );
}
```

Código 2.116: Eliminar un artículo

Con este último se ha ofrecido ya una panorámica general de toso los pasos de edición posibles de un Entity Bean.

2.19.3.4. **Métodos del Ciclo de vida**

El control real de un Entity Bean reside en el **EntityManager**. Este informa a la instancia del bean con determinadas acciones, de modo que esta pueda reaccionar en caso necesario. Para ello el **EntityManager** llama a los denominados métodos **lifecycle-callback**, si la clase bean los ha implementado. Como es usual esto se hace mediante la anotación correspondiente o con el descriptor de despliegue.

PrePersist

Este método se llama cuando el método **persist()** se ha finalizado correctamente, y por lo tanto el bean está preparado para el guardado. También la llamada del método **merge()** puede llevar a este método, en concreto cuando se ha constatado que aún no existe la correspondiente línea en la base de datos.

```
@PrePersist
public void nuevaEntrada()
{
…
}
```

Código 2.117: Ejemplo de una anotación @PrePersist

Para la descripción de Entity Bean hay un Deployment Descriptor propio con el nombre **orm.xml**, que debería encontrarse en el índice **META-INF**. Si se trabaja exclusivamente con anotaciones, se puede prescindir por completo de este archivo. Su estructura exacta se describe posteriormente en esta sección.

```
<entity class = “server.Articulo”>
…
<pre-persist method name = “nuevaEntrada”/>
…
</entity>
```

Código 2.118: Definicion XML de un método Pre-Persist

PostPersist

La llamada de este método muestra que el Entity Bean acaba de incluirse en el banco de datos. Esto no es sin embargo ninguna garantía de que permanezca ahí. Si se produjera

en lo que queda de transacción algún tipo de error, se produciría un Rollback y se eliminaría esta fila de la base de datos.

```
@PostPersist
public void nuevaEntrada()
{
…
}
```

Código 2.119: Ejemplo de una anotación @PostPersist

```
<entity class = “server.Articulo”>
…
<post-persist method name = “nuevaEntrada”/>
…
</entity>
```

Código 2.120: Definicion XML para un método Post-Persist

PreRemove

La llamada de este método muestra que el Entity Bean está preparado para eliminación. El método **remove()** puede ejecutrase sin errores. La anotación correspondiente se llama @**PreRemove** la etiqueta en el Deployment Descriptor **<pre-remove/>**.

PostRemove

En este estado se acaba de transmitir al sistema de la base de datos una orden para la eliminación de la fila correspondiente. Como en **PostPersist** aquí tampoco se asegura realmente que esto se produzca. Un Rollback posterior puede deshacer toda la acción. La anotación correspondiente es @**PostRemove**, la etiqueta en el Deployment Descriptor es **<post-remove/>**.

PreUpdate

Para este evento no hay ningún momento definido porque no hay ningún método correspondiente en el EntityManager. Según la implementación el servidor de aplicaciones llamada el método después de cada modificación de un atributo, o tan solo una vez, cuando ha reconocido que debe modificar la línea correspondiente en el banco de datos. La anotación es @**PreUpdate**, la etiqueta en el Deployment Descriptor **<pre-update/>**.

PostUpdate

La llamada de este método documenta que se le acaba de indicar al banco de datos que modificque la línea correspondiente. Pero tampoco aquí existe garantía alguna. Un

Rollback podría instaurar de nuevo el estado original. La anotación es @**PostUpdate**, la etiqueta en el Deployment Descriptor **<post-update>**.

PostLoad

El último método de esta serie es el método **PostLoad**. Comunica al Entity Bean que acaban de cargarse datos, que ya se pueden leer en el banco de datos.

2.19.3.5. XML Deployment Descriptor

No existe ninguna necesidad de entrar datos mediante un Entity Bean en un Deployment Descriptor. Solo cuando se prescinde de anotaciones o se tienen que sobreescribir sus datos el Deployment Descriptor resulta una opción adecuada.

Las entradas de datos se hacen en un archivo XML, que se llama por defecto **orm.xml**. Se trata de hecho más de un archivo mapping, a través del cual se puede determinar a qué tabla de base de datos pertenece cada clase Entity Bean y cómo se distribuyen los atributos en las filas. Se debe mencionar que junto a este archivo puede haber otros archivos de mapping, cuyos nombres deben introducirse en la definición **PersistenceUnit** el archivo **persistence.xml**. En el Código 2.121 se reproduce una de estas definiciones.

```
<? xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
version="2.0">
<persistence-unit name="JavaEE" transaction-type="JTA">
<jta-data-source>java:DefaultDS</jta-data-source>
<mapping-file>META-INF/propio_mapping.xml</mapping-file>
</persistence-unit>
</persistence>
```

Código 2.121: PersistenceUnit con archivo de mapping propio.

No tienen que hacerse muchas entradas de datos. Al fin y al cabo se obtienen todos a partir de las anotaciones mostradas. Así la etiqueta **column** puede disponer de muchos datos, como ya se han explicado al presentar la anotación @**Column**. Una entrada en el Código 2.122 es muy importante. En la etiqueta **entity** se encuentra la definición **meta-data-complete = "true"**, que dice que todas las anotaciones de la clase bean deben ser ignoradas y solo cuenta lo que se introduzca aquí. Si se quisiera por ejemplo definir de otro modo tan solo el nombre de una columna, entonces no se debe hacer esta entrada de datos o debe configurarse como **"false"**. Si por el contrario el Entity Bean no contiene ninguna anotación, vuelve a ser necesario de nuevo el dato **"true"**, porque al contrario se intentará por lo menos leer el atributo desde la clase bean, que disponga de la anotación @**Id**. Esto provocará inevitablemente un mensaje de error.

```
<? xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_2_0.xsd" version="2.0">
<entity class=”server.Articulo” access=”PROPERTY” metadata-complete=”true”>
<table name=”ARTICULO”/>
<named-query name=”Articulo.buscarTodos”>
<query>
SELECT a FROM Articulo a ORDER BY a.artnr
</query>
</named-query>
<attributes>
<id name=”artnr”>
<column name=”ARTNR”/>
</id>
<basic name=”des”>
<column name=”DES”/>
</basic>
<basic name=”precio”>
<column name=”PRECIO”/>
</basic>
<basic name=”cantidad”>
<column name=”CANTIDAD”/>
</basic>
</attributes>
</entity>
</entity-mapping>
```

Código 2.122: orm.xml

En el Código 2.122 se reproduce el Deployment Descriptor **orm.xml** al completo, que para el Entity Bean Articulo da por hecho que la clase bean no dispone de ningún tipo de anotación.

2.19.4. Consultas y EJB QL

2.19.4.1. **Panorama general**

En la siguiente sección se detallará el complejo ámbito temático de las consultas. La búsqueda de los Entity Beans goza de la misma importancia que tiene su almacenamiento. En apartados anteriores se mostró que este tipo de búsquedas se pueden insertar mediante los métodos **createQuery()** y **createNativeQuery()** de la clase **EntityManager**. Los nombres de los métodos ya indican que existen básicamente dos maneras de buscar datos. La primera consiste en utilizar el lenguaje de consultas EJB QL (Enterprise JavaBeans Query Language), que se presentará con detalle en esta sección. Es siempre preferible puesto que ya no trabaja a nivel de tablas sino a nivel de los Beans y por lo tanto es totalmente independiente de la estructura de base de datos. La otra posibilidad consiste en recurrir de nuevo a SQL nativas. No obstante esto debería hacerse solamente si no hay otro remedio. Algo que puede motivar su uso podría se que se deben incluir en la consulta columnas de las tablas de base de datos que no están administradas por ningun Entity Bean. A estas columnas no se puede acceder mediante EJB QL.

El EntityManager también ofrece el método **createNamedQuery()**, que entra en acción cuando la consulta se deposita mediante una de las anotación **@NamedQuery** o **@NamedNativeQuery**. En el caso ideal se encuentra la anotación en la clase Entity Bean que le proporciona como resultado y se sitúa así en un lugar central. Si se constata que durante el funcionamiento productivo de la aplicación es necesario retocar algún acceso a la base de datos, y de hecho este casi siempre sucede, entonces se puede resolver Entity Bean por Entity Bean y no se debe recorrer toda la aplicación para optimizar un acceso en algún punto en concreto. Ya se ha hecho referencia en algún apartado anterior al hecho de que el nuevo estándar permite, desafortunadamente programar consultas en cualquier parte de un Session Bean.

Independientemente de qué método se utilice siempre se devuelve una instancia de la clase Query que se describirá y explicará a continuación. Con esta instancia se transmiten parámetros necesarios a la consulta, que será ejecutada después también con ayuda de los métodos correspondientes. Si se utiliza SQL clásica los parámetros se escribirán, como es usual, en forma de interrogación en el String de la consulta. En los EJB QL se puede escoger entre parámetros de posición (una interrogación seguida de una cifra) o parámetros denominados (dos puntos seguidos de un denominador).

2.19.4.2. **Interfaz Query**

Si se llama uno de los métodos **createQuery()** o **createNamedQuery()** de la clase **EntityManager** siempre se obtiene como resultado una instancia de **Query**, siempre y cuando la sintaxis de la consulta transmitida fuera correcta.

```
package javax.persistence;
import java.util.Calendar;
import java.util.Date;
import java.util.List;
public interface Query
{
public List getResultList();
public Object getSingleResult();

public int executeUpdate();

public Query setMaxResults(int maxResult);
public Query setFirstResult(int startPosition);

public Query setHint(String hintName, Object value);

public Query setParameter (String name, Object value);
public Query setParameter (String name, Date value, TemporalType temporalType);
public Query setParameter (String name, Calendar value, TemporalType temporalType);
public Query setParameter (int position, Object value);
public Query setParameter (int position, Date value, TemporalType temporalType);
public Query setParameter (int position, Calendar value, TemporalType temporalType);

public Query setFlushMode(FlushModeType flushMode);
}
```

Código 2.123: Interfaz Query

Cada uno de los métodos de la interfaz **Query**aparecen en el Código 2.123 y se verán a continuación con más detenimiento.

2.19.4.2.1. getResultList()

El método de más importancia es **getResultList()** con el cual se lleva a cabo una consulta y se obtiene su resultado. Siempre devuelve una instancia de **java.util.List**, que puede estructurarse de modo diferente según la consulta. En el caso más sencillo se selecciona mediante una consulta una lista de Entity Beans, como muestra el Código 2.124.

```
@Entity
@NamedQueries({
@NamedQuery(name=”Socio.buscarTodo”,
query=”SELECT p FROM Socio p ORDER BY p.SocioNr”)
})
@Table(name=”SOCIO”)
public class Socio
{ …. }

public List getSocio()
{
Query query = manager.createNamedQuery(“Socio.buscarTodo”);
List lista = query.getResultList();
return lista;
}
```

Código 2.124: Consulta de todos los Socios

Si mediante una consulta, se selecciona una lista con tan solo un valor por fila, como en el Código 2.124, entonces la instancia devuelta **List** está constituida de esos valores. Un **Iterator**, en el mismo ejemplo, devuelve en cada acceso una instancia de **Socio**.

```
@Entity
@NamedQueries({
@NamedQuery(name=”Socio.buscarNombresTodos”,
query=”SELECT p.name FROM Socio p”)
})
@Table(name=”SOCIO”)
public class Socio
{ …. }

public List getNombresSocios()
{
Query query = manager.createNamedQuery(“Socio.buscarNombresTodos”);
List lista = query.getResultList();
return lista;
}
```

Código 2.125: Selección de todos los nombres de los Socios

En el Código 2.125 también se selecciona tan solo un valor por fila, pero aquí no se trata de todo el objeto **Socio** sino solo de su nombre. La instancia **Lista** devuelta consta de Strings, que se pueden obtener pieza a pieza con un **Iterator**.

```
@Entity
```

```
@NamedQueries({  @NamedQuery(name=”Socio.buscarNombresTodos”,
    query=”SELECT p.nombre, p.apellido FROM Socio p”)  })
@Table(name=”SOCIO”)
public class Socio
{…}
public List getNombresSocios()
{
Query query = manager.createNamedQuery(“Socio.buscarNombresTodos”);
List lista = query.getResultList();
return lista;
}
```

Código 2.126: Selección de Nombre y Apellido

Si se seleccionan como en el Código 2.126 varios valores por fila, en este caso nombre y apellido, la lista de resultado consta de objetos del tipo **Object[ ]**. Si por ejemplo hay diez socios, la lista consta de diez de estas tablas de objetos. Cada tabla tiene, según el ejemplo, dos entradas, un String para el nombre y un String para el apellido. Si están permitidos los valores null para las columnas seleccionadas, la instancias entregada también será null.

```
@Entity
@NamedQueries({
@NamedQuery(name=”Socio.buscarPedidoSocio”,
query=”SELECT p b FROM Socio s, Pedido p WHERE s.SocioNr = p.SocioNr”)
})
@Table(name=”SOCIO”)
public class Socio
{…}
public List getNombresSocios()
{
Query query = manager.createNamedQuery(“Socio.buscarPedidoSocio”);
List lista = query.getResultList();
return lista;
}
```

Código 2.127: Socios y sus Pedidos

En el último ejemplo de el Código 2.127 se selecciona en cada fila tanto socios como también pedidos. Las lista de resultados consta de nuevo de instancias **Object[ ]**, cada primer elemento de estas lleva una instancia de **Socio** y cada segundo elemento una instancia de **Pedido**.

Si en una consulta se mezclan tipos Entity Bean con simples tipos de columnas, la tabla de objetos se estructurará consiguientemente.

Si se lleva a cabo una consulta SQL clásica (NativeQuery), el contenido de la lista de resultados resulta de las columnas seleccionadas. Para ello sirven las mismas normas ya representadas. Si se definen estas consultas con ayuda de la anotación **@NamedNativeQuery**, sus resultados se explicarán o bien mediante el parámetro **resultClass** o mediante **resultSetMapping**. Si se selecciona tan solo una lista del tipo Entity Bean, esta se definirá mediante **resultClass** y la lista de resultados constará de este tipo de instancias. Si se consulta por otros tipos de datos o por una mezcla de ambos, se

combinará la estructura del resultado con la anotación **@SqlResultSetMapping**. La construcción de la lista de resultados se erige exactamente según esta descripción.

```
@SqlResultSetMapping
(
name=”SocioDireccion”,
entities = { @EntityResult(entityClass=server.Socio.class),
    @EntityResult(entityClass=server.Direccion.class) }
)
```

Código 2.128: Selección de dos tipos de Entity Bean

La lista devuelta consta en el caso del Código 2.128 de instancias **Object[ ]**, que lleva cada una un Socio y una Direccion.

```
@SqlResultSetMapping
(
name=”ValoracionAportacion”,
columns={
@ColumnResult(name=”AVG_APORTACION”),
@ColumnResult(name=”MAX_APORTACION”),
@ColumnResult(name=”MIN_APORTACION”)
}
)
```

Código 2.129: Selección de tres tipos dobles

Por el contrario la lista del Código 2.129 consta de instancias **Object[ ]** que hacen referencia cada una a tres instancias dobles.

2.19.4.2.2. getSingleResult()

Una vez comprobado que una consulta solo puede referirse a una línea de resultado es posible, recogerla con ayuda del método **getSingleResult()**. El resultado es del tipo **Object** y depende de cuántas columnas se hayan seleccionado realmente. Como ya se ha descrito en el método **getResultList()** se trata o bien de una única instancia del tipo seleccionado y/o un **Object[ ]**, que representa la lista de selección.

Si el resultado debe abarcar más de una línea, se produce una **NonUniqueResultException**. Si no se puede seleccionar nada, el resultado será **null**.

2.19.4.2.3. executeUpdate()

Aparte de las consultas también se pueden utilizar, con ayuda de EJB QL, instrucciones **UPDATE** y **DELETE**, que pueden influenciar a toda una serie de Entity Beans. El método utilizado para ello **executeUpdate()** devuelve el numero de filas eliminadas.

Este tipo de modificaciones masivas deben efectuarse siempre al inicio de una transacción, antes de que se empiece a seleccionar beans por separado.

Es interesante que este tipo de instrucciones también se pueden definir mediante una anotación @**NamedQuery** en la clase Bean.

2.19.4.2.4. setMaxResults() y setFirstResult()

Si se quisiera leer una lista de resultados en fragmentos o por paginas, se pueden utilizar los métodos **setMaxResults()** y **setFirstResult()**. Con **setMaxResults()** se transmite el número de filas que se quieren obtener como máximo en el resultado. Con **setFirstResult()** se puede fijar la posición inicial en la lista, empezando por 0, a partir de la cual debe empezar la selección de las filas n.

Ambos métodos devuelven el Query como resultado y se pueden combinar entre ellos.

```
public List getNombresSocios()
{
Query query = manager.createNamedQuery(“Socio.buscarNombresTodos”).setFirstResult(100).setMaxResults(25);
List lista = query.getResultList();
return lista;
}
```

Código 2.130: Restricción de los resultados

2.19.4.2.5. setHint()

Si se desea incluir requisitos específicos en la consulta, se puede utilizar el método **setHint()**. Espera como primer parámetro una cadena de simbolos con el nombre de la propiedad y en el segundo parámetro una instancia **Object** con el valor correspondiente.

```
public List getNombresSocios()
{
Query query = manager.createNamedQuery(“Socio.buscarNombresTodos”).
setHint(“org.hibernate.timeout”, new Integer(1000));
List lista = query.getResultList();
return lista;
}
```

Código 2.131: Fijar propiedades específicas

2.19.4.2.6. setParameter(…)

Para transmitir parametos necesarios a una consulta, se dispone en total de seis métodos diferentes con el nombre setParameter. Se dividen en tres versiones, con las cuales se pueden fijar los parámetros denominados, y en otras tres para parámetros dependientes de la posición.

```
@Entity
@NamedQueries({
@NamedQuery(name=”Socio.buscarPorNombre”,
query=”SELECT s FROM Socio s WHERE s.nombre = :nombre”)
})
@Table(name=”SOCIO”)
public class Socio
{
…
}

public List getSocioSegundoNombre(String nombreBuscado)
{
Query query = manager.createNamedQuery(“Socio.buscarPorNombre”);
query.setParameter(“nombre”, nombreBuscado);
List lista = query.getResultList();
return lista;
}
```

Código 2.132: Fijar un parámetro denominado

Un parámetro denominado se define dentro de una expresión EJB QL mediante dos puntos seguidos de una denominación. En el Código 2.132 se trata de **“:nombre”.** Este parámetro se fija con **setParameter(String, Object)**.

```
@Entity
@NamedQueries({
@NamedQuery(name=”Socio.buscarPorNombre”,
query=”SELECT s FROM Socio s WHERE s.nombre = ?1”)
})
@Table(name=”SOCIO”)
public class Socio
{
…
}

public List getSocioSegundoNombre(String nombreBuscado)
{
Query query = manager.createNamedQuery(“Socio.buscarPorNombre”);
query.setParameter(“nombre”, nombreBuscado);
List lista = query.getResultList();
return lista;
}
```

Código 2.133: Fijar un parámetro por posición

Si por el contrario se trabaja con parámetros de posición, estos se definen con un signo de interrogación con una cifra a continuación. El método necesario para fijar este parámetro es **setParameter(int, Object)**.

Los cuatro métodos restanes se utilizan cuando se tiene que transmitir como valor una instancia **java.util.Date** o n. La indicación mediante el tercer parámetro del tipo **javax.persistence.TemporalType** se realiza si el contenido de la fecha se interpreta como **java.sql.Date**, **java.sql.Time** o **java.sql.Timestamp**.

```
public enum TemporalType
{
DATE,           // java.sql.Date
TIME,           // java.sql.Time
```

```
TIMESTAMP     // java.sql.Timestamp
}
```

Código 2.134: TemporalType

En el Código 2.135 encontramos un ejemplo de la consulta de todos los socios con una determinada fecha de nacimiento.

```
@Entity
@NamedQueries({
@NamedQuery(name="Socio.buscarPorFechaNac",
query="SELECT s FROM Socio s WHERE s.fechaNac = :fechaNac")
})
@Table(name="SOCIO")
public class Socio
{
…
}

public List getSocioSegunFechaNacimiento(java.util.Date datum)
{
Query query = manager.createNamedQuery("Socio. buscarPorFechaNac");
query.setParameter("fechaNac", datum, TemporalType.DATE);
List lista = query.getResultList();
return lista;
}
```

Código 2.135: Consulta por fecha de nacimiento

2.19.4.2.7. setFlushMode()

Cuando se realiza una consulta a la base de datos es importante que todos los campos que se han producido durante la transacción sean conocidos por la base de datos. En programas clásicos este siempre es el caso por el programador programa él mismo sus instrucciones **INSERT**, **UPDATE** y **DELETE** y estasse ejecutan de inmediato. Si por en cambio se trabaja con EntityBeans, el desarrollador tan solo llevará a cabo modificaciones en instancias Java y queda pendiente del EntityManager cuándo se transmiter estos cambios a la clase a la base de datos. Esto sucederá como muy tarde al finalizar la transacción.

Una vez el EntityManager ejecuta una consulta, comprueba si administra actualmente beans en los que se ha producido algún tipo de cambio que la base de datos aún no conoce. Si es este el caso ejecuta primero estas modificaciones y después la consulta en sí. Este sería el procedimiento de la configuración estándar de **FlushModeType.AUTO**.

```
public enum FlushModeType
{
COMMIT,
AUTO
}
```

Código 2.136: FlushModeType

Para una única consulta se puede configurar este **FlushModeType** individualmente. Si el **FlushModeType** general se configura en el EntityManager con **COMMIT**, puede resultar práctico cambiar a **AUTO** en consultas concretas y viceversa. El **FlushModeType.COMMIT** hace que las modificaciones se transmitan a la base de datos al finalizar la transacción y no inmediatamente antes de la consulta.

El desarrollador de aplicaciones dispone siempre de la posibilidad de llamar explícitamente al método **flush()** de la clase EntityManager, para actualizar de inmediato la base de datos.

2.19.4.3. **EJB QL**

Como ya se ha mencionado al principio EJB QL no es simplemente otro SQL. Hay una diferencia entre si se consultan simples tablas de bases de datos o una estructura de objeto. EJB QL se ha desarrollador precisamente para esto ultimo y para la definición de su sintaxis utiliza SQL. Debe procurarse no confundir ambos.

La diferencia se hace evidente si se observa la simple sentencia EJB QL **"SELECT s FROM Socio s".** Aquí se seleccionan no solo las columnas de la tabla Socio, sino un objeto entero del tipo Socio. Si también se definieran relaciones de cantidad, se recibirían también de inmediato.

La siguiente diferencia resulta obvia cuando se observa **"SELECT distinct s FROM SocioExt s JOIN s.pedidos p WHERE p.metodospago = 0"**. Aquí se seleccionan socios cuyos pedidos tienen asignado el método de pago 0. Condición previa es una relación **@OneToMany** con el nombre pedidos en la clase **SocioExt**. Si se quisiera conseguir esto mediante SQL, se tendría que vincular la tabla de socios con la de pedidos y asegurar mediante condiciones **WHERE**, que solo se contemplarían aquellas filas en las que coinciden también los números de socios.

Con EJB QL se crea una total independencia de una estructura concreta de base de datos. El mapeo a esta lo realizan los Entity Beans con sus anotaciones, que en caso necesario se pueden sobreescribir en el Deployment Descriptor, si las columnas deben llamarse de forma algo diferente en la siguiente instalación. La aplicación resulta altamente portable. Tampoco se tienen ya más dificultades con los dialectos de bases de datos utilizados. Ahora es indiferente si se usa Oracle o DB2. Es tarea del Entity Manager traduricr las expresiones EJB QL a las SQL requeridas por el sistema receptor.

Para los ejemplos en este apartado se utilizarán las clases **Socio**, **Direccion**, **DatosBancarios**, **TarjetaCredito**, **Articulo**, **Pedido** y **Unidadpos**.

En la Figura XX se reproduce todas ellas junto con sus atributos. Tan solo la clase Unidadpospk no se trata de un Entity Bean, es más bien la clave primarioa de la clase Unidadpos (una clase anidada).

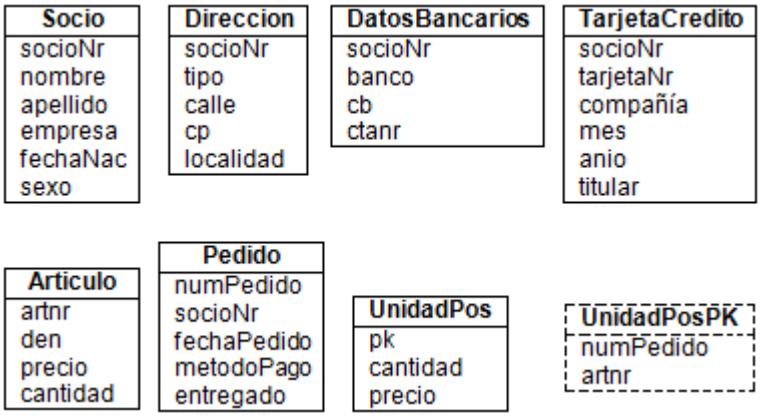


Figura XX: Modelo de datos

En la Figura XXI se puede ver un panorama general de estas clases y el modo de escribir sus atributos. Todos los atributos de relación se han escrito en cursiva. Un asterisco al final del nombre indica que se trata de un Container, que puede contener más de una referencia.

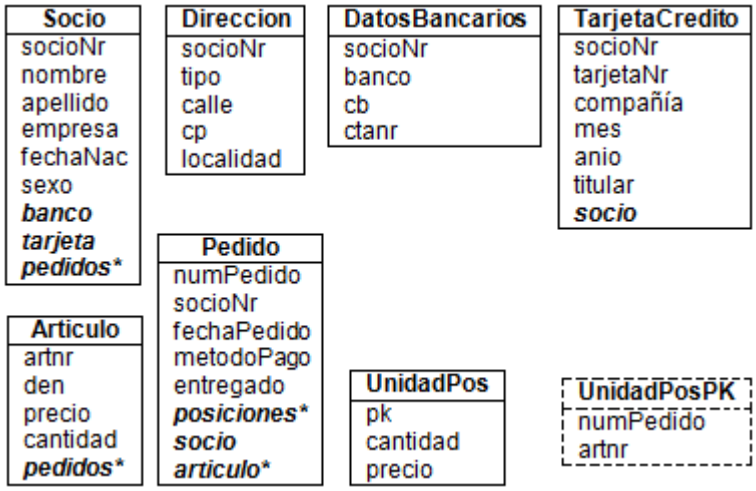


Figura XXI: Modelo de datos ampliado.

2.19.4.3.1. Sintaxis BNF

La sintaxis del lenguaje EJB QL se representa en este apartado en el formato llamado BNF. Esas siglas son la abreviatura de “Backus Naur Form” y esté generalizada. Tenemos un ejemplo en el Código 2.137.

```
select_clause : :=
SELECT [DISTINCT] select_expression {, select_expresion} *
```

Código 2.137: Ejemplo de BNF

La primera secuencia **“: : =”** introduce la descripción sintáctica. La entrada **“select-clause : :”** indica que el fragmento sintáctico **select_clause** tiene el aspecto que sigue a continuación.

Todas las palabras en mayúsculas son palabras clave y deben escribirse exactamente así, las minúsculas no son aceptadas aquí.

Las entradas opcionales van entre conrchetes. También **DISTINCT** puede omitirse. Si se ofrece una lista de selección esta debe ir entre llaves, cada opción se separa de la otra mediante un símbolo pipe (|). Un ejemplo sería la lista **{AVG | MAX | MIN | SUM}**. Un asterisco tras algún elemento de la sintaxis expresa que ese elemento debe aparecer una vez, a menudo o nunca en esa posición. En el ejemplo anterior la llave se ha utilizado erróneamente para expresar que la combinación de coma y **select_expression** puede aparecer de 0 a n veces.

2.19.4.3.2. Tipos de instrucciones

El lenguaje EJB QL conoce tres tipos de instrucciones. Junto a **SELECT**, se dispone también de **UPDATE** y de **DELETE**. La sintasis BNF correspondiente se encuentra en el Código 2.138.

```
QL_statement : : =
select_statement | update_statement | delete_statement
```

Código 2.138: Cláusula QL

Una instrucion **SELECT** sirve para la consuta de Beans y debe disponer como minimo del complemento **FROM** que indica qué fuentes deben consultarse. Además está la cláusula opcional **WHERE** con la que se puede formular una condición que restrinja los resultados, una clausula opcional **GROUP BY**, con la que se agrupan los datos consultados y después pueden consultarse por grupos, y otra cláusula también opcional **HAVING**, que aplica condiciones a grupos ya creados, y por último otra cláusula opcional **ORDER BY** que ordena el resultado al final.

```
select_statement : : =
select_clause from_clause [where_clause] [groupby_clause] [having_clause] [orderby_claus]
```

Código 2.139: Cláusula SELECT

Las instrucciones **UPDATE** y **DELETE** tienen una arquitectura más simple y consta cada una de una cláusula opcional **WHERE**, con la que se puede delimitar a qué objetos deben afectar los cambios. Si se cambia el apellido con **UPDATE** a Muller y se olvida la cláusula **WHERE** todos los socios se llamarán de repente Muller.

```
update_statement : : = update_clause [where_clause]
delete_statement : : = delete_clause[where_clause]
```

Código 2.140: Cláusula UPDATE y DELETE

2.19.4.3.3. FROM

La primera clausula evaluada es la clausula FROM. Pone a disposición los datos básicos para toda la consulta.

```
from-clause : : =
FROM identification_variable_declaration
{, {identification_variable_declaration | collection_member_declaration}}*
identification_variable_declaration : : =
range_variable_declaration { join | fetch_join } *

range_variable_declaration : : =
abstract_schema_name [AS] identification_variable

join : : =
join_spec join_association_path_expression [AS] identification_variable

fetch_join : : =
join_spec FETCH join_association_path_expression

join_association_path_expression : : =
join_collection_valued_path_expression |
join_single_valued_association_path_expression

join_spec : : =
[ LEFT [OUTER] | INNER ] JOIN

collection_member_declaration : : =
IN (collection_valued_path_expression) [AS] identification_variable
```

Código 2.141: Cláusula FROM

En su forma más simple la palabra clave **FROM** sigue una range_variable_declaration. Esta consta de un **abstract _schema_name**, de la palabra clave opcional AS y de una **identification_variable**. A primera vista esto puede parecer más complicado de lo que es en realidad. Como **abstract_schema_name** se entiende del nombre del Entity Bean. Coincide normalmente con el nombre de clase del bean, si no existe ningun otro nombre en la anotación **@Entity**. La palabre clase **AS** se explica por si solo. Una **identification_variable** es cualquier signo o palabra que no debe ser ni una palabra clave reservada ni el nombre del Entity Bean.

La cláusula **FROM** para la selección de **Socio** es simplemente **"FROM Socio s"** o **"FROM Socio AS s"**. Al introducir el nombre del Entity Bean se debe prestar mucha atención a las mayúsculas y minúsculas. Lo que se declare como variable de identificación es indiferente, aquí se ha utilizado la minúscula **s**. si se compara esto con una SQL, se da un alias. Mientras esto es opcional en SQL, es en cambio para EJB QL totalmente necesario.
Un concepto que aparece siempre en la descripción de la sintaxis es **path_expression**. Se trata de una variable de identificación seguido de un punto y después de un atributo del bean. Se hace referencia al nombre de un **Socio** mediante la expresión **"s.nombre"**. De nuevo aquí es importante escribir en mayúsculas o en minúsculas. Si se trata del atributo en un campo de relación, se habla generalmente de una **valued_path_expression**. El nombre del atributo es un **single_valued_association_field** en una relación **@OneToOne** o **@ManyToOne**, y un **collection_valued_association_field** en una relación **@OneToMany** o **@ManyToMany**.

[INNER] JOIN

En la cláusula FROM se pueden seleccionar también varios Entity Beans. Para ello se escriben consecutivamente, separados por comas. La instrucción **"select s, p from Socio s, Pedido p"** selecciona todos los socios y todos los pedidos y los vincula unos con otros. Esta vinculación se llama **JOIN** y se especifica la mayoría de las veces con una cláusula **WHERE:** si se quisiera tener por ejemplo todos los socios solo con sus propios pedidos, entonces se formula **"select s, p from Socio s, Pedido p where s.socioNr = p.socioNr".**

Si hubiera un socio que no ha realizado aún ningún pedido, no será seleccionado. Por consiguiente este **JOIN** se llama también **INNER JOIN**. Los dos beans seleccionados deben encontrarse juntos dentro del conjunto.

En EJB QL se utiliza esta forma de **JOIN** más bien poco. Es más usual definir las relaciones dentro de los beans. Así por ejemplo la clase **SocioExt** posee un atributo pedido que se trata de una colección de los pedidos de este socio. Si se quisiera tener todos los socios con los pedidos que aún no han sido entregados, se selecciona **"select s, p from SocioExt s join s.pedidos p where p.entregado is null"**. Aquí también se trata de un **INNER JOIN**, lo que debe explicitarse por escrito.

```
select s, p
from SocioExt s inner join s.pedidos p
where p.entregado is null
```

Código 2.142: Ejemplo de INNER JOIN

LEFT [OUTER] JOIN

Existe también la variante de **LEFT OUTER JOIN**, con la cual también se seleccionan las instancias de objetos que se encuentran a la izquierda de la palabra **JOIN** y para las que no hay ningún bean, a la derecha de **JOIN** al que se haga referencia. Si se quisieran todos los socios y aquellos que tienen pedidos, con sus informaciones de pedido, entonces se formula como en el Código 2.144.

```
select s, p
from SocioExt s left outer join s.pedidos p
where p.entregado is null
```

Código 2.144: Ejemplo de un LEFT OUTER JOIN

FETCH JOIN

Sirve en realidad para consultar una referencia sino para leer de inmediato los datos de una relación. Si se quieren obtener todos los socios, y en caso de que dispongan de pedidos, incluir estos también, basta con incluir todos los socios. Con la instrucción N se alcanza este objetivo. Por supuesto también se obtienen los pedidos acabados de entregar. Como se trata de una relación @**OneToMany**, que por defecto es **LAZY**, solo se leerán primero los datos de los socios. Los datos de los pedidos llegan justo después del

primer acceso. Si por el contrario se desean cargar inmediatamente los datos de los pedidos, se define un **FETCH JOIN**.

```
select s, p
from SocioExt s left join fetch s.pedidos p
```

Código 2.145: Ejemplo de FETCH JOIN

IN

Por ulitmo la **collection_memeber_declaration** con la palabra clave **IN**. Proviene de EJB 2.1 y no es otra cosa que un **JOIN** normal. Para seleccionar todos los socios que aún no tienen pedidos entregados junto con sus datos de los pedidos, se puede proceder como muestra el Código 2.146.

```
select s, p
from SocioExt s, in(s.pedidos) p
where p.entregado is null
```

Código 2.146: collection_memeber_declaration

Como el nuevo estándar también puede tratar con jerarquías de herencia de Entity Beans, es posible para la cláusula **FROM** seleccionar todos los beans de un tipo, incluso si se trata de beans derivados.

2.19.4.3.4. WHERE

La función de la cláusula **WHERE** es delimitar la búsqueda.

```
where_clause : : = WHERE conditional_expression
```

Código 2.147: Cláusula WHERE

Se transmite una condición simple o compuesta a la cláusula **WHERE**, que debe referirse a todos los objetos y datos que estén a disposición de la cláusula **FROM**.

Dentro de la cláusula **WHERE** pueden aparecer parámetros variables, cuyos valores se fijan durante el tiempo de ejecución. Se puede tratar de parámetros de posición (**?1**) o parámetros denominados **(:nombre**).

Dentro de las expresiones condicionales que pueden utilizar están: BETWEEN, IN, LIKE, NULL, EMPTY,MEMBER [OF], EXIST, ALL, ANY, SOME.
Para una comparación simple se utilizan los signos de comparación habituales +, <, >, <=, >= y <> para la diferencia. Además se permite ejecutar operaciones aritméticas simples con los operadores de comparación. Para ello se dispone de los operadores +, -, * y / así como de los prefijos * y /.

De igual manera se pueden realizar subqueries (subconsultas), además el lenguaje EJB QL conoce algunas funciones que se pueden usar en las cláusulas **WHERE** o **HAVING**.

Hay funciones que devuelven cadenas de simbolos, otras producen valores numéricos y las ultimas devuelven una fecha, una hora o ambos.

Por mencionar una función tenemos **CONCAT**, que devuelve una nueva cadena de simbolos, que resulta de la yuxtaposición de la primera y la segunda cadena de simbolos devueltas. La función **TRIM** separa los signos vacíos que aparecen por defecto al principio y al final de la cadena de simbolos. Con **LOCATE** se puede investigar si la primera cadena de simbolos está contenida en la segunda. La función **SQRT** obtiene la raíz cuadrada del parámetro transmitido, las funciones **CURRENT_DATE**, **CURRENT_TIME** y **CURRENT_TIMESTAMP** devuelven la fecha actual, la hora o un Timestamp que consta de la fecha y horas actuales.

2.19.4.3.5. GROUP BY

Después de seleccionar con la cláusula **FROM** el conjunto de datos relevantes para la consulta actual y de restringir con ayuda de la cláusucla opcional **WHERE** este conjunto, se pueden ordenar los datos restantes con la ayuda de **GROUP BY** en diferentes grupos.

Cada grupo se construirá mediante los atributos entrados en la cláusula **GROUP BY**. Si se utiliza en lugar de un atributo una varialbe de identificación, se establece una referencia a un determinado tipo de bean, lo que es equiparable a la entrada de la clave primaria de la correspondiente clase Bean.

Una vez se han creado los grupos se pueden seleccionar en la cláusula **SELECT** tan solo aquellos elementos que aparecen en las agrupaciones de la entrada **GROUPBY**. Si se agrupan por ejemplo los artículos según su cantidad, será la cantidad lo único que se obtendrá como resultado de la lista. En la cláusula **SELECT** solo puede aparecer por lo tanto la cantidad.

```
select a.cantidad
from Articulo a
group by a.cantidad
```

Código 2.148: Ejemplo de GROUP BY

El resultado de la consulta en el Código 2.148 son tantos grupos de artículos, como articulo con diferentes cantidades hay. Todos los artículos con la misma cantidad pertenecen a un grupo. Lo único común dentro de cada uno de estos grupos es la cantidad, por ese motivo esta información es lo único que se puede seleccionar.

Otro ejemplo es la consulta sobre el precio medio de cada artículo por cantidad. También aquí se crea un grupo por cantidad y se aplica la función de grupo **AVG**.

```
select a.cantidad, avg(a.precio)
from Articulo a
```

```
group by a.cantidad
```

Código 2.149: Precio medio por cantidad

2.19.4.3.6. HAVING

La cláusula **HAVING** es la cláusula **WHERE** de la cláusula **GROUP BY**. En otras palabras, selecciona los grupos que se crearán con ayuda de **GROUP BY**.

Primero la cláusula **FROM** comunica el total de todos los datos que deben ser evaluados. Después la cláusula **WHERE** restringe esta cantidad, antes de que esta sea compactada en grupos por **GROUPBY**. La cláusula **HAVING** que sigue a continuación reduce de nuevo la cantidad resultante, por lo que es comparable a **WHERE**. La diferencia consiste en que **HAVING** se refiere a los grupos creados, mientras que **WHERE** a cada una de las filas coincidentes con la búsqueda que la aplicación ha encontrado.

En la descripción de la cláusula **GROUPBY** ya se ha transmitido el precio medio de todos los artículos por cantidad. Suponiendo que se quisiera esta información solo para aquellos artículos, cuyo precio medio por cantidad sea de más de 10 euros, entonces se pueden eliminar con la cláusula **HAVING** el resto de grupos. Esto no se puede realizar con la cláusula **WHERE**, porque el precio medio por cantidad se podrá comunicar solamente cuando se hayan ordenado los artículos según su cantidad.

```
select a.cantidad, avg(a.precio)
from Articulo a
group by a.cantidad
having avg(a.precio) > 10
```

Código 2.150: Ejemplo de HAVING

Si se utiliza la cláusula **HAVING** sin **GROUPBY** todos los datos seleccionados hasta ahora se contemplarán como un grupo, bajo la condición que se aplique la cláusula **HAVING**. El resultado entonces puede constrar de ninguna o de una fila como máximo. El estándar Java EE dice expresamente que los fabricantes no deben soportar esta combinación **HAVING** sin **GROUPBY**. Por lo tanto tampoco debe utilizarse.

2.19.4.3.7. SELECT

Aunque se empiece a escribir una consulta con la cláusula SELECT, se la explica ahora, porque su función consiste en seleccionar de los datos que aún quedan, columnas u objetos determinados.

Si se quiere hacer referencia a un solo atributo, se debe definir en la cláusula **SELECT** la ruta a este atributo. El nombre de un Socio se obtiene mediante **“select s.name from**

Socio s”. Si el atributo se trata de una relación **@OneToOne** o **@ManyToOne**, es posible moverse con ayuda de la ruta correspondiente también a los atributos del bean relacionado. El nombre de un socio desde su pedido se obtiene a partir de **“select p.socio.namefromPedidoExt.p”**.

DISTINCT

Si en una consulta se utiliza la palabra clave **DISTINCT** se resumen filas idénticas en una sola. Si se quiere tener todos los socios para los que existe también un pedido y no hay ninguna relación definida entre Socio y Pedido, con **“select s from Socio s, Pedido p where s.socioNr = p.socioNr”** se obtendrían todos estos socios, pero si uno de ellos ha realizado cinco pedidos, aparecerá cinco veces en la lista. Para limpiar estas repeticiones, se utiliza **DISTINCT**.

```
Select distinct s
from Socio s, Pedido p
where s.socioNr = p.socioNr
```

Código 2.151: Ejemplo de DISTINCT

La palabra clave también se puede utilizar en combinación con las funciones **AVG**, **MIN**, **MAX**, **SUM**, **COUNT**. Aquí se aístan aquellos valores iguales. Si se desea obtener la media de todos los precios de los artículos, se puede transmitir con **“avg(distinct a.precio)”**. La función **AVG** recibe tan solo precios diferentes. **DISTINCT** también es posible con **MIN** y **MAX** pero no tiene mucho sentido.

OBJECT

Con esta palabra clase se hace referencia explícita al hecho de que aquí se selecciona un objeto. De hecho es opcional y se puede obviar. La cláusula **“select object (a) from Articulo a”** es equivalente a **“select a from Articulo a”**.

Tipos de resultados

Los resultados de una cláusula **SELECT** se corresponden siempre con el tipo de elementes seleccionados.

Si se hace referencia atributos simples como el nombre, el tipo de resultado coincidirá con el tipo de esta columna, es decir **String**.

Si se hace referencia a un tipo de Bean, en el que se ha seleccionado una varialbre de identificación, el resultado se corresponderá con la clase del Entity Bean. En **“select a from Articulo a”** la clase es Articulo.

Si se selecciona una relación de conjunto como por ejemplo **"select a.pedidos from ArticuloExt a"** el tipo del resultado será del tipo del conjunto seleccionado, es decir **PedidoExt**.

Si se utiliza una función como **AVG** o **COUNT**, el resultado será del tipo de la función correspondiente que puede volver a ser dependiente de los parámetros de las funciones. **MIN** devuelve un resultado numérico para valores numéricos, pero un **String** para cadenas de símbolos.
No es posible mezclar diferentes tipos de resultados. Si se intenta seleccionar un solo valor con un conjunto se producirá un error de sintaxis. La siguiente consulta **"select a.artnr, a.pedidos from ArticuloExt a"** no está permitida, porque solo hay un número de articulo por articulo, pero este puede disponer de varios pedidos. Tampoco a la inversa se puede acoplar un solo valor con el resultado de una función aditiva, puesto que la función devuelve un solo valor para todas las filas coincidentes, un **"select a, min(a.precio) from Articulo a"**. Un solo valor como el número de artículo solo se puede seleccionar junto con el resultado de una función aditiva cuando el conjunto encontrado se resume en grupos y el valor individual es una parte del concepto de agrupación, como se reproduce en el Código 2.152.

```
select a.artnr, count(b)
from ArticuloExt a join a.pedidos p
group by a
```

Código 2.152: Selección de un solo valor y una función aditiva

NEW

El resultado de una cláusula **SELECT** también puede ser la instancia de una clase cualquiera. Puede ser cualquiera, porque aquí no solo se pueden utilizar las clases Entity Bean. Para crear estas instancias se utiliza en la cláusula **SELECT** la palabra clave **NEW**, como en Java. A continuación sigue la llamada del constructor correspondiente, para lo que el nombre de la clase siempre debe estar totamente cualificado. Como paramtero se pueden transmitir al constructor cualesquiera elementos seleccionados, solo deben corresponderse con los tipos esperados.

```
select new server.tiempo(t.mes, t.anio)
from TarjetaCredito t
```

Código 2.153: Ejemplo de NEW

Funciones de adición

En la cláusula **SELECT** se pueden utilizar las cinco funciones **AVG**, **MIN**, **MAX**, **SUM** y **COUNT**. Siempre se refieren a las filas seleccionadas y a partir de ahí dan un único resultado.

Una vez aparece una de estas funciones en la cláusula **SELECT**, el resultado final de la consulta puede constar de cómo máximo una solo fila, excepto si se ha trabajado simultáneamente con **GROUPBY**. Si se agrupa, las funciones de adicion se aplicarán a cada grupo y se obtendrán como resultado tantas filas como grupo seleccionados. Los ejemplos correspondientes se encuentran en la explicación sobre **GROUPBY**.

Solo se puede transmitir una relación de conjuntos a la función **COUNT**, el resto de funciones esperan atributos simples.

Los paramteros de las funciones **SUM** y **AVG** deben ser numéricos. En los parámetros de la función **MIN** y **MAX** debe tratarse de tipos de datos ordenables, como por ejemplo **String**, **Integer**, **BigDecimal** o **Date**.

El resultado de la función **COUNT** es del tipo **Long**. Las funciones **MIN** y **MAX** devuelven el tipo, que se les ha transmitido como parámetro. En **AVG** siempre es **Double**, en **SUM** es **Long**, si el parámetro es un número entero, entonces **Double**. Si a la función **SUM** se transmiten parámetros del tipo **BigInteger** o **BigDecimal**, el resultado también será del tipo correspondiente.

Si se transmite a una de las funciones **AVG**, **MAX**, **MIN** o **SUM** un conjunto vacío, el resultado será **NULL**. Solo la función **COUNT** devolverá en ese caso el valor 0.

Los valores nulos se eliminan antes de la transmisión a una de estas funciones. Por lo tanto, en las funciones **AVG**, **MIN**, **MAX** o **COUNT** no desempeñan ningún papel.

2.19.4.3.8. ORDER BY

Después de seleccionar todos los datos, se puede ordenar el conjunto coincidente con la búsqueda según uno o varios criterios.

Los elementos según los que se debe realizar la ordenación, deben estar al final del conjunto, pero no es necesario que la cláusula **SELECT** lo muestre obligatoriamente. Si solo se quiere tener la denominación del artículo, ordenado según su precio, también es posible.

```
select a.den
from ArticuloExt a
order by a.precio
```

Código 2.154: Ejemplo de ORDER BY

Los elementos afectados no deben ser relacionados de conjunto. Solo se permiten atributos simples o relación **@ManyToOne** y **@OneToOne**. El Código 2.155 selecciona todos los pedidos y el número del socio que ha hecho el correspondiente pedido y ordena la muestra de resultados según los números de socios. No es necesario referirse explícitamente al número de socio en la cláusula **ORDER BY**. Escribiendo s.socio el sistema ordena automáticamente los datos según la clave primaria del objeto correspondiente.

```
select b.pedidoNr, p.socio.socioNr
from PedidoExt p
order by p.socio.
```

Código 2.155: Ordenacion según el número de socio

Por defecto se ordena siempre en orden creciente. Este puede modificarse entrando la palabra clave **DESC**.

Si se entran varios conceptos de orden, el conjunto se ordenará según el primer concepto especificado. Si los valores no son precisos, los valores iguales se ordenarán siguiendo el segundo término de orden, etc. En cada criterio de ordenación es posible una entrada **ASC** o **DESC**, si no se especifica nada prevalecerá **ASC** y se ordenará en dirección ascendente.

2.19.4.3.9. UPDATE

Con el lenguaje de consulta EJB QL también se pueden llevar a cabo cambios masivos. Para ello se utiliza la instrucción **UPDATE** pero debe tenerse en cuenta eventualmente que los Beans acabados de seleccionar que ya se encuentran en el acceso han dejado de ser actuales. Por ese motivo es posible transmitir este tipo de Beans al método **refresh()** del EntityManager para actualizarlos. Por otro lado es recomendable ejecutar estos cambios masivos siempre al principio o como única acción dentro de una transacción.

Una instrucción **UPDATE** solo se puede relacionar siempre con una clase bean determinada. Tras la palabra clase **SET** sigue la lista de atributos que deben ser modificados, con sus correspondientes atribuciones de valores. Se puede entrar opcionalmente una cláusula **WHERE**, pero la mayoría de las veces esto no es necesario. Si se omite la cláusula WHERE, los cambios se referirán a todos los beans del tipo correspondiente.

```
Update Articulo a
Set a.precio = a.precio + 1
Where a.cantidad < 10
```

Código 2.156: Ejemplo de UPDATE

Una instrucción de este tipo se puede definir mediante la anotación **@NamedQuery** dentro de una clase Entity Bean. Para ejecturala se utiliza primero el método **createNamedQuery()** de la clase EntityManager y se le devuelve una instancia de **Query**. Esta ofrece el método **executeUpdate()**, que ejecuta la instrucción.

2.19.4.3.10. DELETE

La instrucción **DELETE** elimina una o más filas de la base de datos. Como la instrucción **UPDATE**, puede definirse solo para un tipo determinado de Entity Bean y debería ejecutarse siempre en una transacción propia o al inicio de esta.

Siempre empieza con las palabras clave **DELETE FROM** seguidas de la clase bean a la que debe aplicarse. De modo opcional se puede utilizar una cláusula **WHERE**, para restringir los beans a eliminar. Si se elimina la cláusula **WHERE**, se eliminan todos los beans del tipo correspondiente.

```
delete from SocioExt s
where s.pedido is empty
```

Código 2.157: Ejemplo de DELETE

En el Código 2.157 se eliminan todos los socios que todavía no han pedido nada.

Una instrucción **DELETE** de este tipo solo se ejecutará en la clase indicada y en todas las clases derivadas de esta. Las clases dependientes no se eliminan. Hay una diferencia entre si se elimina en Entity Bean mediante el método **remove()** del EntityManager y se reenvía esta orden mediante **cascade =CascadeType.REMOVE** a las clases dereivadas, o si se utiliza la instrucción **DELETE**. Si se ha provisto en la base de datos a la tabla correspondiente de la regla RI **ON DELETE CASCADE**, el sistema de base de datos asumirá la eliminación de la información dependiente.

Una instrucción de este tipo se puede definir dentro de una clase Entity Bean mediante la anotación @**NamedQuery**. Para ejecutarla se utiliza el método **createNamedQuery()** de la clase EntityManager y se recibe una instancia de **Query**. Esta ofrece el método **executeUpdate()**, que ejecuta la instrucción. Ocurre lo mismo que con la instrucción **UPDATE**. De hecho también se tiene que utilizar el método **executeUpdate()** para **DELETE**, no existe un **executeDelete()**.

2.20. Transacciones

2.20.1. Panorama general

Ninguna aplicación moderna se presenta completamente sin transacciones. Con el uso de las transacciones se asegura que una función compleja del sistema se ejecuta o no. Esto significa que los datos en la base de datos solo se modifican cuando se puede realizar esta tarea por completo y sin errores. Si mientras tanto se produjera un error, la transacción se ocuparía de que todas las modificaciones realizadas hasta el momento en la base de datos se cancelaran. Si no hubiera todavía ninguna protección de transacción se tendría que volver a establecer los datos modificados en caso de una cancelación, mediante la última copia de seguridad. Un estado, totalmente insostenible para una aplicación online.

La protección de transacción se refiere siempre al contenido de la base de datos no al estado de las clases Java en la memoria principal. Una vez se ha llamado un método en un Session Bean, se inicia esta transacción. Memoriza prácticamente todo lo que el EntityManager modifica en la base de datos durante el proceso. Siempre que se coloca una cláusula **UPDATE** o **INSERT** se ejecutará esto en la base de datos. Según la

configuración estos cambios permanecerán primero invisibles para todos los usuarios, porque las correspondientes filas de la base de datos para la transacción modificadora están bloqueadas. El resto de transacciones que deseen casualmente trabajar con los mismos datos deben esperar, hasta que la transacción que realiza los cambios haya terminado con su trabajo. También se puede configurar una base de datos de manera que las transacciones lectoras reciban los datos aún no liberados de otra transacción. Entonces sin embargo no se puede asegurar que los datos al final se introduzcan exactamente así en las tablas. Si finaliza el método llamado, finaliza normalmente también la transacción. Si no se produce ningún error, se llega a lo conocido como **COMMIT** y todos los cambios producidos se liberan para el resto. En caso de error se produce por el contrario un **ROLLBACK**, lo que dispone a la base de datos a anular todas las modificaciones. Según los cambios que haya habido en una transacción, esto ocupará más o menos.

Si dos usuarios independientes entre sí llaman simultáneamente el mismo método desde el servidor, se inicia para ambos una transacción propia. Cada uno realiza sus cambios independientemente del otro. Si ambos necesitan la misma fuente el que llega más tarde tendrá que esperar hasta que el otro termine. En este punto se puede llegar a un deadlock. La transacción de un usuario A necesita un conjunto, que el usuario B ha modificado pero aun no ha liberado y viceversa. Los sistemas de bases de datos reconocen este tipo de conflictos y por lo tanto los solucionan automáticamente, deteniendo una o ambas transacciones, lo que conduciría al correspondiente mensaje de error para el usuario.

Mediante la protección de transacción se garantiza que los datos en la base de datos siempre se encuentran en un estado consistente. Las modificaciones realizadas en la memoria principal no se ven afectadas. Si ya se ha memorizado en un atributo de la clase bean que el pedido ha terminado correctamente y ya ha sido guardado, y se produce entonces un rollback, el atributo afectado no se cancelará automáticamente del mismo modo que el sistema de base de datos. Al final de la sesión se describirá como los Session Beans pueden informarse del éxito o del fracaso de la transacción, la cancelación de los atributos será tarea del desarrollador.

El comportamiento descripto hasta aquí, en el que se inicia la transacción con una llamada al método y se mantiene hasta el final del método no tiene por qué ser siempre así. Se puede cambiar este comportamiento estándar mediante la entrada de anotaciones concretas. Más adelante se describirán todas las posibilidades de las que se dispone. Es posible incluso manejar manualmente la administración de la transacción, entonces hablamos de Bean Managed Transaction. Esto solo es necesario en casos excepcionales y no muy recomendables en cuanto a una arquitectura de la aplicación.

2.20.2. Container Managed Transaction (Contenedor Manejador de Transacciones)

Bajo el concepto Container Managed Transaction (CMT) se entiende que el servidor de aplicaciones se ocupa del control completo de la transacción según las prescripciones del desarrollador. El sistema mismo reconoce cuándo iniciar una transacción y cuándo finalizarla. Para ello el desarrollador da diferentes atributos de transacción en el nivel de métodos, a través de los cuales puede controlar este comportamiento. Si no se denomina ningún atributo, servirán los supuestos estándar. Según estos cada método de un Session

Bean, llamado desde un cliente, necesitará protección de la transacción. Si ya se ha iniciado una transacción para el usuario, el método llamado la utilizará, si no es así creará una nueva transacción. Si finaliza el método que ha creado la transacción también finaliza la transacción. Si finaliza el método que ha creado la transacción también finaliza la transacción. Si durante la transacción se llega a una excepción o el desarrollador ha comunicado al administrador de la transacción que todos los cambios deben cancelarse, al final se llega a un **ROLLBACK**, sino a un **COMMIT**. Este comportamiento es el predefinido para todos los Message-Driven Beans.

2.20.2.1. @**TransactionAttribute**

Cada método publico de un Session Bean o Message-Driven Bean puede dotarse de la anotación **@TransactionAttribute**. De esto modo se configura cómo controla el servidor de aplicaciones el comportamiento de la transacción.

```
public enum TransactionAttributeType
{
MANDATORY,
REQUIRED,
REQUIRES_NEW,
SUPPORTS,
NOT_SUPPORTED,
NEVER
}

@Target({METHOD, TYPE})
public @interface TransactionAttribute
{
TransactionAttributeType value() default TransactionAttributeType.REQUIRED;
}
```

Código 2.158: @TransactionAttribute

La anotación puede realizarse en el nivel de los métodos o de las clases. Si se quisiera determinar que cada método de un Session Bean siempre conduzca a una nueva transacción se puede proceder como indica la Tabla 2.160.

```
@Stateful
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public class testContadorBean implements testContadorRemote
{
…
}
```

Código 2.159: Ejemplo de control de una transacción en la extensión de clase

Normalmente se entra esta anotación en el nivel de los métodos. Sin embargo se debe tener presente que estos atributos de transacción solo son efectivos cuando el método correspondiente se llama mediante el servidor de aplicaciones. Este es el caso cuando un cliente, desde dónde se encuentre, ha realizado un lookup en la interfaz local o remota de la clase bean y llama entonces un método. Si por el contrario nos encontramos ya en un

Session Bean y este llama su propio método, al servidor de aplicaiones no se le comunica nada al respecto. Se trata al fin y al cabo de una llamada de método normal. El atributo de la transacción del método así llamado se ignora. En el Código 2.160 se reproduce el ejemplo correspondiente. Aunque ambos métodos dicen que ambos necesitan una nueva transacción, al llamar el primer método se inicia tan solo una transacción, aunque durante su ejecución se utilice también el segundo método. Los atributos de la transacción actúan por lo tanto siempre en una llamada a través del cliente, en la que un Session Bean puede ser de todas formas cliente de otro.

```
@Stateful
public class testContadorBean implements testContadorRemote
{
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void primerMetodo()
{
//si un cliente llama este método, se inicia una nueva transacción
segundoMetodo();
}

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void segundoMetodo()
{
// Si este método es llamado por primerMetodo()
// no se crea ninguna transacción nueva.
// Se utilizará la transacción del primer método.
// Si es el cliente el que llama este método, se iniciará
// una nueva transacción.
…
}
}
```

Código 2.161: Llamada de un método Business propio

Como alternativa se pueden depositar los atributos de la transacción también en el Deployment Descriptor. Como es habitual se sobreescriben así todos los datos de la anotación. La etiqueta **<assembly-descriptor>** del archivo **ejb-jar.xml** consta de varias entradas de **<container-transaction>,** que asignan un atributo de transacción a uno o más métodos. La etiqueta **<method>** puede aparecer varias veces en una **<container-transaction>**. Consta por su parte de como minimo las entradas **<ejb-name>** y **<method-name>**, con lo cual se determina, para qué método y desde qué bean se tiene que realizar aquí una entrada. Como es posible que en un bean puedan haber varios métodos con el mismo nombre, que solo se diferencian entre si por sus parámetros, la etiqueta **<method>** puede disponer también de la entrada **<method-param>**. Aquí se determinan cada uno de los parámetros con ayuda de **<method-param>**. Después de entrar uno o más métodos se fija finalmente el auténtico atributo de transacción mediante **<trans-attribute>**.

```
<ejb-jar
xmlns=”http://java.sun.com/xml/ns/javaee”
xmlsn:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:SchemaLocation=”http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd”
versión=”3.1”>
<enterprise-beans>
<session>
<ejb-name>AdministracionArticuloBean</ejb-name>
…
```

```
</session>
</enterprise-beans>
<assembly-descriptor>
<container-transaction>
<method>
<ejb-name>AdministracionArticuloBean</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
<method>
<ejb-name>AdministracionArticuloBean</ejb-name>
<method-name>getArticulo</method-name>
</method>
<method>
<ejb-name>AdministracionArticuloBean</ejb-name>
<method-name>getArticulo</method-name>
<method-params>
<method-param>int</method-param>
</method-params>
</method>
<trans-attribute>NotSupported</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

Código 2.162: Deployment Descriptor

El Código 2.162 es un bean ejemplo de la definición del atributo de transacción para el Session Bean **AdministracionArticuloBean**. Primero encuentra una entrada **<method>** para este bean, en el que el **<method-name>** es un asterisco. Esto significa que se incluyen todos los métodos de este bean. Como atributo de la transacción se da finalmente **REQUIRED**, lo que significa en total, que todos los métodos de este bean necesitan protección para la transacción. Después sigue otra entrada para el mismo bean, que se refiere dos veces al método **getArticulo**. En el primer caso el método no espera ningún parámetro, en el segundo un valor int. El atributo de transacción para los métodos **getArticulo()** y **getArticulo(int)** se llama **NotSupported**, que dice que ninguno de los dos métodos ofrece ninguna protección de transacción. Al fin por lo tanto solo quedan dos métodos sin protección.

2.20.2.2. **NOT_SUPPORTED**

Si un cliente llama un método con este atributo, se suspende la actual transacción, es decir no se continúa en principio con su ejecución. Todas las acciones que se ejecuten ahora carecerán de protección de transacción. Si finaliza el método, volverá a activarse cualquier posible transacción suspendidad anteriormente.

Tiene sentido utilizar este atributo para accesos de tan sólo lectura. Si un método proporciona solo datos y los transmite siempre al cliente, no se necesario iniciar una transacción. Esto contribuye a un mejor rendimiento.

El atributo de la transacción lleva el nombre **NotSupported** en el Deployment Descriptor.

2.20.2.3. **SUPPORTS**

Si se equipa un método con este atributo, apoyará a una posible transacción. Si no existiera ninguna, también se ejecutaría el método sin protección de transacción.

Como los atributos de la transacción solo actúan en una llamada de un cliente, puede imaginar el siguiente escenario: una página JSP llama un método business o un session bean provisto con un atributo de la transacción **Required**. De este modo se inicia una transacción. El método así llamado utiliza otro session bean también como cliente y llama un método con el atributo **Supports** a través de su interfaz local o remota. Como existe protección para la transacción y esta es soportada por los dos métodos llamados, todos los cambios de la base de datos funcionan bajo esta transacción. Si por el contrario la página JSP llama un método caracterizado con **Supports** no existe allí ninguna protección de transacción.

El atributo de transacción lleva el nombre **Supports** en el Deployment Descriptor.

2.20.2.4. **REQUIRED**

Este es el atributo de transacion utilizado más frecuentemente. Dice que el método necesita protección de transacción. Si durante la llamada aún no existe ninguna transacción se iniciará una, si no, se seguirá utilizando la existente.
El atributo de la transacción lleva el nombre de **Required** en el Deployment Descriptor.

2.20.2.5. **REQUIRES NEW**

Si es necesario ejecuta un método llamándolo a través de un cliente en una transacción autónoma, se le debe proveer del atributo **RequiresNew**. Si hasta ahora no funcionaba ninguna transacción para el usuario, se iniciará una. Si el cliente que realiza la llamada ya tiene protección de la transacción, se suspende y se inicia una nueva transacción.
Una vez el método proveisto con **RequiresNew** finaliza, se detiene también la transacción iniciada de nuevo y se vuelve a ejecutar la transacción suspendida anteriormente. Carece de importancia para la transacción externa cómo finaliza la interna. Incluso si una transacción finaliza con un **ROLLBACK**, puede llegar la otra a un **COMMIT** y viceversa.
El atributo de transacción se llama **RequiresNew** en el Deployment Descriptor.

2.20.2.6. **MANDATORY**

El atributo **Mandatory** determina que el método llamado debe ser siempre parte de una transacción existente. Debe haber ya una transacción iniciada, si se utiliza uno de estos métodos por un cliente. Si por el contrario no existe ninguna protección de transacción, se interrumpe la llamada del método y se produce una **javax.ejb.EJBTransactionRequiredException**.

El atributo de transacción lleva el nombre **Mandatory** en el Deployment Descriptor.

2.20.2.7. **NEVER**

El atributo de transacción **Never** es algo particular. Si se llama un método así caraterizado desde un cliente con protección de transacción, se producirá una **EJBException**. Solo si no existe ninguna protección se puede trabajar el método como es debido. Tamposco se inicia ninguna nueva transacción para el método. Todos los cambios en la base de datos, que se ejecutan en un método de este tipo se escriben de inmediato en las tablas de modo visible y permanecen allí independientemente de lo que ocurra en el método.

2.20.3. Administrador de Bean de Transacciones

Si fuera necesario alguna vez controlar uno mismo el comportamiento de la transacción, se puede proveer un session bean con la anotación **@TransactionManagement** y a determinar, que el bean debe hacerse cargo del control.

```
public enum TransactionManagerType
{
BEAN,
CONTAINER
}
@Target({TYPE})
@Retention(RUNTIME)
public @interface TransactionManagement
{
TransactionManagerType value() default TransactionManagerType.CONTAINER;
}
```

Código 2.163: @TransactionManger

Si entonces se llama un método de este bean el Contenedor ya no se preocupará de la protección de la transacción. Si existiera alguna porque el cliente se ejecuta bajo una transacción, se seguirá utilizando. Si por el contrario el método llamado quiere iniciar una transacción debe poseer una instancia de la clase **userTransaction**. Lo más fácil es hacerlo mediante el método **getUserTransaction()** de la clase **EJBContext**.

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class testContadorBean implements testContadorRemoto
{
@Resource
SessionContext ejbContext;

public void unMetodo()
{
boolean errorAparecido = false;
userTransaction ut = ejbContext.getUserTransaction();
try
{
ut.begin();
…
if( errorAparecido )
ut.rollback();
```

```
else
ut.commit();
}
catch( IllegalStateExceptio e )
{…}
catch( SecurityException e )
{…}
catch( NotSupportedException e )
{…}
catch( SystemException e )
{…}
catch( RollbackException e )
{…}
catch( HeuristicMixedException e )
{…}
catch( HeuristicRollbackException e )
{…}
}
}
```

Código 2.164: Fijar propiedades específicas

En el Código 2.164 se repoduce un ejemplo de un Stateless Session Bean que se ocupa él mismo de la protección de su transacción. En este caso la transacción debe finalizar en el mismo método en el que se ha creado. Esta restricción no es válida para Stateful Session Beans.

En lugar de ir al **EJBContext**, se puede inyectar directamente en un bean una **userTransaction**.

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class testContadorBean implements testContadorRemoto
{
@Resource
UserTransaction ut;
}
```

Código 2.165: UserTransaction como fuente

La tercera posibilidad para llegar a un UserTransaction consiste finalmente en colocar un lookup en **“java:comp/env/userTransaction”**. De este modo un cliente también puede participar en la administración de la transacción desde fuera de un servidor de aplicaciones. De este modo es posible iniciar una transacción en el cliente, llamar uno o más métodos de los Session Beans y después finalizar la transacción en el cliente.

```
UserTransaction ut;
try
{
InitialContext ctx = new InitialContext();
ut = (UserTransaction) ctx.lookup(“java:comp/env/UserTransaction”);
}
catch( NamingException e )
```

```
{…}
```

Código 2.166: Lookup en UserTransaction

Los métodos más importantes de la clase **UserTransaction** son **begin()** para iniciar una transacción, **commit()** para finalizar limpiamente y **rollback()** para anular todos los cambios. Adeas también existe el método **setRollbackOnly()**, que finaliza aún la transacción pero ya indica que debe terminar en todo caso con un rollback. Mediante **setTransacttionTimeout()** se puede determinar tras cuántos segundos se debe detener la transacción. Si no utiliza este método prevalecerán los valores estandasr del manager de la transacción que se pueden configurar mediante el servidor de aplicaciones. Por último, con el método **getStatus()** se puede conocer el estado actual de la transacción. Devuelve constantes desde la interfaz **javax.transaction.Status**. Valores interesantes son **STATUS_ACTIVE**, **STATUS_COMMITED** o **STATUS_MARKED_ROLLBACK**.
También un Message-Driven Bean puede asumir él mismo el control de la transacción. Todo lo que se ha mostrado aquí se puede aplicar sin cambios al método **onMessage()** de este tipo método de bean. Como en los Stateless Session Beans la transacción debe terminar en el mismo método en el que se ha iniciado. No es posible recoger varias llamadas del método **onMessage()** en una transacción.

2.20.4. EJBs sin transacción

No es necesario que para cada método exista protección de transacción. Un Stateless Session Bean para la validación de un número de tarjeta de crédito o el número de un billete no necesita ninguna transacción. Si un bean consta tan solo de estos métodos, se puede describir toda la clase con **NOT_SUPPORTED**. Esto no significa obligatoriamente que los métodos no realicen ningún tipo de acceso a la base de datos. Siempre y cuando estos sean tan solo de lectura por ejemplo para comprobar la existencia de un número de cuenta, también funcionará correctamente sin protección de la transacción. En todo caso es mucho más grave olvidar la protección de transacción de un método que la necesita que dársela a un método que no va a sacarle ninguna utilidad. Caracterizar todos los métodos con **NOT_SUPPORTED** para darles después por separado un **REQUIRED** es peor que preocuparse primero de que todos los métodos tengan protección de transacción y después cambiar algunos de ellos. Los valores estándar de cada aportación apoyan más bien este segundo modo de proceder.

2.20.5. Accesos competitivos

En este apartado se probarán y se evaluarán diferentes procedimientos con los que se pueden evitar accesos competitivos en una aplicación. Para ello se crean dos tablas de bases de datos con las que se puede probar el comportamiento de un sistema cuando dos clientes intentan a la vez acceder a los mismos datos.

```
CREATE TABLE TEST_CONTADOR
{
NR INTEGER NOT NULL PRIMARY KEY,
```

```
VERSION INTEGER,
VALOR INTEGER NOT NULL
};

CREATE TABLE TESTTABLA
{
NR INTEGER NOT NULL PRIMARY KEY
};
```

Código 2.167: Tablas para test

La tabla **TEST_CONTADOR** consta de las columnas **NR** y **VALOR**. La columna **VERSION** se explicará más tarde. La clave primaria es **NR** y contiene en los siguientes tests siempre el número 1. Un cliente debe desempeñar la tarea de leer quinientas veces la línea con el número 1, aumentar el valor encontrado en una unidad y reescribir este cambio en la base de datos. Además debe realizar una entrada en la **TABLATEST** con el valor encontrado originalmente, cuya única columna es también su clave primaria, que como ya es sabido debe ser precisa.

Si se inician dos clientes de este tipo a la vez, ambos leen la misma fila de la tabla **TEST_CONTADOR**. Si se da por hecho que ambos trabajan paralelamente, los dos leerán el mismo valor, por ejemplo 100. Ambos lo aumentarán hasta 101 e intentarán reescribirlo. Solo esto no tiene por qué conducir a ningún error. Aunque hayan actuado los dos clientes, en la base de datos estará el valor 101. Será una vez intenten crear una fila con 101 en la **TESTTABLA** cuando fallará el segundo. Solo si se procura que sea siempre solo uno el que lea la fila de la tabla **TEST_CONTADOR** y trabaje con ella, se podrá resolver la tarea sin errores.

Para ejecutar los tests se programa un Stateful Session Bean con el nombre **TestContadorBean**. Mediante el método **initBaseDatos()** se prepara la base de datos para el test. Durante esta preparación se elimina el contenido de ambas tablas y después se inserta en la tabla **TEST_CONTADOR** una nueva fila con el número 1, versión 1 y valor 1.

```
package server.tr;

import javax.ejb.*;
import javax.persistence.*;

@Stateful
public class TestContadorBean implements TestContadorRemote
{
@PersistenceContext( unitName=”JavaEE” )
EntityManager manager;

public void initBaseDatos( )
{
manager.createNamedQuery(“TestContador.borrarTodo”).executeUpdate();
manager.createNamedQuery(“TestTabelle.borraTodo”).executeUpdate();
CounterTestVersion ct = new CounterTestVersion();
ct.setNr(1);
ct.serVersion(1);
ct.setValor(1);
manager.persist(ct);
}
…
}
```

Código 2.168: TestContadorBean

Para la tabla **COUNTER_TEST** se escribe el correspondiente Entity Bean con el nombre **TestContador**.

```
package server.tr;
import javax.persistence.*;
@Entity
@NamedQueries({  @NamedQuery(name=”TestContador.borrarTodo”,
query=”delete from TestContador ct”)  })
@Table(name=”TEST_CONTADOR”)
public class TestContador
{
private int nr;
private int valor;
@Id
public int getNr()
{
return nr;
}
public void setNr(int nr)
{
this.nr = nr;
}

public int getValor()
{
return valor;
}
public void setValor(int valor)
{
this.valor = valor;
}
}
```

Código 2.169: EntityBean TestContador

Falta tan sólo un Entity Bean para la tabla de prueba. También esta clase se presenta muy fácilmente. Para poder contar al final cuantas filas se han escrito en la tabla de prueba, la clase **TestTabla** dispone de la consulta **“TestTabla.cantidadFilas”,** que con ayuda de **count(*)** comunica la cantidad.

```
package server.tr;

import javax.persistence.*;

@Entity
@NamedQueries({
@NamedQuery(name=”TestTabla.borrarTodo”,
query =”delete from TestTabla tt”),
@NamedQuery(name=”TestTabla.cantidadFilas”,
query =”select count(*) from TestTabla tt”)
})
@Table(name=”TESTTABLA”)
public class TestTabla
{
private int nr;

@Id
public int getNR()
{
```

```
return nr;
}
public void setNr(int nr)
{
this.nr = nr;
}
}
```

Código 2.170: Entity Bean TestTabla

2.20.5.1. **Trabajar sin bloqueo de conjuntos**

En el primer test se tiene que trabajar completamente sin bloqueos, para mostrar que ocurre cuando varios clientes trabajan simultáneamente las mismas fuentes. Para ello se guarda un método con el nombre **testContadorSinLock()** en el **TestContadorBean**, que se reproduce en el Código 2.171.

```
@Stateful
public class testContadorBean implements testContadorRemote
{
…
public Boolean testContadorSinLock()
{
try
{
CounterTest ct = manager.buscar(TestContador.class, 1);
int valor = ct.getValor();
ct.setValor(valor + 1);
TestTabla tt = new TestTabla();
tt.setNr(valor);
manager.persist(tt);
manager.flush();
return true;
}
catch( Throwable t )
{
return false;
}
}
…
}
```

Código 2.171: Método testContadorSinLock()

El método busca primero la instancia **TestContador** con el número 1, memoriza el número encontrado que después, una vez aumentado hasta 1, reescribe. A continuación se ejecuta con este número una nueva grabación de **TestTabla** y se guarda. La llamada del método **flush()** al EntityManager no es en sí explícitamente necesaria pero se ocupa de interceptar una posible excepción y así evitar perder la instancia de bean.

2.20.5.2. **Versionar automáticamente**

Para posibilitar un trabajo en concurrencia los datos debe bloquearse. Esto puede realizarse en modo optimista o pesimista. Como bloqueo de conjunto optimista se entiende que se parte de la base que se encontrarán más bien escasos en concurrencia, por lo que

los datos no se bloquean en un principio. El momento en qué debe reconocerse si se produce un acceso en concurrencia es solo una vez se deben reescribir realmente las modificaciones. Con este objetivo se guarda en la correspondiente tabla de base de datos una marca distintiva de la versión en forma de un número continuo o indicando la hora. Si se quiere sobreescribir un cambio, primero se comprueba si los datos en la base de datos tienen la misma señal distintiva que en la memoria principal. Si es así, no existe ningún acceso en concurrencia y se pueden sobreescribir los datos, sin olvidar que la marca distintiva debe coincidir con la de la base de datos. Si es al contrario, se informará al usuario de que los datos han sido modificados mientras tanto y se le posibilitará leerlos de nuevo. Un sistema realmente bueno memoriza qué cambios ha introducido el usuario y le conduce a los nuevos datos leídos. Este estado entonces se le mostrará al usuario, que podrá guardarlo o bien eliminarlo.

El nuevo estándar soporta un desarrollador en la administración de este tipo de señalizadores de versión. Si guarda una columna del tipo **int**, **Integer**, **short**, **Short**, **long**, **Long** o **Timestamp** en su Entity Bean, entonces puede otorgarse una anotación **@Version**.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Version{ }
```

Código 2.172: @Version

Se puede caracterizar con la anotación **@Version** como máximo una columna por Entity Bean. Si el bean se extiende a lo largo de varias tablas, la correspondiente columna debe estar incluida en la tabla primaria. Es el servidor de la aplicación el qué se ocupará personalmente del contenido de esta columna. Si se ha optado por un valor numérico, lo aumentará cada vez en una unidad cuando se le reescriba alguna modificación. Antes comprueba si el número de versión en la base de datos sigue siendo el mismo que en la instancia del Entity Bean. Si no es así, lanza una **OptimisticLockExceptioin**, que conducirá a un **rollback**, incluso cuando se intercepte la exception.

Si por el contrario se ha realizado un cambio masivo con ayuda de la instrucción **UPDATE** es el mismo desarrollador el que debe ocuparse de la administración de la señalización de la versión. Esto puede hacerse por ejemplo especificando **SET VERSION = VERSION + 1**.

La tabla **TEST_CONTADOR** dispone de una columna con el nombre **VERSION**, que podría llamarse también de otro modo, que sirve como señalización de la versión. El Entity Bean **TestContadorVersion** del Código 2.173 incluye esta columna.

```
package server.tr;

import javax.persistence.*;
@Entity
@Table(name="TEST_CONTADOR")
public class TestContadorVersion
{
private int nr;
private int versión;
private int valor;
```

```
@Id
public int getNr()
{
return nr;
}
public void setNr( int nr )
{
this.nr = nr;
}
@Version
public int getVersion()
{
return version;
}
public void setVersion( int version )
{
this.version = version;
}
public int getValor()
{
return valor;
}
public void setValor ( int valor )
{
this.valor = valor;
}
}
```

Código 2.173: Entity Bean con controles de versión

Si se quisiera prescindir de anotacions, se tiene que definir la columna de la versión en el Deployment Descriptor con la etiqueta **<version>**. El Código 2.174 muestra la entrada correspondiente.

```
<entity class=”server.tr.TestContadorVersion” Access=”PROPERTY”
metadata-complete=”true”>
<table name=”TEST_CONTADOR”/>
<attributes>
<id name=”nr”>
<column name=”NR”/>
</id>
<basic name=”valor”>
<column name=”VALOR”/>
</basic>
<version name=”version”>
<column name=”VERSION”/>
</version>
</attributes>
</entity>
```

Código 2.174: Columna de versión en el Deployment Descriptor

Lo siguiente que debe comprobarse es lo que ocurre en un Entity Bean versionado de este tipo cuando se da un acceso concurrente de tres clientes. Para ello se programa en la clase **TestContadorBean** un método con el nombre **testContadorConVersion()**, reproducido en el Código 2.175.

```
@Stateful
public class TestContadorBean implements TestContadorRemote
{
@PersistenceContext(unitName=”JavaEE”)
EntityManager = manager;
```

```
public boolean testContadorConVersion()
{
try
{
TestContadorVersion ctv =
 manager.buscar(TestContadorVersion.class, 1);
int valor = ctv.getValor();
ctv.setValor(valor + 1);
manager.flush();
TestTabla tt = TestTabla();
tt.setNr(valor);
manager.persist(tt);
manager.flush();
return true;
}
catch( OptimisticLockException e )
{
System.out.println(“>>> Conflicto de version !”);
return false;
}
catch( Throwable t )
{
System.out.println( t.getMessage() );
return false;
]
}
…
}
```

Código 2.175: Método de prueba testContadorConVersion()

Para que el cliente pueda ser informado correctamente de este conflicto de versiones se intercepta la **OptimisticLockException** y se llama al método **flush()** del EntityManager en el bloque **try**. Si no fuera así sería una vez finalizado el método cuando se produciría la excepción mencionada, transmitida después como tal al cliente. Aunque la excepción sea interceptada, se producirá un **rollback** de toda la transacción. Por lo tanto es respondabilidad del cliente reaccionar ante este caso inusual y consultar de nuevo los datos. Si se intentara consultar los datos en el servidor justo después de la **OptimisticLockException** se produciría un error.

2.20.5.3. **Trabajar con bloqueo de conjuntos**

La última opción al enfrentarse con el problema de los accesos concurrentes consiste en bloquear los datos leídos en la base de datos hasta que se sobreescriban las modificaciones. Con este objetico el EntityManager ofrece el método **lock()**, que se describe a continuación. Un bloqueo de este tipo se levanta cuando el conjunt se ha sobreescrito o la transacción ha finalizado. Y es justo aquí donde reside el problema. En una aplicación online el usuario llamará primero un método para leer los datos para que este los pueda mostrar con tranquilidad. Si se bloquea el conjunto durante la lectura, este bloqueo se pierde cuando los datos se devuelven al cliente. Si envía finalmente sus cambios, es posible que un concurrente haya sido más rápido y nos encontraremos con el problema descrito. Si no se soluciona esto siempre ganará aquel que escriba el último, lo que no resulta muy propio de una aplicación profesional. Suponiendo que dos operarios

leen simultáneamente el mismo socio. Uno modifica el apellido y mando los datos. Si ahora otro modifica la calle y los guarda por su parte, el cliente volverá a llamarse como antes.

Un bloqueo por lo tanto tendrá sentido como máximo si los datos se envían en la misma transacción, si se trata pues de un procesamiento por lotes.

Las especifacion preescribe que un Entity Bean que debe ser bloqueado mediante el método **lock()** debe disponer de una señalización de versión, que es también remarcable mediante la anotación **@Version**. Al método **lock()** se le tienen que transmitir dos parámetros. El primero es el Entity Bean a bloquear, el segundo el tipo de bloqueo, reproducido mediante la enumeración **LockModeType**. Este ofrece los valores **READ** y **WRITE**. Se diferencian tan solo porque con **WRITE** se actualiza también la marca de versión al fijar el bloqueo.

Para comprobar los bloqueos se amplía la clase **TestContadorBean** con el método **testContadorConLock()**, reproducido en el Código 2.176.

```
@Stateful
public class TestContadorBean implements TestContadorRemote
{
@PersistenceContext(unitName=”JavaEE”)
EntityManager = manager;

public boolean testContadorConLock()
{
try
{
TestContadorVersion ctv =
 manager.buscar(TestContadorVersion.class, 1);
manager.lock(ct, LockModeType.WRITE);
int valor = ctv.getValor();
ctv.setValor(valor + 1);
TestTabla tt = TestTabla();
tt.setNr(valor);
manager.persist(tt);
manager.flush();
return true;
}
catch( Throwable t )
{
return false;
]
}
…
}
```

Código 2.176: Método testContadorConLock()

La llamada del método **flush()** se ocupa tan solo de que el método no finalice con una excepción, porque si ocurriera esto en el caso de Stateful Session Bean conllevaría que el servidor la liberaría.

Debido a su implementación, la llamda del método **lock()** no es ninguna garantía para evitar operaciones concurrentes, puesto que los datos ya se han leído antes del bloqueo. También una aplicación en bloque debe ser capaz de manejar este tipo de situaciones. O

bien escribe una entrada en un protocolo en la que se indica que el conjunto no se pudo editar o lo intenta nuevamente con la consulta y otro cambio. En ese caso debe tenerse en cuenta que esto no ocurra en la misma transacción. Por lo tanto debe haber un Session Bean en el servidor que intente con ayuda de otro Session Bean, ejecutar el procesamiento por lotes. El bean dependiente debe disponer por lo tanto de su propio control de la transacción, es decir los métodos deben estar descritos con **RequiresNew**.

2.20.6. Rollback en EJBContext

Una situación especial se da cuando un método debe ocuparse debido a un error técnico de que la transacción en la base de datos se invierta. Para ello la clase **EJBContext** dispone del método **setRollbackOnly()**. Si se llama se producirá al finalizar la transacción un **rollback**. Un ejemplo de esta llamada se encuentra en el Código 2.177. Como alternativa la aplicación puede producir una excepción. Entonces toda acción finaliza de inmediato, mientras que el método en el caso de **setRollbackOnly()** puede seguir trabajando.

```
@Stateful
public class TestContadorBean implements TestContadorRemote
{
@Resource
SessionContext ejbContext;
public void unMetodo()
{
…
if( error )
ejbContext.setRollbackOnly();
…
}
}
```

Código 2.177: Llamada de setRollbackOnly()

2.20.7. Transacciones y excepciones

Un tema importante en relación con las transacciones son las excepciones lanzadas por una aplicación. Cogestionan entre otras cosas si se produce un rollback y también son responsables en el caso de los Stateful Session Bean de si el servidor de aplicaciones desecha la instancia del bean de manera que el cliente tenga que registrarse de nuevo.

Se diferencia entre tres tipos de excepciones. Las más extensas son las llamadas errores del sistema. Se trata de la clase **java.lang.RuntimeException** y de todas las clases derivadas de esta. Java no prescribe la intercepción de estas excepciones puesto que se trata normalmente de errores técnicos que provocan este tipo de excepciones. Si aparece una de estas excepciones la transacción terminará con un **rollback**. Esto significa además para un Stateful Session Bean que será liberado de inmediato y que el cliente deberá registrarse de nuevo.

Totalmente diferente es el tema cuando un método anuncia un error de aplicación. En este caso se da una **java.lang.Exception** o una clase derivada de esta. Para este error Java

prescribe un blouqe **try-catch**. Una excepción de este tipo conduce a un **commit** y el Stateful Session Bean sobrevive. Aquí es ante todo interesante el hecho de que la transacción termina exitosamente y permanecen todos los datos escritos hasta el momento en la base de datos.

En la tercera versión el desarrollador provee a la excepción de la anotación **@ApplicationException**, mediante la cual puede controlar el comportamiento de la transacción. Con el parámetro **rollback** se determina cómo finaliza la transacción. Por defecto está en **false**. Si se da esta anotación a una **RuntimeException** y no se producen más indicaciones, se producirá un **commit** de la transacción y el Stateful Session Bean sobrevivirá. Así se manejará un error de sistema como un error de aplicación. Si por el contrario se fija el parámetro **rollback** en **true**, permanece un error de aplicación, pero de base de datos se limpiará. Por lo tanto es bastante razonable dotar a una excepción técnica también de esta anotación para asegurar que los datos en la base de datos permanecen consistentes.

Para poder comprobar el comportamiento, se han programado cuatro excepciones. La clase **GraveExcepcionSinRollback** se ha derivado de **RuntimeException** y se ha provisto de la anotación **@ApplicationException**. Si se produce, llevará esto a un **commit** y el Stateful Session Bean sobrevivirá.

```
package server.tr;
import javax.ejb.ApplicationException;
@ApplicationException
public class GraveExcepcionSinRollback extends RuntimeException
{
private static final long SerialVersionUID = 1L;

public GraveExcepcionSinRollback( String msg )
{
super(msg);
}
}
```

Código 2.178: Error de sistema con Commit

La clase **GraveExcepcionConRollback** está derivada de **RuntimeException**, pero se le ha otorgado una anotación **@ApplicationException(rollback=true)**. Si se produjera, un stateful session bean sobrevivirá pero se eliminarían los datos en la base de datos.

```
package server.tr;

import javax.ejb.ApplicationException;

@ApplicationException(rollback=true)
public class GraveExcepcionConRollback extends RuntimeException
{
private static final long SerialVersionUID = 1L;

public GraveExcepcionConRollback( String msg )
{
super(msg);
}
}
```

Código 2.179: Error de sistema con Rollback

Si se observa con detenimiento la clase **SimpleExcepcionSinRollback**, se comprueba que se trata de una excepcion técnica clásica. No tiene más anotación y si se produjera conduciría a un **Commit**. Un Stateful Session Bean sobrevive a esto sin problemas.

```
package server.tr;

public class SimpleExcepcionSinRollback extends Exception
{
private static final long SerialVersionUID = 1L;

public SimpleExcepcionSinRollback ( String msg )
{
super(msg);
}
}
```

Código 2.180: Error de aplicación con Commit

Con la clase **SimpleExcepcionConRollback** nos encontramos por primera vez con una excepcion técnica que también conlleva un **rollback**. Aquí la anotación con **ApplicationException(rollback=true)** es absolutamente indispensable. Mientras debería estar claro que un stateful session bean sobrevive una excepcion de este tipo. Para la aplicación concreta esto significa sin embargo que todas las clases de errores deberían estar programadas de este modo. Entonces es razonable detener un procesamiento, grabar los datos acopiados hasta ese momento y realizar las modificaciones.

```
package server.tr;

import javax.ejb.ApplicationException;

@ApplicationException(rollback=true)
public class SimpleExcepcionConRollback extends Exception
{
private static final long SerialVersionUID = 1L;

public SimpleExcepcionConRollback ( String msg )
{
super(msg);
}
}
```

Código 2.181: Error de aplicación con Rollback

Si se desea precindir de nuevo de las anotaciones o sobreescribirlas, debe definir el error de aplicación como tal en el Deployment Descriptor **ejb-jar.xml**. En el Código 2.182 se muestra cómo se lleva esto a cabo.

```
<ejb-jar
xmlns=”http://java.sun.com/xml/ns/javaee”
xmlsn:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:SchemaLocation=”http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd” versión=”3.1”>
<enterprise-beans>
…
</enterprise-beans>
<assembly-descriptor>
<application-exception>
```

```
<exception-class>
server.tr.SimpleExcepcionConRollback
</exception-class>
<rollback>true</rollback>
</application-exception>
<application-exception>
<exception-class>
server.tr.GraveExcepcionConRollback
</exception-class>
<rollback>true</rollback>
</application-exception>
<application-exception>
<exception-class>
server.tr.SimpleExcepcionSinRollback
</exception-class>
</application-exception>
<application-exception>
<exception-class>
server.tr. GraveExcepcionSinRollback
</exception-class>
</application-exception>
</assembly-descriptor>
</ejb-jar>
```

Código 2.182: Deployment Descriptor

2.20.8. Session Beans transaccionales

La protección de la transacción se extiende tan solo a lo largo de la base de datos. Los atributos de los session beans no se cancelan solo porque se haya producido un rollback, pero existe la posibilidad de involucrar un bean en la acción de la transacción. Para ello tiene que implementar la interfaz **java.ejb.SessionSynchronization**, para lo que en total se requieren tres métodos.

El método **despuesDeIniciar()** se llamara simpre cuando se inicie una transacción para el session bean. No recibe más parámetros y devuelve void.

Mediante el método **antesDeCompletar()** se informa al bean de que la transacción finalizará con un **Commit**. En este punto esto no está sin embargo cien por cien garantizado. Si se produjera ahora un rollback, este método no llegaría a ejecutarse nunca. Tampoco espera recibir ningún parámetro y devuelve **void**.

El método más interesante es **despuesDeCompletar(boolean)**. Cuando se llama queda fijado cómo finaliza la transacción. El parámetro transmitido es **true** cuando se da un **Commit** y **false** en caso de un **rollback**. Tampoco estos métodos devuelven ningún parámetro.

En el Código 2.183 se amplica la clase **TestContadorBean** con la interfaz **SessionSynchronization** y los métodos necesarios para ello. Al final de cada transacción se da como resultado la relación entre transacciones exitosas y fracasadas. Si se observa el ejemplo de **TestContadorConLock**, llama la atención que siempre hay bloques en los que funcionan algunas transacción en esta parte y después se llega a rollbacks.

```
@Stateful
public class TestContadorBean implements TestContadorRemote, SessionSynchronization
{
```

```
private int commit = 0;
private int rollback = 0;
…
public void despuesDeIniciar() throws EJBException, RemoteException
{
}

public void antesDeCompletar() throws EJBException, RemoteException
{
}

public void despuesDeCompletar( Boolean committed)
throws EJBException, RemoteException
{
if( committed )
commit++;
else
rollback++;
System.out.println("commit/rollback: " + commit + "/" + rollback);
}
}
```

Código 2.183: SessionBean transaccional

Puede ser tarea del método **despuesDeCompletar()** rectificar el estado interno del Stateful Session Bean en caso de que la transacción no pase. Para Stateless Session Bean o Message-Driven Bean esto carece de relevancia.

2.21. Interceptores y Entity Listener

2.21.1. Panorama general

Un Interceptor, es decir alguien que intercepta algo, es una simple clase Java, cuyos métodos siempre se ejecutan cuando se llama de hecho el método de otra clase totalmente diferente. Se activa prácticamente entre el que llama un método y los métodos llamados. Así está en situación de supervisar el sistema. Podría por ejemplo calcular el tiempo que necesita una llamada de un método o simplemente protocolar en que sucesión se necesitan qué métodos, cuándo y por quién son requeridos. El sentido de un interceptos recae precisamente en mantener alejadas estas actividades de administración y protocolo del auténtico proceso. Si se quisiera saber cuánto tiempo necesita una llamada de método en nuestro session bean, se puede programar el código fuente necesario para la cronometracion del tiempo en el método business. Por ese motivo, este se ocupa a veces de cosas de las que en realidad no es responsable en absoluto. Además en cada método bean se encuentra siempre la misma estructura de piezas lógicas. Un interceptor sirve para liberar los métodos de servicio de este tipo de actividades. Rodea un session bean e intercepta todas las llamadas de los métodos. Ahora se puede implementar la lógica para la medición del tiempo desde el punto central y averiguar si realmente se tienen que medir los tiempos.

Un interceptor se puede configurar para un session bean o para un message-driven bean. Los entity beans son tabú para él.

Para entity beans está el Entity Listener con los que se pueden interceptar los métodos del ciclo de vida **@PostLoad** o **@PostPersist**. De este modo no es posible un cálculo del tiempo, más una documentación de cuándo se ha llamado uno de estos entity beans por el EntityManager y para qué entity beans. También aquí puede programar uno mismo esta documentación en el entity bean, pero esto duplicaría el código fuente y ocuparía las clases con tareas con las que no tienen nada que ver. El Entity Listener centraliza todos estos métodos.
Tanto para el Interceptor como para el Entity Listener es la clase a vigilar la que debe decidir por quién es controlada. Solo usando un Deployment Descriptor se puede definir un estándar para todos, es decir predefinir una sola vez que todos los session beasn y message-driven beans sean controlados por un Interceptor determinado y todos los entity beans por un Entity Listener determinado. La reducción a un solo controlador de este tipo de hecho no es muy acertada. Puede haber varios de ellos, de modo que se colocan sucesivamente. Lo que no funciona es activar o desactivar este tipo de clases de supervsion durante el tiempo de ejecución. Si se han definido mediante anotaciones o en el Deployment Descriptor, siempre se llamarán. Por lo tanto queda a disposición de la lógica de estas clases estar activadas o no. De este modo sin embargo supondrá un Overhead considerable, que cuesta su tiempo.

2.21.2. Interceptor

2.21.2.1. **Anotaciones**

La manera más fácil de programar un interceptor consiste en dotar a un método determinado de la anotación **@AroundInvoke**. Este método debe corresponder con la signatura representada en el Código 2.184.

```
@AroundInvoke
Object <METHOD>(InvocationContext) throws Exception
```

Código 2.184: Signatura de un método interceptor

```
@Target({METHOD})
@Retention(RUNTIME)
public @interface AroundInvoke{ }
```

Código 2.185: @AroundInvoke

El método puede llevar cualquier nombre. Debe dar como resultado una instancia de **Object** y espera como único parámetro una instancia de **InvocationContext**.

```
public interface InvocationContext
{
public Object getBean();
public Method getMethod();
public Object[ ] getParameters();
public void setParameters( Object[ ] params);
public java.util.Map<String, Object> getContextData();
```

```
public Object proceed() throws Exception;
}
```

Código 2.186: Interfaz InvocationContext

Mediante estos parámetros el método a la escucha obtiene acceso al objeto, que debe llamarse (**getBean()**), al método objetivo en concreto (**getMethod()**) y sus parámetros (**getParameters()**). Es posible incluso modificar los parámetros (**setParameters()**) antes de que tenga lugar la llamada en sí (**Proceed()**).

Un Interceptor no se informa tan solo mediante una llamada de método, sino que se activa mientras tanto. Si el interceptor no llama el método **proceed()**, tampoco entrará el método business. Tampoco otro interceptor que se haya activado mientras tanto obtendrá ningún control.

Para que un interceptor no intervenga en la lógica del programa debe llamar al método **proceed()** y devolver desde allí el objeto obtenido como resultado, incluso cuando el mismo método business no tiene ningún valor de devolución.

Con el método **getContextData()** se puede dar al método interceptor una instancia de la clase **Map**, que existe activamente para esta llamada de métodos. Mediante estos **Map** varias clases interceptor puede intercambiar información, si todas ellas esuchan las mismas llamadas de métodos. Así por ejemplo se puede vigilar que la protocolización se dé solo una vez para cada llamada de método y el resto se informen a partir de esta. En el Código 2.187 se representa una clase interceptor, que devuelve la duración de una llamada cualquiera de método de un session bean o un message-driven bean.

```
package server.in;
import javax.interceptor.*;

public class MedirTiempos
{
@AroundInvoke
public Object timeTrace( InvocationContext invocation ) throws Exception
{
long start = System.currentTimeMillis();
try
{
return invocation.proceed();
}
finally
{
long final = System.currentTimeMillis( );
String clase = invocation.getBean().toString();
String metodo = invocation.getMethod().getName();
System.out.println(class + ":" + metodo + " -> " + (final-inicio) + "ms");
}
}
}
```

Código 2.187: Clase Interceptor

El bloque **finally** se coloca en el método de medición por si se da el caso de que el método llamado finaliza con una excepción.
Excepto si se ha definido un interceptor estándar para todos los beans en el Deployment Descriptor, cada uno de ellos debe comunicar por qué tipo de clases quieren ser controlados. Para ello se crea la anotación **@Interceptors**, mediante la que se puede dar una lista de las clases a la escucha.

```
@Target({CLASS, METHOD})
@Retention(RUNTIME)
public @interface Interceptors
{
Class[ ] value();
}
```

Código 2.188: @Interceptors

La anotación se puede utilizar tanto en el nivel de los métodos como en el de las clases. En consecuencia se supervisarán solo algunos métodos concretos de la clase bean o bien todos.

```
@Stateful
@Interceptors(MedirTiempos.class)
public class TiempoAdministracionSocioBean implements TiempoAdministracionSocioRemote
{
…
}
```

Código 2.189: @SecondaryTables

En el Código 2.189 se programa un Session Bean que quiere que la llamada de todos sus métodos sea controlada por un interceptor con el nombre **MedirTiempos**. Se incluyen también los métodos del ciclo de vida.

Están también las anotaciones **@ExcludeDefaultInterceptors** y **@ExcludeClassInterceptors** que pueden ser discutidas en relación con el Deployment Descriptor (vea el apartado 2.10.2.2) puesto que solo allí tienen sentido.

2.21.2.2. **Descriptor de Despliegue**

Como es habitual todas las entradas de las anotaciones pueden rescribirse o sustituirse por completo mediante el Deployment Descriptor. En el caso de las clases Interceptors el Deployment Descriptor puede más incluso que las anotaciones. Aquí se puede configurar una clase interceptor estándar para todos los beans.
Primero es importante describir uno mismo la clase interceptor. Esto se realiza mediante la etiqueta **<interceptor>** en el archivo **ejb-jar.xml**.

La etiqueta **<interceptor>** consta de una o más etiquetas **<interceptor>**, que enumeran de nuevo el nombre de la clase interceptor y después la lista de todos los métodos de esta

clase. Como la signatura de estos métodos está firmemente fijada basta para cada uno con el nombre del método. En el Código 2.190 se reproduce cómo se define la clase **MedirTiempo** como interceptor en el Deployment Descriptor.

```
<interceptors>
<interceptor>
<interceptor-class>server.in.MedirTiempos</interceptor-class>
<around-invoke>
<method-name>timeTrace</method-name>
</around-invoke>
</interceptor>
</interceptors>
```

Código 2.190: El interceptor MedirTiempos

En el siguiente paso se fija el interceptor correspondiente para el bean en cuestión. Esto sucede dentro de la etiqueta **<assembly-descriptor>** bajo **<interceptor-binding>**. Ahí se conecta el nombre del bean con la clase interceptor. Si deben supervisarse todos los métodos del bean no se realizan más entradas, sino puede definirse para cada **<interceptor-binding>** un método que debe reaccionar al interceptor.

```
<interceptor-binding>
<ejb-name>TiempoAdministracionSocioBean</ejb-name>
<interceptor-class>server.in.MedirTiempos</interceptor-class>
<method>
<method-name>addSocio</method-name>
<method-params>
<method-param>java.util.vector</method-param>
</method-params>
</method>
</interceptor-binding>
```

Código 2.191: Interceptor para un método

En el Código 2.191 se reproduce cómo se supervisa solo un método determinado de un bean mediante un interceptor. El Código 2.192 determina por el contrario todos los métodos de la clase como supervisables.

```
<interceptor-binding>
<ejb-name>TiempoAdministracionSocioBean</ejb-name>
<interceptor-class>server.in.MedirTiempos</interceptor-class>
</interceptor-binding>
```

Código 2.192: Interceptor para todo un bean

Como ya se ha mencionado en un Deployment Descriptor también se puede definir un interceptor estándar para todos los beans. Esto se consigue simplemente incluyendo un asterisco en la etiqueta <ejb-name>.

```
<interceptor-binding>
<ejb-name>*</ejb-name>
<interceptor-class>server.in.MedirTiempos</interceptor-class>
</interceptor-binding>
```

Código 2.193: Interceptor estándar

Es posible que una determinada clase bean intente oponer resistencia a un interceptor estándar. Si la clase **TiempoAdministracionSocio** no quiere dejarse supervisar por un interceptor, esto puede administrarse con la anotación **@ExcludeDefaultInterceptor** o la etiqueta **<exclude-default-interceptors>**.

```
@Stateful
@ExcludeDefaultInterceptors
@Interceptors( MedirTiempos.class )
public class TiempoAdministracionSocioBean implements TiempoAdministracionSocioRemote
{…
}
```

Código 2.194: @ExcludeDefaultInterceptors

```
<interceptor-binding>
<ejb-name>AdministracionSocioBean</ejb-name>
<interceptor-class>server.in.MedirTiempos</interceptor-class>
<exclude-default-interceptors>true</exclude-default-interceptors>
</interceptor-binding>
```

Código 2.195: Etiqueta <exclude-default-interceptors>

Tanto la anotación como la etiqueta se pueden dar en toda la clase, como en los Códigos 2.194 y 2.195, o también a nivel de los métodos, para activar el interceptor estándar solo para determinados métodos.
Para el resto también está la anotación **@ExcludeClassInterceptors** con la etiqueta correspondiente, que solo es práctica en el nivel de los métodos. Así se determina que un interceptor definido para una clase no afecta a este método.

Un interceptor debe observarse como un Stateless Session Bean pero al que puede accederse desde fuera. No tiene ningún sentido guardar en los atributos de una clase de este tipo cualquier información sobre una llamada de método determinada, puesto que la misma instancia puede ser utilizada de inmediato por otro usuario y por otro método totalmente diferentes. Para la llamada de un método se encuentra la clase interceptor en el mismo contexto y en la misma transacción que el método cuya llamada ha sido desviada. De ahí que pueda resultar práctico inyectar el **SessionContext** o el **PersistenceContext**. Están permitidas las anotaciones necesarias y el Deployment Descriptor prevé las entradas correspondientes. También es posible el acceso a otras fuentes.

2.21.3. Entity Listener

2.21.3.1. **Anotaciones**

Un interceptor no está permitido en un Entity Bean. Pero este tipo de bean también puede ser supervisado por otras clases. La supervisión se limita sin embargo a los métodos del ciclo de vida del Entity Bean que puede dotarse de una de las anotaciones representadas en el Código 2.196.

```
@javax.persistence.PrePersist
@javax.persistence.PostPersist
@javax.persistence.PostLoad
```

```
@javax.persistence.PreUpdate
@javax.persistence.PostUpdate
@javax.persistence.PreRemove
@javax.persistence.PostRemove
```

Código 2.196: Anotaciones del Ciclo de Vida

Un Entity Listener es una clase Java normal que dispone de uno o más métodos caracterizados con anotaciones del ciclo de vida. Un ejemplo de ello aparece en el Código 2.197.

```
package server.li;

import javax.persistence.*;

public class DocumentarActividades
{
@PostPersist
public void postPersist( Object entity )
{
System.out.println("Entity guardado: " + entity.getClass().getName());
}

@PostLoad
public void postLoad( Object entity )
{
System.out.println("Entity cargado: " + entity.getClass().getName());
}

@PostUpdate
public void postUpdate( Object entity )
{
System.out.println("Entity modificado: " + entity.getClass().getName());
}

@PostRemove
public void postRemove( Object entity )
{
System.out.println("Entity eliminado: " + entity.getClass().getName());
}
}
```

Código 2.197: Entity Listener

Para que un Entity Listener de este tipo también este activo debe haber por lo menos un entity bean que accepte esta clase como Listener. La excepcion también aquí será la definición de un Entity Listener para todos los Entity Beans, algo que puede realizarse, eso sí, desde un Deployment Descriptor.

```
@Entity
@Table(name="SOCIO")
@EntityListeners( DocumentarActividades.class )
public class TiempoSocio
{
…
}
```

Código 2.198: Entity con Listener

En el Código 2.198 el Entity Bean **TiempoSocio** permite que la clase **DocumentarActividades** le supervise. La anotación **@EntityListerners** puede darse en el nivel de las clases.

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface EntityListeners
{
Class[ ] value();
}
```

Código 2.199: @EntityListeners

Puede ser que tanto un Entity Bean como también el Listerner dispongan de los mismos métodos del ciclo de vida. En este caso se llama primero el método del Listener y después el del bean. Además pueden activarse varios listerner en un mismo bean.

La anotación **@ExcludeDefaultListeners** solo es relevante en relación con un Deployment Descriptor, por lo que también se explicará allí.

Si un Entity Bean está derivado de otro y la clase base posee un Entity Listener, con la anotación **@ExcludeSuperClassListeners** la clase derivada puede controlar que no sea ella misma supervisada. Un Entity Listener se transmite sino a lo largo de todas las clases dependientes.

2.21.3.2. **Descriptor de Despliegue**

Un Entity Listener se define en el Deployment Descriptor o bien dentro de la etiqueta **<entity>** o dentro de **<persistence-unit-defaults>**. Según el caso pertenece a un entity bean determinado o sirve por defecto para todos. En el Código 2.200 se define el Entity Listener **DocumentarActividades** y se conecta a su vez con el Entity Bean **TiempoSocio**. Si otro Entity Bean necesitar del mismo listerner, se tendrían que repetir las mismas entradas en el otro bean.

```
<entity-class=”server.li.TiempoSocio” Access=”PROPERTY”
metadata-complete=”true”>
<table name=”SOCIO”/>
<entity-listerners>
<entity-listener class=”server.li.DocumentarActividades”>
<post-persist method-name=”postPersist”/>
<post-remove method-name=”postRemove”/>
<post-update method-name=”postUpdate”/>
<post-load method-name=”postLoad”/>
</entity-listener>
</entity-listerners>
</entity>
```

Código 2.200: Definición de un Entity Listener

Con las etiquetas **<pre-persist>**, **<post-persist>**, etc, se representa cada evento del ciclo de vida en métodos determinados del Listener. Si se utiliza el mismo Entity Listener para todos los Entity Beans se crea este simplemente dentro de la etiqueta **<persistence-unit-defaults>**, como puede observarse en el Código 2.201.

```
<persistence-unit-metadata>
<persistence-unit-defaults>
<entity-listeners>
<entity-listener class=”server.li.DocumentarActividades”>
<post-persist method-name=”postPersist”/>
<post-remove method-name=”postRemove”/>
<post-update method-name=”postUpdate”/>
<post-load method-name=”postLoad”/>
</entity-listener>
</entity-listeners>
</persistence-unit-defaults>
</persistence-unit-metadata>
```

Código 2.201: Entity Listener global

Si un Entity Bean está derivado de otro y la clase base posee un Entity Listener, con la etiqueta **<exclude.superClassListener>** la clase derivada puede controlar que no sea ella misma supervisada. Un Entity Listener se transmite sino a lo largo de todas las clases dependientes.

```
<entity class=”server.li.TiempoSocio” access=”PROPERTY”
metadata-complete=”true”>
<table name=”SOCIO”/>
…
<exclude-superclass-listeners/>
…
<attributes>
…
</attributes>
</entity>
```

Código 2.202: Destituir el Listener de la clase base.

2.22. Seguridad

2.22.1. Asegurar llamadas de métodos

2.22.1.1. @RolesAllowed

Con la anotación **@RolesAllowed** se fija tanto en e nivel de las clase como en el de los métodos que papel debe desempeñar un usuario para poder llamar los métodos, si se realiza la entrada a nivel de las clases, será válido para todos los métodos de esta clase, excepto el método los escribe con una anotación propia.

```
@Target(value={METHOD,TYPE})
@Retention(value=RUNTIME)
public @interface RolesAllowed
{
```

```
String[ ] value();
}
```

Código 2.203: @RolesAllowed

Si se encuentran varios roles con esta anotación, el parámetro **value** será una tabla de instancias **String**, el cliente que realice la llamada deberá disponer de cómo minimo uno de estos roles, no de todos.

```
@Stateful
@RolesAllowed(“ADMIN”)
public class RealizarPedidoBean implements RealizarPedidoRemote
{
@RolesAllowed({“SOCIO_REGISTRADO”, “ADMIN”})
public void asignarSocio(Vector<GrupoDatos> datos) throws PedidoException
{
…
}
…
}
```

Código 2.204: Ejemplo de @RolesAllowed

En el Código 2.205 se asegura el método **AsignarSocio()** con los roles **SOCIO_REGISTRADO** y **ADMIN**. La llamada de este método por lo tanto solo le permitirá cuando es cliente disponga de alguno de estos dos roles. El resto de métodos de esta clase se mantienen frente al administrador.

Como alternativa a las anotaciones también se puede utilizar el Deployment Descriptor para definir roles o para sobreescribir los roles fijados en las anotaciones.

```
<ejb-jar
xmlns=”http://java.sun.com/xml/ns/javaee”
xmlsn:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:SchemaLocation=”http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd”
versión=”3.1”>
…
<assembly-descriptor>
<security-role>
<role-name>SOCIO_REGISTRADO</role-name>
</security-role>
<security-role>
<role-name>ADMIN</role-name>
</security-role>
…
<method-permission>
<role-name>SOCIO_REGISTRADO</role-name>
<role-name>ADMIN</role-name>
<method>
<ejb-name>RealizarPedidoBean</ejb-name>
<method-name>asignarSocio</method-name>
<method-params>
<method-param>java.util.Vector</method-param>
</method-params>
</method>
```

```
</method-permission>
<method-permission>
<role-name>ADMIN</role-name>
<method>
<ejb-name>RealizarPedidoBean</ejb-name>
<method-name>*</method-name>
</method>
</method-permission>
…
</assembly-descriptor>
</ejb-jar>
```

Código 2.205: Adjudicados de roles mediante el Deployment Descriptor

En el Código 2.205 se reproduce cómo la etiqueta **<assembly-descriptor>** fija en la clase RealizarPedidoBean que se prescriba el rol **ADMIN** para la llamada de todos los métodos. Esto se consigue con el asterisco en la etiqueta **<method-name>**, con el cual se adjudican los roles a todos los métodos del bean. A su vez se encuentra también una entrada para el método **asignarSocio()** que espera una instancia de la clase **java.util.Vector**. Esta entrada especial rescribe las globales y permite también para este método el rol **SOCIO_REGISTRADO**. En el estándar Java EE cada rol que se utiliza dentro de la etiqueta **<assembly-descriptor>** debe ejecutarse primero mediante la etiqueta **<security-role>**.

2.22.1.2. @**PermitAll**

Si no se ha asignado ningún rol de usuario a un método y este tampoco lo ha obtenido desde la clase bean, entonces todo usuario estará autorizado a llamarlo. Esto puede expresarse también explícitamente mediante la anotación **@PermitAll**, que puede usarse tanto a nivel de los métodos como también de las clases.

```
@Target(value={METHOD, TYPE})
@Retention(value=RUNTIME)
public @interface PermitAll
```

Código 2.206: @PermitAll

Es poco común dotar a cada método accesible con esta anotación. Se suele utilizar más bien cuando se quiere anular una entrada válidad globalmente. Si se ha dotado una clase bean con la anotación **@RolesAllowed**, después estos roles servirán para todos los métodos de esta clase. Si esta clase contiene pues un método que puede ser llamado por cualquier usuario, entonces se le provee de la anotación **@PermitAll**.

```
@Stateful
@RolesAllowed("SOCIO_REGISTRADO")
public class RealizarPedidoBean implements RealizarPedidoRemote
{
…
@PermitAll
public Vector<GrupoDatos> getEstructuraSocio() throws PedidoException
{
```

```
…
}

public void asignarSocio( Vector<GrupoDatos> datos ) throws PedidoException
{
…
}
}
```

Código 2.207: Ejemplo de @PermitAll

También esta anotación se puede fijar o rescribir mediante el Deployment Descriptor. Dentro de la etiqueta **<assembly-descriptor>** se define un método como **<un-checked/>,** con el que se excluye toda comprobación de roles para este método.

```
<ejb-jar
xmlns="http://java.sun.com/xml/ns/javaee"
xmlsn:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:SchemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd" versión="3.1">
…
<assembly-descriptor>
<security-role>
<role-name>SOCIO_REGISTRADO</role-name>
</security-role>
…
<method-permission>
<unchecked/>
<method>
<ejb-name>RealizarPedidoBean</ejb-name>
<method-name>getEstructuraSocio</method-name>
</method>
</method-permission>
<method-permission>
<role-name>SOCIO_REGISTRADO</role-name>
<method>
<ejb-name>RealizarPedidoBean</ejb-name>
<method-name>*</method-name>
</method>
</method-permission>
…
</assembly-descriptor>
</ejb-jar>
```

Código 2.208: Definicion de un método <unchecked/>

En el Código 2.208 se crea la misma situación del Código 2.207. Para cada método es necesario el rol **SOCIO_REGISTRADO**, excepto para **getEstructuraSocio()**, que debe llamarlo todos.

2.22.1.3. @**DenyAll**

A veces resulta sorprendete la anotación **@DenyAll**, que prohíbe todos los accesos. Da igual qué rol tenga el usuario, simplemente no va a poder llamar este método. Es cuestionable entonces por qué se ha incluido este método en la interfaz local o remota. Si

el método no apareciera allí, tampoco se podría llamar. Pero si se ha derivado un session bean de otro, entonces se heredan todos los métodos de la clase base junto con sus definiciones de seguridad. Si entonces se quisiera evitar las llamadas a un método, se recargarían en la clase derivada y se dotaría de la entrada **@DenyAll**.

```
@Target(value={METHOD, TYPE})
@Retention(value=RUNTIME)
public @interface DenyAll
```

Código 2.209: @DenyAll

La anotación también puede utilizarse en el nivel de las clases. Así primero no se puede utilizar ningún método de esta clase hasta que no se le otorgan algunos con otras anotaciones.

```
@Stateful
public class RealizarPedidoBean implements RealizarPedidoRemote
{
…
@DenyAll
public void setPosiciones( HashMap<Integer, Unidadpos> posiciones )
{
}
}
```

Código 2.210: Ejemplo de @DenyAll

El método **setPosiciones()** del Código 2.210 no puede llamarse por ningún cliente. En el Deployment Descriptor se expresa la anotación **@DenyAll** mediante **<exclude-list>** dentro de **<assembly-descriptor>**. Todos los métodos aquí ejecutados de los beans indicados están excluidos de ser usados. La definición de una lista de este tipo en un Deployment Descriptor debería aparecer más a menudo que la utilización de la anotación. Si por ejemplo utilizamos un software comprado y deben excluirse algunos métodos o beans completos, se amplia simplemente el Deployment Descriptor como corresponde.

2.22.1.4. @**RunAs**

Junto a los roles necesarios para poder llamar cada método de un bean, se puede fijar con la anotación **@RunAs** bajo qué roles funcionan los métodos de la clase bean. Esto es relevante entonces cuando también se puede aludir a estos métodos desde otros beans.

La anotación tan solo se puede otrorgar en el nivel de las clases y es válidad siempre para todos lo métodos de esta clase, es decir también para los internos como para ejemplo los métodos del ciclo de vida.

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
public @interface RunAs
{
String value();
```

```
}
```

Código 2.211: @RunAs

Esta anotación también es interesante para los Message-Driven Beans. Como no se puede llamar nunca directamente a través de un usuario no tienen ningún tipo de contexto **Security**. Si deben llamarse cotros beans que dispongan de la anotación **@RolesAllowed**, es necesario asignar un rol de llamada a la clase bean del message-driven bean con ayuda de **@RunAs**.

```
@MessageDriven(…)
@RunAs(“BEAN_INTERNO”)
public class ChatBean implements MessageListener
{
public void onMessage( Message message )
{ … }
}
```

Código 2.212: Ejemplo de @RunAs

En el Código 2.212 se ha proveisto al message-driven bean **ChatBean** del rol **BEAN_INTERNO**. Por lo tanto será posible llamar el método de otro bean dentro del método **onMessage()**, en caso de que el método llamado no disponga de ninguna Security concreta o de que se hay previsto en su lista de roles también el rol **BEAN_INTERNO**.

Si se quisiera otorgar la anotación **@RunAs** a través del Deployment Descriptor se podría hacer con ayuda de la etiqueta **<security-identity>** en la descripción del bean. En el Código 2.213 se provee al message-driven bean **ChatBean** del rol **BEAN_INTERNO**. Del mismo modo esta entrada se realiza también para el resto de tipos de bean dentro del Deployment Descriptor **ejb-jar.xml**.

```
<ejb-jar
xmlns=”http://java.sun.com/xml/ns/javaee”
xmlsn:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:SchemaLocation=”http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd” versión=”3.1”>
<enterprise-beans>
<message-driven>
<ejb-name>ChatBean</ejb-name>
<security-identity>
<run-as>
<role-name>BEAN_INTERNO</role-name>
</run-as>
</security-identity>
</message-driven>
</enterprise-beans>
</ejb-jar>
```

Código 2.213: Deployment Descriptor con etiqueta <run-as>

También es posible anular la anotación **@RunAs** en el Deployment Descriptor. Para un session bean se puede utilizar también dentro de la etiqueta **<security-identity>** en la entrada **<use-caller-identity>** en lugar de **<run-as>**, para expresar que todos los métodos de este bean deben transcurrir bajo el rol del cliente que ha realizado la llamada. En

message-driven beans no se permite esta entrada, porque estos, no tienen nunca un solo usuario asignado.

2.22.2. Comprobación técnica de derechos

Con los métodos descritos hasta ahora tan solo se puede establecer básicamente si un usuario puede llamar un método. Para ello nos orientamos tan solo por los roles asignados a cada usuario y no por cuestiones técnicas. Podrá ralizar pedidos todo aquel que se haya registrado. Se recogerán pedidos de más e 100 euros pero tan solo los que provengan de clientes fijos. Por lo tanto debemos estar en situación de poder consultar dentro de un método en una determinada constelación los roles del usuario, para poder evitar en caso de error que se prosiga trabajando mediante una excepción.

El método **isCallerInRole()** del SessionContext ofrece el valor **true**, cuando se ja asignado a quien realiza la llamada el rol que trasmitimos al método.

```
@Stateful
@DeclareRoles("CLIENTE_FIJO")
public class RealizarPedidoBean implements RealizarPedidoRemote
{
@Resource
SessionContext context;

@RolesAllowed("SOCIO_REGISTRADO")
public void asignarSocio( Vector<GrupoDatos> datos ) throws PedidoException
{
…
if( unidadPos != null && unidadPos.size > 0 )
{
pedido = new Pedido();
…
BigDecimal valorPedido = new BigDecimal(0);
Set<Integer> keys = unidadPos.keySet();
for( Integer key : keys )
{
unidadPos = unidadPos.get(key);

unidadposPk posPk = pos.getPk();
posPk.setPedidoNr(pedido.getPedidoNr());
valorpedido = valorpedido.add(pos.getPrecio().multiply(new BigDecimal(pos.getCantidad())));
manager.merge(pos);
}

if( valorpedido.compareTo(new BigDecimal(1000)) > 0
&& context.isCallerInRole("CLIENTE_FIJO") == false )
{
throw new PedidoException("Valor mercancía demasiado alto", datos);
}
…
}
}
}
```

Código 2.214: Comprobacion del valor de pedido

2.22.2.1. @**DeclareRoles**

Para poder asignar a un servidor de aplicaciones un rol como **CLIENTE_FIJO**, antes debemos darle un nombre. Esto se consigue con la anotación **@DeclareRoles**, que debe usarse en el nivel de las clases. Enumera uno o más roles que son consultados dependiendo de la clase con el método **isCallerInRole()**.

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface DeclareRoles
{
String[ ] value();
}
```

Código 2.215: @DeclareRoles

En el Deployment Descriptor se encuentra la declaración de un rol en la etiqueta **<security-role-ref>** de la clase bean. Si deben definirse varios roles, se repite la entrada de esta etiqueta. En el Código 2.216 se vincula el rol **CLIENTE_FIJO** con la clase **RealizarPedidoBean**.

```
<ejb-jar
xmlns="http://java.sun.com/xml/ns/javaee"
xmlsn:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:SchemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
versión="3.1">
…
<enterprise-beans>
<session>
<ejb-name>RealizarPedidoBean</ejb-name>
<remote>server.sec.RealizarPedidoRemote</remote>
<ejb-class>server.sec.RealizarPedidoBean</ejb-class>
<session-type>Stateful</session-type>
<persistence-context-ref>
<persistence-context-ref-name>
JavaEE
</persistence-context-ref-name>
<persistence-unit-name>
JavaEE
</persistence-unit-name>
<injection-target>
<injection-target-class>
server.sec.RealizarPedidoBean
</injection-target-class>
<injection-target-name>
manager
</injection-target-name>
</injection-target>
</persistence-context-ref>
<security-role-ref>
<role-name>CLIENTE_FIJO</role-name>
</security-role-ref>
</session>
…
</enterprise-beans>
</ejb-jar>
```

Código 2.216: Definición del rol CLIENTE_FIJO

CAPÍTULO III

ANÁLISIS, DISEÑO E IMPLEMENTACIÓN

3.5. Metodología de Desarrollo de Software

3.5.1. Procesos de Desarrollo

El objetivo de un proceso de desarrollo es subir la calidad del software (en todas las fases por las que pasa) a través de una mayor transparencia y control sobre el proceso.

La implantación de un proceso de desarrollo es una labor más a medio-largo plazo que una labor de resultados inmediatos. Cuesta tiempo que los trabajadores se adapten al proceso, pero una vez superado la inversión se recupera con creces. Es por ello que no tiene sentido ajustarse a un proceso al pie de la letra, sino que hay que adaptarlo a las necesidades y características de cada empresa, equipo de trabajo y hasta a cada proyecto.

En los últimos tiempos la cantidad y variedad de los procesos de desarrollo ha aumentado de forma impresionante, sobre todo teniendo en cuenta el tiempo que estuvo en vigor como ley única el famoso desarrollo en cascada. Se podría decir que en estos últimos años se han desarrollado dos corrientes en lo referente a los procesos de desarrollo, los llamados métodos pesados y los métodos ligeros. La diferencia fundamental entre ambos es que mientras los métodos pesados intentan conseguir el objetivo común por medio de orden y documentación, los métodos ligeros (también llamados métodos agiles) tratan de mejora la calidad del software por medio de una comunicación directa e inmediata entre las persona que intervienen en el proceso.

Para el presente trabajo de desarrollo se ha tomado la opción de adoptar el Proceso Unificado de Racional, RUP (Rational Unified Process), con UML (Unified Modeling Language) como complemento ideal para integrar una metodología acorde a las necesidades del proyecto.

3.5.2. **RUP (Rational Unified Process)**

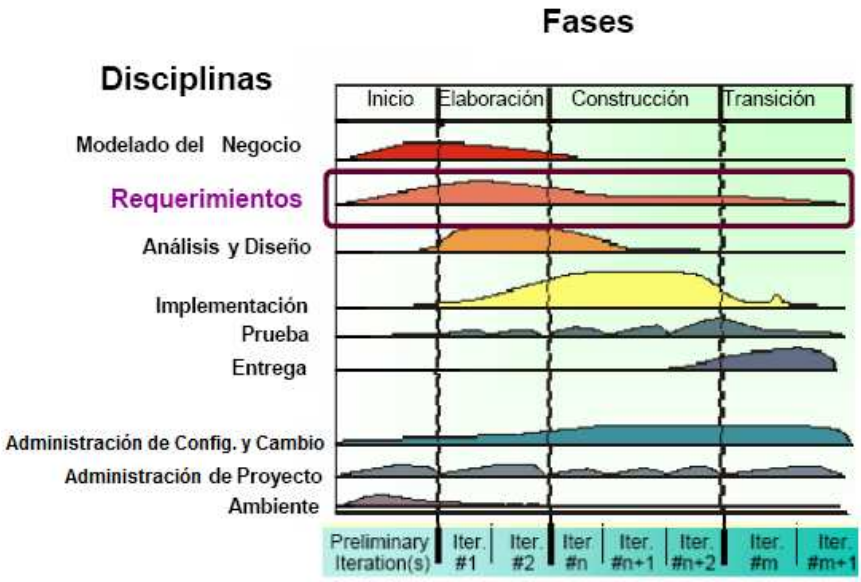


Figura XXII: Vista general de RUP

RUP define varias actividades a realizarse en cada fase del proyecto.

- Modelado de negocio
- Análisis de requisitos
- Análisis y diseño
- Implementación
- Test
- Distribución
- Gestión de configuración y cambios
- Gestión del proyecto
- Gestión del entorno

Y el flujo de trabajo (workflow) entre ellas en base a los llamados diagramas de actividad. El proceso define una serie de roles que se distribuyen entre los miembros del proyecto y que definen las tareas de cada uno y el resultado (artefactos en la jerga de RUP) que se espera de ellos.

Figura XXIII: Flujos de Trabajo de RUP

RUP se basa en casos de uso para describir lo que se espera del software y esta muy orientado a la arquitectura del sistema, documentádose lo mejor posible, basándose en UML (Unified Modeling Language) como herramienta principal.

3.5.2.1. Características del RUP

El RUP es un proceso de desarrollo de software dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental. RUP pretende implementar las mejores prácticas en ingeniería de software, con el objetivo de asegurar la producción de software de calidad, dentro de plazos y presupuestos predecibles.

3.5.2.1.1. Desarrollo Iterativo

Permite una compresión creciente de los requerimientos, a la vez que se va haciendo crecer el sistema. RUP sigue un modelo iterativo que aborda las tareas más riesgosas primero. Así se logra reducir los riesgos del proyecto y tener un subsistema ejecutable tempranamente.

3.5.2.1.2. Administración de Requerimientos

RUP describe cómo: obtener los requerimientos, organizarlos, documentar los requerimientos de funcionalidad y restricciones, rastrear y documentar las decisiones; y cómo captar y comunicar los requerimientos del negocio.

3.5.2.1.3. Arquitecturas basadas en Componentes

El proceso se basa en diseñar tempranamente una arquitectura base ejecutable. Esta arquitectura debe ser: flexible, fácil de modificar, intuitivamente comprensible, y debe promover la reutilización de componentes.

3.5.2.1.4. **Modelamiento Visual**

RUP propone un modelamiento visual de la estructura y el comportamiento de la arquitectura y las componenetes. En este esquema, los bloques de construcción deben ocultar detalles, permitir la comunicación en el equipo de desarrollo, y permitir analizar la consistencia entre las componentes, entre el diseño y entre la implementación. UML es la base del modelamiento visual de RUP.

3.5.2.1.5. **Verificación de la calidad de software**

No sólo la funcionalidad es esencial, también el rendimiento y la confiabilidad. RUP ayuda a planificar, diseñar, implementar, ejecutar y evaluar pruebas que verifiquen estas cualidades.

3.5.2.1.6. **Control de cambios**

Los cambios son inevitables, peros es necesario evaluar si éstos son necesarios y también es necesario rastreas su impacto. RUP indica como controlar, rastrear y monitorear los cambios dentro del proceso iterativo de desarrollo.

3.5.2.2. **Fases del RUP**

RUP divide el proceso de desarrollo en ciclos, donde se obtiene un producto al final de cada ciclo. Cada ciclo se divide en cuatro Fases: Concepcion, Elaboracion, Contruccion y Transaccion. Cada fase concluye con un hito bien definido donde deben tomarse ciertas decisiones.

3.5.2.2.1. **Fase de Concepción (Inicio)**

En esta fase se establece la oportunidad y alcance del proyecto. Se idenfican todas las entidades externas con las que se trata (actores) y se define la interaccion en un alto nivel de abstracción: se deben identificar todos los casos de uso, y se deben describir algunos en detalle. La oportunidad del negocio

incluye: definir los criterios de éxito, identificación de riesgos, estimación de recursos necesarios, y plan de las fases incluyento hitos.

3.5.2.2.2. **Fase de Elaboración**

Definir y validar una arquitectura estable. Se hace un refinamiento de la Visión del Sistema, basándose en nueva información obtenida durante esta fase, se establece una sólida compresión de los casos de uso más críticos que definen las decisiones arquitectónicas y de planificación.

Creación de los planes de desarrollo detallados para las iteraciones de la fase de construcción.

3.5.2.2.3. **Fase de Construcción**

Gestión de los recursos, optimización y control de los procesos de construcción del software. Se completa el desarrollo de los componentes y/o subsistemas, probándolos contra un conjunto definido de criterios aprobados al inicio del proyecto.

3.5.2.2.4. **Fase de Transición**

Ejecución de los planes de implantación. Se finalizan los manuales de usuario y mantenimiento. Pruebas del sistema en el entorno de explotación. Creación de una ***release*** del sistema. Validación del sistema por los usuarios. Ajuste fino del sistema según la validación con el usuario. Se facilita la transición del sistema al personal de mantenimiento. Se pone el producto a disposición del usuario final.

3.5.3. **UML (Unified Modeling Language)**

Es un lenguaje de modelado visual que se usa para especificar, visuarlizar, construir y documentar modulos de un sistema de software. Se usa para entender, diseñar, configurar, mantener y controlar la información sobre los sistemas a construir.

UML es un lenguaje de propósito general para el modelado orientado a objetos. Es también un lenguaje de modelamiento visual que permite una abstracción del sistema y sus componentes. UML es independiente del ciclo de desarrollo a seguir, puede encajar en un tradicional ciclo de cascada, o en un evolutivo ciclo en espiral o incluso en los métodos ágiles de desarrollo.

3.5.3.1. **Objetivos del UML**

- UML es un lenguarje de modelado de propósito general que pueden usar todos los modeladores. No tiene propietario y está basado en el común acuerdo de gran parte de la comunidad informática.

- UML no pretende ser un método de desarrollo completo. No incluye un proceso de desarrollo paso a paso. UML incluye todos los conceptos que se consideran necesarios para utilizar un proceso moderno iterativo, basado en construir una sólida arquitectura para resolver requisitos dirigidos por casos de uso.

- Ser tan simple como sea posible pero manteniendo la capacidad de modelar toda la gama de sistemas que se necesita construir. UML necesita ser lo suficiente expresivo para manejar todos los conceptos que se originan en un sistema moderno, tales como la concurrencia y distribución, asi como también los mecanismos de la ingeniería de software, como son la encapsulación y componentes.

- Debe ser un lenguaje universal, como cualquier lenguaje de propósito general. Imponer un estándar mundial.

3.5.3.2. **Arquitectura de UML**

Arquitectura de cuatro capas, definido a fin de cumplir con la especificación Meta Object Facility del OMG:

- Meta-meta-modelo: define el lenguaje para especificar meta-modelos.
- Meta-modelo: define el lenguaje para especificar modelos.
- Modelo: define el lenguaje para describir un dominio de información.
- Objetos de usuario: define un dominio de información específico.

3.5.3.3. **Áreas conceptuales de UML**

Los conceptos y modelos de UML pueden agruparse en las siguientes areas conceptuales:

3.5.3.3.1. **Estructura estática**

Cualquier modelo preciso debe primero definir su universo, esto es, los conceptos clave de la aplicación, sus propiedades internas, y las relaciones entre cada una de ellas. Este conjunto de construcciones es la estructura estática. Los conceptos de la aplicación son modelados como clases, cada una de las cuales describe un conjunto de objetos que almacenan información y se comunican para implementar un comportamiento.

3.5.3.3.2. **Comportamiento dinámico**

Hay dos formas de modelar el comportamiento, una es la historia de la vida de un objeto y la forma como interactúa con el resto del mundo y la otra es por los patrones de comunicación de un conjunto de objetos conectados, es decir la forma en que interactúan entre sí. La visión de un objeto aislado es una máquina de estados, muestra la forma en que el objeto responde a los eventos en función de su estado actual. La visión de la interaccion de los objetos se representa con los enlaces entre objetos junto con el flujo de mensajes y los enlaces entre ellos.

3.5.3.3.3. **Construcciones de implementación**

Los modelos UML tienen significado para el análisis lógico y para la implementación física. Un componente es una parte física reemplazable de un sistema y es capaz de responder a las peticiones descritas por un conjunto de interfaces. Un modo es un recurso computacional que define una localización durante la ejecución de un sistema. Puede contener componentes y objetos.

3.5.3.3.4. **Organización del modelo**

La información del modelo debe ser dividida en piezas coherentes, para que los equipos puedan trabajar en las diferentes partes de forma concurrente. El conocimiento humano requiere que se organice el contenido del modelo en paquetes de tamaño modesto. Los paquetes son unidades organizativas, jerárquicas y de propósito general de los modelos de UML. Pueden usarse para almacenamiento, control de acceso, gestión de la configuración y construcción de bibliotecas que contengan fragmentos de código reutilizable.

3.5.3.4. **Diagramas de UML**

Un modelo captura una vista de un sistema del mundo real. Es una abstracción de dicho sistema, considerando un cierto propósito. Así, el modelo describe completamente aquellos aspectos del sistema que son relevantes al propósito del modelo, y a un apropiado nivel de detalle.

Un proceso de desarrollo de software debe ofrecer un conjunto de modelos que permitan expresar el producto desde cada una de las perspectivas de interés. UML contempla un manejo detallado de diagramas entre los cuales se pueden mencionar los siguientes:

Área	Vista	Diagramas	Conceptos Principales
Estructural	Vista Estática	Diagramas de Clases	Clase, asociación, generalización, dependencia, realización, interfaz
	Vista de Casos de Uso	Diagramas de Casos de Uso	Caso de Uso, Actor, asociación, extensión, generalización.
	Vista de Implementación	Diagramas de Componentes	Componente, interfaz, dependencia, realización.
	Vista de Despliegue	Diagramas de Despliegue	Nodo, componente, dependencia, localización.
Dinámica	Vista de Estados	Diagramas de Estados	Estado, evento, transacción, acción.
	Vista de Actividades	Diagramas de Actividad	Estado, actividad, transición, determinación, división, unión.
	Vista de Interacción	Diagramas de secuencia	Interaccion, objeto, mensaje, activación.
		Diagramas de Colaboración	Colaboración, interaccion, rol de colaboración, mensaje.
Administración o Gestión de Modelo	Vista de Gestion de Modelo	Diagramas de Clases	Paquete, subsistema, modelo.
Extencion de UML	Todas	Todos	Restricción, estereotipo, valores, etiquetados.

Tabla III.II: Diagramas de UML

3.5.3.4.1. **Diagramas de Objetos**

Objeto es una entidad discreta con límite bien definido y con identidad, es una unidad atómica que encapsula estado y comportamiento. La encapsulación en un objeto permite una alta cohesion y un bajo acoplamiento. El Objeto puede ser reconocido también como una instancia de la clase a la cual pertenece dicho objeto.

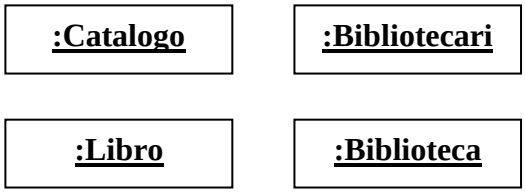


Figura XXIV: Diagrama de Objetos

Un objeto se puede ver desce dos perspectivas relacionadas: la primera, como una entidad de un determinado instante de tiempo que posee un valor especifico (Un objeto puede caracterizar una entidad física -coche-) y la segunda, como un poseedor de identidad que tiene distintos valores a lo largo del tiempo (abstracta -ecuación matemática-).

3.5.3.4.2. **Diagramas de Clases**

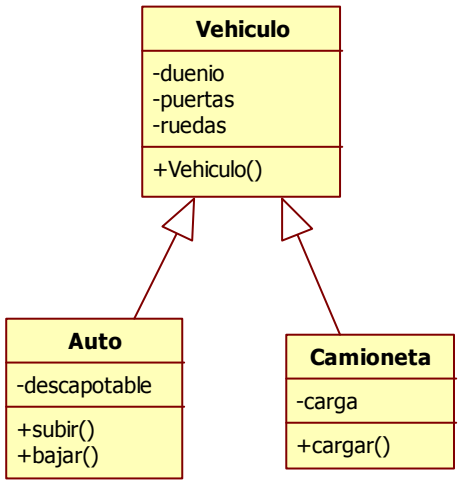


Figura XXV: Diagrama de Clases

El Diagrama de Clases es el diagrama principal para el análisis y diseño. Un diagrama de clases presenta las clases del sistema con sus relaciones estructurales y de herencia. La definición de clase incluye definiciones para establecer las clases, objetos, atributos y operaciones.

Cada clase se representa en un rectángulo con tres compartimientos:

- Nombre de la Clase
- Atributos de la Clase
- Operaciones de la Clase

Los atributos de una clase no deberían ser manipulables directamente por el resto de objetos. Por esta razón se creación niveles de visibilidad para los elementos que son:

- (-) Privado: es el máa fuerte. Esta parte es totalmente invisible (excepto para clases friends en terminología C++).
- (#) Los atributos/operaciones protegidos están visibles para las clases fiends y para las clases derivadas de la original.
- (+) Los atributos/operaciones públicos son visibles a otras clases (cuando se trata de atributos se está transgrediendo el principio de encapsulacion).

Los enlaces entre objetos se representan entre la respectivas clases y sus formas de relación son:

- Asociación y Agregacion (vista como un caso particular de asociación)
- Generalización/Epecialización

3.5.3.4.3. Diagramas de Caso de Uso

Casos de Uso es una técnica para capturar información de cómo un sistema o negocio trabaja, o de cómo se desea que trabaje. No pertenece estrictamente al enfoque orientado a objetos, es una técnica para captura de requisitos.

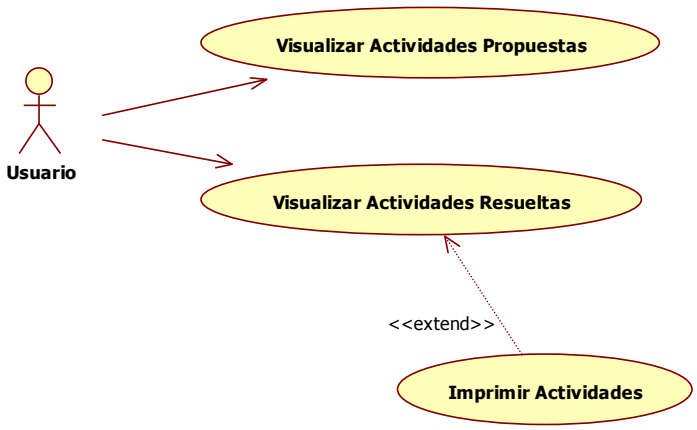


Figura XXVI: Diagrama de Casos de Uso

- Los Casos de Uso (Ivar Jacobson) describen bajo la forma de acciones y reacciones el comportamiento de un sistema desde el punto de vista del usuario.

- Permiten definir los límites del sistema y las relaciones entre el sistema y el entorno.

- Los Casos de Uso son descripciones de la funcionalidad del sistema independientes de la implementación.

- Cubren la carencia existente en métodos previos (OMT, Booch) en cuanto a la determinación de requisitos.

- Están basados en el lenguaje natural, es decir, es accesible por los usuarios.

3.5.3.4.4. Diagramas de Actividades

El Diagrama de Actividad es una especialización del Diagrama de Estado, organizado respecto de las acciones y usado para especificar:

- Un método
- Un caso de uso
- Un proceso de negocio (workflow)

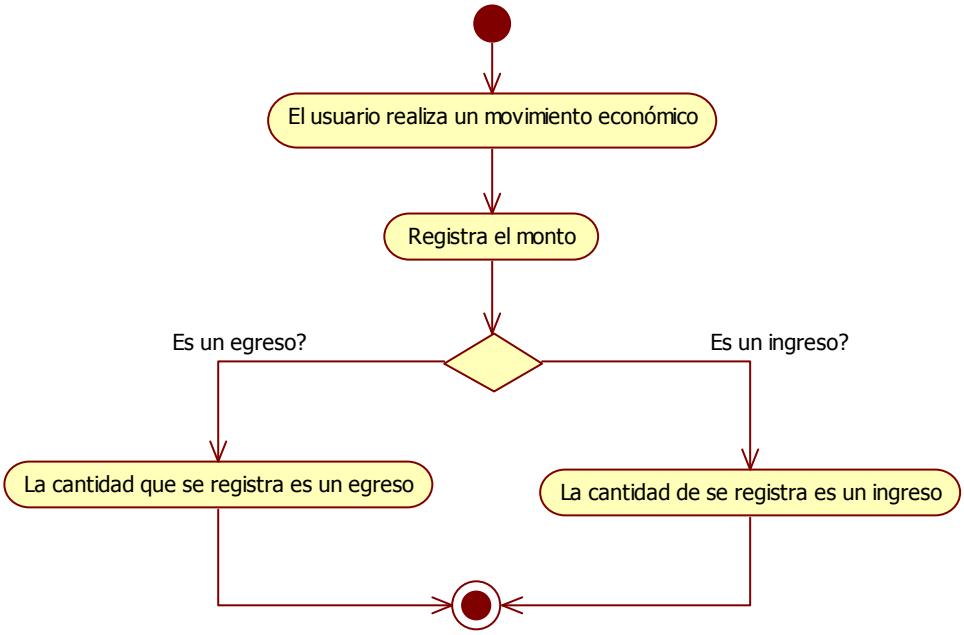


Figura XXVII: Diagrama de Actividades

Un estado de actividad representa una actividad: un paso en el flujo de trabajo o la ejecución de una operación. Las actividades se enlazan por transiciones automáticas. Cuando una actividad termina se desencadena el paso a la siguiente actividad. Un diagrama de actividades es provechoso para entender el comportamiento de alto nivel de la ejecución de un sistema, sin profundizar en los detalles internos de los mensajes.

3.5.3.4.5. **Diagramas de Estado**

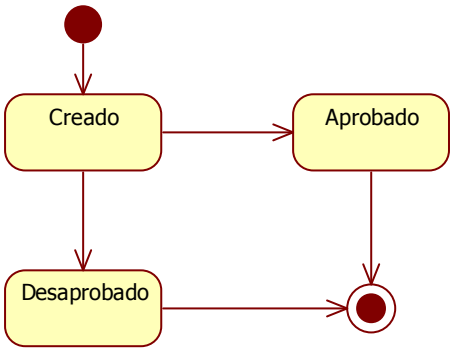


Figura XXVIII: Diagrama de Estado

Muestra el conjunto de estados por los cuales pasa un objeto durante su vida en una aplicaion, junto con los cambios que permiten pasar de un estado a otro. Son útiles sólo para los objetos con un comportamiento significativo. Cada objeto está en un estado en cierto instante. El estado está caracterizado parcialmente por los valores de los atributos del objeto. El estado en el que se encuentra un objeto determina su comportamiento. Cada objeto sigue el comportamiento descrito en el Diagrama de Estados asociado a su clase. Los Diagramas de Estados y escenarios son complementarios, los Diagrmas de Estados son autómatas jerárquicos que permiten expresar concurrencia, sincronización y jerarquías de objetos, son grafos dirigidos y deterministas.

3.5.3.4.6. Diagramas de Interacción

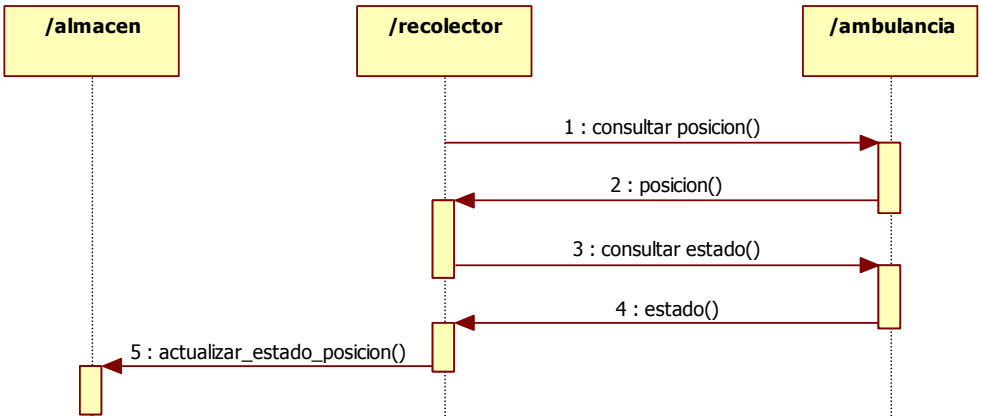


Figura XXIX: Diagrama de Interacción

La vista de interaccion describe secuencias de intercambios de mensajes entre los roles que implementan el comportamiento de un sistema. Un rol clasificador, es la descripción de un objeto, que desempeña un determinado papel dentro de una interaccion, distinto de los otros objetos de la misma clase. Los diagramas de interaccion muestran cómo se comunican los objetos en una interacción.

3.5.3.4.7. Diagramas de Secuencia

Representa una interaccion, un conjunto de comunicaciones entre objetos organizadas visualmente por orden tempora. A diferencia de los diagramas de colaboración, los diagramas de secuencia incluyen secuencias temporales pero no incluyen las relaciones entre objetos.

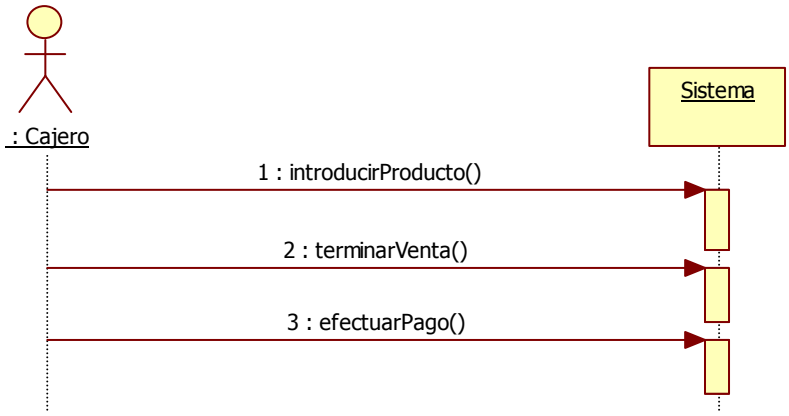


Figura XXX: Diagrama de Secuencia

Un diagrama de secuencia puede mostrar un escenario, es decir, una historia individual de transacción. Un uso de un diagrama de secuencia es mostra la secuencia del comportamiento de un caso de uso.

3.5.3.4.8. Diagramas de Colaboración

Un Diagrama de Colaboración es también un diagrama de clases que mediante roles de clasificador y roles de asociación en lugar de sólo clasificadores y asociaciones. Los roles de clasificador y los de asociación describen la configuración de los objetos y de los enlaces que pueden ocurrir cuando se ejecuta una instancia de la colaboración.

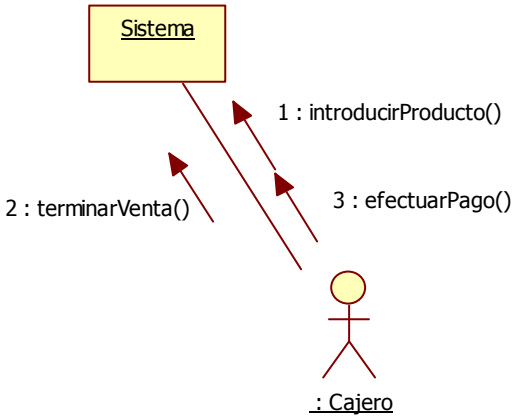


Figura XXXI: Diagrama de Colaboración

Un uso de un diagrama de colaboración es mostrar la implementación de una operación. La colaboración muestra los parámetros y las variables locales de la operación, así como asociaciones más permanentes.

3.5.3.4.9. Diagramas de Componentes

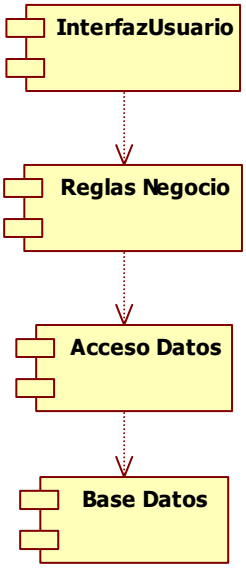


Figura XXXII: Diagrama de Componentes

Los Diagramas de Componentes describen los elementos físicos del sistema y sus relaciones. Muestran las opciones de realización incluyendo código fuente, binario y ejecutables. Los componentes representan todos los tipos de elementos de software que entran en la fabricación de aplicaciones informáticas. Pueden ser simples archivos, paquetes, bibliotecas cargadas dinámicamente, etc. Las relaciones de dependencia se utilizan en los diagramas de componentes para indicar que un componente utiliza los servicios ofrecidos por otro componente.

Un diagrama de componentes representa las dependencias entre componentes software, incluyendo componentes de código fuente, componentes del código binario y componentes ejecutables. Un módulo de software se puede representar como componente. Algunos componentes existen en tiempo de compilación, algunos en tiempo de enlace y algunos en tiempo de ejecución, otros en varias de estás.

3.5.3.4.10. **Diagramas de Despliegue**

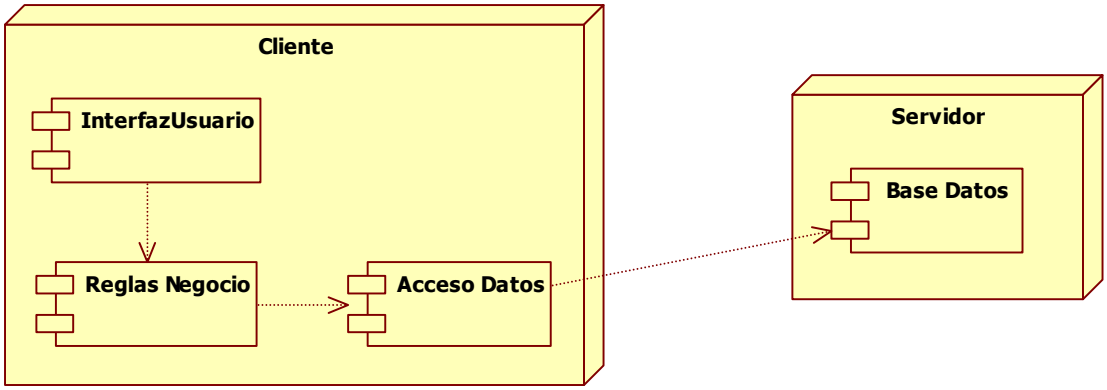


Figura XXXIII: Diagramas de Despliegue

Los Diagramas de Despliegue muestran la disposición física de los distintos nodos. La vista de despliegue representa la disposición de las instancias de componentes de ejecución en instancias de nodos conectados por enlaces de comunicación. Un nodo es un recurso de ejecución.

3.5.3.4.11. Diagrama de Paquetes

Cualquier sistema grande se debe dividir en unidades más pequeñas, de modo que las personas pueden trabajar con una cantidad de información limitada, a la vez y de modo que los equipos de trabajo no interfieran con el trabajo de los otros.

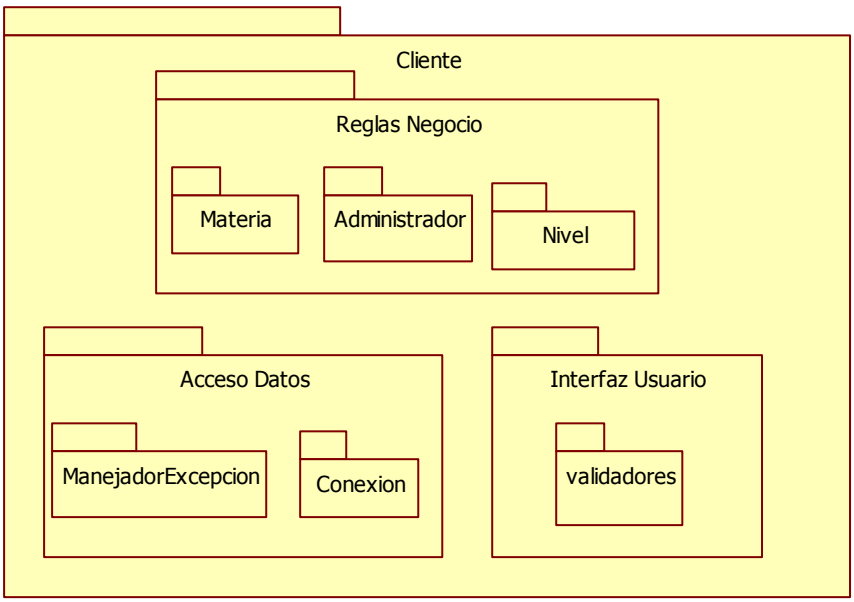


Figura XXXIV: Diagrama de Paquetes

Un paquete es una parte de un modelo. Cada parte del modelo debe pertenecer a un paquete. Pero para ser funcional, la asignación debe seguir cierto principio racional, tal como funcionalidad común, implementación relacionada y punto de vista común. UML no impone una regla para componer los paquetes.

3.6. **Requerimientos**

En la disciplina de Requerimientos nos permitirá gestionar las necesidades del proyecto en forma estructurada, mejorar la capacidad de predecir cronogramas de proyectos, así como sus resultados, mejorar la calidad del software, es decir cumplir con un conjunto de funcionalidad, facilidad de uso, confiabilidad, desempeño, etc.

Para más detalles sobre la disciplina de Requerimientos ver el **Anexo 2** sección **Requerimientos – Especificación de Requerimientos de Software (SRS)**.

3.7. **Análisis y Diseño**

Con el Análisis se puede explicar y pronosticar la actuación de los sistemas bajo condiciones variables y así poder prever medidas administrativas adecuadas y oportunas. A través de este proceso se obtiene una visión de la interacción del sistema con su medio ambiente, dando como resultado una retroalimentación.

En cuanto al Diseño de un sistema de información, puede decirse su importancia resalta por cuanto, responde a la forma en la que el sistema cumplirá con los requerimientos identificados durante la fase de análisis. El Diseño de un sistema de información debe responder a los requerimientos que fueron el resultado de la fase de análisis, dando como resultado un sistema o proceso con suficientes detalles como para permitir su interpretación y realización física.

Para más detalles sobre el Análisis y Diseño ver el **Anexo 2Documentación Técnica** sección **Análisis y Diseño**.

3.8. **Implementación**

La Implementación (ver **Anexo 2**, **Documentación Técnica** sección **Implementación**) forma parte de la Fase de Construcción del RUP, en esta disciplina se detallan los recursos hardware y software utilizados para la misma, además se implementaron los requerimientos citados en la Especificación de Requerimientos (Ver **Anexo 2**, **Documentación Técnica**sección **Requerimientos – Especificación de Requerimientos de Software (SRS)**), dando un control al desarrollo adecuado de cada uno de estos.

CAPÍTULO IV

ESTUDIO DE RESULTADOS

En muchos estudios, incluidos la mayoría de los desarrollos de software, es necesario comparar ciertas características en dos o más herramientas de programación; o a su vez en dos o más tecnologías de desarrollo. Tal sería el caso, por ejemplo, si pensamos que un proceso nuevo puede tener un porcentaje de mejoría que otro, o cuando nos planteamos si utilizar una tecnología que mejora la productividad al momento de desarrollar un sistema.

Con el fin de reducir el tiempo de desarrollo se ha planteado el siguiente escenario:

Se tiene dos aplicaciones web, **la primera** que fue desarrolla de una manera tradicional creando clase (java beans), y creando métodos que interactúen con esas clases, denominada la Capa de Acceso a Datos, las mismas que se enlazan mediante referencias con páginas jsp para la Capa de Presentación, en este caso se ha utilizado como gestor de base de datos a SQL Server 2000, y la **segunda** en el que tenemos una aplicación creada con tecnología EJB (Enterprise Java Beans), aplicación que posee: **EJB's de Entidad** para la denominda Capa de Persistencia, **EJB de Sesiones** para la implementación de la Lógica de Negocio, **servlets** como controladores que interactúan con las **jsp**, todo esto ensamblado sobre un servidor de aplicaciones GlassFish 3.1 y con gestor de base de datos a Oracle 10g XE.

Para ambos casos tenemos un conjunto de clases y métodos, la diferencia está en que EJB proporciona un conjunto de servicios adicionales, y es aquí donde se necesitan de más o menos líneas de código, además de menos o más tiempo, con lo cual se tomarán en cuenta a estas dos variables (de las tres) para la comparación entre ambas formas de desarrollo.

4.8. Modificaciones entre formas de desarrollo

Como se mencionó, la primera aplicación está constituida por una **Capa de Acceso a Datos** utilizando clases java (java beans) haciendo uso de jdbc (Java Database Conectivity), la **Capa de Presentación** está realizada con **jsp**, las cuales realizan referencias a los java beans necesarios dentro de cada una de ellas.

Se puede observar en la imagen a continuación, parte de los archivos contenidos en cada una de las capas.

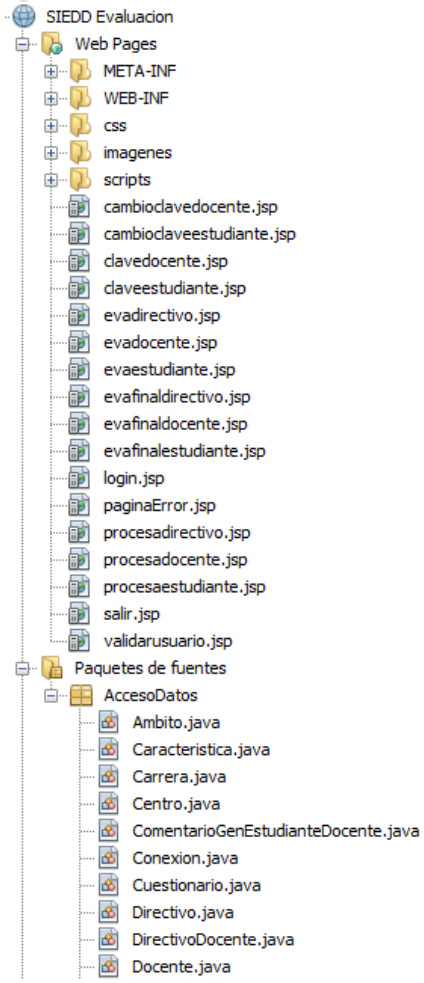


Figura XXXV. Capas de Aplicación sin tecnología EJB

Ahora, en la segunda aplicación en la que se utilizó tecnología EJB se instauraron los siguientes cambios:

Las Capa de Persistencia, que está constituida por EJB's de Entidad remplaza a la Capa de Acceso a Datos, pero en está se tienen no solo las tablas en si de la base de datos, sino que se tienen también cada una de las referencias existentes en la misma a través del Mapeo Objeto-Relacion. Está capa implementa una Unidad de Persistencia para la comunicación de los cambios a la base de datos (en este caso Oracle 10g XE).

La Capa de Lógica de Negocios, está compuesta por EJB's de Sesión, que implementan referencias hacia las entidades de EJB's necesarias a través de interfaces locales o

remotas, que son intermediarios en la comunicación entre la Capa de Lógica de Negocios y la Capa de Presentación.

Tanto la Capa de Lógica de Negocios como la Capa de persistencia, conforman el módulo EJB.

Dentro del Módulo Web, se tiene la Capa de Presentación en donde se utilizan tanto servlets como jsp, ya que los servlets son los que se comunican con el módulo EJB para obtener y establecer cambios dentro de los datos, o también para ciertas tareas de presentación de datos; mientras que las jsp se comunican con los servlets simplemente para presentación de datos.

En la Figura XXXVI y XXXVII, se pueden observar los módulos, las capas y parte de los objetos mencionados anteriormente.

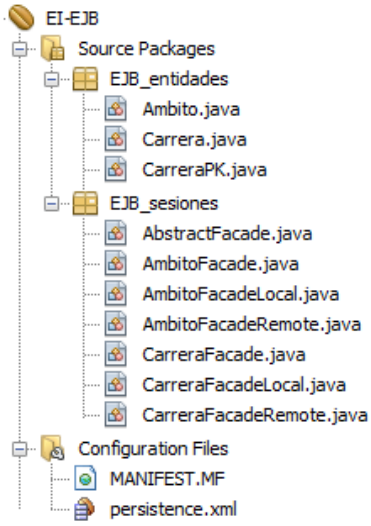


Figura XXXVI. Módulo EJB y sus Capas de Persistencia y Lógica de Negocios.

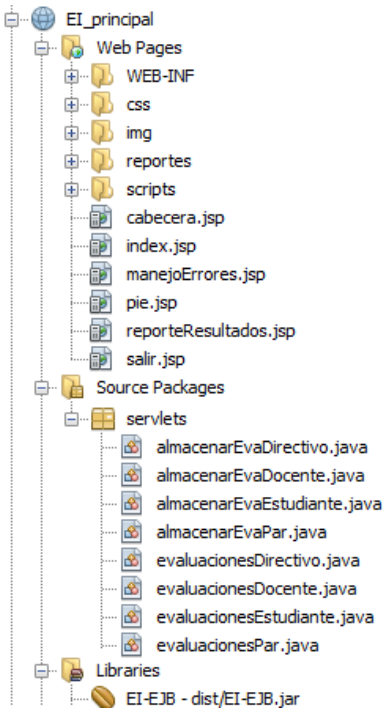


Figura XXXVII. Módulo Web con Servlets y JSP.

4.9. Establecimiento de variables

Para realizar la comparación de resultados entre el primer método de desarrollo (sin EJB) y el segundo (con EJB) se establecieron 3 variables, con la siguiente ponderación:

Variable	Peso
Total LDC del Programa	10%
Tiempo	40%
LDC al Programar	50%
Total	100%

Tabla IV.III: Ponderación de Variables

4.10. Definición de las variables

A continuación se describen cada una de las variables y como se las utilizarán en la comparación:

4.10.1. Total LDC del Programa

La definición de Línea de Código (LDC), aunque es básica para muchas métricas del software, es ambigua. El significado de línea de código varía de un lenguaje a otro, pero también dentro de un mismo lenguaje de programación.

En la plataforma Java, por ejemplo, una línea de código puede ser:

String nombre = “”, apellido = “”, dirección = “”;

Para contar el total de líneas de código de las aplicaciones se utilizará la herramienta “Practiline Source Code Line Counter”.

4.10.2. Tiempo

El tiempo es la magnitud física que mide la duración o separación de las cosas sujetas a cambio, de los sistemas sujetos a observación, esto es, el periodo que transcurre entre el estado del sistema cuando éste aparentaba un estado X y el instante en el que X registra una variación perceptible para un observador. Es la magnitud que permite ordenar los sucesos en secuencias, estableciendo un pasado, un presente y un futuro.

4.10.3. LDC al Programar

Esta variable se tomará en cuenta, ya que al momento de programar, se puede optar por tecnologías o herramientas de programación que nos generan código, que probablemente al finalizar la aplicación, esta puede tener más líneas de código que una que no haya ocupado estas herramientas o tecnologías.

Como para ambos casos se hace uso del IDE NetBeans 6.9, ha sido necesario tomar en cuenta esta variable.

Entonces para poder comparar esta variable se ha sacado una muestra de métodos programados para la lógica del negocio directamente en la aplicación de un total de 74, que aplicando la fórmula de muestreo queda:

$$n = \frac{N \cdot \sigma^2 \cdot Z^2}{(N-1)E^2 + \sigma^2 \cdot Z^2}$$

n = Tamaño de la muestra
N= Universo de la población
σ^2= Varianza (0.5)
Z= Nivel de confianza
E= Límite aceptable del error muestral

Fórmula 1: Tamaño de la muestra

Para el Nivel de Confianza (Z) utilizamos la distribución Gaussiana.

Coeficiente de confianza	50%	68,27%	90%	95%	95,45%	99%	99,37%

Z	0,647	1,00	1,645	1,96	2,00	2,58	3,00

Tabla IV.IV: Coeficientes de confianza

Datos:

n = X
N= 74
σ^2= $(0.5)^2$
Z= 1,96
E= 0,05

Aplicando la Fórmula 1:

$$n = \frac{(74) \cdot (0{,}5)^2 \cdot (1{,}96)^2}{(74-1)(0{,}05)^2 + (0{,}5)^2 \cdot (1{,}96)^2}$$

$$n = \frac{71{,}0696}{0{,}1825 + 0{,}9604}$$

$$n = \frac{71{,}0696}{1{,}1429}$$

$$n = 62{,}1836 \cong 62$$

$$n = 62$$

Por lo tanto, la muestra con la cual se trabajará en el apartado 4.4.3 son 62 métodos, los mismos que se tabularán con la herramienta MINITAB 15.0, para la generación del reporte de media, moda, desviación estándar, error típico y su gráfico correspondiente.

4.11. Generación de resultado de las aplicaciones

4.11.1. Variable “Total LDC del programa”

Los resultados arrojados por la aplicación “Practiline Source Code Line Counter” se muestran en las figuras XXXVIII y XXXIX:

File Name	Nominal Lines	Source Code Lines	Source Code Lines (%)	Comment Lines	Comment Lines (%)	Blank Lines	Blank Lines (%)	Total Lines
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\centro.jsp	63	58	92,06 %	0	0,00 %	3	4,76 %	63
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\cerrarproceso.jsp	34	26	76,47 %	4	11,76 %	2	5,88 %	34
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\css\principal.css	25	25	100,00 %	0	0,00 %	0	0,00 %	25
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\encuestasdocente.jsp	72	68	94,44 %	0	0,00 %	2	2,78 %	72
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\encuestasestudiante.jsp	72	67	93,06 %	0	0,00 %	3	4,17 %	72
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\listarcentros.jsp	77	66	85,71 %	6	7,79 %	3	3,90 %	77
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\listarmaterias.jsp	187	159	85,03 %	16	8,56 %	10	5,35 %	187
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\listarprogramas.jsp	85	74	87,06 %	6	7,06 %	3	3,53 %	85
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\login.jsp	59	56	94,92 %	0	0,00 %	1	1,69 %	59
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\META-INF\context.xml	2	2	100,00 %	0	0,00 %	0	0,00 %	2
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\opcionesadmin.jsp	68	59	86,76 %	0	0,00 %	7	10,29 %	68
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\participacion.jsp	111	92	82,88 %	12	10,81 %	5	4,50 %	111
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\participaciondocentes.jsp	98	86	87,76 %	6	6,12 %	4	4,08 %	98
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\participacionestudiantes.jsp	99	86	86,87 %	6	6,06 %	5	5,05 %	99
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\programa.jsp	19	15	78,95 %	0	0,00 %	2	10,53 %	19
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\reseteardocente.jsp	24	20	83,33 %	0	0,00 %	2	8,33 %	24
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\reseteardocente.jsp	24	20	83,33 %	0	0,00 %	2	8,33 %	24
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\resetearclaveestudiante.jsp	74	69	93,24 %	0	0,00 %	3	4,05 %	74
D:\Aplicaciones\SIEDD_UED\SIEDD Administrador\build\web\resetearclavedocente.jsp	74	69	93,24 %	0	0,00 %	3	4,05 %	74
Total:	**30785**	**25348**	**82,34 %**	**2542**	**8,26 %**	**2691**	**8,74 %**	**30785**

Figura XXXVIII: Resultados “Total LDC del programa” sin EJB.

File Name	Nominal Lines	Source Code Lines	Source Code Lines (%)	Comment Lines	Comment Lines (%)	Blank Lines	Blank Lines (%)	Total Lines
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\cabecera.jsp	13	12	92,31 %	0	0,00 %	1	7,69 %	13
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\css\calendario\css\calendario.css	264	224	84,85 %	4	1,52 %	36	13,64 %	264
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\css\calendario\css_notes\dwsync.xml	4	4	100,00 %	0	0,00 %	0	0,00 %	4
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\css\calendario\img_notes\dwsync.xml	14	14	100,00 %	0	0,00 %	0	0,00 %	14
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\css\calendario\js\Calendario.js	545	488	89,54 %	6	1,10 %	51	9,36 %	545
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\css\calendario\js\Fecha.js	482	424	87,97 %	10	2,07 %	48	9,96 %	482
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\css\calendario\js_notes\dwsync.xml	7	7	100,00 %	0	0,00 %	0	0,00 %	7
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\css\Principal.css	91	79	86,81 %	5	5,49 %	7	7,69 %	91
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\index.jsp	190	189	99,47 %	0	0,00 %	1	0,53 %	190
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\manejoErrores.jsp	140	136	97,14 %	0	0,00 %	2	1,43 %	140
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\pie.jsp	18	18	100,00 %	0	0,00 %	0	0,00 %	18
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\reporteCertificado.jsp	43	28	65,12 %	6	13,95 %	5	11,63 %	43
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\reporteCuestionarios.jsp	62	47	75,81 %	6	9,68 %	5	8,06 %	62
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\reporteGeneral.jsp	136	100	73,53 %	9	6,62 %	7	5,15 %	136
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\reporteListadoSinEvaluar.jsp	70	45	64,29 %	9	12,86 %	8	11,43 %	70
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\reporteParticipacion.jsp	118	86	72,88 %	9	7,63 %	7	5,93 %	118
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\reporteResultados.jsp	45	30	66,67 %	6	13,33 %	5	11,11 %	45
E:\Mis Documentos\NetBeansProjects\EI_administracion\build\web\salir.jsp	28	22	78,57 %	2	7,14 %	2	7,14 %	28
Total:	**45620**	**39763**	**87,16 %**	**1854**	**4,06 %**	**3600**	**7,89 %**	**45620**

Figura XXXIX: Resultados “Total LDC del programa” con EJB.

En la Tabla IV.3, se resumen los resultados obtenidos con la aplicación “Practiline Source Code Line Counter”, de la cual sólo se van a tomar los valores marcados con la palabra **OK**, ya que son los valores que resaltan para comprobar nuestro objetivo.

Detalle	Valor sin EJB	Valor con EJB
Líneas en Blanco	2691	3600
Líneas comentadas	2542	1854
Líneas de Código Fuente	**25348 (ok)**	**39763 (ok)**
Total	30785	45620

Tabla IV.V: Resumen de resultado obtenidos con “Practiline Source Code Line Counter”

4.11.2. Variable “Tiempo”

Para la variable Tiempo los valores tomados en cada una de las aplicaciones son la cantidad de tiempo (en semanas) dedicado en programar cada una de las aplicaciones con y sin EJB.

Ts = 6 semanas (Tiempo sin EJB)
Tc = 4 semanas (Tiempo con EJB)

4.11.3. Variable "LDC al Programar"

Para obtener las LDC al Programar se utilizará el resultado obtenido en la definición de la variable "LDC al Programar" (Ver apartado 4.3.3) que es 62, donde se han obtenido los siguientes resultados para cada uno de los métodos en sus respectivas clases:

- **Clase CARRERA**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
buscarCarrerasDeEscuela	27	4
recuperarTodos	1	0
Total	**28**	**4**

Tabla IV.VI: Resultados LDC al Programar en la Clase CARRERA

- **Clase DIRECTIVODOCENTE**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Actualizar	31	0
buscarPorCedulaDirectivo	34	4
recuperarTodos	11	0
Total	**76**	**4**

Tabla IV.VII: Resultados LDC al Programar en la Clase DIRECTIVODOCENTE

- **Clase DIRECTIVO**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	25	0
Actualizar	25	0
buscarPorCuenta	28	4
recuperarTodos	11	0
Total	**89**	**4**

Tabla IV.VIII: Resultados LDC al Programar en la Clase DIRECTIVO

- **Clase DOCENTEDOCENTE**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	24	0
Actualizar	24	0
buscarPorCedulaDocente	32	4
recuperarTodos	12	0
Total	**92**	**4**

Tabla IV.IX: Resultados LDC al Programar en la Clase DOCENTEDOCENTE

- **Clase DOCENTE**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	25	0
Actualizar	27	0
buscarPorCuenta	29	4
recuperarTodos	11	0
Total	**92**	**4**

Tabla IV.X: Resultados LDC al Programar en la Clase DOCENTE

- **Clase ESCUELA**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
buscarEscuelasDeFacultad	29	4
recuperarTodos	11	0
Total	**40**	**4**

Tabla IV.XI: Resultados LDC al Programar en la Clase ESCUELA

- **Clase ESTADISTICAGENERAL**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
calcularResultados	27	2
Total	**27**	**2**

Tabla IV.XII: Resultados LDC al Programar en la Clase ESTADISTICAGENERAL

- **Clase ESTANDAR**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	28	0
Actualizar	30	0
Eliminar	26	0
buscarEstandaresDelAmbito	31	4
recuperarTodos	11	0
Total	**126**	**4**

Tabla IV.XIII: Resultados LDC al Programar en la Clase ESTANDAR

- **Clase ESTUDIANTEDOCENTE**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	31	0
Actualizar	34	0
buscarPorCedulaEstudiante	35	4
recuperarTodos	11	0
Total	**111**	**4**

Tabla IV.XIV: Resultados LDC al Programar en la Clase ESTUDIANTEDOCENTE

- **Clase ESTUDIANTE**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	25	0
Actualizar	27	0
buscarPorCuenta	28	4
recuperarTodos	11	0
Total	**91**	**4**

Tabla IV.XV: Resultados LDC al Programar en la Clase ESTUDIANTE

- **Clase INDICADOR**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	30	0
Actualizar	33	0
Eliminar	26	0
buscarIndicadoresDelAmbitoEstandar	36	5
recuperarTodos	11	0
Total	**136**	**5**

Tabla IV.XVI: Resultados LDC al Programar en la Clase INDICADOR

- **Clase MATERIA**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
buscarDatosDocentePorCedula	14	4
Total	**14**	**4**

Tabla IV.XVII: Resultados LDC al Programar en la Clase MATERIA

- **Clase OPCIONES**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	34	0
Actualizar	37	0
Eliminar	25	0
buscarPorPregunta	39	7
recuperarTodos	11	0
Total	**146**	**7**

Tabla IV.XVIII: Resultados LDC al Programar en la Clase OPCIONES

- **Clase PREGUNTA**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	32	0
Actualizar	33	0
Eliminar	24	0
buscarPorTipoEvaluacion	34	4
recuperarTodos	11	0
buscarPreguntaPorAmbitoEstandarIndicadorTipo	16	6
Total	**150**	**10**

Tabla IV.XIX: Resultados LDC al Programar en la Clase PREGUNTA

- **Clase PROCESOEVALUACION**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	34	0
Actualizar	34	0
Eliminar	31	0
buscarTodos	32	3
Total	**131**	**3**

Tabla IV.XX: Resultados LDC al Programar en la Clase PROCESOEVALUACION

- **Clase PROCESO**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	27	0
Actualizar	28	0
Eliminar	24	0
buscarProcesoVigente	29	4
recuperarTodos	11	0
Total	**119**	**4**

Tabla IV.XXI: Resultados LDC al Programar en la Clase PROCESO

- **Clase USUARIO**

Método	LDC al Programar sin EJB	LDC al Programar con EJB
Insertar	33	9
buscarPorCuenta	25	4
recuperarTodos	29	3
Total	**87**	**16**

Tabla IV.XXII: Resultados LDC al Programar en la Clase USUARIO

De los totales obtenidos en cada tabla de resultados, se puede visualizar que el número de líneas escritas con EJB es menor con relación a las escritas sin hacer uso.

Con la muestra ingresada en el programa MINITAB 15.0, se obtuvieron los siguientes resultados y que se pueden revisar en el **Anexo 1** denominado **Resultados Estadísticos**:

- Sin EJB

Media = 25,0806

Variabilidad de la desviación = 8,8786

Varianza = 78,829

- Con EJB

Media = 1,4032

Variabilidad de la desviación = 2,2137

Varianza = 4,9003

4.12. Estudio comparativo de los resultados de ambas aplicaciones

Con los resultados obtenidos en el apartado anterior se ha calculado el porcentaje de mejora o decremento de la aplicación empresarial desarrollada con tecnología EJB, respecto a la que no utilizó dicha tecnología, para lo cual se hará uso de la siguiente fórmula de regla de tres:

$$A \rightarrow B$$

$$X \rightarrow Y$$

$$Y = \frac{B \cdot X}{A}$$

Fórmula 2: Regla de tres

Interpretandolo la Fórmula 2, diríamos que desarrollando la aplicación $\boldsymbol{A}$ sin EJB se obtuvo una eficiencia de $\boldsymbol{B}$ por ciento, entonces que porcentaje $\boldsymbol{Y}$ se obtendrá si se lo realiza con la aplicación $\boldsymbol{X}$ con EJB.

Con lo cual se obtuvieron los siguientes resultados:

Variable “Total LDC del programa”

Sin EJB = 25348

Con EJB = 39763

25348 100%

39763 Y

Y = 156,8684%

Mejora = 100% - 156,8684%

Mejora = -56,8684%

Interpretando el resultado obtenido, se puede decir que: haciendo uso de EJB no se disminuye el Total de LCD del Programa, más bien incremento en un 56,87%.

Variable “Tiempo”

Sin EJB = 6 semanas

Con EJB = 4 semanas

6 100%

4 Y

Y = 66,67%

Mejora = 100% - 66,67%

Mejora = 33,33%

Interpretando el resultado obtenido, se puede mencionar que: haciendo uso de la tecnología EJB se mejoró (redujo) el Tiempo empleado para el desarrollo de la aplicación, dedicando un 33,33% menos de tiempo.

Variable “LDC al Programar”

En los dos casos, se toma el valor obtenido en la **Media** para comparación.

Sin EJB = 25,0806 LDC al Programar

Con EJB = 1,4032 LDC al Programar

25,0806 100%

1,4032 Y

Y = 5,5948%

Mejora = 100% - 5,5948%

Mejora = 94.41%

Interpretando el resultado obtenido, se puede mencionar que: el uso de EJBs permite mejorar (disminuir) el número de LCD al Programar, con lo cual el trabajo por parte del desarrollador es menor.

4.13. Resultado final

Tomando como base la Tabla 4.1 “Ponderacion de variables”, y los resultados obtenidos en la comparación de ambas aplicaciones (Ver apartado 4.5), se aplica el siguiente proceso:

Variable “Total LDC del programa”

MejoraFinal = (-56,8684%)(10%)
MejoraFinal = -5,69%

Variable “Tiempo”

MejoraFinal = (33,33%)(40%)
MejoraFinal = 13,33%

Variable “LDC al Programar”

MejoraFinal = (94,41%)(50%)
MejoraFinal = 47,21%

Con lo cual, los resultados finales para determinar si hubo o no una mejora se mencionan en la Tabla IV.XXIII.

Variable	Peso
Total LDC del Programa	-5,69%
Tiempo	13,33%
LDC al Programar	47,21%
Total	54,85%

Tabla IV.XXIII: Ponderación de la mejora final

Sumando todos los resultados de mejoras obtenidos para cada variable, nos da un total de **54,85%** de mejora, que interpretándolo de mejor manera, sería: que el uso de EJB permite mejorar el desarrollo de un proyecto respecto a la no utilización de esta tecnología, con respecto a las variables tomadas en cuenta.

4.14. Demostración de la Hipótesis

Para la demostración de la hipótesis se ha tomado la variable **LDC al Programar**, debido a que no toma el mismo tiempo programar una función sin EJB que tenga mayor número de líneas con respecto a una función con EJB, junto con la distribución normal Z y la teoría de Hipótesis.

Hipotesis: "La implementación de componentes EJB que se acoplen al Sistema de Evaluación Docente de la ESPOCH permitirá optimizar el tiempo de desarrollo destinado en la lógica de negocios del mismo."

H0 = Implementar un sistema sin EJB lleva igual tiempo de desarrollo destinado en la lógica de negocios que un sistema implementado con EJB.

H1 = Implementar un sistema sin EJB lleva diferente tiempo de desarrollo destinado en la lógica de negocios que un sistema implementado con EJB.

H0 => μSEJB = μCEJB
H1 => μSEJB <> μCEJB

Datos sin EJB

ns = 62
Media ($\overline{X_1}$) = 25,0806
Desviación estándar (Ss) = 8,8786

Datos con EJB

nc = 62
Media ($\overline{X_2}$) = 1,4032
Desviación estándar (Sc) = 2,2137

Error Estándar = 5%

Cálculo de la Desviación Estándar:

$$S_{\bar{X}_1-\bar{X}_2} = \sqrt{\frac{Ss^2}{nc} + \frac{Sc^2}{ns}}$$

$$S_{\bar{X}_1-\bar{X}_2} = \sqrt{\frac{(8{,}8786)^2}{62} + \frac{(2{,}2137)^2}{62}}$$

$$S_{\bar{X}_1-\bar{X}_2} = \sqrt{\frac{78{,}8295}{62} + \frac{4{,}9005}{62}}$$

$$S_{\bar{X}_1-\bar{X}_2} = \sqrt{1{,}2714 + 0{,}0790}$$

$$S_{\bar{X}_1-\bar{X}_2} = \sqrt{1{,}3504}$$

$$S_{\bar{X}_1-\bar{X}_2} = 1{,}1621$$

Cálculo de Z

$$Z = \frac{\overline{X_1} - \overline{X_2}}{S_{\overline{X_1}-\overline{X_2}}}$$

$$Z = \frac{25{,}0806 - 1{,}4032}{1{,}1621}$$

$$Z = \frac{23{,}6774}{1{,}1621}$$

$$Z = 20{,}3747$$

Utilizando el software MINITAB 15.0 se obtuvo el siguiente resultado:

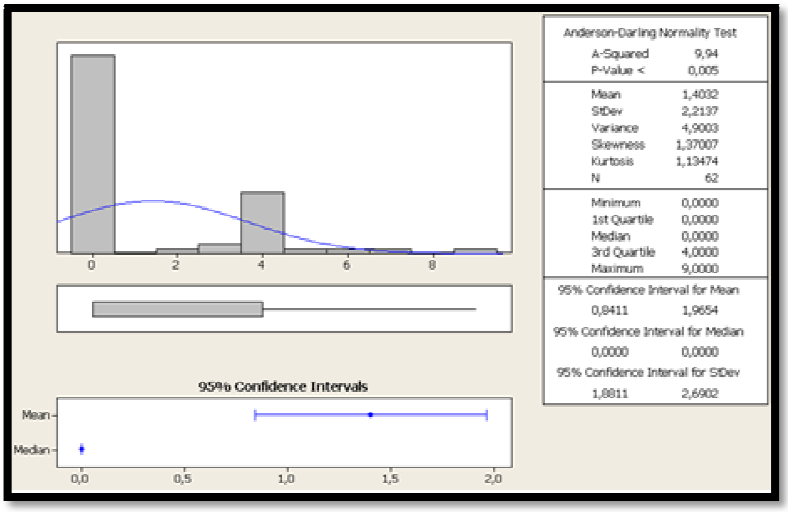


Figura XL: Gráfico de Demostración de la Hipótesis

Basándose en el gráfico de la demostración de la hipótesis representado en la Figura XL, con un nivel de significación de 0,05 equivalente al +/- 1,96 se rechaza la hipótesis verdadera “Implementar un sistema sin EJB lleva igual tiempo de desarrollo destinado en la lógica de negocios que un sistema implementado con EJB”, por encontrarse el valor de Z en la cola derecha se concluye que:

“Implementar un sistema con EJB requiere menor tiempo de desarrollo destinado en la lógica de negocios que un sistema sin EJB.”

Por tanto, si se requiere menos tiempo y el resultado es el mismo entonces:

“La Implementación de componentes EJB que se acoplen al Sistema de Evaluación Docente de la ESPOCH permite optimizar el tiempo de desarrollo destinado en la lógica de negocios del mismo”

CONCLUSIONES

La realización del estudio de la tecnología EJB (Enterprise Java Beans) para la implementación de componentes EJB que se acoplen al sistema de evaluación docente de la ESPOCH, permitió determinar que:

- EJB es un modelo de componentes o framework que permite crear aplicaciones sin tener que reinventar servicios como las transacciones, seguridad, persistencia.

- Un EJB es un código que se ejecuta en un entorno de ejecución especial llamado contenedor EJB, el mismo que proveé funcionalidades comunes (persistencia automática) en forma de servicios y cuya configuración se realiza sencillamente con anotaciones. Las anotaciones convierten a un POJO (Plain Old Java Object – Objeto Plano Java Antiguo) en un EJB.

- Existen dos tipos de EJB que pueden ser utilizados para el desarrollo de aplicaciones empresariales, que son: EJB de Sesiones (Stateless, Stateful y Singleton), EJB Manejadores de Mensajes.

- En EJB 3.1, el estándar actual, se define a los EJB de Entidades como parte de la Java Persistence API (JPA) para gestión de persistencia automática.

- En EJB para la gestión de persistencia se hace uso de una técnica llamada Mapeo Objeto-Relacional (ORM), que mapea los datos de un objeto con las tablas de una base de datos mediante una configuración, de forma que se evita escribir el código JDBC.

- Respecto a una aplicación que no utiliza tecnología EJB, el tiempo destinado al desarrollo, disminuye en un 33,33%.

- En escenarios similares en los que se necesitan crear métodos en las mismas condiciones y necesitando el mismo resultado, la cantidad de LDC al programar se redujo en un 94,41% al hacer uso de la tecnología EJB.

- El **Total de LDC del programa**, que muchas metodologías se basan para estimar los costos de uno u otro sistema, aumento en un 56,87% al utilizar la tecnología EJB.

- De las pruebas realizadas con estudiantes de la Facultad de Administración de Empresas, actualmente el sistema de evaluación docente nos permite realizar un máximo de 400 a 450 evaluaciones simultáneas, incrementando de esta manera en 250 el número de conexiones, pudiendo con esto establecer un lapso menor de tiempo en el Proceso de Evaluación docente en la ESPOCH.

RECOMENDACIONES

- Utilizar la tecnología EJB en el desarrollo de aplicaciones, ya que esta permite agregar funcionalidad (manejo de transacciones, seguridad, persistencia, mapeo objeto-relacional, despliegue distribuido, despliegue en muchos entornos operativos) a nuestra aplicación con solo agregar anotaciones, además de reducir la cantidad de líneas de código que tenemos que programar directamente y por consecuencia permite disminuir el tiempo de desarrollo con lo cual se puede ganar mayor productividad.

- Se debe conocer a profundidad las dos maneras de implementar una aplicación (con y sin EJB) por parte del desarrollador para que la comparación, análisis de resultados y comprobación de la hipótesis sea válida.

- Al momento de realizar la comparación y análisis de resultados entre las aplicaciones, se debe definir y calcular correctamente los parámetros para obtener conclusiones válidas.

- Se recomienda que se imparta clases referentes a esta tecnología de desarrollo en la Escuela de Ingeniería en Sistemas, por la facilidad de implementación y las múltiples funcionalidades (transaccionalidad, persistencia, concurrencia) que pueden ser agregadas a las aplicaciones con solo agregar anotaciones.

- Realizar un estudio de la concurrencia que permite obtener la utilización de la tecnología EJB.

RESUMEN

En la realización del Estudio de la arquitectura de los componentes EJB (Enterprise Java Beans), caso aplicativo en el Sistema de Evaluación Docente de la Escuela Superior Politécnica, se determinó el decremento en el tiempo de desarrollo destinado en la lógica de negocios del mismo.

Para la demostración de la Hipótesis fueron utilizados los métodos científicos deductivo y comparativo además de técnicas estadísticas aplicadas en todo el estudio basado en parámetros. Mientras que para su realización se utilizó como herramientas lo siguiente: la plataforma Java Entreprise Edition (JEE) con JDK 1.6, Oracle 10g XE como gestor de base de datos y GlassFish Server 3.1 como Contenedor EJB y Servidor de Aplicaciones. Además como metodología de desarrollo se utilizó Rational Unified Process (RUP).

Unas vez finalizado el estudio de la arquitectura de los Enterprise Java Beans, se ha llegado a obtener los siguientes resultados: La productividad que se gana al momento de programar utilizando la tecnología EJB respecto a Líneas de Código (LDC) al programar es de un 94,41% y en tiempo destinado se disminuyó un 33,33%, comprobándose que la utilización de EJB mejora (disminuye el tiempo) en un 54,85% al momento de desarrollar un sistema.

Se concluye que "**La Implementación de componentes EJB que se acoplen al Sistema de Evaluación Docente de la ESPOCH permite optimizar el tiempo de desarrollo destinado en la lógica de negocios del mismo"**.

Se recomienda que, al momento de desarrollar aplicaciones empresariales, se utilice la tecnología EJB a pesar de ser necesario un amplio conocimiento de Java Enterprise Edition (JEE), esta nos brinda un conjunto de servicios adicionales por si sola, como seguridad, transaccionalidad, mayor concurrencia.

SUMMARY

"Study of the software architecture components EJB (Enterprise Java Beans), Case applicative for Teacher's Evaluation System of Escuela Superior Politécnica de Chimborazo (ESPOCH)."

Free software is available and it has a low cost, it is easy to modify and redistribute. The ESPOCH is interested to use free software on its Teachers Evaluation System to optimize time of development destined to its own business logic.

The number of users that the actual Evaluation System support is 120, and that's the reason why this research developed components EJB to increase the number of users.

The main goal is to increase the number of permitted connections to the Teachers' Evaluaction System of the ESPOCH.

The methods applied to determine the hypothesis were scientific deductive and comparative, beside other statistics techniques based on parameters; besides, this research developed a Rational Unified Process (RUP) as a development technology. For the execution the research used the following tools: platform JEE (Java Enterprise Edition) with JDK (Java Development Kit) 1.6, Oracle 10g XE as a database management and Glass Fish Sever 3.1 as an EJB container and application server.

Once the study of the EJB architecture ended, it obtained the following results:

Productivity is achieved at the moment of programming using the EJB technology compared to Source Lines of Code (SLOC).

Programming goes from 94,41% and in the destined time it decreased a 33,33%. This fact proves that the using of EJB decrease in a 54,85% at the moment to develop a system.

This research shows that the implementation of EJB components adapted to the Teachers' Evaluation Sytem of the ESPOCH permits to optimize the time of development destined to its business logic.

It is important the EJB technologies be applied at the moment of developing of enterprise applications. It is also essential to amplify the Java Enterprise Edition knowledge. This technology offers a group of additional services by its own security, transaction processing, and concurrent computing.

GLOSARIO

ESPOCH.- Escuela Superior Politécnica de Chimborazo.

UTEI.- Unidad Técnica de Evaluación Interna.

SIEDD V2.0.- Sistema Integrado de Evaluación del Desempeño Docente Versión 2.0.

EJB.- Enterprise Java Beans

JDK.- Java Development Kit

Cliente.- Persona o institución que recibe y usa un producto.

Interfaz.- conexión física y funcional entre dos sistemas.

Prototipo.- ejemplar origianl o primer molde en que se fabrica una figura u otra cosa.

Proyecto.- conjunto de actividades planificadas, coordinadas y presupuestadas que se emprenden para alcanzar objetivos específicos.

Requisitos.- circunstancias o condición necesaria para satisfacer alguna necesidad.

Servicio.- producto intangible que es el resultado de realizar por lo menos una actividad en la interfaz.

Sistema.- conjunto de elementos interrelacionados e interactuantes en uno o mas de los procesos que proporcionan la capacidad de satisfacer las necesidades u objetivo definido.

Modelo.- guía que ha sido aprobada y aceptada para seguir en un proceso de ingeniería de software.

Usuario.- persona que usa el sistema para realizar una función especifica.

Misión.- objetivo de cada empresa o institucion

Visión.- plantea el futuro de una empresa o institución.

Fines.- termino, remate o consumación de una cosa. Objeto o motivo con que se ejecuta una cosa.

ANEXOS

ANEXO 1

RESULTADOS OBTENIDOS

CON MINITAB 15.0

C1 -> Sin EJB

C2 -> Con EJB

Data Display

```
C1
   27    1   31   34   11   25   25   28   11   24   24   32   12   25   27
   29   11   29   11   27   28   30   26   31   11   31   34   35   11   25
   27   28   11   30   33   26   36   11   14   34   37   25   39   11   32
   33   24   34   11   16   34   34   31   32   27   28   24   29   11   33
   25   29
```

Data Display

```
C2
   4   0   0   4   0   0   0   4   0   0   0   4   0   0   0   4   0   4   0
   2   0   0   0   4   0   0   0   4   0   0   0   4   0   0   0   0   5   0
   4   0   0   0   7   0   0   0   0   4   0   6   0   0   0   3   0   0   0
   4   0   9   4   3
```

Descriptive Statistics: C1; C2

Variable	N	N*	Mean	SE Mean	StDev	Variance	CoefVar	Minimum	Q1
C1	62	0	25,08	1,13	8,88	78,83	35,40	1,00	22,00
C2	62	0	1,403	0,281	2,214	4,900	157,76	0,000	0,000

ANEXO 2

DOCUMENTACIÓN TÉCNICA

INTRODUCCIÓN

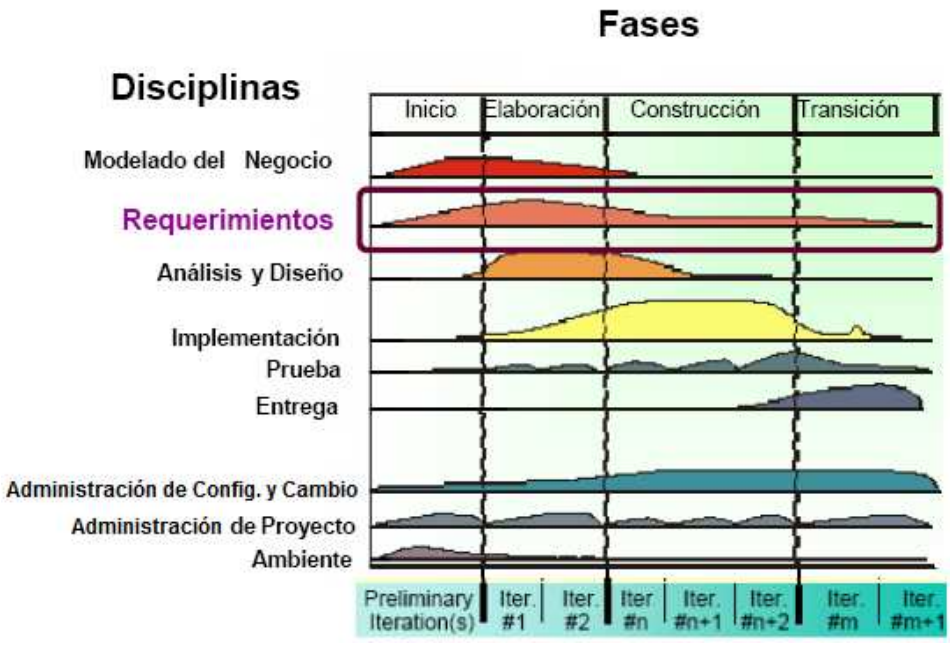
¿Por qué del SIEDD V2.0?

En la actualidad la Escuela Superior Politécnica de Chimborazo cuenta con un Sistema de Evaluación Docente, el mismo que se ve limitado en cuanto a conectividad de usuarios (un máximo de 150 conexiones concurrentes) por motivo de licencias, existiendo ocasiones en las que el mismo ha colapsado, razón por la cual se ha visto la necesidad de implementar una aplicación que se ejecute bajo un sistema operativo de licencia libre, o independiente de la plataforma y el mismo que estará constituido por componentes Enterprise JavaBeans.

¿Para qué usar la Metodología RUP?

Rational Unified Process o simplemente RUP define varias actividades (disciplinas) a realizarse en cada fase del proyecto.

- Modelado de negocio
- Análisis de requisitos
- Análisis y diseño
- Implementación
- Test
- Distribución
- Gestión de configuración y cambios
- Gestión del proyecto
- Gestión del entorno



Vista general de RUP

Y el flujo de trabajo (workflow) entre ellas en base a los llamados diagramas de actividad. El proceso define una serie de roles que se distribuyen entre los miembros del proyecto y que definen las tareas de cada uno y el resultado (artefactos en la jerga de RUP) que se espera de ellos.

Flujos de Trabajo de RUP

RUP se basa en casos de uso para describir lo que se espera del software y está muy orientado a la arquitectura del sistema, documentándose lo mejor posible, basándose en UML (Unified Modeling Language) como herramienta principal.

Entre las principales características de esta metodología, está que es un proceso de desarrollo de software dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental. RUP pretende implementar las mejores prácticas en ingeniería de software, con el objetivo de asegurar la producción de software de calidad, dentro de plazos y presupuestos predecibles.

RUP divide el proceso de desarrollo en ciclos, donde se obtiene un producto al final de cada ciclo. Cada ciclo se divide en cuatro Fases: Concepcion, Elaboracion, Contruccion y Transaccion. Cada fase concluye con un hito bien definido donde deben tomarse ciertas decisiones.

¿Por qué usar la notación UML?

Se usará la notación UML (Unified Modeling Language) dentro de este proyecto de desarrollo debido a que es un estándar de facto que utiliza la orientación a objetos y sobre todo porque RUP se basa en esta.

Para realizar los diversos diagramas que se mostrarán en este documento se contará con la ayuda de la herramienta StarUML que es idónea para nuestro propósito, la misma que aprovecha la notación UML.

OBJETIVOS

Objetivo General

- Proporcionar un sistema para que facilite a los estudiantes de la Escuela Superior Politécnica de Chimborazo la Evaluación al Docente.

Objetivos Específicos

- Procesar y analizar de manera correcta y precisa los requerimientos funcionales y no funcionales del problema para un correcto análisis y diseño de los procesos llevado a cabo en la evaluación docente.

- Implementar una solución basada en la Metodología RUP, de manera que el sistema presente características de confiabilidad, eficiencia, portabilidad, escalabilidad y sobre todo un buen rendimiento, de manera que se pueda dar un mantenimiento oportuno periódicamente.

- Documentar técnicamente todos los requisitos del sistema para poder dar un seguimiento adecuado al mismo.

- Representar de forma clara, concreta y precisa todos los diagramas necesarios para cada uno de los procesos que se llevan a cabo en la evaluación docente, para la codificación del sistema en sí.

- Utilizar PROJECT 2003 con el fin de establecer parámetros tales como la duración, actividades y recursos, y poder realizar una estimación y planificación adecuada conforme al cumplimiento de tareas.

- Plasmar los conocimientos adquiridos en lo que tiene que ver a la Ingeniería de Software, para poderlos aplicar correctamente al proyecto mediante una buena elección del modelo de ciclo de vida y de la adecuada notación de los diagramas para acercarnos a una solución óptima.

MARCO TEÓRICO

A continuación se presentan y se detallan algunas de las disciplinas, como Modelado del Negocio, Requerimientos, Análisis y Diseño, y la disciplina de Implementación, que forman parte de la metodología de desarrollo RUP (Rational Unified Process).

I. MODELADO DEL NEGOCIO

1. ESPOCH

1.1. Antecedentes

La Escuela Superior Politécnica de Chimborazo (ESPOCH), tiene su origen en el Instituto Tecnológico Superior de Chimborazo, creado mediante Ley No.6090, expedida por el Congreso Nacional, el 18 de abril de 1969. Inicia sus actividades académicas el 2 de mayo de 1972 con las Escuelas de Ingeniería Zootécnica, Nutrición y Dietética e Ingeniería Mecánica. Se inaugura el 3 de abril de 1972.

El 28 de septiembre de 1973 se anexa la Escuela de Ciencias Agrícolas de la PUCE, adoptando la designación de Escuela de Ingeniería Agronómica.

1.2. Visión

Ser una institución universitaria líder en la Educación Superior y en el soporte científico y tecnológico para el desarrollo socioeconómico y cultural de la provincia de Chimborazo y del país, con calidad, pertinencia y reconocimiento social.

1.3. Misión

Formar profesionales competitivos, emprendedores, conscientes de su identidad nacional, justicia social, democracia y preservación del ambiente sano, a través de la generación, transmisión, adaptación y aplicación del conocimiento científico y tecnológico para contribuir al desarrollo sustentable de nuestro país.

1.4. Objetivos

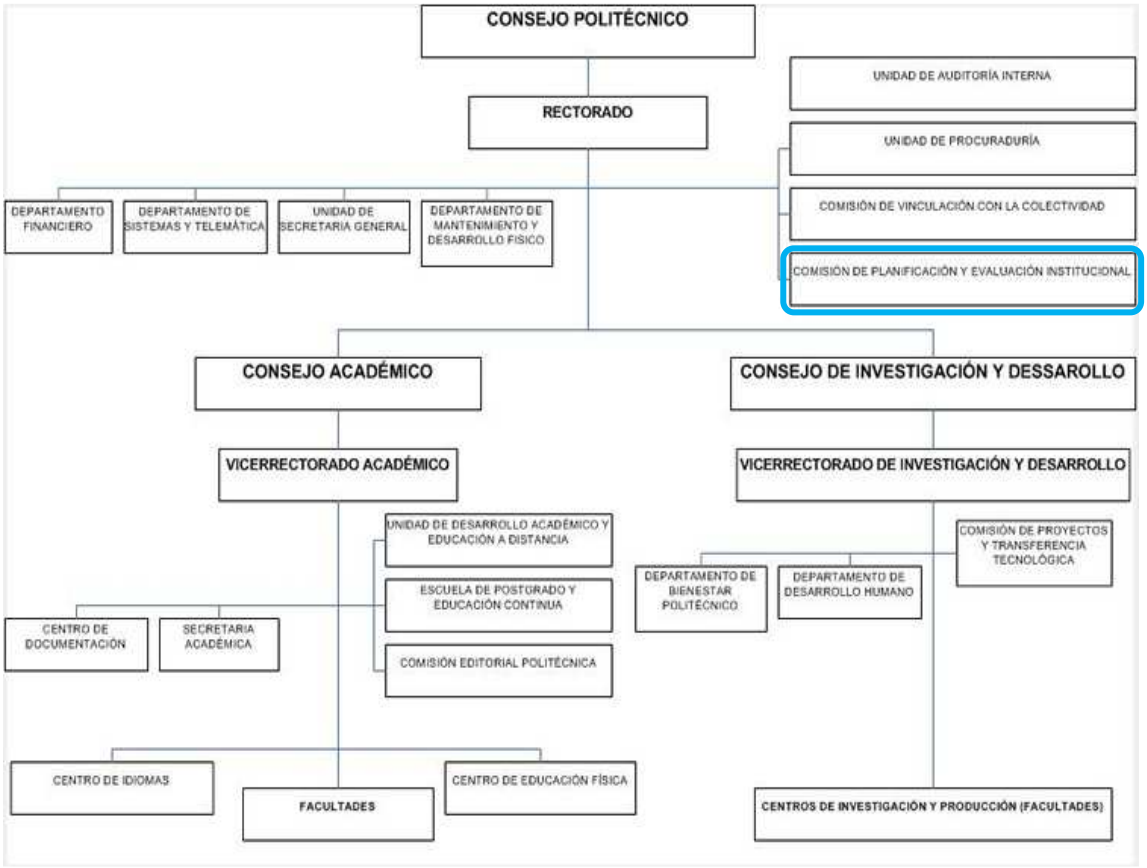
- Lograr una administración moderna y eficiente en el ámbito académico, administrativo y de desarrollo institucional.
- Establecer en la ESPOCH una organización sistémica, flexible, adaptativa y dinámica para responder con oportunidad y eficiencia a las expectativas de nuestra sociedad.
- Desarrollar una cultura organizacional integradora y solidaria para facilitar el desarrollo individual y colectivo de los politécnicos.
- Fortalecer el modelo educativo mediante la consolidación de las unidades académicas, procurando una mejor articulación entre las funciones universitarias.

- Dinamizar la administración institucional mediante la desconcentración de funciones y responsabilidades, procurando la optimización de los recursos en el marco de la Ley y del Estatuto Politécnico.

- Impulsar la investigación básica y aplicada, vinculándola con las otras funciones universitarias y con los sectores productivos y sociales.

- Promover la generación de bienes y prestación de servicios basados en el potencial científico-tecnológico de la ESPOCH.

1.5. Orgánico Funcional



2. UTEI – Unidad Técnica de Evaluación Interna

2.1. Antecedentes

LA ESPOCH de conformidad con la Constitución Política del Ecuador, la Ley de Educación Superior, su Estatuto, y ante todo por propia voluntad y decisión política de las autoridades, el 9 de Febrero de 2001, conforma la Comisión de Evaluación Interna actualmente denomina como Unidad Técnica de Evaluación Interna.

2.2. Misión

Evaluar las Funciones: Docencia, Investigación, Vinculación con la colectividad y Gestión Administrativa, a fin de asegurar niveles de calidad en todos los procesos.

2.3. Objetivos

Planificar y coordinar, así como asesorar al H. Consejo Politécnico sobre los procesos de Autoevaluación institucional, Evaluación Externa y Acreditación en concordancia con el mandato de la Ley de Educación Superior y las reglamentaciones que a nivel del Sistema Universitario Nacional se establezcan, así como en función de la reglamentación interna que la ESPOCH determine.

2.4. Funciones

- Promover la cultura de evaluación en la ESPOCH.
- Implementar la base de datos sobre la información institucional y del medio externo
- Proponer al Consejo Politécnico, políticas de evaluación y acreditación institucional
- Determinar los criterios, indicadores e instrumentos que han de aplicarse en la planificación y evaluación institucional.
- Vigilar que los procesos de autoevaluación se realicen de conformidad con las normas y procedimientos que para el efecto se establezcan; propendiendo que sus resultados sean fruto de una absoluta independencia e imparcialidad.
- Asesorar el establecimiento y ejecución de la evaluación en las diferentes unidades académicas y administrativas
- Presentar a las autoridades respectivas los resultados de las evaluaciones efectuadas.
- Presentar anualmente al Consejo politécnico los informes de planificación y evaluación.
- Asesorar a las autoridades y organismos institucionales sobre la evaluación.

II. REQUERIMIENTOS

En el presente apartado se detallan los requerimientos del Sistema Integrado de Evaluación del Desempeño Docente V2.0 (SIEDD) para la EPEC de la ESPOCH a través de una SRS (Especificación de Requerimientos de Software).

Tal y como se mencionó anteriormente, la notación utilizada para los distintos modelos, es la proporcionada por UML, que se ha convertido en el estándar de facto en cuanto tiene que ver a la notación orientada a objetos.

Especificación de Requerimientos de Software (SRS)

1. Introducción

1.1. Propósito

Este apartado describe los requerimientos del software para el Sistema de Evaluación Docente de la ESPOH para Pregrado, con el cual se trata de dar solución al problema planteado.

Además, se define el sistema a nivel de usuarios y los casos de uso.

1.2. Alcance

El sistema a ser desarrollado será llamado Sistema Integrado de Evaluación del Desempeño Docente V2.0, por sus siglas SIEDD V2.0.

Este documento alcanza al proyecto completo, tanto las aplicaciones FRONT como BACK.

1.3. Visión general

El documento de SRS describe los requerimientos del software a construir. Se incluyen los requerimientos funcionales y no funcionales, así como también diagramas de los casos de uso del sistema. Su especificación existe en otros documentos, a los que se hará referencia en su momento.

1.4. Definiciones, acrónimos y abreviaturas

ESPOCH.- Escuela Superior Politécnica de Chimborazo.

UTEI.- Unidad Técnica de Evaluación Interna.

SIEDD V2.0.- Sistema Integrado de Evaluación del Desempeño Docente Versión 2.0.

SRS.- por sus siglas en inglés Specification Requirements Software (Especificación de Requerimientos de Software).

UML.- por sus siglas en inglés Unified Modeling Language (Lenguaje Unificado de Modelado).

Aplicación web FRONT.- Aplicación a disposición de estudiantes y docentes para rendir la evaluación docente.

Aplicación web BACK.- Aplicación a disposición del Administrador y Directivos.

HTTP.- por sus siglas en inglés HiperText Transfer Protocol (Protocolo de Transferencia de Hipertexto).

JDK.- Java Development Kit. (Herramientas de Desarrollo Java).

JVM.- Java Virtual Machine (Máquina Virtual de Java)

Clases: Descripción de objetos con comportamientos y características.

Desarrollador.- Se encarga del desarrollo del software.

Métodos: Es la descripción del comportamiento de los objetos.

Objetos: Es único, es algo que ocupa un lugar en el espacio, tiene sus propias características, puede ser definido intelectualmente.

Reportes: Prueba litográfica que sirve para plasmar las actividades económicas de un servicio.

Tabla: Es la representación de objetos o clases con sus respectivos atributos y métodos.

1.5. Resumen

En general en el presente documento se realizará una definición de las acciones fundamentales que debe realizar el software al recibir información, procesarla y producir resultados.

2. Descripción general

El proyecto SIEDD V2.0 constará de dos aplicaciones empresariales, una FRONT que permitirá tanto a Estudiantes, Docentes, Directivos y Pares rendir una evaluación. También existirá una aplicación BACK para tareas de administración, como, cierre del proceso de evaluación, restablecer contraseñas de todos los tipos de usuarios, emisión de reportes estadísticos de participación, certificados de evaluación, seguimiento de participación en la evaluación docente, etc. Adicionalmente, permitirá el acceso a Secretarias de cada una de las Escuela las cuales podrán revisar ciertos reportes.

2.1. Perspectivas del producto

El SIEDD V2.0 es una aplicación empresarial independiente y sin ningún tipo de relación con los diferentes sistemas existentes en la ESPOCH.

2.2. Funcionalidad del producto

El sistema en forma general tendrá que permitir generar diversas tipos de encuestas de acuerdo a las características de los evaluadores, dentro de las cuales existirán diversos Ámbitos a ser considerados, las mismas que serán puestas a disposición de los estudiantes, docentes, directivos y/o pares, con lo cual se podrá obtener datos estadísticos sobre el desempeño de cada uno de los docentes que imparten clases en la ESPOCH.

2.3. Características de los usuarios

El sistema presentado lo podrán utilizar las siguientes personas:

Estudiante.- Serán los encargados de realizar la evaluación a cada uno de los docentes.

Docente.- Los docentes estarán encargados de la realización de una autoevaluación.

Directivos.- Son aquellos que evalúan el desempeño del docente a través de la aplicación FRONT y además controlan el proceso de evaluación, a través de reportes en la aplicación END.

Pares.- Es el docente que evalúa a otro docente, luego de haber realizado un seguimiento presencial al desenvolvimiento del mismo dentro del aula.

Secretaria.- Será la responsable de la emisión de Certificados de Evaluación a los docentes.

Administrador.- Es la persona a cargo de la configuración de los parámetros iniciales para conformar un cuestionario, es decir será el que genere la estructura

(preguntas) de la encuesta (ámbitos, estándares, indicadores, preguntas, opciones). Además, será el responsable de la extracción de datos indispensables para la conformación de la evaluación (facultades, escuelas, carreras, niveles, materias, estudiantes, docentes, directivos, pares), y podrá también obtener reportes generales de la participación en el proceso de evaluación vigente. Y podrá administrar cada una de las cuentas de los diferentes tipos de usuarios, tanto para la aplicación FRONT como la BACK.

2.4. Restricciones

Para la ejecución del siguiente proyecto se hará uso de la metodología RUP (Rational Unified Process) que hace uso del Lenguaje de Modelado Unificado (UML).

Además para la implementación del mismo se lo realizará utilizando la plataforma JAVA, con lo cual el sistema podrá ser implantado bajo cualquier sistema operativo en el que esté a disposición la JVM (Java Virtual Machine – Máquina Virtual de Java), se utilizará JSP para el módulo web, y componentes EJB para el módulo del mismo nombre, y para el despliegue de la aplicación se utilizará GlassFish Server 3.1.

3. Requerimientos específicos

3.1. Requerimientos comunes de los interfaces

3.1.1. Interfaces de usuario

La aplicación tendrá una interfaz de usuario muy ligero, apto para la web, además esta deberá ser manejada a través de vínculos o links para la navegación entre los diferentes niveles, reportes, y demás funcionalidades del sistema.

3.1.2. Interfaces de hardware

Las interfaces de hardware necesarias para ejecutar la aplicación serán:

- Tarjeta de red 10/100 Mbps.
- Mouse o Teclado.
- Monitor

3.1.3. Interfaces de software

El sistema no estará integrado a ningún otro sistema de los existentes en la Escuela Superior Politécnica de Chimborazo.

Pero, hará consumo de los servicios web disponibles y necesarios para la extracción de datos de Facultades, Escuelas, Carreras, Niveles, Materias, Estudiantes y Docentes.

Adicionalmente, se podrá hacer uso de la aplicación utilizando cualquiera de los navegadores existentes.

3.1.4. Interfaces de comunicación

La comunicación del sistema se la realizará bajo **Http**, y además no existirá ningún tipo de comunicación con otro sistema.

3.2. Requerimientos Funcionales

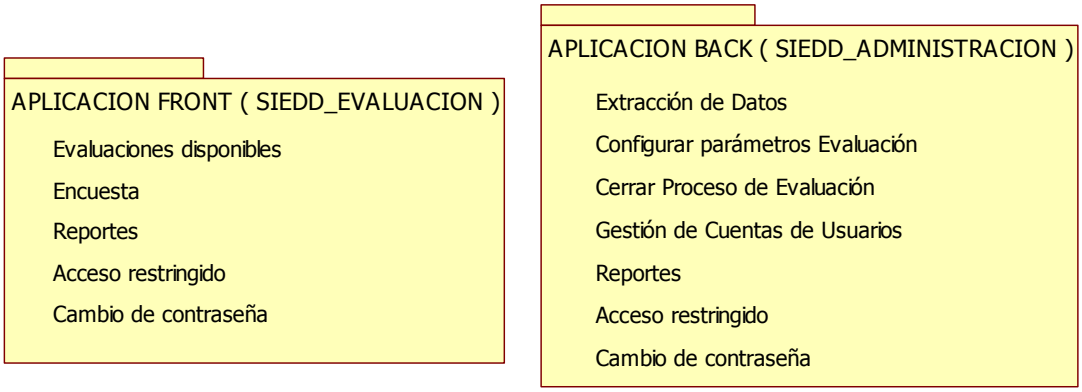


Diagrama de paquetes FRONT y BACK

3.2.1. Aplicación FRONT

La aplicación FRONT es la que utilizarán tanto estudiantes, docentes, directivos como los pares, para realizar la Evaluación del Desempeño Docente.

3.2.1.1. RF1. Evaluaciones disponibles

El sistema debe mostrar al Estudiante un listado de todas las Materias por las cuales está cursando actualmente con el nombre del docente que la imparte y el estado de la evaluación; al Docente debe mostrar todas las materias y el paralelo al que dicta clase y el estado de la evaluación; al Directivo deberá mostrar los Docentes a los que evalúa con el nombre de la Carrera de la que forma parte el docente; finalmente a los Pares tiene que mostrar los Docentes con la respectiva Materia y Paralelo de la que forma parte, a los cuales realizó un seguimiento directo en clase.

3.2.1.2. RF2. Encuesta

Encuesta es el núcleo de la aplicación. La misma está estructurada por Ámbitos que a su vez están constituidos por diversos Estándares y estos por Indicadores, y este último tiene una pregunta por tipo de evaluador (Estudiante, Docente, Directivo, Par), adicionalmente las preguntas serán

cerradas y mutuamente excluyentes, y en ocasiones pueden contar con comentarios (Directivos y Pares), así también, ciertas preguntas se relacionarán con otra de las mismas características. Finalmente, el número de opciones que se tiene como alternativas de respuesta puede variar en cada una de ellas.

3.2.1.3. RF3. Acceso restringido

Acceso de acuerdo a un tipo de usuario (Estudiante, Docente, Directivo, Par), su cuenta de usuario (número de cédula con guión) y la contraseña (inicialmente número de cédula sin guión).

3.2.1.4. RF4. Cambio de contraseña

El sistema debe permitir a cualquier tipo de usuario autenticado, modificar su contraseña en cualquier momento.

3.2.1.5. RF5. Reportes

El sistema debe permitir a los usuarios de tipo Docente obtener reportes por Ámbitos, Estándares o Indicadores de los resultados obtenidos en el proceso de evaluación y categorizados en cada una de las Carrera que imparte clases.

3.2.2. Aplicación BACK

La aplicación BACK será utiliza por el Administrador, Director de Escuela o su Secretaria, y la Secretaria de la UTEI.

El sistema le permitirá al Administrador, extraer datos de Facultades, Escuelas, Carreras, Niveles, Materias, Estudiantes y Docentes desde el sistema OASIS. Podrá Configurar parámetros de la evaluación como Ámbitos, Estándares, Indicadores, Preguntas, Opciones, Proceso, y Procesos de Evaluación por Carrera. También podrá Gestionar cuentas de usuario de cualquier tipo (Estudiantes, Docentes, Directivos, Pares, Secretaria UTEI, Director Escuela). Y tendrá acceso a reportes y certificados de evaluación, además podrá cerrar el proceso de evaluación.

Por su parte los Directores de Escuela o su Secretaria, estarán a cargo de controlar la participación de los estudiantes en la evaluación docente además podrán tener acceso a reportes de resultados de la evaluación.

Finalmente la Secretaria de la UTEI emite Certificados de Evaluación.

3.2.2.1. RF6. Extracción de Datos

El Administrador será el responsable de la ejecución del proceso de extracción de datos, para Facultades, Escuelas, Carreras, Niveles, Materias, Estudiantes y Docentes.

Los datos extraídos para Facultades serán: Código y Nombre.

Los datos extraídos para Escuelas serán: Código, Nombre y Código de Facultad a la que pertenece.

Los datos extraídos para Carreras serán: Código, Nombre y los Códigos de Facultad y Escuela a la que pertenece.

Los datos extraídos para Niveles serán: Código, Nombre y los Códigos de Facultad, Escuela y Carrera a la que pertenece.
Los datos extraídos para Materias serán: Código, Paralelo, Nombre y Cédula del Docente que dicta la materia, y los Códigos de Facultad, Escuela, Carrera y Nivel al que pertenece.

Los datos extraídos para Estudiantes serán: Número de Cédula, Nombres y Apellidos completos, y se establecerá inicialmente la clave con su número de cédula sin guión.

Los datos extraídos para Docentes serán: Número de Cédula, Nombres y Apellidos completos, Tipo y se establecerá inicialmente la clave con su número de cédula sin guión.

A partir de la extracción de datos completamente finalizada, se generan las evaluaciones para Estudiantes y Docentes.

3.2.2.2. RF7. Configurar Parámetros de Evaluación

El Administrador gestionará (Altas, Bajas, Modificaciones, Listados) de Ámbitos, Estándares, Indicadores, Preguntas, Proceso y Procesos de Evaluación por carrera.

Los campos a manejarse por Ámbito y Estándar son: Código, Detalle y Orden.

Los campos a manejarse por Indicador son: Código, Detalle, Orden y Ponderación.

Los campos a manejarse por Pregunta son: Código, Tipo (a quien pertenece la pregunta Estudiante, Docente, Directivo o Par), Detalle, Orden, Ponderación y pregunta con la que se relaciona.

Los campos a manejarse por Opción serán: Código, Detalle, Orden y Ponderación.

Los campos a manejarse por Proceso serán: Código, Detalle y Vigencia.

Cada Proceso de Evaluación de ser configurado por Carrera y tener Fecha de Inicio, Fecha Final y Fecha de Cierre.

3.2.2.3. RF8. Cerrar Proceso de Evaluación

Una vez concluidas las evaluaciones a nivel general, el Administrador será el responsable de la ejecución del proceso de cierre, para los cálculos de resultados.

3.2.2.4. RF9. Gestión de Cuentas de Usuarios

El Administrador será en responsable de la gestión de las cuentas de usuario, de los usuarios que utilizan la aplicación BACK gestionará Altas, Bajas y Modificaciones mientras que de las cuentas utilizadas por los usuarios de la aplicación FRONT solo podrá restablecer la clave de estos de manera automática (se genera a partir del número de cédula sin guión).

3.2.2.5. RF10. Reportes

El Administrador podrá tener acceso a reportes de Participación en general; Encuestas Generadas por el SIEDD para Estudiantes, Docentes, Directivos o Pares; estructura de la configuración de los Cuestionarios para los diferentes tipos de evaluadores; estudiantes y docentes que no han realizado la evaluación; Certificados de Evaluación.

Los Directores de Escuela o sus Secretarias, podrán tener acceso a reportes de Participación de su Escuela; y, Estudiantes y Docentes que no han realizado la evaluación.

La Secretaria de la UTEI solo podrá emitir Certificados de Evaluación.

3.2.2.6. RF11. Acceso restringido

Los usuarios que podrán utilizar la aplicación FRONT de acuerdo a un perfil o tipo (Administrador, Secretaria UTEI, Director Escuela o Secretaria), su cuenta de usuario (número de cédula con guión) y la contraseña (inicialmente número de cédula sin guión).

3.2.2.7. RF12. Cambio de contraseña

El sistema debe permitir a cualquier tipo de usuario autenticado, modificar su contraseña en cualquier momento.

3.3. Requerimientos No Funcionales

3.3.1. Usabilidad

3.3.1.1. Interfaz intuitiva

Debido a que muchos de los Estudiantes, Docentes Directivos y Pares se encontrarán por única y quizá por primera vez, frente a la aplicación FRONT

para la evaluación docente, será necesario que la aplicación disponga de una interfaz lo más simple, clara e intuitiva posible.

3.3.2. Seguridad

3.3.2.1. Manejo de Sesiones

Las dos aplicaciones harán uso de sesiones para gestionar y restringir los accesos de los usuarios al sistema.

3.3.3. Portabilidad

3.3.3.1. Publicación

La aplicación podrá ser puesta a disposición de los usuarios a través de un servidor de aplicaciones, y además será necesario tener un JDK adecuado para la plataforma del sistema operativo, sin que sea necesario modificar el código fuente de la misma.

3.3.4. Rendimiento

3.3.4.1. Número de conexiones

El sistema debe permitir conexiones concurrentes de al menos unos 150 evaluadores.

3.3.4.2. Almacenamiento seguro y completo

El sistema debe almacenar de manera segura y total las respuestas a cada una de las preguntas de todas las encuestas.

3.3.5. Disponibilidad

El producto estará disponible para cada todas las personas a través del sitio institución de la ESPOCH, pero solo podrán hacer uso de esta los usuarios que dispongan de credenciales tanto la aplicación FRONT como la BACK.

3.3.6. Restricciones de diseño

Deberá desarrollarse una aplicación para la web.

III. ANÁLISIS Y DISEÑO

En el presente apartado se realizará un análisis a través de Casos de Uso en formato expandido, para su posterior diseño a través del Modelado de la Base de Datos, como también diagramas de secuencia, colaboración, estado, componentes y despliegue del Sistema Integrado de Evaluación del Desempeño Docente en su versión 2.0 (SIEDD V2.0) para la ESPOCH a través de sus respectivos formatos de modelado.

En el siguiente diagrama se muestran los procesos a llevarse a cabo de una manera muy general (para la aplicación FRONT y BACK), y que se mencionaron en el apartado anterior.

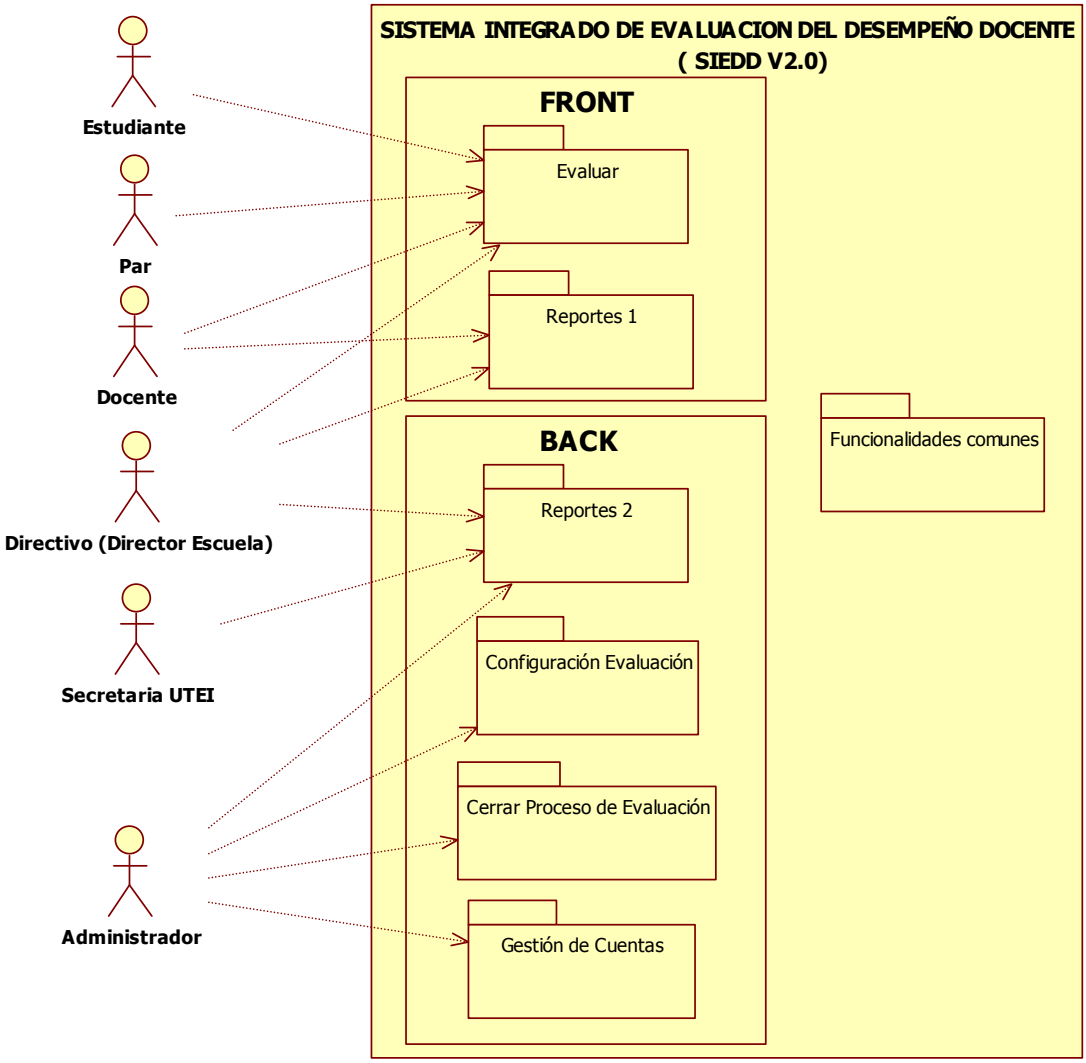


Diagrama de Paquetes SIEDD V2.0

1. Casos de Uso en formato expandido

En este punto del documento se dará a conocer los principales casos de uso y de los principales actores que intervendrán en el mismo. Se realizará una descripción a través de Casos de Uso en formato expandido utilizando un lenguaje natural para una mayor y mejor comprensión.

1.1. Evaluar

CASO DE USO: EVALUAR

Identificador de Caso de Uso:	CU001
Nombre de Caso de Uso:	Evaluar
Actores :	Estudiante, Docente, Directivo, Par
Propósito:	Permitir realizar la evaluación docente.
Visión General:	El Estudiante será el que evalué a los docentes de cada materia por las que cursa actualmente, el Docente se realizará una autoevaluación, el Directivo evaluará a los docentes que forman parte de su escuela, y el Par será el que evalué al docente que haya dado seguimiento presencial en clases.
Tipo:	Esencial
Referencias:	RF1, RF2

CURSO TÍPICO DE EVENTOS	
ACCIÓN DEL ACTOR	**RESPUESTA DEL SISTEMA**
1. El Caso de Uso comienza cuando el Estudiante, Docente, Directivo o Par desean llevar a cabo su evaluación docente. 2. Detallará los datos para la autenticación (Ver Caso de Uso FUNCIONALIDADES COMUNES). 3. Selecciona la tarea correspondiente a realizar.	4. Presenta un listado de las evaluaciones disponibles de acuerdo al tipo de usuario autenticado.
5. El usuario seleccionará la Materia de acuerdo al docente que la dicta para realizar la evaluación.	6. Genera la respectiva encuesta, que constará de un número indefinido de preguntas (mutuamente excluyentes), y cada una de las cuales tendrá hasta 5 alternativas de respuesta, que tendrán una valoración diferente.
7. El usuario seleccionará las respuestas adecuadas de acuerdo a su criterio respecto del desempeño del docente en cada una de las preguntas. 8. El usuario procederá a guardar la encuesta.	9. Almacenar las respuestas a cada pregunta y el comentario si lo hubiera.
CURSOS ALTERNATIVOS	
Línea 8. Muestra un error mediante un mensaje, de acuerdo al problema presentado. **-** Todas las preguntas deben ser respondidas.	

1.2. Armar Encuesta

CASO DE USO: ARMAR ENCUESTA

Identificador de Caso de Uso:	CU002
Nombre de Caso de Uso:	Armar Encuesta
Actores :	Administrador
Propósito:	Permitir configurar una encuesta.
Visión General:	El Administrador será el responsable de generar la estructura de la encuesta para cada tipo de evaluador

	(estudiantes, docentes, directivos y pares).
Tipo:	Esencial
Referencias:	RF7
CURSO TÍPICO DE EVENTOS	
ACCIÓN DEL ACTOR	**RESPUESTA DEL SISTEMA**
1. El Caso de Uso comienza cuando el Administrador desea generar una nueva encuesta para un proceso de evaluación 2. Detallará los datos para la autenticación (Ver Caso de Uso FUNCIONALIDADES COMUNES). 3. Selecciona tarea correspondiente a realizar. 4. Registra datos para nuevo proceso de evaluación. 5. Envía a guardar datos del proceso de evaluación.	6. Procederá a almacenar datos del proceso de evaluación.
7. Registra datos para nuevo Ámbito. 8. Envía a guardar datos del Ámbito.	9. Procederá a almacenar datos del ámbito.
10. Registra datos para el nuevo Estándar. 11. Envía a guardar datos del estándar.	12. Procederá a almacenar datos del estándar.
13. Registra datos para el nuevo Indicador. 14. Envía a guardar datos del indicador.	15. Procederá a almacenar datos del indicador.
16. Registra datos para la nueva Pregunta. 17. Envía a guardar datos de la pregunta.	18. Procederá a almacenar datos de la pregunta.
19. Registra datos para la nueva Opción. 20. Envía a guardar datos de la opción.	21. Procederá a almacenar datos de la opción.
CURSOS ALTERNATIVOS	
Línea 5, 8, 11, 14, 17, 20. Muestra un error mediante un mensaje, de acuerdo al problema presentado. **-** Todos los datos son obligatorios.	

1.3. Cerrar Proceso de Evaluación

CASO DE USO: CERRAR PROCESO DE EVALUACIÓN

Identificador de Caso de Uso:	CU003
Nombre de Caso de Uso:	Cerrar Proceso de Evaluación
Actores :	Administrador
Propósito:	Permitir que se cierre (cálculo de resultados) el proceso de evaluación vigente.

Visión General:	El Administrador será el responsable de cerrar el proceso de evaluación docente.
Tipo:	Esencial
Referencias:	RF8
CURSO TÍPICO DE EVENTOS	
ACCIÓN DEL ACTOR	**RESPUESTA DEL SISTEMA**
1. El Caso de Uso comienza cuando ha concluido el proceso de evaluación y el Administrador desea obtener los resultados de las evaluaciones. 2. Detallará los datos para la autenticación (Ver Caso de Uso FUNCIONALIDADES COMUNES). 3. Selecciona tarea correspondiente a realizar. 4. Confirma la tarea a ejecutar.	5. Procesa las respuestas de cada una de las evaluaciones en cada carrera procediendo a almacenar los respectivos resultados por indicador.
CURSOS ALTERNATIVOS	
Línea 3. Muestra un mensaje solicitando confirmación para la tarea a llevar a cabo. **-** Está seguro que quiere cerrar el proceso de evaluación y calcular los resultados.	

1.4. Gestionar Cuentas de Usuario

CASO DE USO: GESTIONAR CUENTAS DE USUARIO

Identificador de Caso de Uso:	CU004
Nombre de Caso de Uso:	Gestionar Cuentas de Usuario
Actores :	Administrador
Propósito:	Permitir agregar, eliminar, actualizar cuentas de usuario, de acuerdo a un tipo y restablecer contraseñas.
Visión General:	Ingresar cuentas de usuario para autenticarse al sistema.
Tipo:	Esencial
Referencias:	RF9
CURSO TÍPICO DE EVENTOS	
ACCIÓN DEL ACTOR	**RESPUESTA DEL SISTEMA**
1. El Caso de Uso comienza cuando el Administrador desea registrar una nueva cuenta de usuario. 2. Ingresa los datos de la cuenta de usuario. 3. El Administrador guarda los datos establecidos.	4. Valida los datos establecidos. 5. Ingreso de los valores en la BD. 6. Presenta en forma detallada los datos de la cuenta creada.
CURSOS ALTERNATIVOS	
Línea 4. Muestra un error mediante un mensaje, de acuerdo al problema presentado. **-** Todos los datos son obligatorios. – Cuenta de usuario existente.	

1.5. Visualizar Estadísticas

CASO DE USO: VISUALIZAR ESTADÍSTICAS

Identificador de Caso de Uso:	CU005
Nombre de Caso de Uso:	Visualizar Estadísticas
Actores :	Docente, Directivo, Administrador, Secretaria UTEI
Propósito:	Permitir visualizar las estadísticas generadas en el proceso de evaluación vigente

Visión General:	El Directivo será el responsable de controlar el porcentaje de participación en el proceso de evaluación actual.
Tipo:	Esencial
Referencias:	RF5, RF10

CURSO TÍPICO DE EVENTOS

ACCIÓN DEL ACTOR	RESPUESTA DEL SISTEMA
1. El Caso de Uso comienza cuando el cualquiera de los usuarios desea obtener un reporte del proceso de evaluación. 2. Detallará los datos para la autenticación (Ver Caso de Uso FUNCIONALIDADES COMUNES). 3. Selecciona el reporte correspondiente:	4. Genera un archivo con la información correspondiente:
a) El Docente selecciona el reporte de Resultados por Carrera.	a) Contiene: Facultad, Escuela, Carrera, Docente, puntaje obtenido en cada Indicador agrupado por Estándar y Ámbito.
b) El Directivo selecciona el reporte de Participación.	b) Contiene: Escuela, Carrera, Porcentaje de Participación en cada Carrera.
c) El Directivo selecciona el reporte de Estudiantes o Docentes sin evaluar.	c) Contiene: Carrera, Nivel, Materia, Paralelo, Nombres y Número de Cédula de los estudiantes o docentes.
d) El Directivo selecciona el reporte de Resultados por Carrera.	d) Contiene: Facultad, Escuela, Carrera, Docente, Porcentaje del docente en la evaluación vigente, el Total de Participación.
e) El Administrador o la Secretaria de la UTEI selecciona el reporte de Certificados de Evaluación. Y selecciona la Facultad y especifica el C.I.	e) Contiene: Nombres y C. I. del Docente, Facultad, Escuelas, Carrera, Porcentaje obtenido.
f) El Administrador selecciona el reporte de Resultados por Facultad.	f) Contiene: Facultad, Docente, Porcentaje del docente en la evaluación vigente, el Total de Participación.
g) El Administrador selecciona el reporte de Participación por Facultad.	g) Contiene: Facultad, Porcentaje de Participación en cada Facultad.
h) El Administrador selecciona el reporte de Evaluaciones disponibles para Estudiantes, Docentes, Directivos o Pares. Y especifica C.I.	h) Contiene: Facultad, Escuela, Carrera, Nombres del evaluador, Nombre del Docente a evaluar, Estado de la evaluación.
i) El Administrador selecciona el reporte de Cuestionarios configurados.	i) Contiene: Detalle de Preguntas con sus respectivas opciones de respuesta, y categorizadas de acuerdo al Ámbito, Estándar, Indicador al que pertenece.

CURSOS ALTERNATIVOS

Línea 3 opción e. Muestra un error mediante un mensaje, de acuerdo al problema presentado.
- No se puede generar el reporte, seleccione una Facultad y especifique el C.I. del

docente. **Línea 3 opción h.** Muestra un error mediante un mensaje, de acuerdo al problema presentado. **-** No se puede generar el reporte, especifique el C.I. del docente.

1.6. Funcionalidades comunes

1.6.1. Autenticarse al Sistema

CASO DE USO: INGRESAR AL SISTEMA

Identificador de Caso de Uso:	CU006
Nombre de Caso de Uso:	Autenticarse al Sistema
Actores :	Administrador, Directivo, Docente, Estudiante, Par, Secretaria UTEI.
Propósito:	Permitir ingresar al sistema de acuerdo a un Tipo de Usuario (Perfil).
Visión General:	Los usuarios podrán acceder al sistema a través del uso de su Perfil, Cuenta de Usuario y Clave.
Tipo:	Esencial
Referencias:	RF3, RF11
CURSO TÍPICO DE EVENTOS	
ACCIÓN DEL ACTOR	**RESPUESTA DEL SISTEMA**
1. El Caso de Uso comienza cuando cualquiera de los usuarios quiere hacer uso del sistema. 3. Detalla los datos solicitados. 4. Decide Iniciar validación de datos.	2. Presenta una pantalla de autenticación, que solicitará: Tipo de Usuario, Nombre del Usuario y Clave. 5. Valida los datos del usuario. 6. Presenta listado de tareas disponibles de acuerdo al Tipo de Usuario que se autentico.
CURSOS ALTERNATIVOS	
Línea 4. Muestra un error mediante un mensaje, de acuerdo al problema presentado. **-** Todos los datos son obligatorios. **Línea 6.** Muestra un error mediante un mensaje, de acuerdo al problema presentado. **-** Datos de usuario no válidos.	

1.6.2. Cambiar contraseña

CASO DE USO: CAMBIAR CONTRASEÑA

Identificador de Caso de Uso:	CU006
Nombre de Caso de Uso:	Cambiar contraseña
Actores :	Administrador, Directivo, Docente, Estudiante, Par, Secretaria UTEI.
Propósito:	Permitir modificar la contraseña.
Visión General:	El usuario podrá modificar su contraseña cuando así lo

	decida.
Tipo:	Esencial
Referencias:	RF4, RF12.
CURSO TÍPICO DE EVENTOS	
ACCIÓN DEL ACTOR	**RESPUESTA DEL SISTEMA**
1. El Caso de Uso comienza cuando el usuario desea modificar su contraseña actual. 2. Detallará los datos para la autenticación (Ver Caso de Uso FUNCIONALIDADES COMUNES). 3. Selecciona tarea correspondiente a realizar. 5. Detalla los datos para modificar contraseña (contraseña actual, contraseña nueva y confirmación de contraseña nueva). 6. Manda a guardar nuevos datos especificados.	4. Presenta formulario de datos para modificación de contraseña. 7. Almacena en la BD
CURSOS ALTERNATIVOS	
Línea 6. Muestra un error mediante un mensaje, de acuerdo al problema presentado. **-** Todos los datos son obligatorios. **Línea 7.** Muestra un error mediante un mensaje, de acuerdo al problema presentado. **-** No se puedo almacenar, no hay conexión con el servidor.	

2. Diagramas de Casos de Uso

2.1 Evaluar

Involucra los casos de uso para que el Estudiante, Docente, Directivo o Par realicen una evaluación docente.

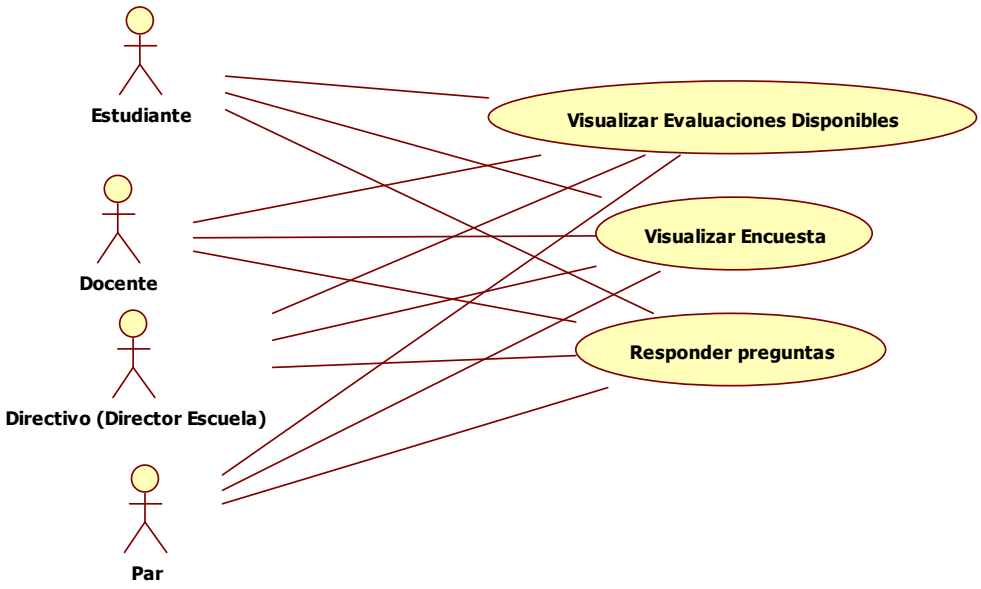


Diagrama de Caso de Uso: Evaluar

2.2 Armar Encuesta

Comprende los casos de uso para poner a disposición de los Estudiantes, Docentes, Directivos o Pares sus respectiva Encuesta.

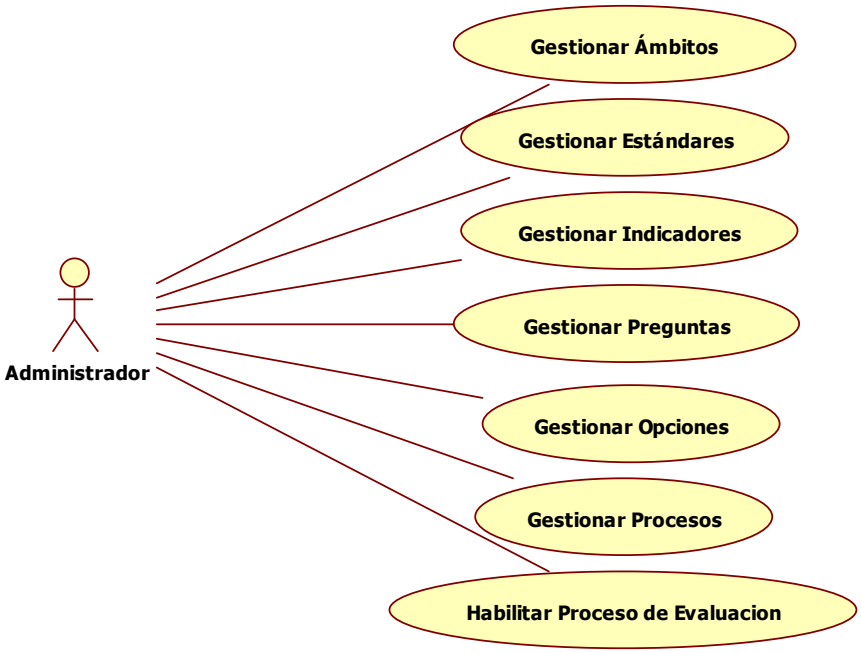


Diagrama de Caso de Uso: Armar Encuesta

2.3 Cerrar Proceso de Evaluación

Involucra los casos de uso para cerrar el proceso de evaluación actualmente vigente.

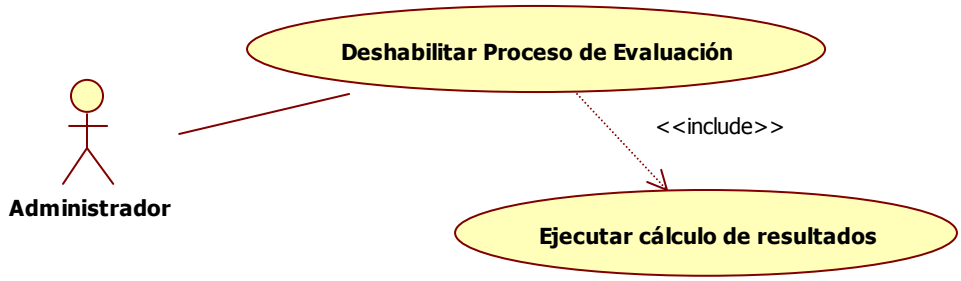


Diagrama de Caso de Uso: Cerrar Proceso de Evaluación

2.4 Gestionar Cuentas de Usuario

Involucra los casos de uso para cerrar el proceso de evaluación actualmente vigente.

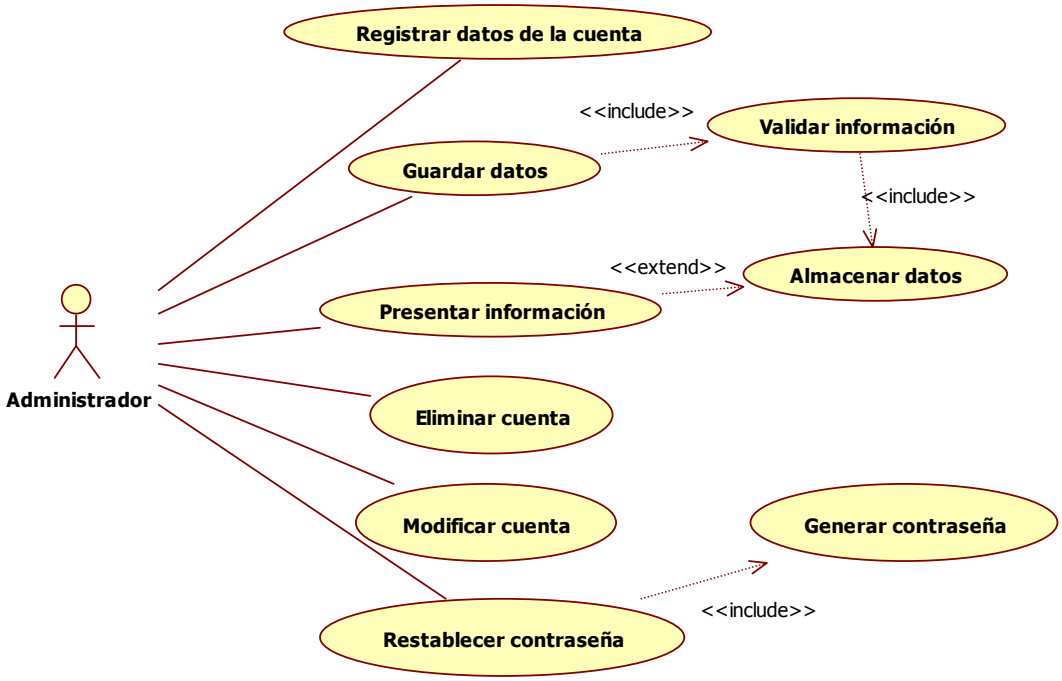


Diagrama de Caso de Uso: Gestionar Cuentas de Usuario

2.5 Visualizar Estadísticas

Abarca los casos de uso para obtener información estadística del proceso de evaluación vigente.

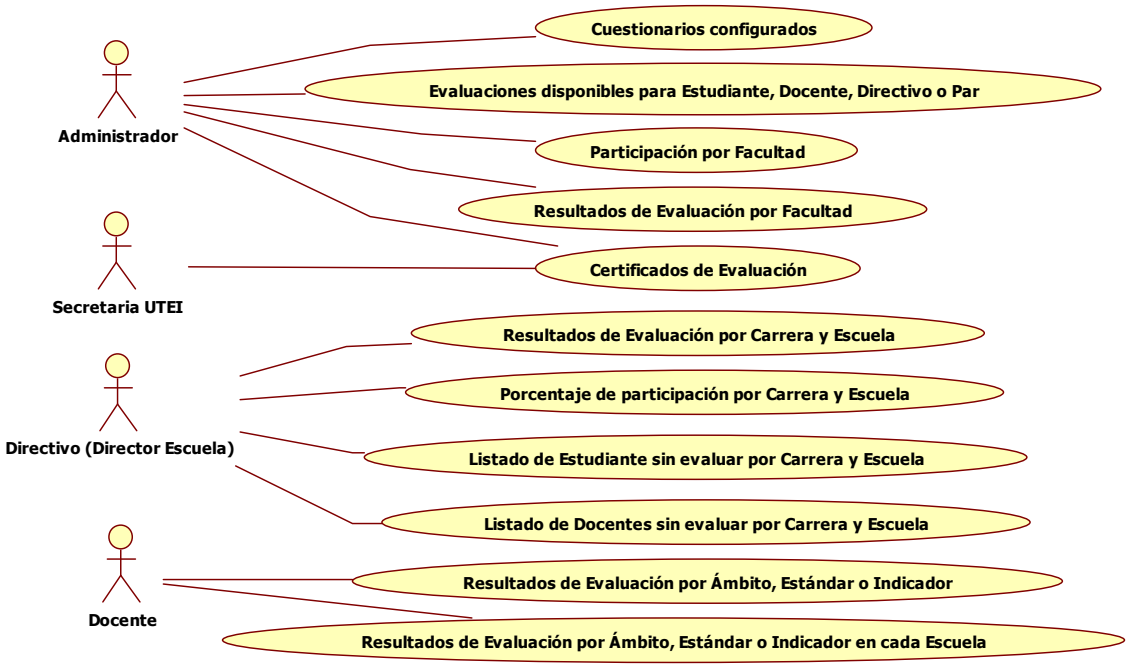


Diagrama de Caso de Uso: Visualizar Estadísticas

2.6 Funcionalidades comunes

Contempla funciones básicas y comunes entre los diferentes actores del sistema.

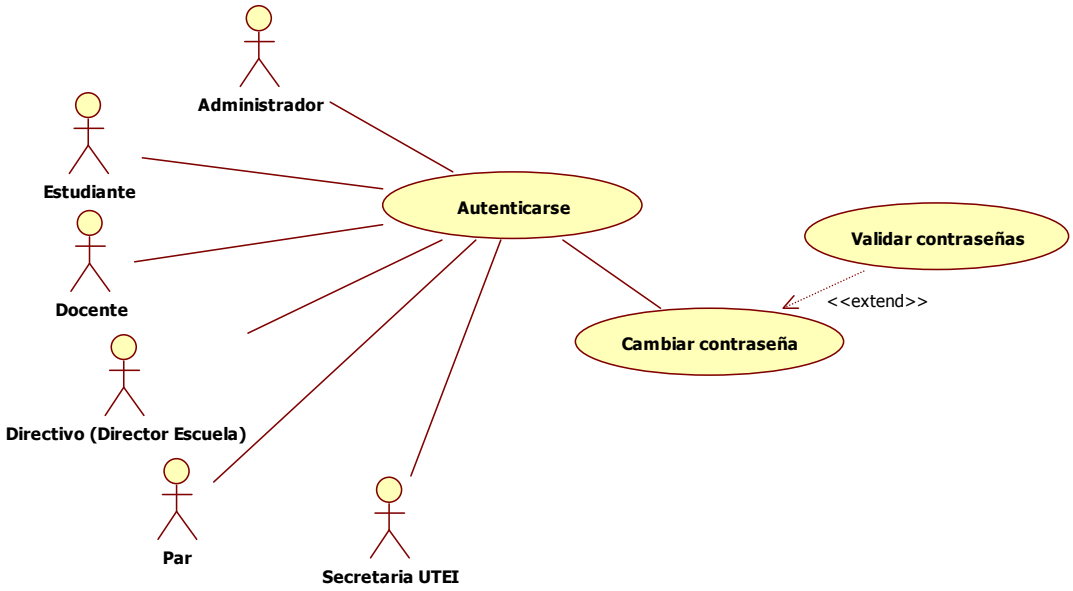
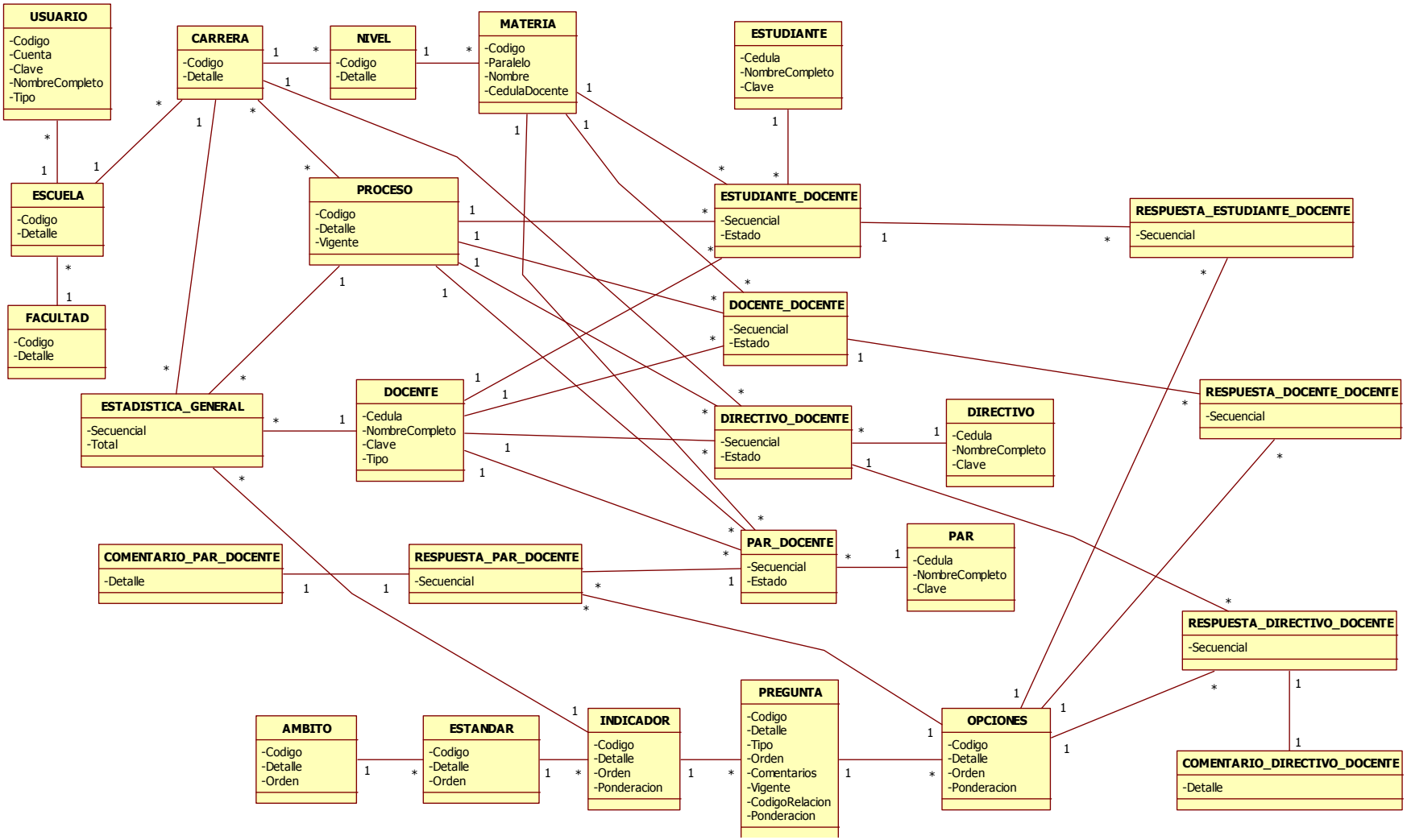


Diagrama de Caso de Uso: Funcionalidades comunes

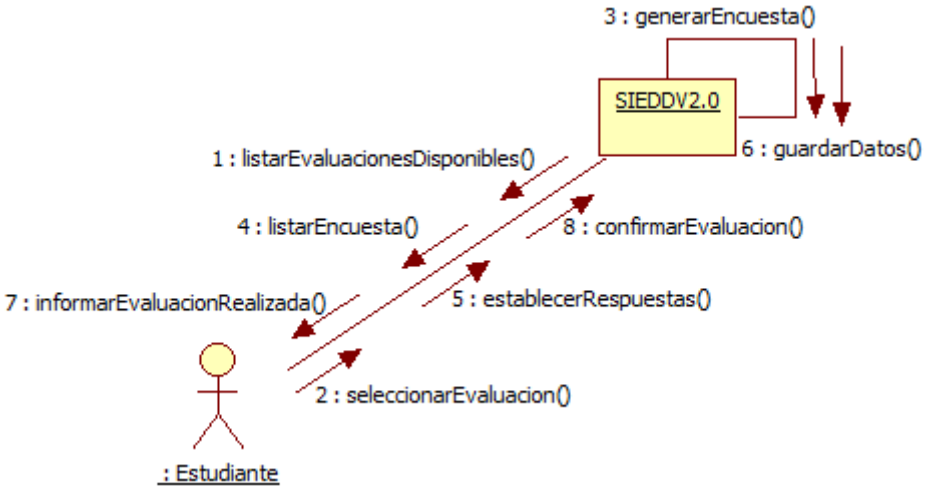
3. Modelo Conceptual



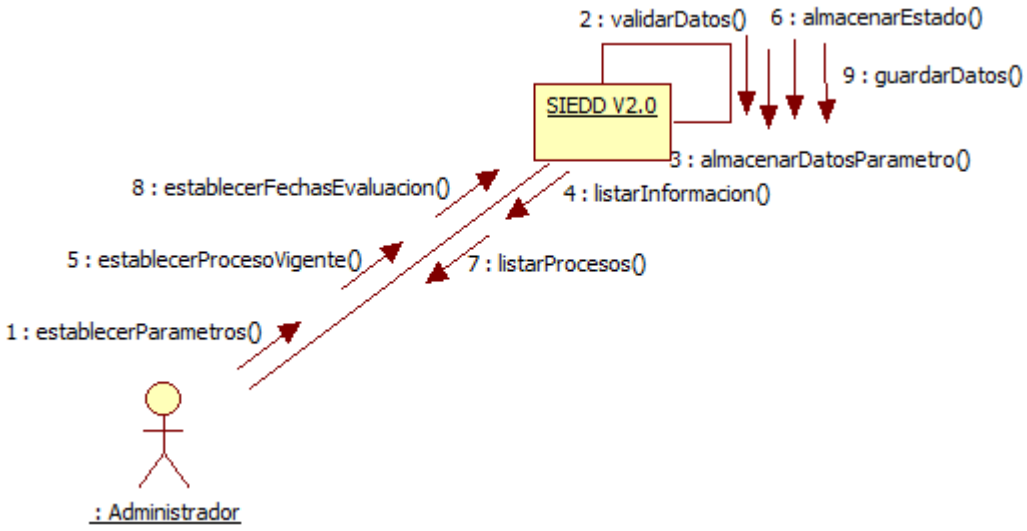
4. Diagramas de Interacción

4.1 Diagramas de Colaboración

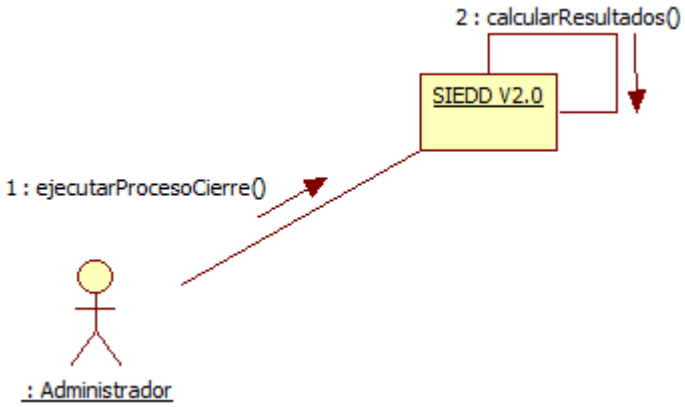
4.1.1. Evaluar



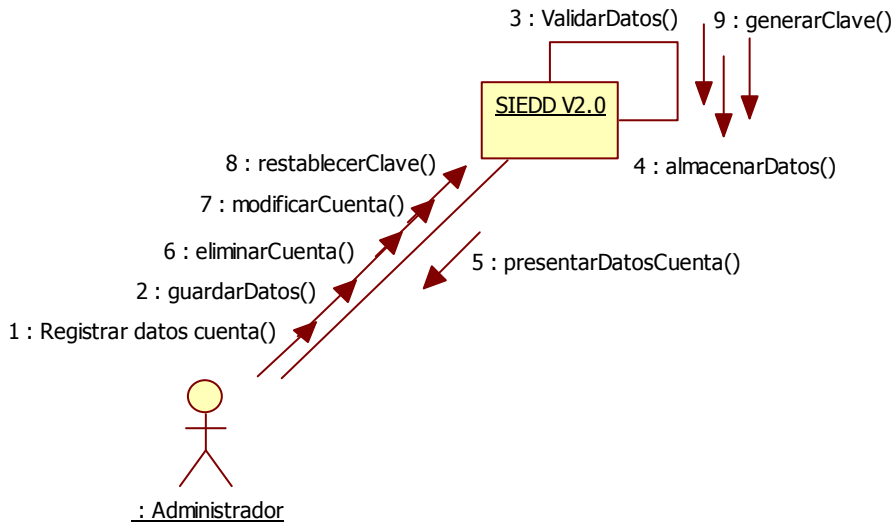
4.1.2. Armar Encuesta



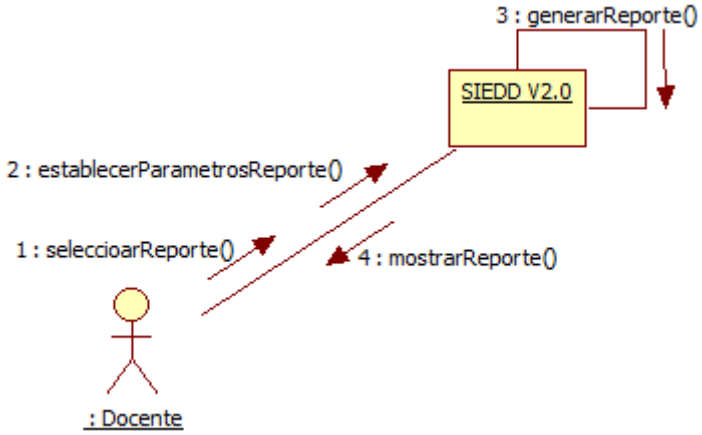
4.1.3. Cerrar Proceso de Evaluación



4.1.4. Gestionar Cuentas de Usuario

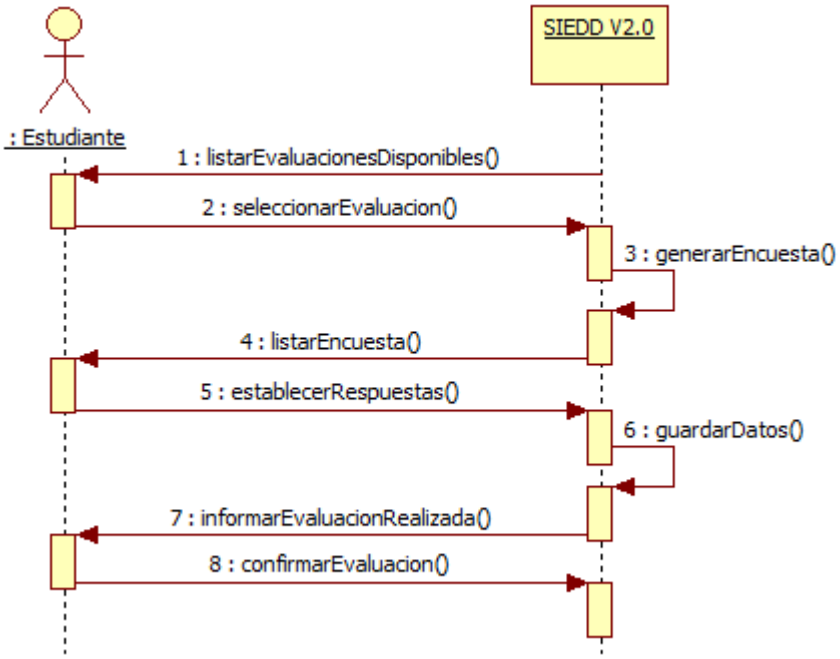


4.1.5. Visualizar Estadísticas

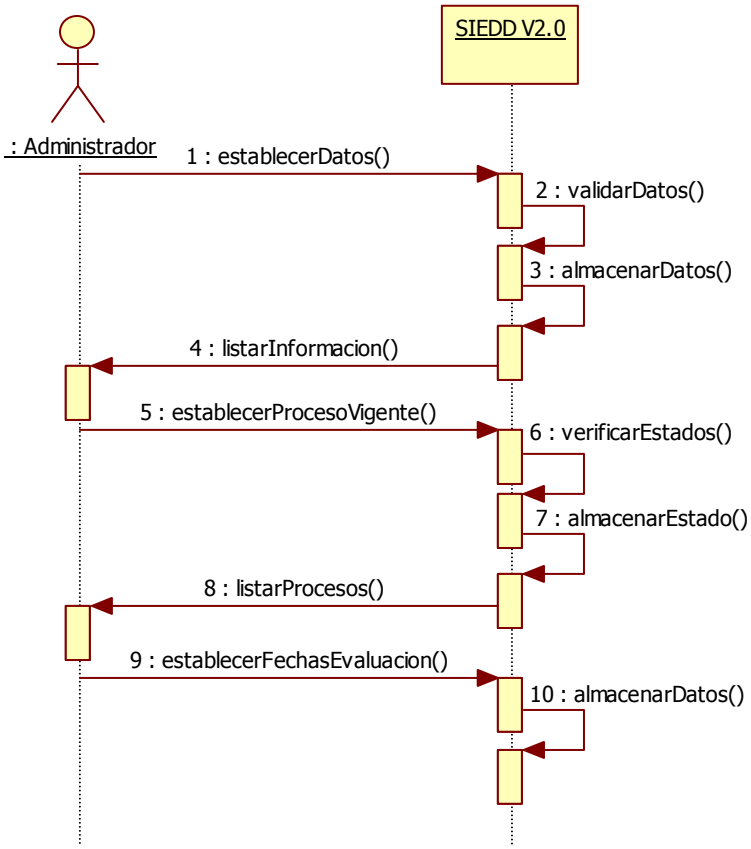


4.2 Diagramas de Secuencia

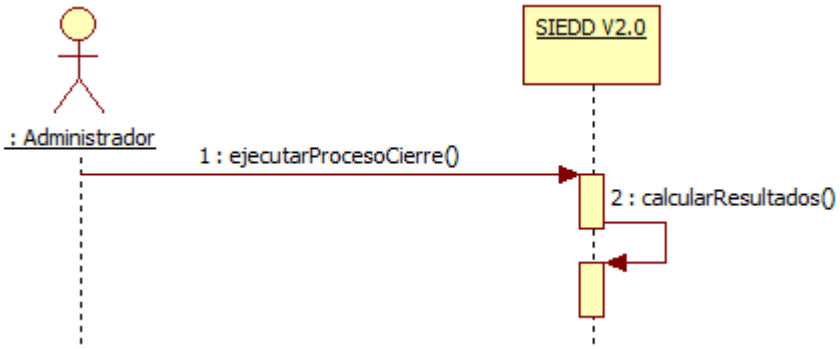
4.2.1 Evaluar



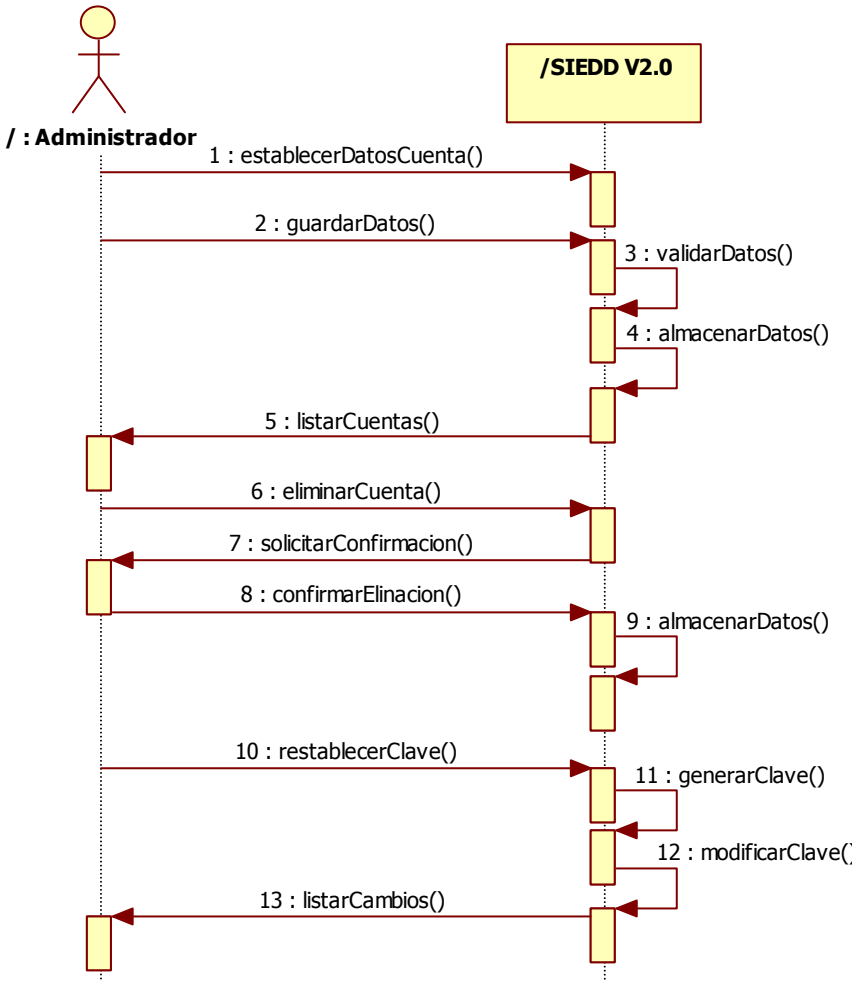
4.2.2 Armar Encuesta



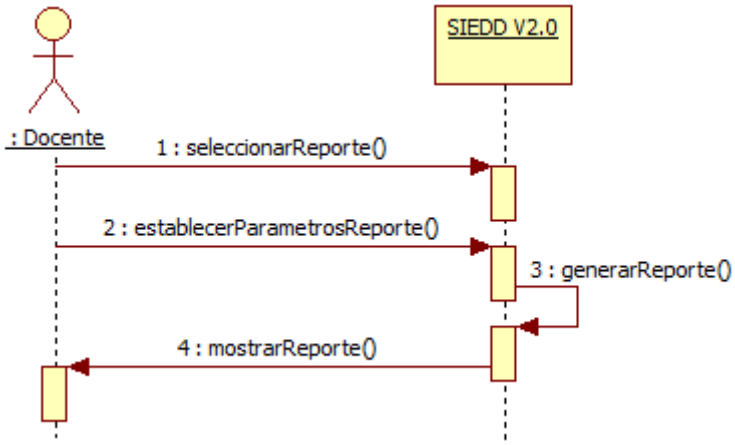
4.2.3 Cerrar Proceso Evaluación



4.2.4 Gestionar Cuentas de Usuarios



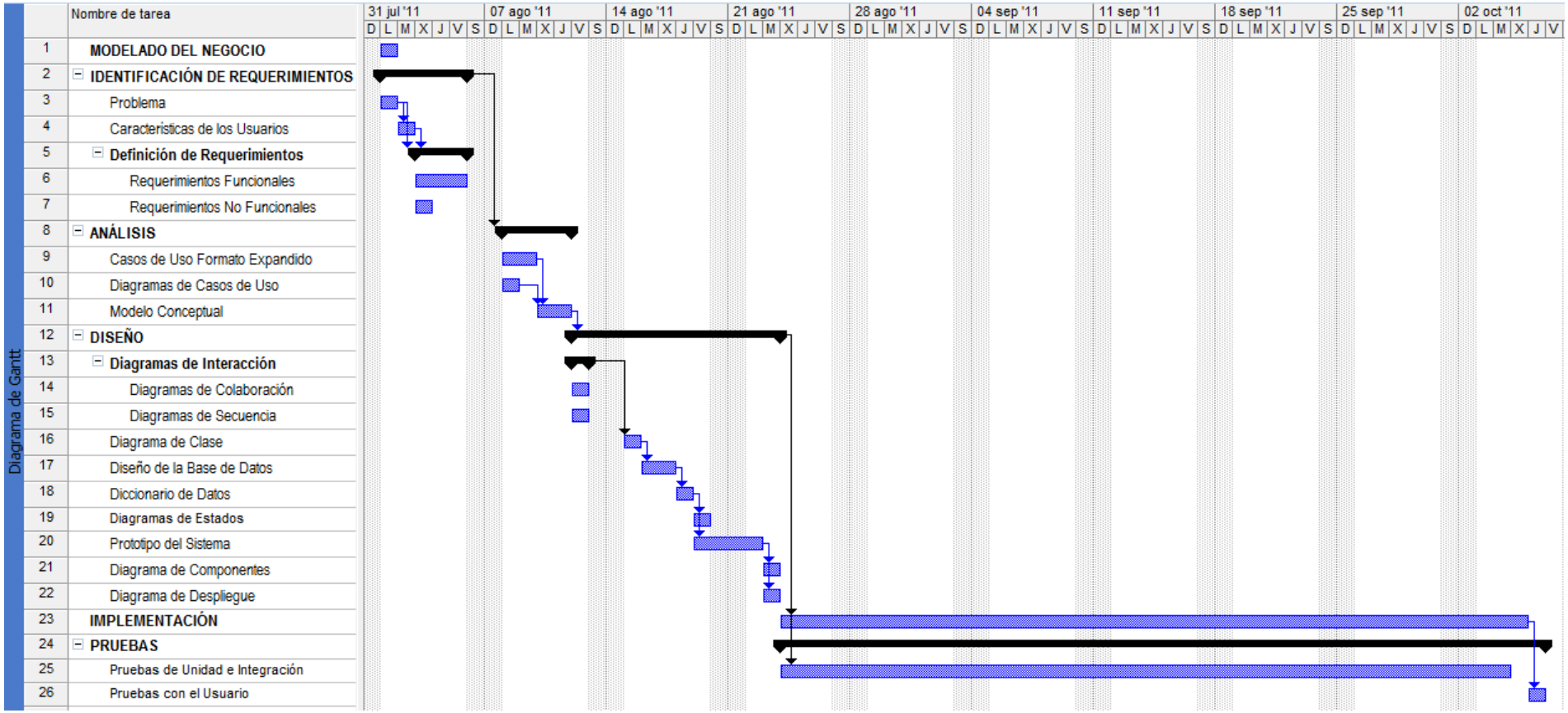
4.2.5 Visualizar Estadísticas



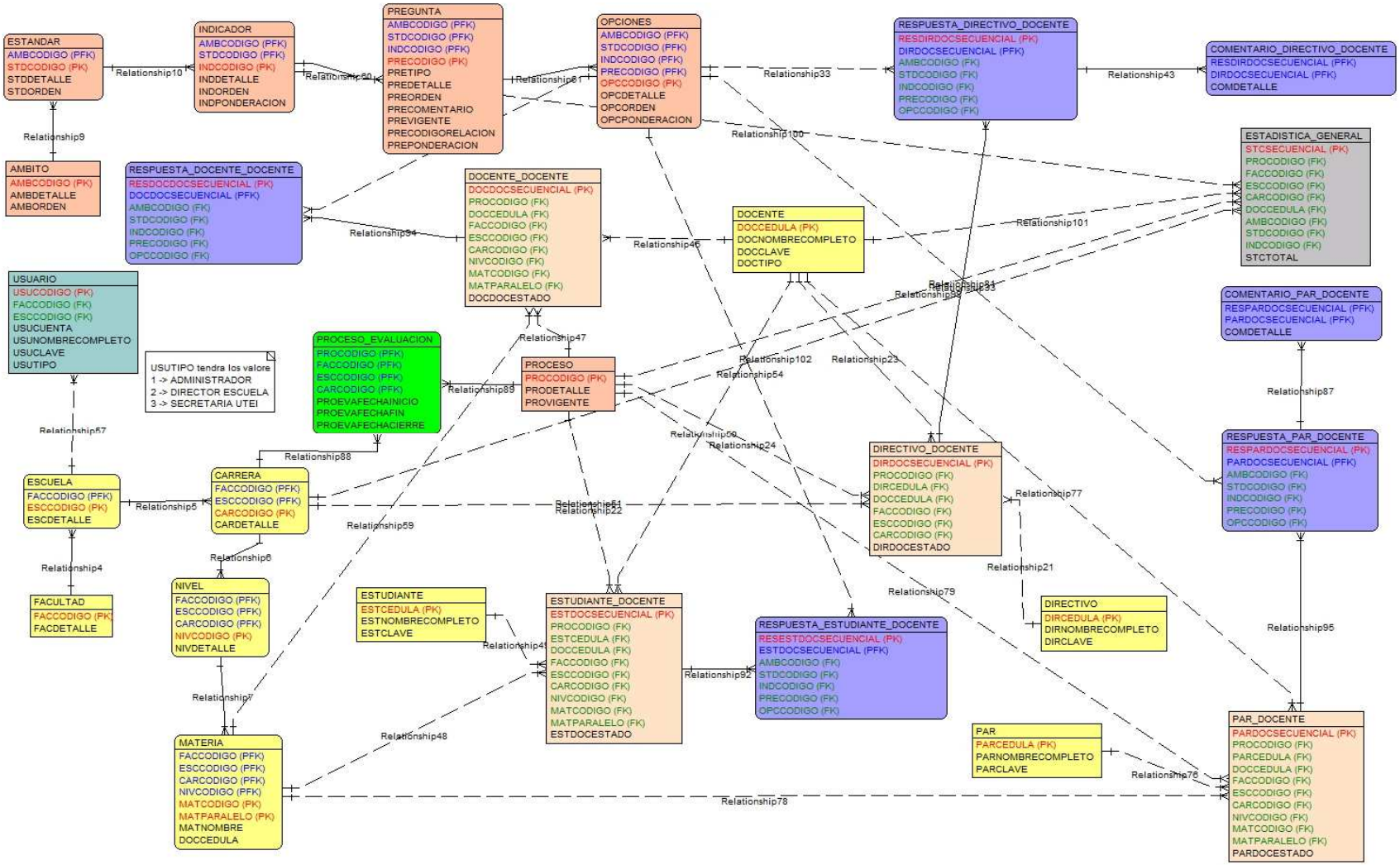
5. Cronograma de Trabajo

	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	**MODELADO DEL NEGOCIO**	1 día?	lun 01/08/11	lun 01/08/11	
2	**IDENTIFICACIÓN DE REQUERIMIENTOS - SRS**	**5 días?**	**lun 01/08/11**	**vie 05/08/11**	
3	Problema	1 día	lun 01/08/11	lun 01/08/11	
4	Caracteristicas de los Usuarios	1 día?	mar 02/08/11	mar 02/08/11	3
5	**Definición de Requerimientos**	**3 días?**	**mié 03/08/11**	**vie 05/08/11**	**3-4**
6	Requerimientos Funcionales	3 días	mié 03/08/11	vie 05/08/11	
7	Requerimientos No Funcionales	1 día?	mié 03/08/11	mié 03/08/11	
8	**ANÁLISIS**	**4 días?**	**lun 08/08/11**	**jue 11/08/11**	**2**
9	Casos de Uso Formato Expandido	2 días	lun 08/08/11	mar 09/08/11	
10	Diagramas de Casos de Uso	1 día?	lun 08/08/11	lun 08/08/11	
11	Modelo Conceptual	2 días	mié 10/08/11	jue 11/08/11	9-10
12	**DISEÑO**	**8 días?**	**vie 12/08/11**	**mar 23/08/11**	**11**
13	**Diagramas de Interacción**	**1 día?**	**vie 12/08/11**	**vie 12/08/11**	
14	Diagramas de Colaboración	1 día?	vie 12/08/11	vie 12/08/11	
15	Diagramas de Secuencia	1 día?	vie 12/08/11	vie 12/08/11	
16	Diagrama de Clase	1 día?	lun 15/08/11	lun 15/08/11	13
17	Diseño de la Base de Datos	2 días	mar 16/08/11	mié 17/08/11	16
18	Diccionario de Datos	1 día?	jue 18/08/11	jue 18/08/11	17
19	Diagramas de Estados	1 día?	vie 19/08/11	vie 19/08/11	18
20	Prototipo del Sistema	2 días	vie 19/08/11	lun 22/08/11	18
21	Diagrama de Componentes	1 día?	mar 23/08/11	mar 23/08/11	20
22	Diagrama de Despliegue	1 día?	mar 23/08/11	mar 23/08/11	20
23	**IMPLEMENTACIÓN**	31 días	mié 24/08/11	mié 05/10/11	12
24	**PRUEBAS**	**32 días**	**mié 24/08/11**	**jue 06/10/11**	
25	Pruebas de Unidad e Integración	30 días	mié 24/08/11	mar 04/10/11	12
26	Pruebas con el Usuario	1 día	jue 06/10/11	jue 06/10/11	23

5.1. Diagrama de Gantt



6. Esquema de la Base de Datos



7. Diccionario de datos

7.1 Tabla AMBITO

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	AMBCODIGO	Varchar2(10)	SI	NO	Clave Primaria de la tabla y es el Código del Ámbito.
	AMBDETALLE	Varchar2(250)	NO	NO	Descripción del Ámbito.
	AMBORDEN	Number(2, 0)	NO	NO	Orden de presentación del Ámbito.

7.2 Tabla ESTANDAR

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	AMBCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria. Además es Clave Foránea con respecto a la tabla AMBITO.
PK	STDCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria y es el Código del Estándar.
	STDDETALLE	Varchar2(250)	NO	NO	Descripción del Estándar.
	STDORDEN	Number(2, 0)	NO	NO	Orden de presentación del Estándar.

7.3 Tabla INDICADOR

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	AMBCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria. Además es Clave Foránea con respecto a la tabla ESTANDAR.
PFK	STDCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria. Además es Clave Foránea con respecto a la tabla ESTANDAR.
PK	INDCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria y es el Código del Indicador.
	INDDETALLE	Varchar2(500)	NO	NO	Descripción del Indicador.
	INDORDEN	Number(2, 0)	NO	NO	Orden de presentación del Indicador.
	INDPONDERACION	Number(6, 4)	NO	NO	Puntaje que tiene el Indicador.

7.4 Tabla PREGUNTA

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	AMBCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria. Además es Clave Foránea con respecto a la tabla INDICADOR.
PFK	STDCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria. Además es Clave Foránea con respecto a la tabla INDICADOR.
PFK	INDCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria. Además es Clave Foránea con respecto a la tabla INDICADOR.
PK	PRECODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria y es el Código de la Pregunta.
	PRETIPO	Number(1, 0)	NO	NO	Identifica al tipo de Pregunta, el cual puede ser: 1 = ESTUDIANTE, 2 DOCENTE, 3 = DIRECTIVO, 4 PAR
	PREDETALLE	Varchar2(500)	NO	NO	Descripción de la Pregunta.
	PREORDEN	Number(2, 0)	NO	NO	Orden de presentación de la Pregunta.
	PRECOMENTARIO	Number(1, 0)	NO	NO	Especifica si la pregunta tiene (1) o no (0) comentario (solo en el caso de DIRECTIVO Y PAR).
	PREVIGENTE	Number(1, 0)	NO	NO	Describe si está o no vigente.
	PRECODIGORELACION	Varchar2(10)	NO	NO	En el caso de tipo PAR especifica el código de la pregunta de tipo DIRECTIVO con que se relaciona. En el caso de tipo DIRECTIVO especifica el código de la pregunta de tipo DOCENTE con que se relaciona. En el caso de tipo DOCENTE especifica el código de la pregunta de tipo ESTUDIANTE con que se relaciona. En el caso de tipo ESTUDIANTE no contiene valor.
	PREPONDERACION	Number(6, 4)	NO	NO	Puntaje que tiene la Pregunta.

7.5 Tabla OPCIONES

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	AMDCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla PREGUNTA.
PFK	STDCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla PREGUNTA.
PFK	INDCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla PREGUNTA.
PFK	PRECODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla PREGUNTA.
PK	OPCCODIGO	Varchar2(5)	SI	NO	Formar parte de la Clave Primaria y es el Código de la Opción.
	OPCDETALLE	Varchar2(50)	NO	NO	Descripción de la Opción.
	OPCORDEN	Number(1, 0)	NO	NO	Orden de presentación de la Opción.
	OPCPONDERACION	Number(11, 6)	NO	NO	Puntaje que tiene la Opción.

7.6 Tabla FACULTAD

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	FACCODIGO	Varchar2(10)	SI	NO	Clave Primaria de la tabla y es el Código de la Facultad.
	FACDETALLE	Varchar2(100)	NO	NO	Nombre de la Facultad.

7.7 Tabla ESCUELA

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	FACCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla FACULTAD.
PK	ESCCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria y es el Código de la Escuela.
	ESCDETALLE	Varchar2(100)	NO	NO	Nombre de la Escuela.

7.8 Tabla CARRERA

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	FACCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla ESCUELA.
PFK	ESCCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla ESCUELA.
PK	CARCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria y es el Código de la Carrera.
	CARDETALLE	Varchar2(100)	NO	NO	Nombre de la Carrera.

7.9 Tabla NIVEL

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	FACCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla CARRERA.
PFK	ESCCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla CARRERA.
PFK	CARCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla CARRERA.
PK	NIVCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria y es el Código del Nivel.
	NIVDETALLE	Varchar2(50)	NO	NO	Nombre del Nivel (Semestre).

7.10 Tabla MATERIA

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	FACCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla NIVEL.
PFK	ESCCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla NIVEL.
PFK	CARCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla NIVEL.
PFK	NIVCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla NIVEL.
PK	MATCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria y es el Código del Materia.
PK	MATPARALELO	Varchar2(2)	SI	NO	Formar parte de la Clave Primaria y describe el Paralelo en donde se imparte la Materia.
	MATDETALLE	Varchar2(50)	NO	NO	Nombre de la Materia.
	DOCCEDULA	Char(11)	NO	NO	Número de Cédula del Docente que dicta la Matera.

7.11 Tabla PROCESO

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	PROCODIGO	Varchar2(10)	SI	NO	Clave Primaria de la tabla y es el Código del Proceso.
	PRODETALLE	Varchar2(50)	NO	NO	Descripción del Proceso.
	PROVIGENTE	Number(1, 0)	NO	NO	Determina si el proceso está o no vigente.

7.12 Tabla PROCESO_EVALUACION

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	PROCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla PROCESO.
PFK	FACCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla CARRERA.
PFK	ESCCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla CARRERA.
PFK	CARCODIGO	Varchar2(10)	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla CARRERA.
	PROEVAFECHAINICIO	Date	NO	NO	Determina la Fecha de Inicio del Proceso de Evaluación.
	PROEVAFECHAFIN	Date	NO	NO	Determina la Fecha de Final del Proceso de Evaluación.
	PROEVAFECHACIERRE	Date	NO	NO	Determina la Fecha de Cierre del Proceso de Evaluación.

7.13 Tabla ESTUDIANTE

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	ESTCEDULA	Char(11)	SI	NO	Clave Primaria de la tabla y es el Número de Cédula del Estudiante.
	ESTNOMBRECOMPLETO	Varchar2(200)	NO	NO	Nombres/Apellidos del Estudiante.
	ESTCLAVE	Varchar2(10)	NO	NO	Contraseña de acceso al sistema (por defecto es el número de cédula sin guión).

7.14 Tabla DOCENTE

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	DOCCEDULA	Char(11)	SI	NO	Clave Primaria de la tabla y es el Número de Cédula del Docente.
	DOCNOMBRECOMPLETO	Varchar2(200)	NO	NO	Nombres/Apellidos del Docente.
	DOCCLAVE	Varchar2(10)	NO	NO	Contraseña de acceso al sistema (por defecto es el número de cédula sin guión).
	DOCTIPO	Number(1, 0)	NO	NO	Describe el tipo del Docente: 1 = NOMBRAMIENTO, TITULARES 2 = CONTRATO, OCASIONALES 3 = INVITADOS

7.15 Tabla DIRECTIVO

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	DIRCEDULA	Char(11)	SI	NO	Clave Primaria de la tabla y es el Número de Cédula del Directivo.
	DIRNOMBRECOMPLETO	Varchar2(200)	NO	NO	Nombres/Apellidos del Directivo.
	DIRCLAVE	Varchar2(10)	NO	NO	Contraseña de acceso al sistema (por defecto es el número de cédula sin guión).

7.16 Tabla PAR

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	PARCEDULA	Char(11)	SI	NO	Clave Primaria de la tabla y es el Número de Cédula del Par Académico.
	PARNOMBRECOMPLETO	Varchar2(200)	NO	NO	Nombres/Apellidos del Par Académico.
	PARCLAVE	Varchar2(10)	NO	NO	Contraseña de acceso al sistema (por defecto es el número de cédula sin guión).

7.17 Tabla ESTUDIANTE_DOCENTE

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	ESTDOCSECUENCIAL	Number	SI	NO	Clave Primaria de la tabla y es el Número de Encuesta generada.
FK	PROCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla PROCESO.
FK	ESTCEDULA	Char(11)	SI	NO	Clave Foránea con respecto a la tabla ESTUDIANTE.
FK	DOCCEDULA	Char(11)	SI	NO	Clave Foránea con respecto a la tabla DOCENTE.
FK	FACCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	ESCCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	CARCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	NIVCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	MATCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	MATPARALELO	Varchar2(2)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
	ESTDOCESTADO	Number(1, 0)	NO	NO	Describe el Estado de la evaluación.

					0 = Sin Evaluar 1 = Evaluada

7.18 Tabla DOCENTE_DOCENTE

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	DOCDOCSECUENCIAL	Number	SI	NO	Clave Primaria de la tabla y es el Número de Encuesta generada.
FK	PROCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla PROCESO.
FK	DOCCEDULA	Char(11)	SI	NO	Clave Foránea con respecto a la tabla DOCENTE.
FK	FACCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	ESCCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	CARCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	NIVCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	MATCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	MATPARALELO	Varchar2(2)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
	DOCDOCESTADO	Number(1, 0)	NO	NO	Describe el Estado de la evaluación. 0 = Sin Evaluar 1 = Evaluada

7.19 Tabla DIRECTIVO_DOCENTE

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	DIRDOCSECUENCIAL	Number	SI	NO	Clave Primaria de la tabla y es el Número de Encuesta generada.
FK	PROCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla PROCESO.
FK	DIRCEDULA	Char(11)	SI	NO	Clave Foránea con respecto a la tabla DIRECTIVO.
FK	DOCCEDULA	Char(11)	SI	NO	Clave Foránea con respecto a la tabla DOCENTE.
FK	FACCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla CARRERA.
FK	ESCCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla CARRERA.
FK	CARCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla CARRERA.
	DIRDOCESTADO	Number(1, 0)	NO	NO	Describe el Estado de la

					evaluación. 0 = Sin Evaluar 1 = Evaluada

7.20 Tabla PAR_DOCENTE

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	PARDOCSECUENCIAL	Number	SI	NO	Clave Primaria de la tabla y es el Número de Encuesta generada.
FK	PROCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla PROCESO.
FK	PARCEDULA	Char(11)	SI	NO	Clave Foránea con respecto a la tabla PAR.
FK	DOCCEDULA	Char(11)	SI	NO	Clave Foránea con respecto a la tabla DOCENTE.
FK	FACCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	ESCCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	CARCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	NIVCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	MATCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
FK	MATPARALELO	Varchar2(2)	SI	NO	Clave Foránea con respecto a la tabla MATERIA.
	PARDOCESTADO	Number(1, 0)	NO	NO	Describe el Estado de la evaluación. 0 = Sin Evaluar 1 = Evaluada

7.21 Tabla RESPUESTA_ESTUDIANTE_DOCENTE

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	RESESTDOCSECUENCIAL	Number	SI	NO	Forma parte de la Clave Primaria es el Secuencial de la respuesta.
PFK	ESTDOCSECUENCIAL	Number	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla ESTUDIANTE_DOCENTE.
FK	AMBCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	STDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	INDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	PRECODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	OPCCODIGO	Varchar2(5)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.

7.22 Tabla RESPUESTA_DOCENTE_DOCENTE

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	RESDOCDOCSECUENCIAL	Number	SI	NO	Forma parte de la Clave Primaria es el Secuencial de la respuesta.
PFK	DOCDOCSECUENCIAL	Number	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla DOCENTE_DOCENTE.
FK	AMBCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	INDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	STDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	PRECODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	OPCCODIGO	Varchar2(5)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.

7.23 Tabla RESPUESTA_DIRECTIVO_DOCENTE

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	RESDIRDOCSECUENCIAL	Number	SI	NO	Forma parte de la Clave Primaria es el Secuencial de la respuesta.
PFK	DIRDOCSECUENCIAL	Number	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla DOCENTE_DOCENTE.
FK	AMBCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	STDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	INDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	PRECODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	OPCCODIGO	Varchar2(5)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.

7.24 Tabla RESPUESTA_PAR_DOCENTE

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	RESPARDOCSECUENCIAL	Number	SI	NO	Forma parte de la Clave Primaria es el Secuencial de la respuesta.
PFK	PARDOCSECUENCIAL	Number	SI	NO	Formar parte de la Clave Primaria Además es Clave Foránea con respecto a la tabla PAR_DOCENTE.
FK	AMBCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	STDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	INDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	PRECODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.
FK	OPCCODIGO	Varchar2(5)	SI	NO	Clave Foránea con respecto a la tabla OPCIONES.

7.25 Tabla COMENTARIO_DIRECTIVO_DOCENTE

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	RESDIRDOCSECUENCIAL	Number	SI	NO	Forma parte de la Clave Primaria y es la Clave Foránea con respecto a la tabla RESPUESTA_DIRECTIVO_DOCENTE.
PFK	DIRDOCSECUENCIAL	Number	SI	NO	Forma parte de la Clave Primaria y es la Clave Foránea con respecto a la tabla RESPUESTA_DIRECTIVO_DOCENTE.
	COMDETALLE	Varchar2(500)	NO	NO	Contiene el comentario.

7.26 Tabla COMENTARIO_PAR_DOCENTE

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PFK	RESPARDOCSECUENCIAL	Number	SI	NO	Forma parte de la Clave Primaria y es la Clave Foránea con respecto a la tabla RESPUESTA_PAR_DOCENTE.
PFK	PARDOCSECUENCIAL	Number	SI	NO	Forma parte de la Clave Primaria y es la Clave Foránea con respecto a la tabla RESPUESTA_PAR_DOCENTE.
	COMDETALLE	Varchar2(500)	NO	NO	Contiene el comentario.

7.27 Tabla ESTADISTICA_GENERAL

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	STCSECUENCIAL	Number	SI	NO	Clave Primaria de la tabla y es el Secuencial de la Estadística (resultado) registrada.
FK	PROCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla PROCESO.
FK	FACCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla CARRERA.
FK	ESCCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla CARRERA.
FK	CARCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla CARRERA.
FK	DOCCEDULA	Char(11)	SI	NO	Clave Foránea con respecto a la tabla DOCENTE.
FK	AMBCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla INDICADOR.
FK	STDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla INDICADOR.
FK	INDCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla INDICADOR.
	STCTOTAL	Number(10, 4)	NO	NO	Contiene el puntaje acumulado en el indicador respectivo.

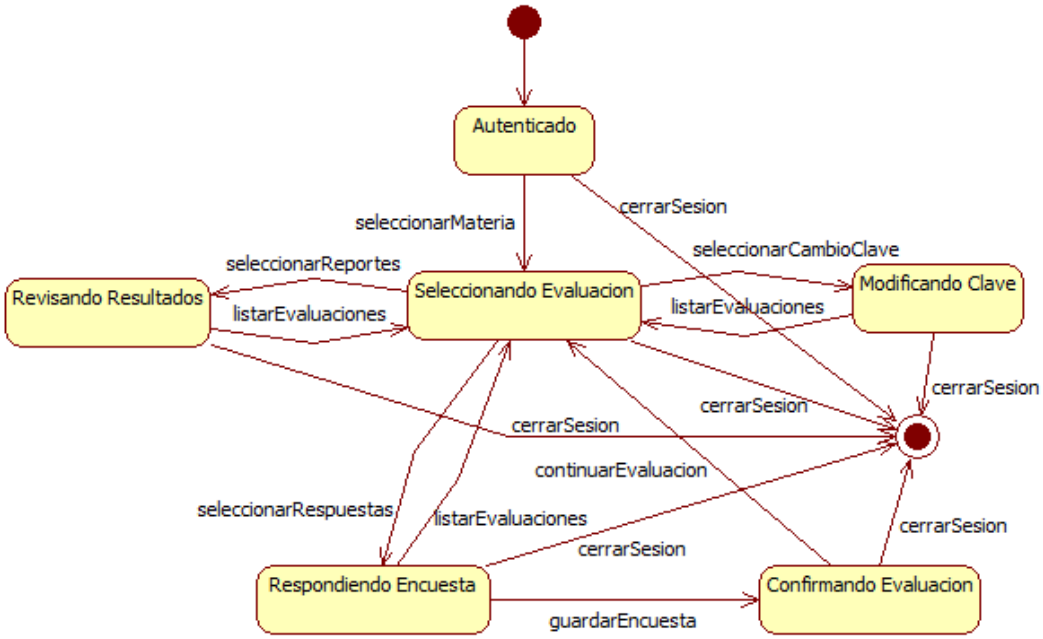
7.28 Tabla USUARIO

Atributos

Clave	Atributo	Tipo de Dato	No nulo	Valor único	Notas
PK	USUCODIGO	Number(3, 0)	SI	NO	Clave Primaria de la tabla y es el Secuencial de la Estadística (resultado) registrada.
FK	FACCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla ESCUELA.
FK	ESCCODIGO	Varchar2(10)	SI	NO	Clave Foránea con respecto a la tabla ESCUELA.
	USUCUENTA	Varchar2(20)	NO	NO	Describe la Cuenta de Usuario con la que accederá al sistema.
	USUNOMBRECOMPLETO	Varchar2(200)	NO	NO	Contiene Nombres/Apellidos del propietario de la cuenta.
	USUCLAVE	Varchar2(10)	NO	NO	Contraseña de acceso al sistema.
	USUTIPO	Number(1, 0)	NO	NO	Determina el tipo de usuario: 1 = ADMINISTRADOR 2 = DIRECTOR ESCUELA 3 = SECRETARIA UTEI

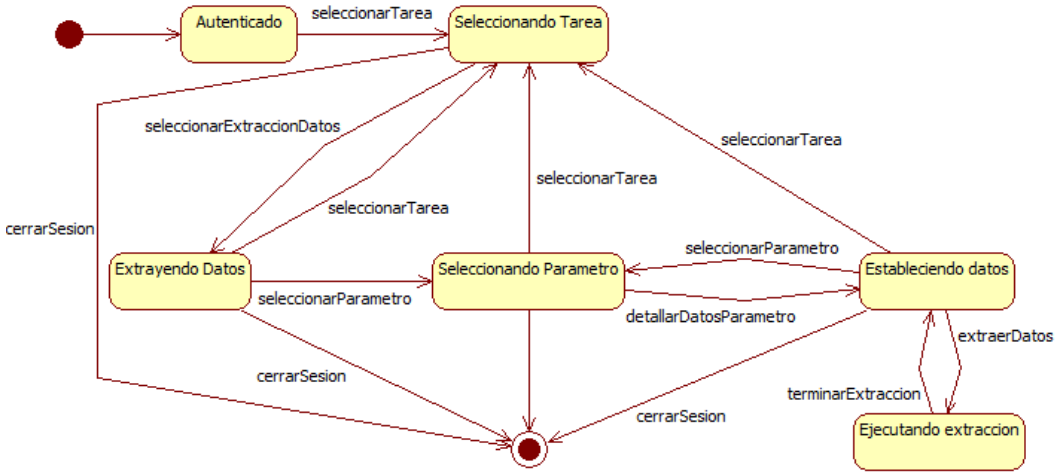
8. Diagramas de Estados

8.1. Evaluación

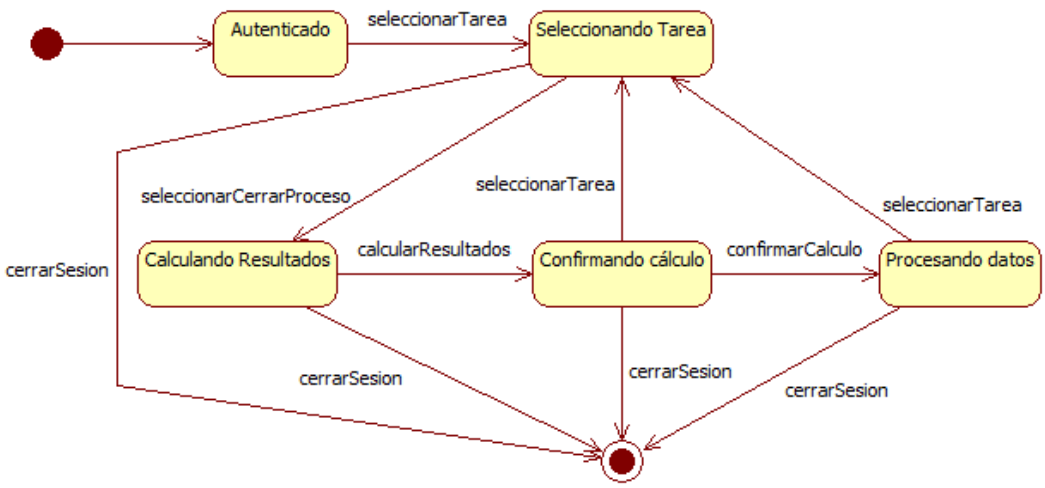


8.2. Administración

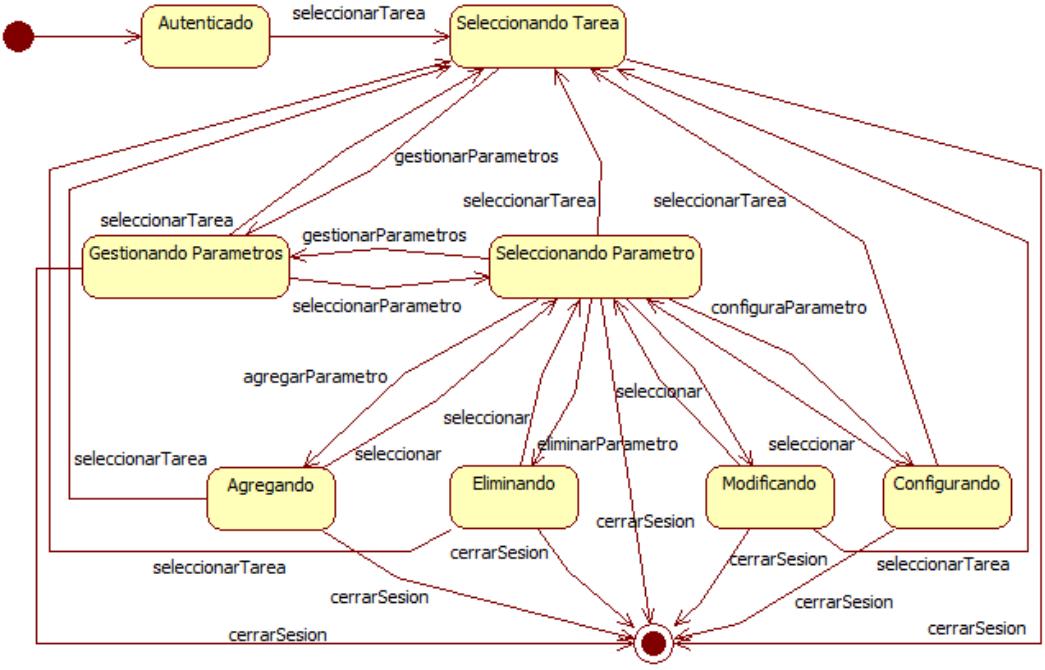
8.2.1. Extracción de Datos



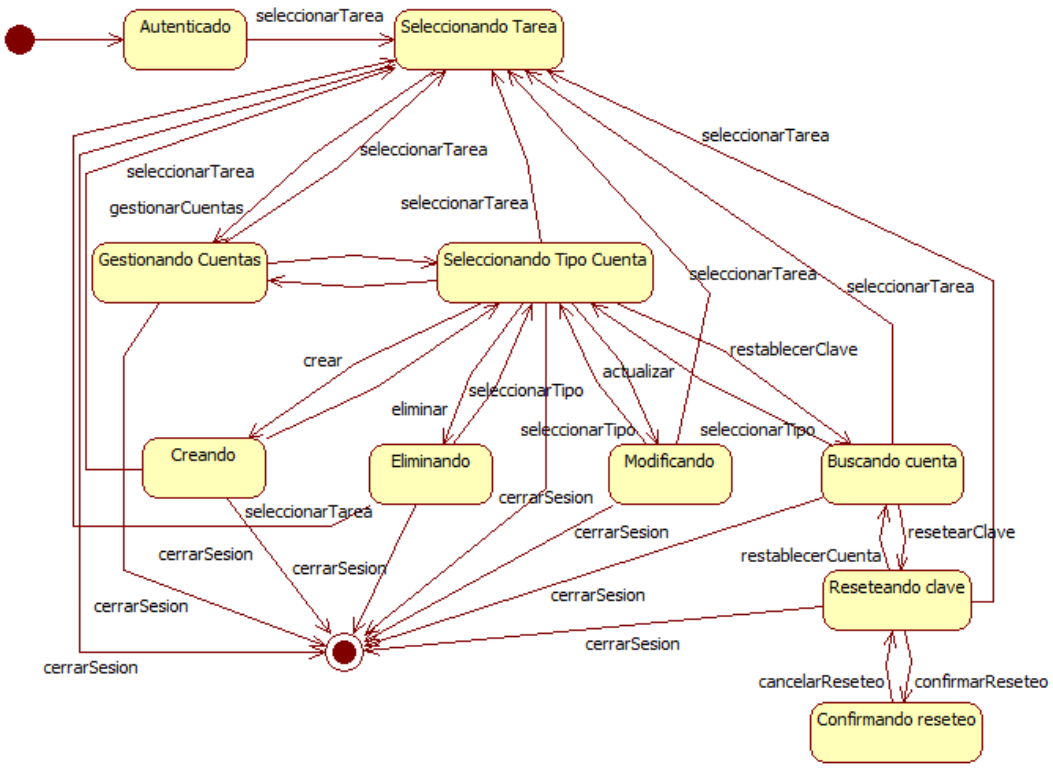
8.2.2. Cerrar Proceso



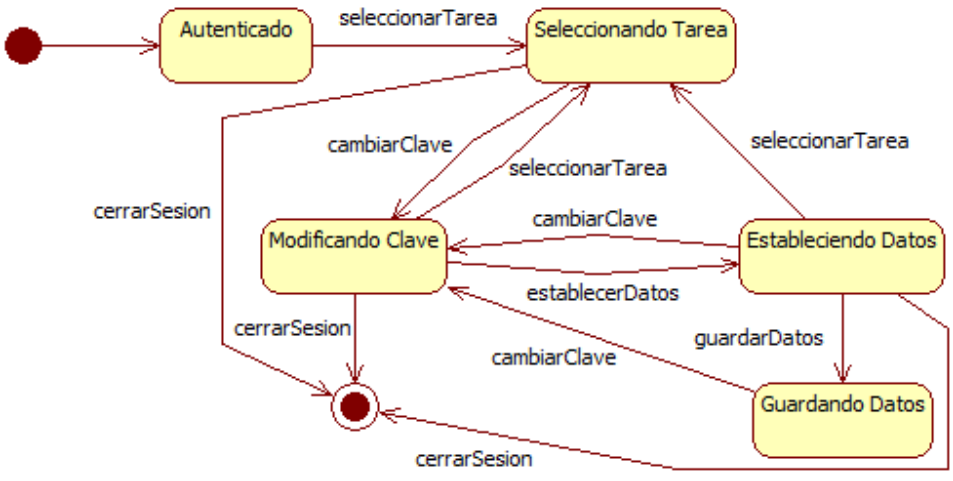
8.2.3. Gestionar Parámetros



8.2.4. Gestionar Cuentas



8.2.5. Modificar Clave



8.2.6. Visualizar Reportes

9. Prototipo del sistema

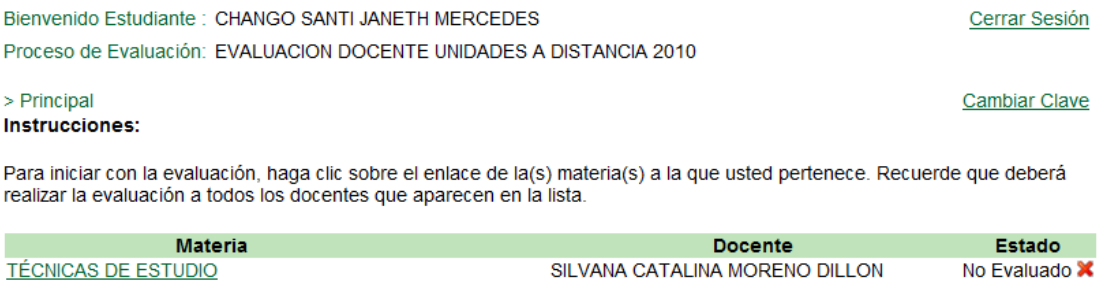
En el presente apartado se bosquejan, ciertas pantallas que conformarán el sistema.

9.1. Página de Autenticación

La manera de autenticarse inicialmente será a través de su número de cédula con guión como Cuenta de Usuario y la Clave de igual manera el número de cédula pero sin guión, y seleccionando el Tipo de Usuario respectivo.

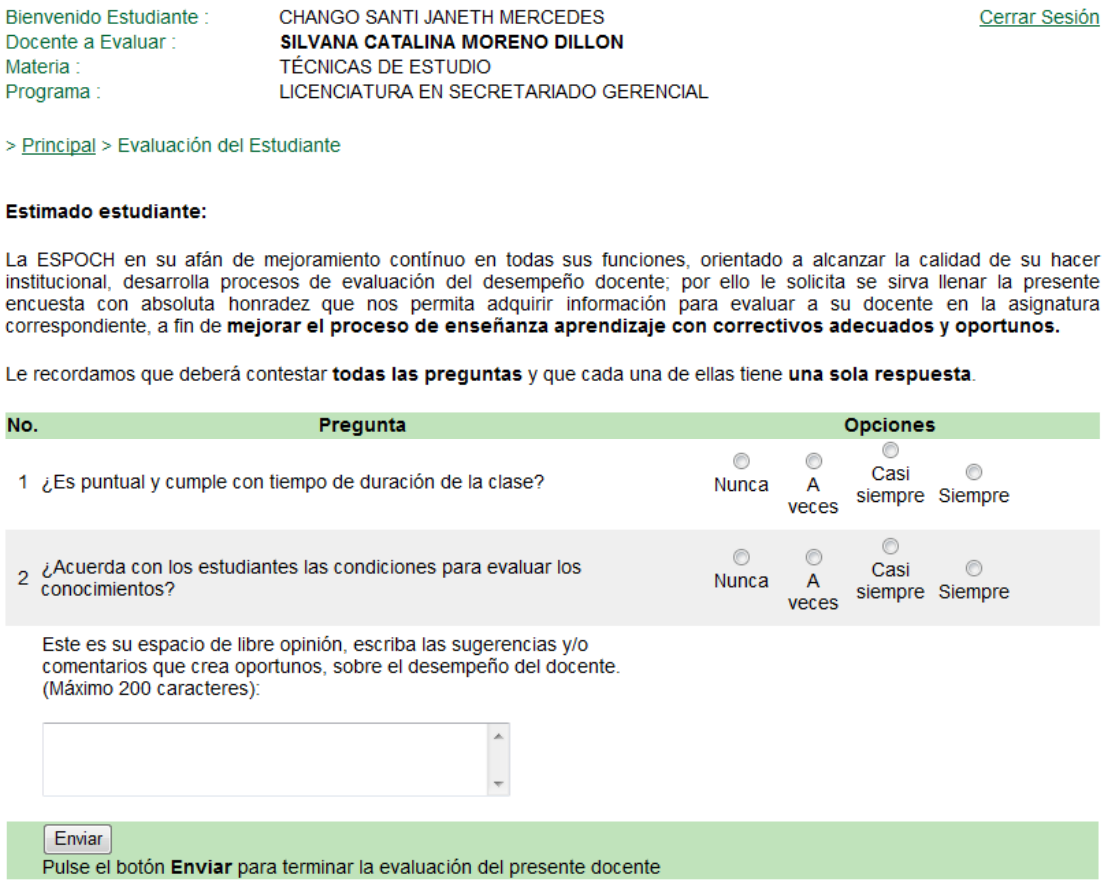
9.2. Página de Evaluaciones Disponibles

Se presentará un listado de las Materias con el nombre del Docente, que toma en estudiante y que podrá evaluar de acuerdo al estado (Evaluado, No Evaluado, Inactivo) en el que se encuentra dicha materia.



9.3. Página de Encuesta

Se presentarán cada una de las preguntas con sus respectivas opciones de respuesta.



Bienvenido Estudiante : CHANGO SANTI JANETH MERCEDES — Cerrar Sesión
Docente a Evaluar : **SILVANA CATALINA MORENO DILLON**
Materia : TÉCNICAS DE ESTUDIO
Programa : LICENCIATURA EN SECRETARIADO GERENCIAL

> Principal > Evaluación del Estudiante

Estimado estudiante:

La ESPOCH en su afán de mejoramiento contínuo en todas sus funciones, orientado a alcanzar la calidad de su hacer institucional, desarrolla procesos de evaluación del desempeño docente; por ello le solicita se sirva llenar la presente encuesta con absoluta honradez que nos permita adquirir información para evaluar a su docente en la asignatura correspondiente, a fin de **mejorar el proceso de enseñanza aprendizaje con correctivos adecuados y oportunos.**

Le recordamos que deberá contestar **todas las preguntas** y que cada una de ellas tiene **una sola respuesta**.

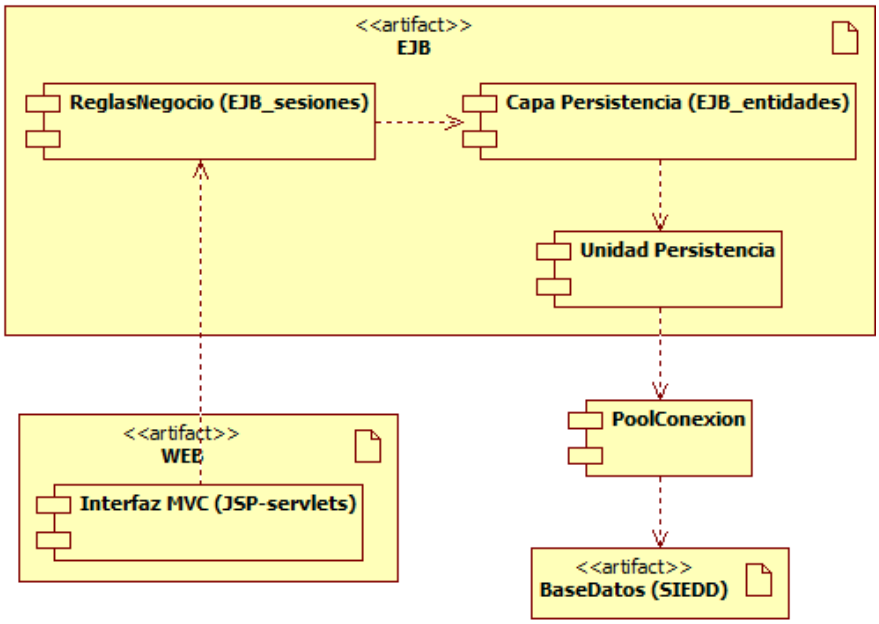
No.	Pregunta	Opciones
1	¿Es puntual y cumple con tiempo de duración de la clase?	Nunca / A veces / Casi siempre / Siempre
2	¿Acuerda con los estudiantes las condiciones para evaluar los conocimientos?	Nunca / A veces / Casi siempre / Siempre

Este es su espacio de libre opinión, escriba las sugerencias y/o comentarios que crea oportunos, sobre el desempeño del docente. (Máximo 200 caracteres):

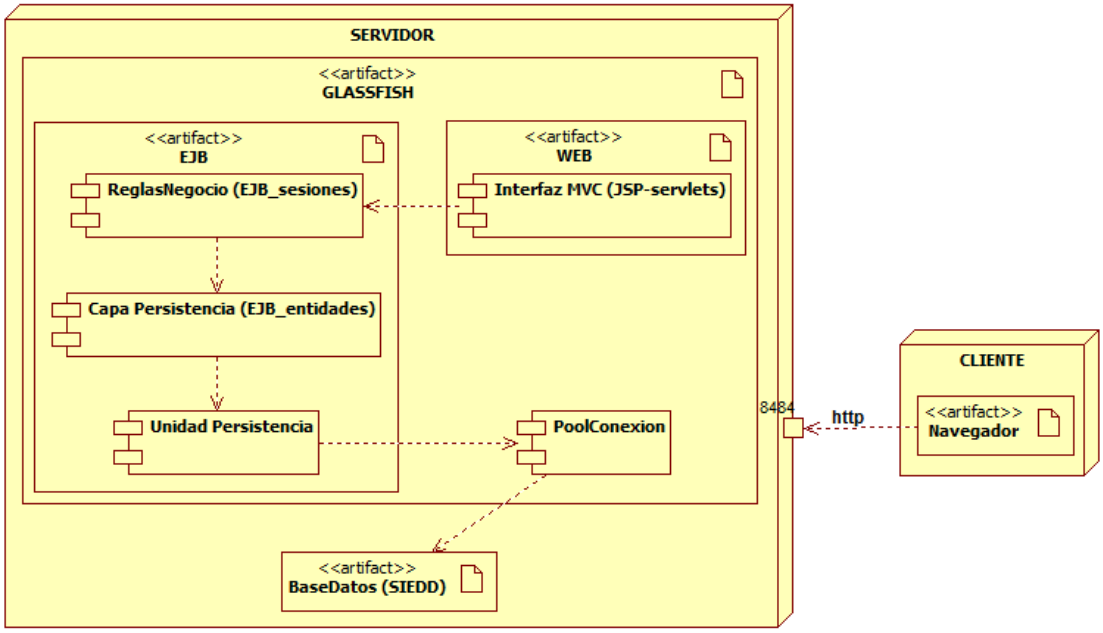
Enviar

Pulse el botón **Enviar** para terminar la evaluación del presente docente

10. Diagrama de Componentes



11. Diagrama de Despliegue



IV. IMPLEMENTACIÓN

1. Herramientas Hardware y Software utilizadas

El framework sobre el que se va a trabajar es JEE (Java Enterprise Edition), el cual tiene como lenguaje de programación base a JAVA, con las funcionalidades y portabilidades propios de esta herramienta. La herramienta de modelado será StarUML 2000 y la de desarrollo será el IDE NetBeans 6.9, con todas las prestaciones que este ofrece.

Como gestor de base de datos se utilizará Oracle 10g XE (Express Edition) de licencia libre.

Por su parte como contenedor o servidor de aplicaciones se hará uso de GlassFish Server 3.1.

Finalmente para la realización de la documentación técnica y los manuales de usuario, se utilizará Microsoft Word 2007, y a su vez para establecer la planificación a llevarse a cabo en el desarrollo del proyecto se lo realizará con Microsoft Project 2003.

En cuanto de herramientas hardware, se hará uso de una computadora portátil, cuyas características se detallan en la Tabla III.1.

HARDWARE	SOFTWARE
1 Laptop – 2 GB en RAM. – 320 GB espacio en disco. – Procesador Intel Centrino Core 2 Duo 2.0 GHz. – Sistema Operativo Windows 7.	- Oracle 10g XE - JDK (Java Development Kit) - IDE NetBeans 6.9 - GlassFish Server 3.1 - StarUML 2000 - Microsoft Office Word 2007 - Microsoft Office Project 2003

Tabla III.1. Resumen de Herramientas Hardware y Software

Los estándares de programación que se seguirán son aquellos propuestos para el desarrollo de software tradicional por Java. Cabe mencionar que Java es una de las mejores herramientas en lo que se refiere a estandarización.

V. PRUEBAS

1. Plan de Pruebas

El Plan de Pruebas permite definir los aspectos a tomarse en cuenta durante las pruebas a las que se someterá el sistema. Permiten organizar la ejecución de las pruebas, de manera que los resultados obtenidos sean de real utilidad para el sistema.

2. Objetivos

- Determinar si el sistema cumple con la funcionalidad planteada.
- Conocer el nivel de satisfacción que brinda a los usuarios.
- Determinar los errores del sistema para su oportuna corrección.

3. Planificación

Las pruebas del sistema se organizarán en dos etapas, determinadas por su distinta orientación. La primera etapa que incluirá dos tipos básicos de pruebas requeridas por la metodología de desarrollo: la segunda etapa se orientará a la interacción de los usuarios con el sistema.

4. Definición de ambientes de Prueba

4.1 Pruebas de Unidad e Integración

Las pruebas de unidad e integración son realizadas en el ambiente de desarrollo, ya que se requiere comprobar la funcionalidad de subsistemas y componentes específicos. Dichas pruebas son realizadas por el desarrollador basándose en parámetros técnicos y la lógica del negocio.

4.2 Pruebas con el Usuario

Para la segunda etapa se define un entorno sencillo y se le solicita al usuario comprobar de manera definitiva el funcionamiento, la presentación, los flujos de trabajo y los procesos del aplicativo. Esta etapa permite la corrección de las deficiencias detectadas por el usuario.

5. Evaluación

5.1. Pruebas de Unidad e Integración

Estas pruebas se realizaron exitosamente, con un total de 20 iteraciones corridas, durante las cuales se realizaron las afinaciones del caso para dejar el sistema en óptimas condiciones de funcionamiento.

Durante esta etapa de pruebas se realizaron las validaciones técnicas necesarias, validando algoritmos, mensajes de error, navegación, tiempos de respuesta, carga, entre otros.

5.2. Pruebas con el Usuario

Estas pruebas fueron definitivas y una comprobación de todo lo trabajado a lo largo de la etapa de desarrollo, los resultados de las mismas dieron un saldo positivo y gran satisfacción por parte de la Unidad Técnica de Evaluación Interna.

El énfasis de estas pruebas se dio en la comprobación de los procesos como tal y los flujos de trabajo acorde a las definiciones en el SRS además de la carga que soporta tanto el gestor de base de datos como el servidor de aplicaciones.

De estas pruebas, se pueden mencionar lo siguiente:

- El sistema SIEDD se ha implantado y probado con estudiantes de Ingeniería en Marketing y de Ingeniería en Comercio de la Facultad de Empresas de la ESPOCH, dándonos como resultado un correcto funcionamiento.

- La autenticación de los usuarios, es la adecuada según lo establecido, y en consecuencia las funciones que puede realizar un usuario está relacionado correctamente con su cuenta de usuario, tanto en la aplicación FRONT como en la BACK.

- El sistema presenta mensajes de advertencia al momento de ingresar datos erróneos o no haber establecido los datos necesarios, como en el caso de si no se ingresa toda la información pedida al momento de autenticarse (campos en blanco).

- Para la debida eliminación de datos, el sistema solicita la respectiva confirmación de la tarea según lo previsto inicialmente.

- Para el almacenamiento de las respuestas el sistema lo realiza adecuadamente para cada una de las preguntas.
- Se comprobó que cada uno de los reportes estadísticos o listados que se pueden generar con el sistema, se obtienen de manera adecuada y consistente.

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

- Al aplicar el proceso de desarrollo orientado a objetos Rational Unified Process (RUP), ha permitido establecer los aspectos clave en el desarrollo del proyecto SIEDD V2.0 para la evaluación docente en la ESPOCH.

- Es muy importante mencionar que el poder formular una especificación de requerimientos completa y consistente, es un paso muy importante para evitar cometer errores en la definición de los requerimientos, ya que los mismos pueden resultar muy caros de corregir una vez desarrollado el sistema.

- El sistema desarrollado está listo para ser utilizado, debido que una vez culminada su implementación y pruebas de unidad e integración se realizaron las respectivas pruebas con el usuario, las cuales arrojaron resultados exitosos, estando apto para la utilización.

- El uso de una metodología ágil y que haga uso de los conceptos de orientación a objetos es realmente importante para poder ordenar las actividades que se desarrollan en cada una de las fases que nos planteé la misma.

- La fase primordial para un buen diseño, es la de análisis de requerimientos, ya que de acuerdo a esto se puede modelar correctamente las diversas funciones que debe cumplir y satisfacer el sistema.

Recomendaciones

- Cabe señalar que en el desarrollo moderno es de suma importancia, en lo posible, el realizar las pruebas de desarrollo con la presencia del usuario, pues esto permite sobre la marcha realizar las correcciones necesarias sugeridas por el mismo; e incluso corregir y mejorar procesos que no se definieron de manera correcta.

- Es preciso tener un conocimiento razonable del dominio en el que se integrará nuestro programa.

- En caso de que se desee agregar funcionalidad al SIEDD, será necesario seguir las normas de desarrollo, como son: la especificación de requerimientos, el diseño de cada uno de los diagramas (casos de uso, colaboración, secuencia, componentes, despliegue, etc.), para finalizar con la respectiva codificación, y por consecuencia la adecuada documentación de los cambios realizados, para poder tener un seguimiento adecuado de las modificaciones o actualización que se den al sistema a lo largo de su utilización para el proceso de evaluación docente.

- Será indispensable para desarrollar adecuadamente un conocimiento de Java, JSP, HTML, Hojas de Estilo, JavaScripts, y por supuesto de todo lo relacionado con la implementación de los EJB, para nuevas funcionalidades del sistema.

BIBLIOGRAFÍA

- Booch, G. – Jacobson, I. – RumbaughT, J., **El Lenguaje Unificado de Modelado**, Addison Wesley, España – 2010.
- Kotonya G. – Sommerville I., **Requirements Engineering. Processes and techniques,** J. Wiley, USA – 2010.
- Larman Craig, **UML Y PATRONES** – Sexta Edición, Prentice Hall, España – 2010.
- Martin Fowler, **UML Gota a Gota,** Prentice Hall, España – 2009.
- Pressman, R., **INGENIERÍA DEL SOFTWARE: Un enfoque práctico** -Octava edición, McGraw-Hill. España – 2010.
- Estándar IEEE Std 830, para realización del SRS.
- Herrera J., Lizka Johany. **"Ingeniería de Requerimientos, Ingeniería de Software".** España – 2010.

ANEXO 3

MANUAL DE ADMINISTRACIÓN

SIEDD

Sistema Integrado de Evaluación del Desempeño Docente

...Hacia La Excelencia Académica

SISTEMA INTEGRADO DE EVALUACIÓN

DEL DESEMPEÑO DOCENTE V2.0

MANUAL DE ADMINISTRACIÓN

AUTOR: Oswaldo Villacrés Cáceres

La información en este documento es propiedad del Autor (Producción) y de la ESPOCH (Centro de Investigación) sujeto a normas legales, no se admite su reproducción parcial o total, pudiendo ser utilizado con el fin de guiarse para la Administración del Sistema de Evaluación del Desempeño Docente V2.0 y darle el adecuado usoen cada uno de los procesos del sistema que son realizados por el Administrador del sistema.

1. Introducción

1.1 Presentación

SIEDD V2.0 es una aplicación empresarial, la cual fue creada con objetivo de aumentar el número de conexiones concurrentes permitidas a la misma además de incrementar sustancialmente el número de evaluadores diarios en el proceso de evaluación de la Escuela Superior Politécnica de Chimborazo.

Adicionalmente, el sistema permite la toma de decisiones a través de los diferentes tipos de reportes que se pueden generar tanto en la aplicación BACK como en la FRONT.

1.2 Requisitos del Sistema

1.2.1 Requisitos Software:

SERVIDOR
- Sistema Operativo Windows XP o superior.
- Gestor de Base de Datos Oracle 10g XE o superior.
- JDK – Java Development Kit.
- GlassFish Server 3.0 o superior.

CLIENTE
- Adobe Reader 8.0 o superior.
- Resolución de pantalla de 1024 x 768.
- Navegador.

1.2.2 Requisitos Hardware:

SERVIDOR
- Procesador: Intel Pentium IV 2.8 GHz (equivalente o superior).
- Memoria RAM: 2GB o superior.

CLIENTE
- Procesador: Intel Pentium III (equivalente o superior).
- Tarjeta de red.

1.3 Instalación

A continuación se detallan todos los pasos a seguir para la instalación (despliegue) y puesta a punto del SIEDD V2.0:

1. Instalar Oracle 10g XE. (Vea el apartado **2 Instalando Oracle 10g XE**).
2. Crear un esquema llamado SIEDD. **(**Veael apartado **3. Crear Esquema SIEDD).**
3. Crear la estructura de la BD e importar datos. **(**Vea el apartado **4. Creando Estructura de la BD e Importando Datos).**
4. Instalar el JDK (Vea el apartado **5. Instalando el JDK**)
5. Instalar GlassFish Server 3.1. (Vea el apartado **6. Instalando GlassFish Server 3.1.).**
6. Crear y Configurar Pool de Conexiones y Datasets. (Vea el apartado **7. Creación y configuración del pool de conexión y dataset**).

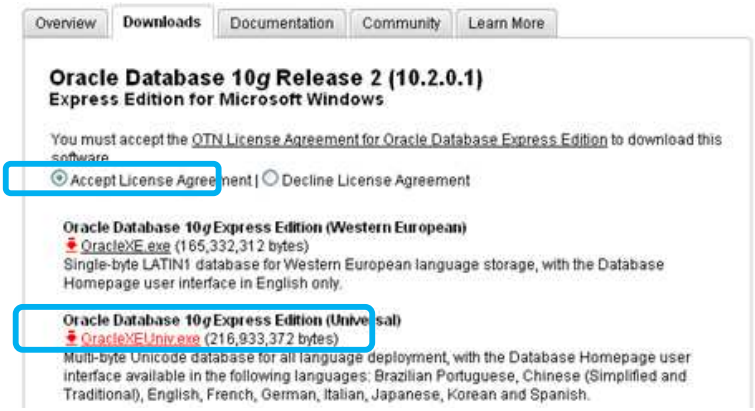
7. Desplegar el SIEDD V2.0. **(**Vea el apartado **8. Desplegando SIEDD V2.0).**
8. Inicie el navegador y tipie **http://localhost:8484/EI_principal** (para Evaluación) o **http://localhost:1223/EI_administracion** (para Administración).

2. INSTALANDO ORACLE 10g XE

PASO 1: Ingresar al sitio de Oracle, para lo cual en la Barra de Dirección del navegador tipiar: http://www.oracle.com/technetwork/database/express-edition/downloads/index.html.

PASO 2: Clic en el link respectivo, para descargar el instalador de Oracle 10g para Windows.

PASO 3: Acepte las condiciones de la Licencia, y seleccione el enlace **OracleXEUniv.exe**, enlace que le permitirá descargar Oracle 10g para varios idiomas, incluido el idioma Español.

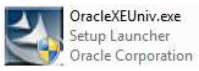


PASO 4: A continuación, se nos solicita registrarnos y una vez que lo hayamos realizado, utilizamos nuestras credenciales para acceder:

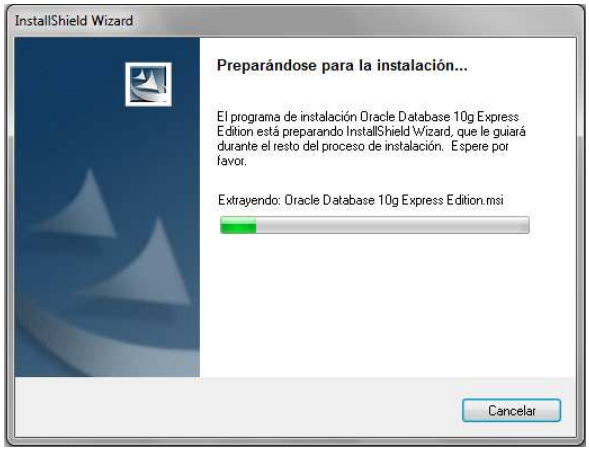
PASO 5: Si todo está correcto, se nos presentará una ventana como muestra la imagen a continuación, en donde daremos clic al botón **Guardar archivo**, para almacenar el instalador en el disco local de nuestra computadora.

Con lo cual iniciará la descarga, misma que puede tardar varios minutos dependiendo el ancho de banda.

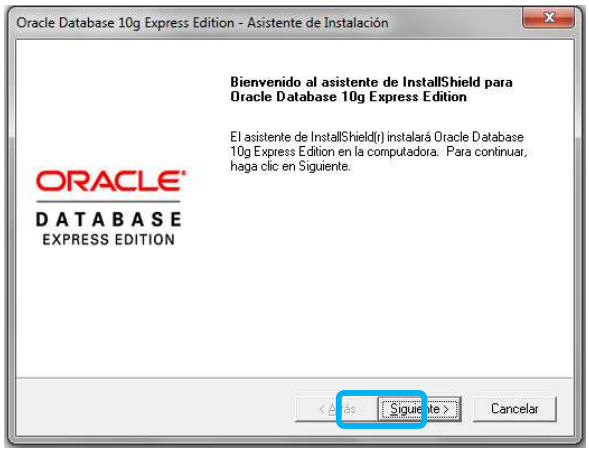
PASO 6: Ya culminada la descarga del respectivo instalador, doble clic sobre este, para iniciar el proceso de instalación.



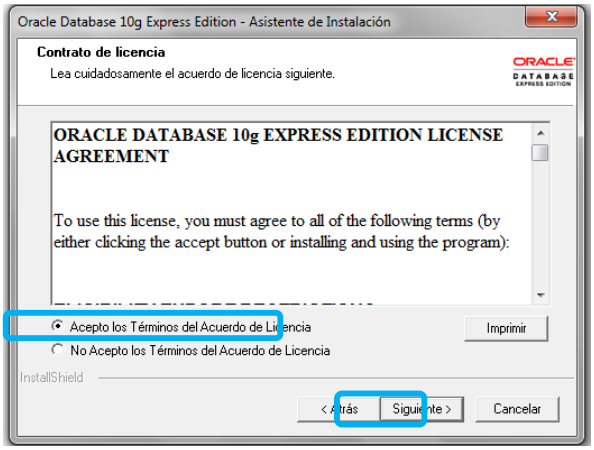
PASO 7: Tendrá que esperar por unos segundos hasta que se prepare el Asistente que nos guiará en el proceso de instalación.



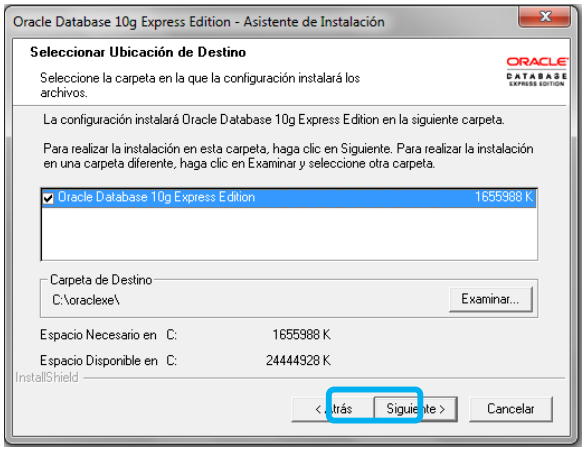
PASO 8: Una vez aparezca el Asistente de Instalación, dar clic al botón **Siguiente**.



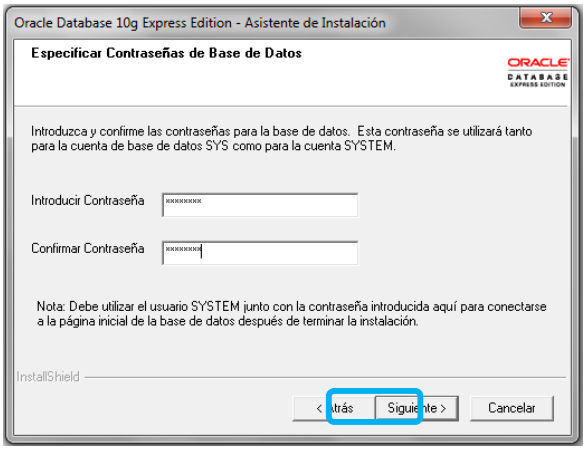
PASO 9: Aceptamos los Términos de la Licencia, y clic al botón Siguiente.



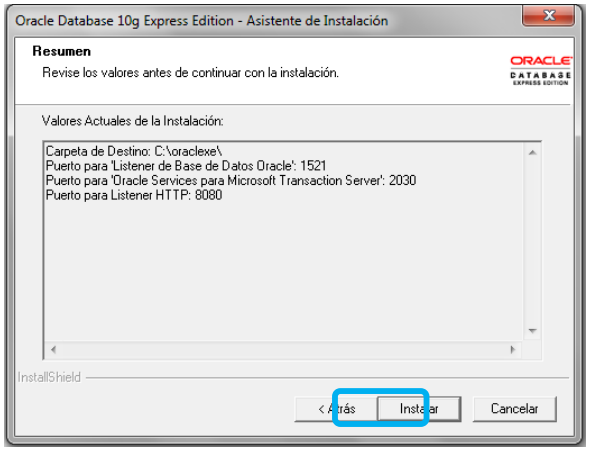
PASO 10: Establecemos el directorio (clic al botón **Examinar…**) en donde se guardará la instalación de Oracle, se recomienda dejar la carpeta de Destino con el valor por defecto, y a continuación clic al botón **Siguiente**.



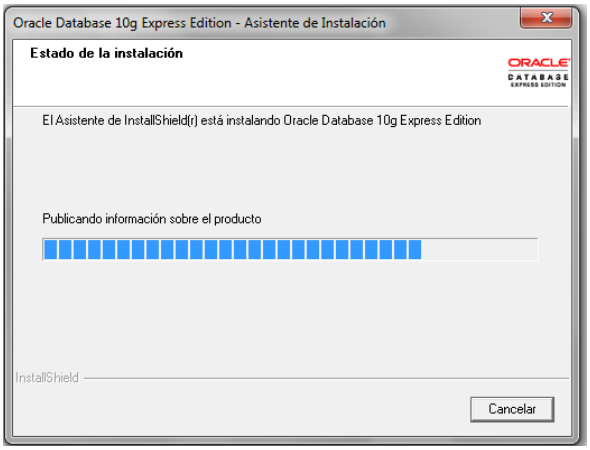
PASO 11: Se nos solicita establecer y confirmar la contraseña para las cuentas SYS y SYSTEM (Administradores de la Base de Datos), y a continuación clic al botón **Siguiente**.



PASO 12: Se nos presentará un resumen de los parámetros establecidos para la instalación, y si todo está de correcto, clic al botón **Instalar**.

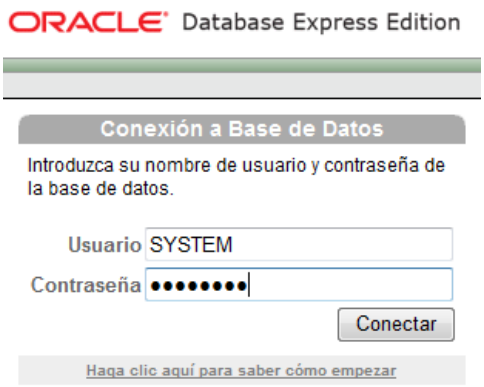


PASO 13: Esperamos por varios minutos hasta que culmine por completo la instalación de Oracle 10g XE.



PASO 14: Una vez culminado el proceso de instalación, se nos presentará una ventana similar a la de la imagen a continuación, en donde daremos clic al botón **Terminar**, y dejaremos habilitado **Iniciar la página inicial de la base de datos** si queremos hacer uso de inmediato de Oracle 10g XE.

PASO 15: Si dejamos marcado **Iniciar la página inicial de la base de datos**, se cargará en nuestro navegador la página para gestionar Oracle. En donde inicialmente tendremos que validarnos haciendo uso de la credencial **SYSTEM** y la contraseña establecida en el proceso de instalación, para conectarnos a Oracle 10g.



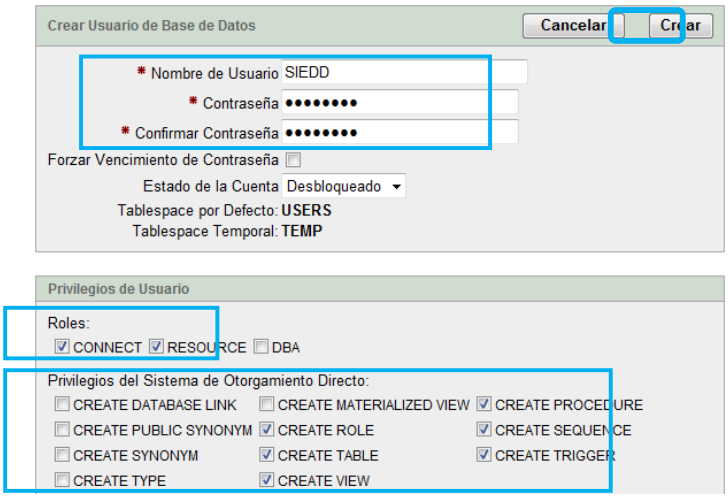
3. CREANDO EL ESQUEMA SIEDD

Antes de todo, hay que tener presente que a un Usuario le corresponde un Esquema con el mismo nombre, por lo tanto se procederá a crear una Cuenta de Usuario.

PASO 1: Nos autenticarnos como usuario administrador **SYSTEM**.

PASO 2: Clic en la de **Administración**, entonces del menú contextual que se despliega seleccionar el ítem, **Usuario de Base de Datos**, y a continuación **Crear Usuario**.

PASO 3: A continuación se presentará el formulario para creación del usuario (propietario del Esquema), esquema en el cual se almacenarán todos nuestros objetos, allí establecer el **Nombre de Usuario**, **Contraseña** y **Confirmar la Contraseña**. Además dejar marcadas los valores por defecto para los Roles, **CONNECT** y **RESOURCE** (para establecer conexión y crear objetos, respectivamente); y activar las casillas como muestra la imagen a continuación para los Privilegios. Y para finalizar la creación, clic al botón **Crear**.



PASO 4: Si todo está correcto, se nos presentará un listado de usuarios, similar al de la imagen siguiente:

PASO 5: Verificar que podemos conectar haciendo uso de la Cuenta de Usuario **SIEDD** que creamos.

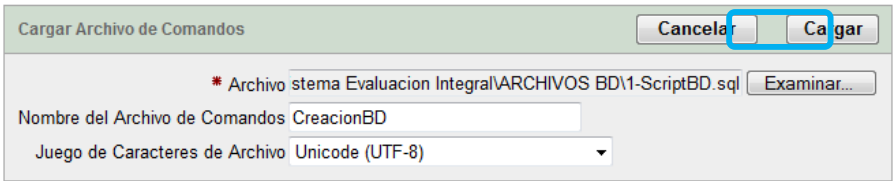
4. CREANDO ESTRUCTURA DE LA BD E IMPORTANDO DATOS

4.1. Crear Estructura de la BD.

PASO 1: Una vez creado el esquema SIEDD, nos autenticamos al mismo haciendo uso de las credenciales respectivas.

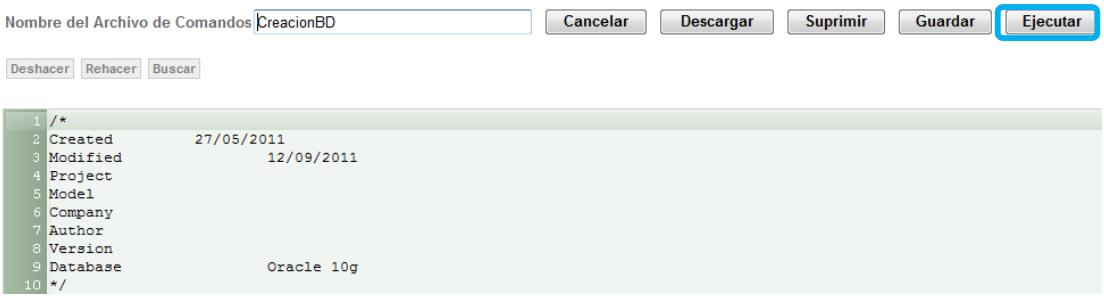
PASO 2: Clic sobre la flecha de **SQL**, nos dirigimos hasta **Archivo deComandos SQL** y a continuación seleccionamos **Cargar**.

PASO 3: Establecemos la ubicación del archivo que contiene el script de creación de la estructura de la base de datos (1-ScriptBD.sql) y opcionalmente detallamos un nombre, y a continuación clic al botón **Cargar**.



PASO 4: Clic sobre la imagen etiquetada con el nombre detallado en el paso anterior.

PASO 5: Clic al botón **Ejecutar** para iniciar con la creación.



PASO 6: Realizaremos los pasos 3, 4 y 5 para cada uno de los archivos de la carpeta **ARCHIVOS BD** y en el orden establecido.

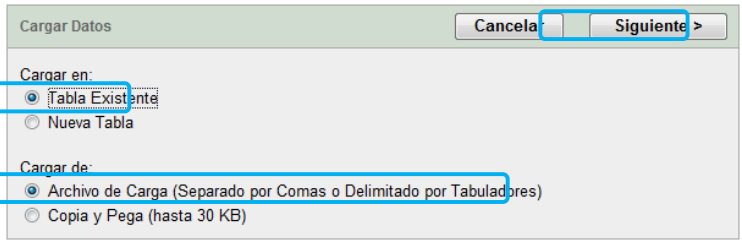
4.2. Importar Datos

PASO 1: Una vez creada la estructura de la base de datos SIEDD, nos autenticamos al mismo haciendo uso de las credenciales respectivas.

PASO 2: Clic sobre la flecha de **Utilidades**, nos dirigimos hasta **Carga/Descarga de Datos** y a continuación seleccionamos **Cargar**.

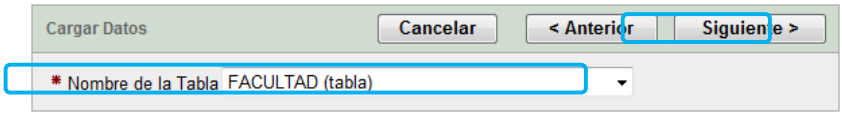
PASO 3: Seleccionamos **Cargar Datos de Texto**.

PASO 4: En la sección **Cargar en** seleccionamos **Tabla Existente** y en la sección **Cargar de** seleccionamos **Archivo de Carga** y a continuación clic al botón **Siguiente >**.

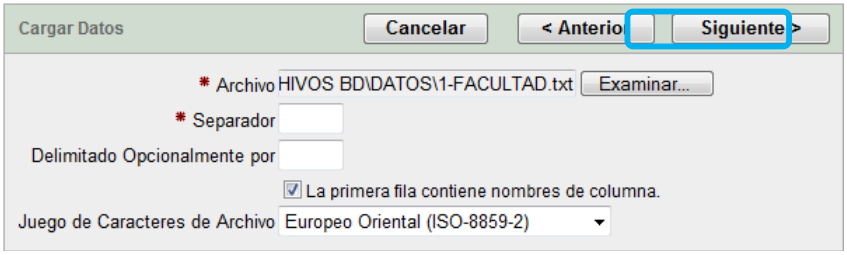


PASO 5: Clic al botón **Siguiente >**.

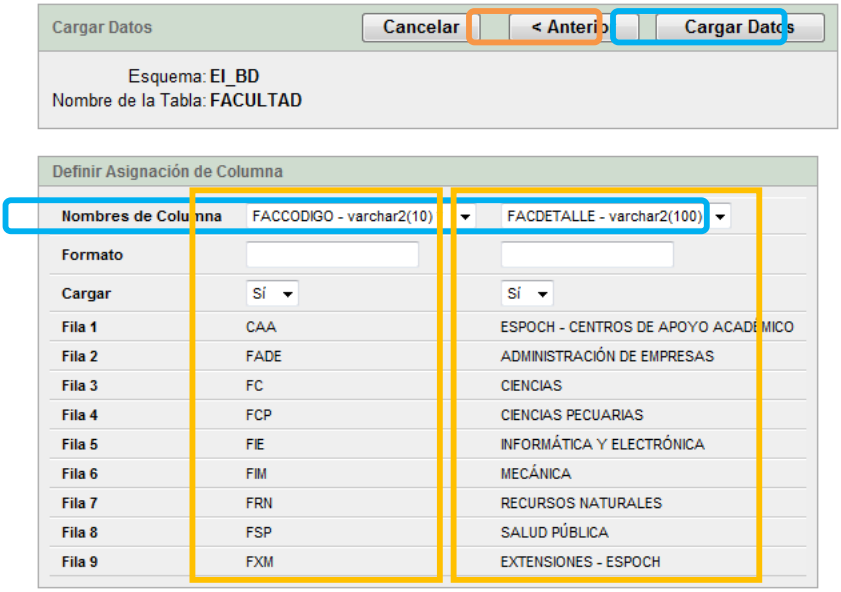
PASO 6: Seleccionamos la Tabla en la cual cargaremos los datos y clic al botón **Siguiente >**.



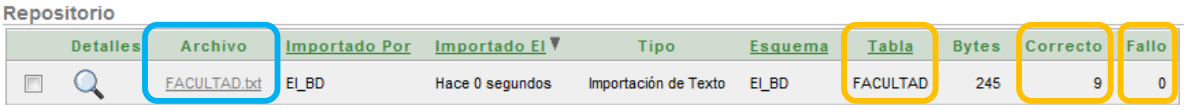
PASO 7: Establecemos la ubicación del archivo que contiene los datos, (1-FACULTAD.txt), establecemos el Separador (**Tab**, abrir el notepad presionar la tecla Tab y copiarlo a la casilla), seleccionamos el Juego de Caracteres, y a continuación clic al botón **Siguiente >**.



PASO 8: Cotejar la columna con el nombre del campo al que pertenezcan los datos de la parte inferior. Y si los datos visualizados en la tabla inferior están correctamente, clic al botón **Cargar Datos**, si no es así, clic al botón **< Anterior** y establecer otro Juego de caracteres hasta observar que los datos a ser cargados tengan el formato adecuado.



PASO 9: Finalmente se presentará el **Repositorio,** en donde se resumen todos los archivos que hemos cargado hasta el momento (**Archivo**), la tabla afectada (**Tabla**), el número de filas insertadas correctamente (**Correcto**) y el número de errores producidos (filas que no se insertaron).

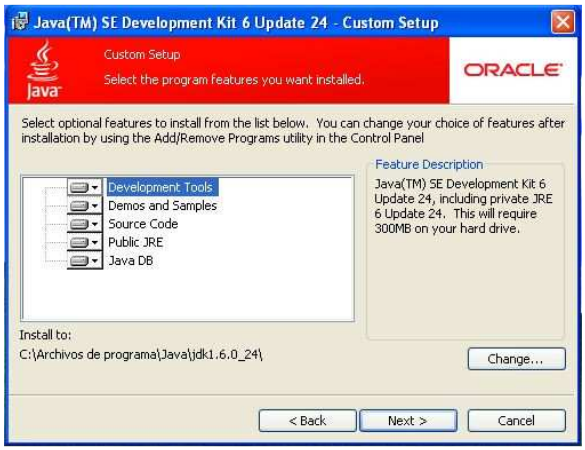


5. INSTALANDO EL JDK

PASO 1: Una vez descargado el JDK desde (**https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=jdk-6u24-oth-JPR@CDS-CDS_Developer**), doble clic sobre el instalador del JDK.

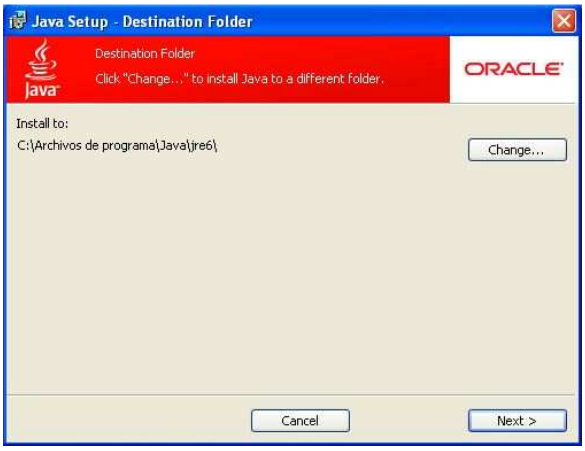
PASO 2: En el asistente que se nos presenta, clic al botón **Next >**.

PASO 3: Establecer el directorio en donde queremos instalar el **jdk**, para este caso se dejará el que se nos da por defecto, y clic al botón **Next >**.



PASO 4: Espera por uno minutos hasta que se instale el JDK.

PASO 5: Establecer el directorio de instalación para el **jre**, y de igual manera se dejará el directorio por defecto, y clic al botón **Next >**.



PASO 6: Nuevamente esperamos por unos minutos hasta que concluya la instalación del **jre**.

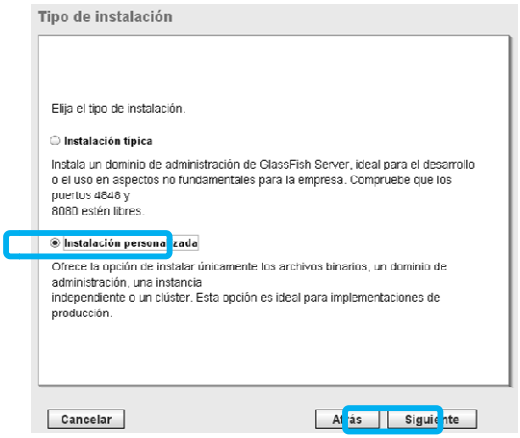
PASO 7: Para finalizar la instalación clic al botón **Finish**.

6. INSTALANDO GLASSFISH SERVER 3.1

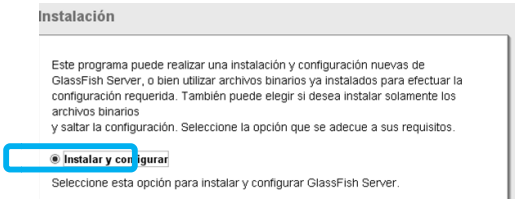
PASO 1: Clic sobre el icono del instalador de GlassFish Server 3.1.

PASO 2: Clic al botón **Siguiente**.

PASO 3: Seleccionar **Instalación personalizada** (ya que necesitaremos configurar el puerto para el http) y clic al botón **Siguiente.**

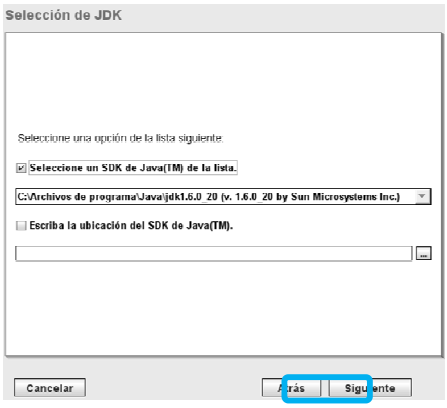


PASO 4: Seleccionamos **Instalar y configurar** y clic botón **Siguiente**.

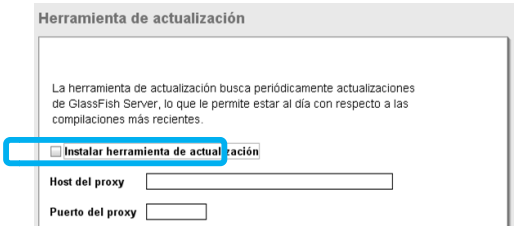


PASO 5: Seleccionamos la ubicación para el GlassFish y clic al botón **Siguiente**.

PASO 6: Esperamos por unos segundos mientras el asistente localice automáticamente la ubicación del JDK (previamente instalado) y clic al botón **Siguiente**.

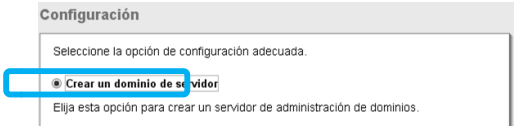


PASO 7: Desactivamos la casilla **Instalar herramientas de actualización** y clic al botón **Siguiente**

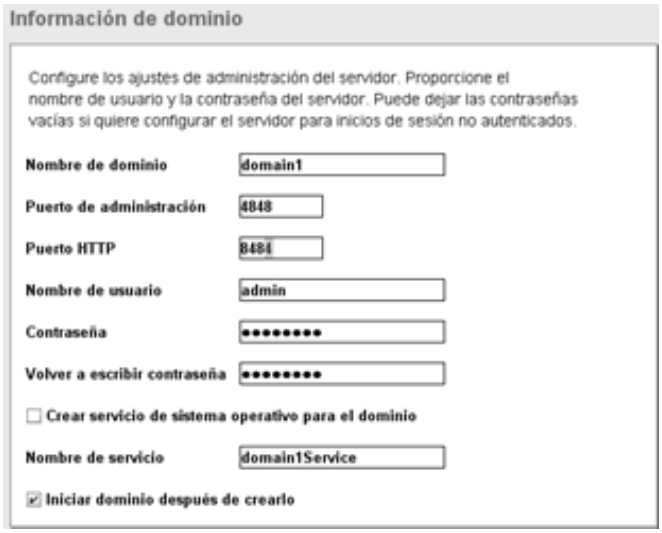


PASO 8: Clic al botón **Instalar** y esperamos por unos segundos mientras se instalan todas las herramientas.

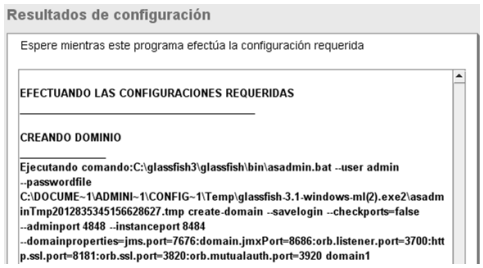
PASO 9: Seleccionamos **Crear un dominio de servidor** y clic al botón **Siguiente**.



PASO 10: A continuación establecemos el nombre del Dominio, los puertos para Administración y para el HTTP, la cuenta del usuario administrador y su contraseña, y damos clic en **Siguiente**.



PASO 11: Esperamos por unos segundos mientras se realizan las configuraciones respectivas.



PASO 12: Podremos observar un mensaje **Estado general: Completo** y clic al botón **Salir** para finalizar la instalación.

7. CREACIÓN Y CONFIGURACIÓN DEL POOL DE CONEXIÓN Y DATASET

7.1 Creación y Configuración del Pool de Conexión

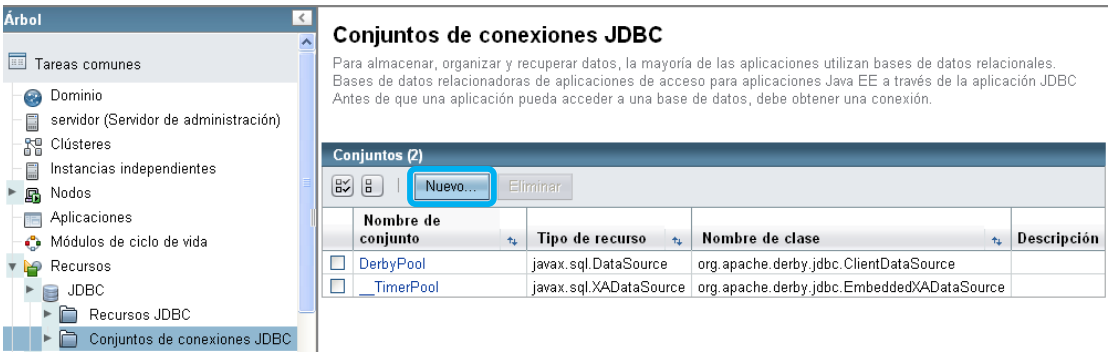
Una vez instalados, configurado e inicializado el servicio del servidor GlassFish, se podrá realizar la creación y configuración del pool de conexión, siguiendo los pasos mencionados a continuación:

PASO 1: Primero, antes de crear y configurar el pool de conexión será necesario agregar la librería **ojdbc14.jar**dentro del directorio **C:\glassfish3\glassfish\domains\domain1\lib\ext**.

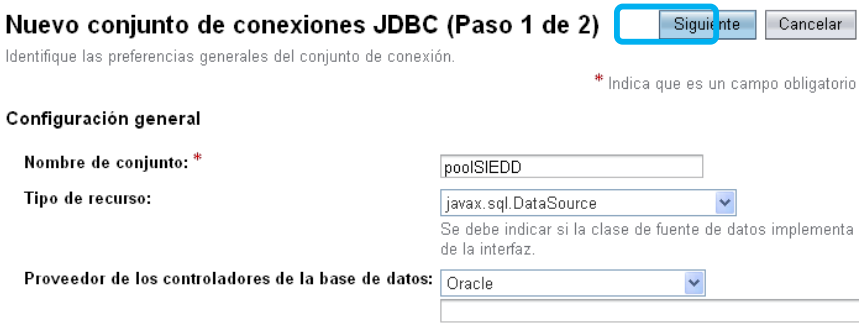
PASO 2: En la barra de direcciones del navegador tipiar, **http://localhost:4848/**.

PASO 3: Nos autenticamos utilizando las credenciales especificadas en el proceso de instalación.

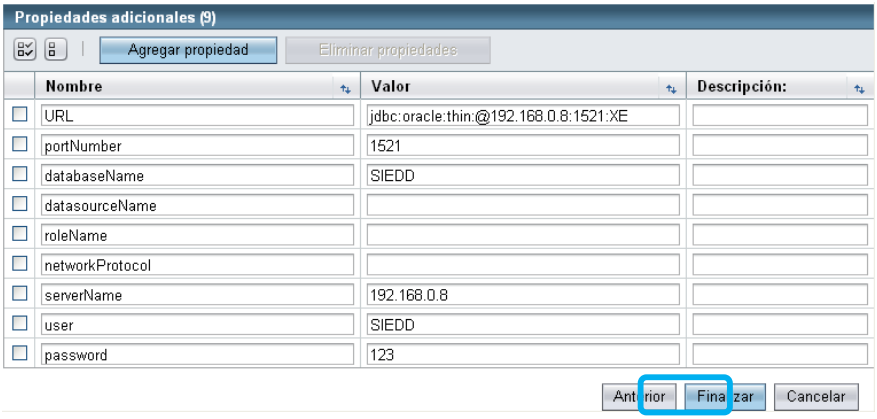
PASO 4: En el Árbol de navegación desplegamos **Recursos/JDBC/Conjuntos de conexiones JDBC** y allí damos clic al botón **Nuevo…**.



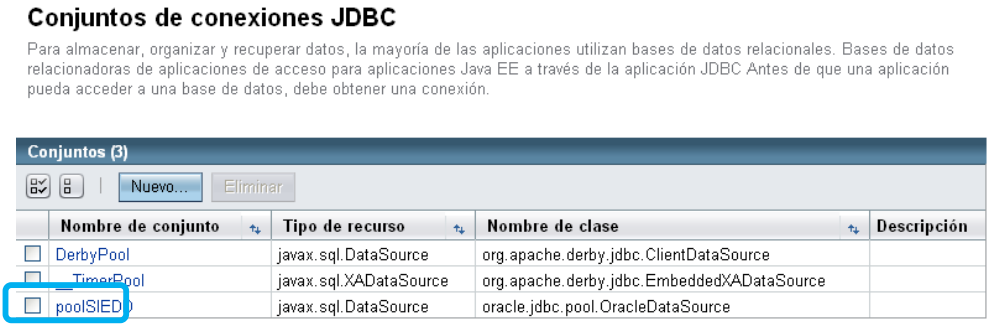
PASO 5: Especificamos el **Nombre**, y seleccionamos el **Tipo de recurso** (javax.sql.DataSource) y el **Proveedor** (Oracle), y a continuación clic al botón **Siguiente**.



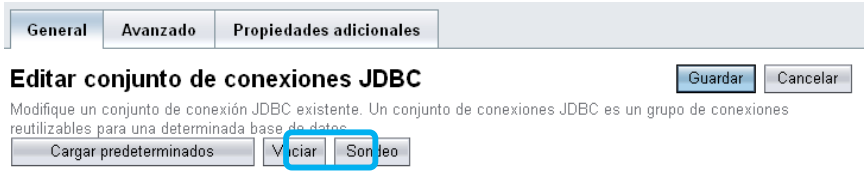
PASO 6: Configuramos los valores como muestra la imagen a continuación (podemos especificar la dirección IP del servidor de base de datos o en el caso que se encuentre en la misma podemos especificar simplemente **localhost**) y clic al botón **Finalizar**.



PASO 7: Podremos observar el nuevo pool creado en el listado.



Clic sobre el nombre pool creado, y a continuación clic sobre el botón Sondeo, para comprobar que se puede establecer comunicación con la base de datos correctamente.



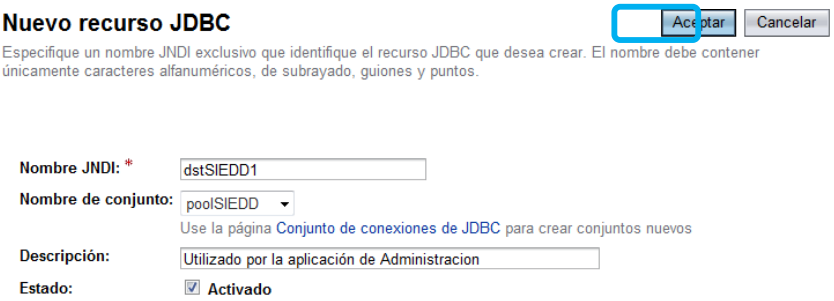
Podremos observar el siguiente mensaje.

7.2 Creación y Configuración del DataSet

Una vez creado y configurado el pool de conexión, podemos crear y configurar el dataset, como se indica en los pasos a continuación:

PASO 1: En el Árbol de navegación desplegamos **Recursos/JDBC/Recurso JDBC** y allí damos clic al botón **Nuevo…**.

PASO 2: Especificamos el **Nombre JNDI** (dstSIEDD1) y seleccionamos el **Nombre de conjunto** (poolSIEDD), y a continuación clic al botón **Aceptar**.



NOTA: Será necesario crear un dataset para cada aplicación (Administración y Evaluación), estrictamente con los siguientes nombres:

dstSIEDD1 para la aplicación de Administratición
dstSIEDD2 para la aplicación de Evaluación

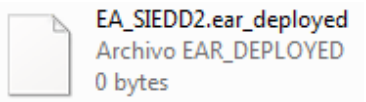
En donde cada dataset puede utilizar o no un mismo pool de conexión, esto dependerá de las necesidades de concurrencia en cada aplicación.

8. DESPLEGANDO SIEDD V2.0

Una vez creados los dos datasets necesarios, podemos desplegar nuestras aplicaciones, siguiendo los pasos mencionados a continuación:

PASO 1: Ubicamos el paquete (EA_SIEDD1 o EA_SIEDD2) de la aplicación respectiva en **C:\glassfish3\glassfish\domains\domain1\autodeploy.**

PASO 2: Esperamos por unos minutos mientras se desempaquetan todos los archivos automáticamente, y si todo está bien podremos observar un archivo similar al de la imagen a continuación, el cual nos indica que la aplicación ha sido desplegada correctamente.



NOTA: Será necesario crear un nuevo dominio en glassfish, para la otra aplicación, si deseamos tener las dos aplicaciones disponibles al mismo tiempo.

PASO 3: En la barra de direcciones del navegador tipiamos **http://localhost:8484/EI_principal** si desplegamos la aplicación **EA_SIEDD2**, y si desplegamos la aplicación **EA_SIEDD1** tipiamos **http://localhost:8484/EI_administracion**.

Para este caso la aplicación desplegada fue **EA_SIEDD2**.

ANEXO 4

MANUAL DE USUARIO

SIEDD

Sistema Integrado de Evaluación del Desempeño Docente

...Hacia La Excelencia Académica

SISTEMA INTEGRADO DE EVALUACIÓN

DEL DESEMPEÑO DOCENTE V2.0

MANUAL DE USUARIO

AUTOR: Oswaldo Villacrés Cáceres

La información en este documento es propiedad del Autor (Producción) y de la ESPOCH (Centro de Investigación) sujeto a normas legales, no se admite su reproducción parcial o total, pudiendo ser utilizado con el fin de guiarse en el uso del Sistema Integrado de Evaluación del Desempeño Docente versión 2.0 SIEDD V2.0 y darle el adecuado usoen cada una de los procesos del sistema.

1. Introducción

1.1 Presentación

SIEDD V2.0 es una aplicación empresarial, la cual fue creada con objetivo de aumentar el número de conexiones concurrentes permitidas a la misma además de incrementar sustancialmente el número de evaluadores diarios en el proceso de evaluación de la Escuela Superior Politécnica de Chimborazo.

Adicionalmente, el sistema permite la toma de decisiones a través de los diferentes tipos de reportes que se pueden generar tanto en la aplicación BACK como en la FRONT.

1.2 Requisitos del Sistema

1.2.1 Requisitos Software:

- Sistema Operativo Windows XP o superior.
- Navegador.
- Adobe Reader 8.0 o superior.
- Resolución de pantalla de 1024 x 768.

1.2.2 Requisitos Hardware:

- Procesador: Intel Pentium III (equivalente o superior).
- Tarjeta de red.

2. SIEDD V2.0

2.1. AUTENTICARSE

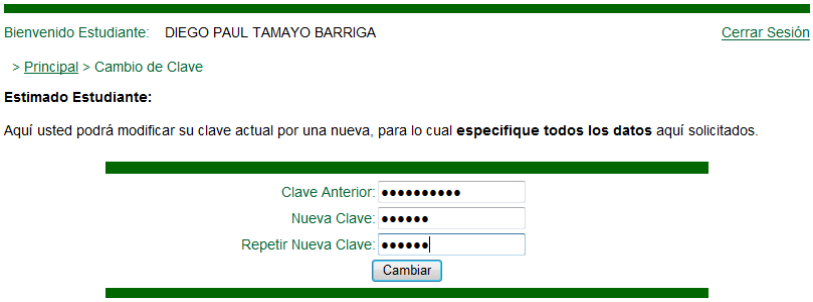
Una vez cargado el SIEDD V2.0 para evaluaciones, podremos observar la página de autenticación, en donde seleccionamos nuestro **Tipo de Usuario** (Estudiante, Docente, Directivo o Par Académico) y establecemos nuestro **Nombre de Usuario** (C. I. con guión) y la respectiva Clave (si no ha modificado será el C. I. sin guión).

Finalmente, clic al botón **Iniciar** o damos **Enter**.

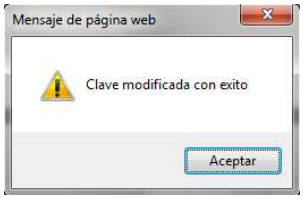
2.2. CAMBIAR CONTRASEÑA

PASO 1: Una vez autenticados, localizamos en la parte superior derecha el vínculo **Cambiar Clave** y damos clic sobre este.

PASO 2: Establecemos nuestra Clave Anterior, y la Nueva Clave la dos casillas siguientes, finalmente clic al botón Cambiar.



PASO 3: Si todo está bien, podremos observar una ventana informativa como la siguiente, en donde daremos clic en **Aceptar**.

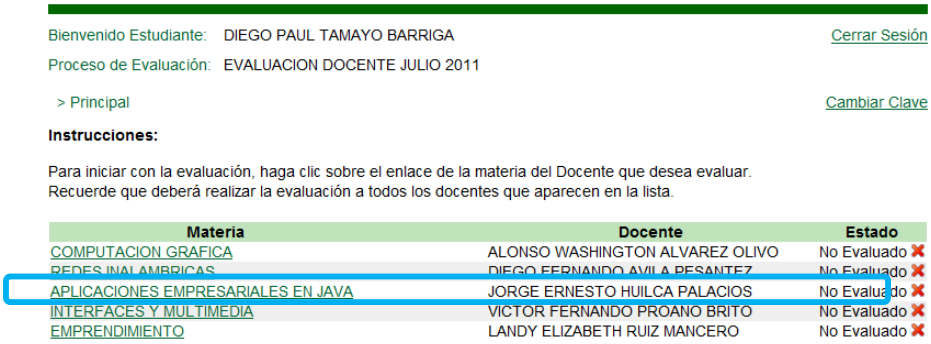


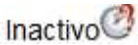
PASO 4: A continuación, localizamos el vínculo **Cerrar Sesión** y damos clic sobre este.

PASO 5: Nos volvemos a autenticar, haciendo uso de nuestra nueva clave.

2.3. EVALUAR

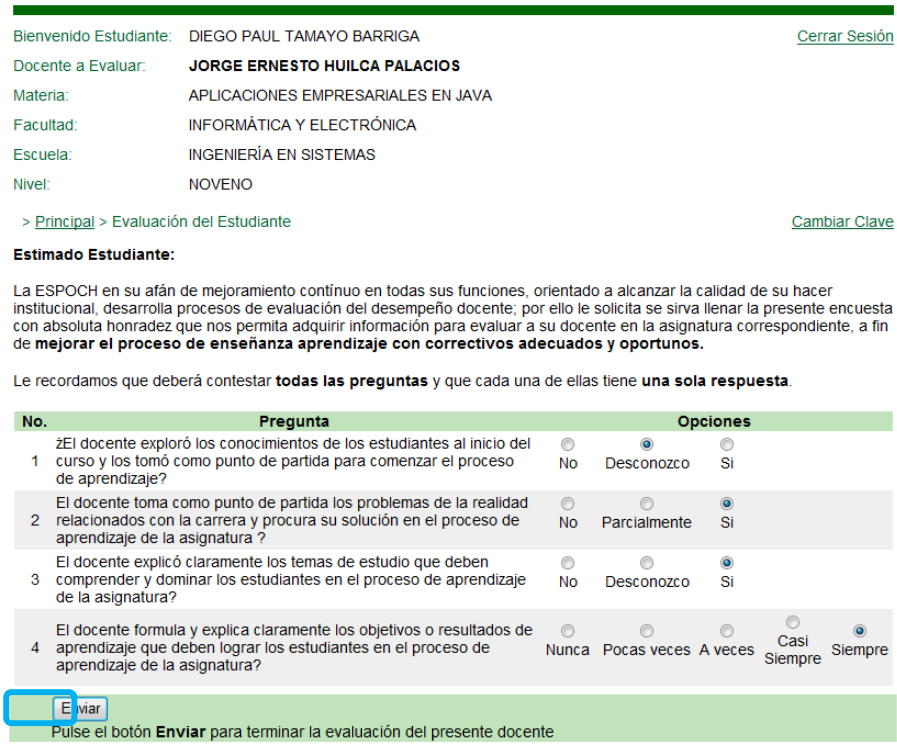
PASO 1: Una vez autenticados, podremos observar el listado de evaluaciones disponibles. De las cuales podemos dar clic sobre el vínculo correspondiente a realizar la evaluación y que no hayamos evaluado.



NOTA: Si observa una imagen similar a está, 

, quiere decir que el Proceso de Evaluación en su Escuela no está habilitado, o a su vez, que la fecha actual está fuera del rango de fechas establecidas para evaluar.

PASO 2: Establecemos nuestras respuestas para cada una de las preguntas dando clic sobre la misma, y finalmente clic al botón **Enviar** ubicado al final del cuestionario.



Bienvenido Estudiante: DIEGO PAUL TAMAYO BARRIGA — Cerrar Sesión

Docente a Evaluar: **JORGE ERNESTO HUILCA PALACIOS**

Materia: APLICACIONES EMPRESARIALES EN JAVA

Facultad: INFORMÁTICA Y ELECTRÓNICA

Escuela: INGENIERÍA EN SISTEMAS

Nivel: NOVENO

> Principal > Evaluación del Estudiante — Cambiar Clave

Estimado Estudiante:

La ESPOCH en su afán de mejoramiento contínuo en todas sus funciones, orientado a alcanzar la calidad de su hacer institucional, desarrolla procesos de evaluación del desempeño docente; por ello le solicita se sirva llenar la presente encuesta con absoluta honradez que nos permita adquirir información para evaluar a su docente en la asignatura correspondiente, a fin de **mejorar el proceso de enseñanza aprendizaje con correctivos adecuados y oportunos.**

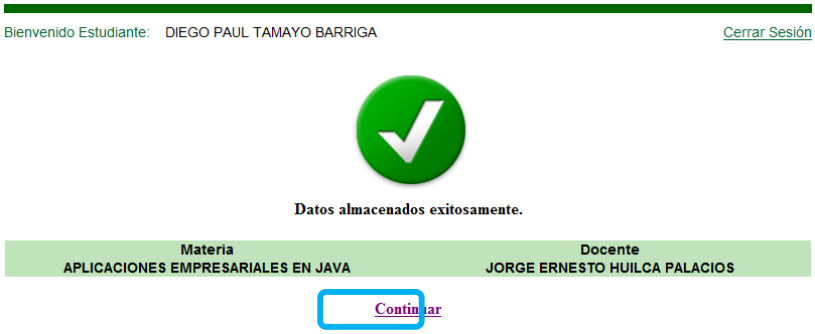
Le recordamos que deberá contestar **todas las preguntas** y que cada una de ellas tiene **una sola respuesta**.

No.	Pregunta	Opciones				
1	żEl docente exploró los conocimientos de los estudiantes al inicio del curso y los tomó como punto de partida para comenzar el proceso de aprendizaje?	No	Desconozco	Si		
2	El docente toma como punto de partida los problemas de la realidad relacionados con la carrera y procura su solución en el proceso de aprendizaje de la asignatura ?	No	Parcialmente	Si		
3	El docente explicó claramente los temas de estudio que deben comprender y dominar los estudiantes en el proceso de aprendizaje de la asignatura?	No	Desconozco	Si		
4	El docente formula y explica claramente los objetivos o resultados de aprendizaje que deben lograr los estudiantes en el proceso de aprendizaje de la asignatura?	Nunca	Pocas veces	A veces	Casi Siempre	Siempre

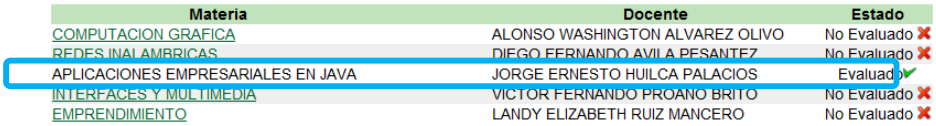
Enviar

Pulse el botón **Enviar** para terminar la evaluación del presente docente

PASO 3: A continuación, se nos informará de la evaluación que acabamos de realizar y damos clic en el vínculo **Continuar** para proseguir con las evaluaciones restantes.



Podremos observar en el listado de evaluaciones que el estado de la materia ha cambiado a **Evaluado**.

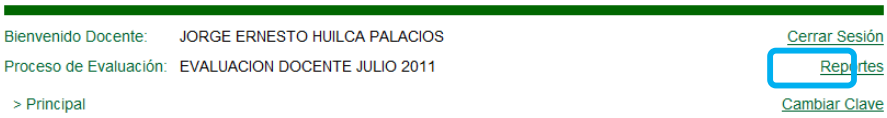


PASO 4: Una vez finalizada la evaluación de todas las materias clic al vínculo **Cerrar Sesión**.

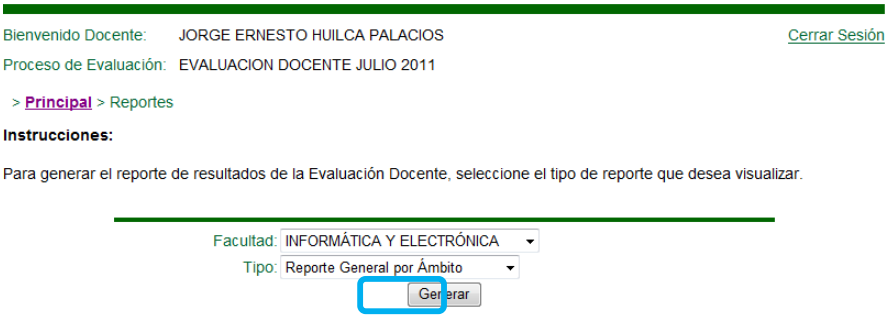
2.4. GENERAR REPORTES

En el caso que seamos Docentes, tendremos a disposición un vínculo para visualizar Reportes del actual Procedo de Evaluación, y que se podrán generar una vez concluido el periodo de evaluación.

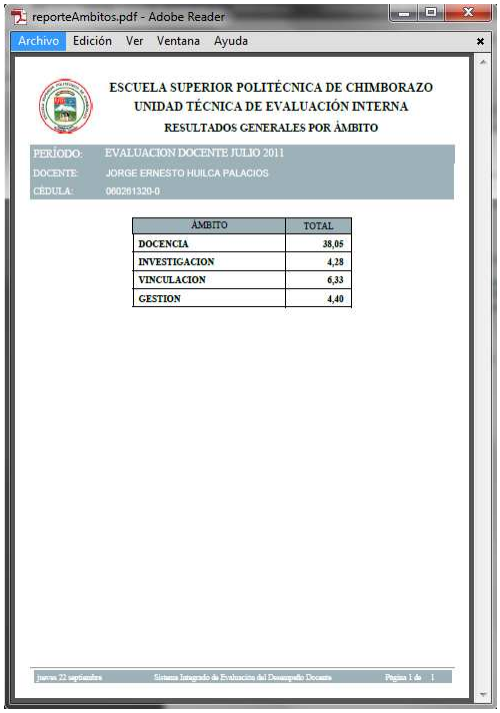
PASO 1: Localizar y dar clic sobre el vínculo **Reportes**, ubicado en la parte superior derecha.



PASO 2: Seleccionamos la **Facultad**, el **Tipo de Reporte** que deseamos obtener y clic al botón **Generar**.



Si tenemos instalado Adobe Reader, podremos observar el correspondiente reporte.



BIBLIOGRAFÍA

LIBROS

- **ANDERSON**., S., Estadísticas para Administración y Economía., 3a. ed., Thomson., México., 2010., Pp. 125 – 180.
- **BOOCH**., G., y. otros., El Lenguaje Unificado de Modelado., España., Addison Wesley., 2010., Pp. 60 – 85.
- **BLUMAN**., A., Statistics., España., Mc Graw Hill., 2010., Pp. 233 – 275.
- **FALKNER**., J., y. otros., Fundamentos Desarrollo Web con JSP. 2a. ed., España., ANAYA MULTIMEDIA., 2010., Pp. 165 – 200.
- **HERRERA**.,J., Ingeniería de Requerimientos, Ingeniería de Software., España., 2010., Pp. 95 – 125.
- **KOTONYA**., G., Requirements Engineering. Processes and techniques., USA., J. Wiley., 2010., Pp. 125 – 165.
- **PRESSMAN**., R., INGENIERÍA DEL SOFTWARE: Un enfoque práctico.,8a. ed., España., McGraw-Hill., 2010., Pp. 20 – 275.
- **ROZANSKI**., U., EJB 3.1., México D.F. – México., Alfaomega., 2010., Pp. 25 – 375.
- **Enterprise JavaBeansTM.**., JSR 318: Versión EJB 3.1.., 2010., Pp. 22 - 470.

1. **Distribución Normal Z.**
http://www.aulafacil.com/CursoEstadistica/
2011/02/18.

2. **Métodos Estadísticos**
http://descartes.cnice.mecd.es/
2011/02/18.

3. **Enterprise con Enterprise Java Beans**
http://www.ruf.rice.edu/~lane/stat_sim/sampling_dist/
2011/02/18.

4. **Singleton**
http://www.monografias.com/trabajos/anaydisesis/
2010/10/10.

5. **Java Persistence API**
http://www.gestiopolis.com/recursos/documentos/Dfulldocs/
2010/10/10.